TAMPEREEN TEKNILLINEN YLIOPISTO

JUSSI MÄKI

TASOLOIKKAPELIN TOTEUTTAMINEN

Diplomityö

Tarkastaja: Tommi Mikkonen
Tarkastaja ja aihe hyväksytty
tieto- ja sähkötekniikan tiedekuntaneuvoston kokouksessa 08.05.2013

TIIVISTELMÄ

TAMPEREEN TEKNILLINEN YLIOPISTO
Tietotekniikan koulutusohjelma
JUSSI MÄKI: Tasoloikkapelin toteuttaminen
Diplomityö, 58 sivua
Tammikuu 2014
Pääaine: Ohjelmistotuotanto
Tarkastajat: Professori Tommi Mikkonen
Avainsanat: Ohjelmistotuotanto, peliohjelmointi

Tasoloikka on suosittu videopeligenre ja monen aloittelevan ohjelmoijan tavoitteena on toteuttaa oma tasoloikkapeli. Tehtävä voi kuitenkin olla vaativa, sillä pelin toteuttamisessa on lukuisia yksityiskohtia ja toteuttajan tulee hallita monta eri osa-aluetta pelin toiminnassa. Ei ole olemassa standardoitua tapaa toteuttaa tasoloikkapeli, vaan toteuttaja joutuu monessa kohtaa valitsemaan erilaisista vaihtoehdoista jonkin asian toteuttamiseen ja väärä valinta voi hankaloittaa pelin kehitystä jatkossa.

Diplomityössä käydään läpi tasoloikkapelin toteutuksen olennaisimpia osa-alueita, jotta lukija saisi käsityksen mitä mihinkin alueeseen liittyy, ennen kuin alkaa itse kirjoittamaan ohjelmakoodia. Käsitellään pelin suunnittelua, kenttäeditoria, pelin logiikkaa, fysiikan mallinnusta, törmäystarkistusta, grafiikkaa, pelin äänimaailmaa, pelaajan antaman syötteen lukemista sekä esimerkkipeliä työssä käsitelleiden asioiden kannalta.

Asioita käsitellään varsin abstraktilla tasolla, sillä toteutus vaihtelee pelikohtaisesti monessa eri alueessa. Ei haluta antaa lukijalle käsitystä, että jokaiseen alueeseen on vain yksi oikea ratkaisu. Asioita käsitellään siten, että toteuttaja saa hyvän yleiskäsityksen asioista ennen kuin alkaa tutkimaan eri osa-alueita sen syvemmin ja toteuttamaan omaa peliään.

ABSTRACT

TAMPERE UNIVERSITY OF TECHNOLOGY
Master's Degree Programme in Information Technology
JUSSI MÄKI : Implementation of a Platform Game
Master of Science Thesis, 58 pages
January 2014
Major: Software Engineering
Examiner: Professor Tommi Mikkonen
Keywords: Software Engineering, Game Programming

Platform games are a popular genre of video gaming and it is many beginning programmer's aspiration to develop his or her own platform game. However, this can be a difficult task, for there are several details and the developer must grasp many different parts of the functionality of the game. There is no one standardised way to implement a platform game and the developer must choose between several choices to implement certain functionality in many points of the process, and a wrong choice can make the development difficult.

The thesis covers the most essential topics of implementation of a platform game to give the reader an understanding of things related to each area, before he or she begins to write any program code. The thesis covers design, level editor, game logic, physics, collision detection, graphics, sound, input and an example game in relation to the topics covered.

The topics are covered in quite an abstract level, for the implementation is game specific. It is not wanted to give the reader an impression that there is only one correct solution for every problem. The topics are covered in a way to give the developer a good general understanding of each area before he or she begins to focus deeper on different areas and begin the actual implementation.

ALKUSANAT

Tämä työ on kirjoitettu tietotekniikan tutkinto-ohjelmassa Tampereen teknillisessä yliopistossa. Työn kirjoittaminen alkoi alunperin syksyllä 2012, mutta se kirjoitettiin myöhemmin lähestulkoon kokonaan uudelleen ja valmistui loppuvuonna 2013. Alunperin ajatuksenani oli dokumentoida harrastusprojektina tekemäni tasoloikkapelin toteutusta suhteellisen matalalla tasolla, mutta nyt työ käsittelee yleisesti tasoloikkapelin toteutusta abstraktimmalla tasolla.

Kiitokset työn ohjaajalle, Tommi Mikkoselle sekä kaikille työtä kommentoineille.

Tampereella 26.11.2013,

Jussi Mäki

SISÄLLYS

1. Johdanto 1
2. Taustaa 3
2.1 Historiaa 3
2.2 Tasoloikkagenre 4
2.3 The Adventures of Nasse 6
3. Suunnittelunäkökohtia 7
3.1 Tasoloikkapelin maailma 8
3.2 Tieto-ohjattu ohjelmointi 10
3.3 Automaatio ja työkalut 10
3.4 Olio-ohjelmointi 11
3.5 Alusta 12
3.6 Pelisilmukka 13
3.7 Koodin tasojen erotus 13
3.8 Suunnittelumallit 14
3.9 Datasisällön jakaminen 14
4. Kenttäeditori 17
4.1 Editorin vaatimukset 17
4.2 Toimintaperiaatteet 17
4.3 Käyttöliittymä 17
4.4 Ikkunointi 18
5. Logiikka 21
5.1 Objektien käsittely 21
5.1.1 Hallinta korkeammalla tasolla 21
5.1.2 Peliobjektin sisäinen toiminta 22
5.1.3 Toiminnallisuuden jaottelu 23
5.2 Kamera 23
6. Fysiikka 25
6.1 Liike 25
6.1.1 Sijainti, s 25
6.1.2 Aika, t 26
6.1.3 Nopeus, v 26
6.2 Kiihtyvä liike ja dynamiikka 26
6.2.1 Kiihtyvyys, a 27
6.2.2 Dynamiikka 27
6.2.3 Gravitaatio 28
6.2.4 Liikemäärä, p ja Impulssi, J 29
6.2.5 Kitka 29

6.2.6 Noste . . . 30
6.2.7 Mallinnuksen toteuttaminen . . . 30
7. Törmäystarkistus . . . 33
7.1 Rajaus . . . 33
7.2 Posteriori (diskreetti) tarkistus . . . 33
7.3 Priori (jatkuva) tarkistus . . . 34
7.4 Törmäysvaste . . . 34
8. Grafiikka . . . 35
8.1 Rasterinäyttö . . . 35
8.1.1 Näyttötilat . . . 35
8.1.2 Ruudunpäivitys . . . 36
8.2 Spritet . . . 36
8.2.1 Toteutus . . . 37
8.2.2 Efektit . . . 37
8.3 Animaatio . . . 37
8.4 Laitteistokiihdytetty grafiikka . . . 38
8.5 Kaksiulotteinen grafiikka 3D-laitteistolla . . . 39
8.6 Vieritys . . . 39
9. Äänet ja musiikki . . . 40
9.1 Digitaalinen ääni . . . 40
9.2 Historiaa . . . 41
9.3 Äänen soittaminen . . . 41
9.3.1 Uudelleennäytteistys . . . 42
9.3.2 Kromaattinen sävelasteikko . . . 42
9.3.3 Miksaus . . . 42
9.3.4 Sekvensoitu ja nauhoitettu musiikki . . . 43
10. Input . . . 46
10.1 Input-laitteet . . . 46
10.2 Syötteen lukeminen ja käsittely . . . 47
11. The Adventures of Nasse . . . 50
11.1 Suunnittelu . . . 50
11.2 Kenttäeditori . . . 50
11.3 Logiikka . . . 52
11.4 Fysiikka . . . 53
11.5 Törmäystarkistus . . . 53
11.6 Grafiikka . . . 54
11.7 Äänet . . . 55
11.8 Input . . . 55
12. Yhteenveto . . . 56

Lähteet 57

1. JOHDANTO

Videopelit ovat kasvattaneet suosiotaan valtavasti ajan kuluessa ensimmäisten pelien ilmestymisen jälkeen. Peliteollisuus on kasvanut jopa elokuvateollisuuden vertaiseksi toimialaksi, ja videopelit ovatkin nykyään yksi suosituimmista viihdemuodoista kuluttajien keskuudessa. Kotitietokoneiden yleistyttyä yhä useammille käyttäjille on tullut mahdollisuus ohjelmoida omia sovelluksia. Nuoret käyttäjät haluavat usein toteuttaa ensimmäisenä sovelluksinaan videopelejä hyötyohjelmien sijaan. Erittäin suosittu valinta ensimmäiseksi peliksi on ohjelmoida tasoloikkagenreä edustava peli.

Vaikka tasoloikkapelit eivät pelattavuudeltaan välttämättä kovin monimutkaisia olekaan, ei niidenkään toteutus ole aivan yksinkertaista, varsinkaan jos toteuttaja ei ole aikaisemmin toteuttanut mitään ohjelmistoprojektia. Vähänkään monimutkaisempi tasoloikkapeli sisältää paljon tietosisältöä, joka kehittäjän pitää toteuttaa, sekä lukuisia pieniä yksityiskohtia niin korkean tason suunnittelussa kuin matalan tason ohjelmakoodissa. Ennen pelin varsinaista toteutusta pitää sen rakenne suunnitella hyvin jotta vältyttäisiin suurilta ongelmilta myöhemmin. Toteuttaakseen ohjelman eri osa-alueet, täytyy ohjelmoijalla olla näistä jonkin asteinen ymmärrys.

Tässä diplomityössä käydään läpi tasoloikkapelin toteutuksen olennaisimpia osa-alueita, jotta lukija saisi käsityksen, mistä osista tällainen peli tyypillisesti koostuu. Työssä on tarkoituksena käydä asiat läpi suhteellisen abstraktilla tasolla, jottei lukijalle jäisi käsitystä, että jokaiseen ongelmaan olisi vain yksi ja ainoa ratkaisutapa. Tämä työ on suunnattu lukijalle, jolla on ymmärrystä ohjelmoinnista ja joka hallitsee sujuvasti jonkin ohjelmointikielen, muttei välttämättä varsinaisesti ymmärrä erityisesti videopelien toteuttamisesta. Työ sopii siis hyvin lukumateriaaliksi vaiheeseen, jossa lukija on ensin oppinut ohjelmoimaan, ja haluaa saada kokonaiskuvan tasoloikkapelien toteuttamisesta ennen kuin alkaa opetella jonkin grafiikkakirjaston käyttöä ja tutkia osa-aiheita syvemmin.

Luvussa 2, ‘Taustaa’, määritellään videopelin käsite, käydään läpi videopelien historiaa sekä esitellään tasoloikkagenre. Luvussa 3, ‘Suunnittelunäkökohtia’, käsitellään pelin korkean tason suunnittelua ja pohditaan kuinka ohjelmistosuunnittelun erinäiset asiat liittyvät pelin toteutukseen. Luvussa 4, ‘Kenttäeditori’, käsitellään yleisellä tasolla yhtä tasoloikkapelien tärkeimmistä työkaluista. Mietitään, mitä editorilla pitäisi pystyä tekemään, sen toimintaperiaatteita sekä annetaan lukijalle käsitystä ikkunointijärjestelmän perusteista. Luvussa 5, ‘Logiikka’, tutkitaan pelin

ydintoiminnallisuutta joka huolehtii, että peli toimii oikein. Pohditaan, mitkä toteutusyksityiskohdat kuuluvat korkeamman tason ohjelmakoodille, ja mitkä asiat toteutetaan yksitellen eri pelihahmoille. Luvussa 6, 'Fysiikka' , käydään läpi suhteellisen yksinkertaisia mekaniikan ilmiöitä, joita mallintamalla saadaan tasoloikan hahmot liikkumaan oikealla tavalla fysiikan kannalta. Luvussa 7, 'Törmäystarkistus', tarkastellaan logiikkaa joka huolehtii etteivät pelihahmot kulje kiellettyjen maaston osien, kuten esimerkiksi seinien, läpi. Luvussa 8, 'Grafiikka', käydään läpi erinäisiä asioita, jotka liittyvät visuaalisen vasteen tuottamiseen pelaajalle. Luvussa 9, 'Äänet ja musiikki', käsitellään pelin äänimaailman eri toteutusyksityiskohtia. Tarkastellaan esimerkiksi mitä eroa on valmiiksi nauhoitetulla ja sekvensoidulla musiikilla, ja mitä jomman kumman käyttäminen pelissä merkitsee. Luvussa 10, 'Input', käydään läpi erilaisia ohjauslaitteita ja kerrotaan, kuinka niiltä saatua informaatiota tulisi käsitellä. Luvussa 11, 'The Adventures of Nasse' käsitellään esimerkkipeliä työn aiheiden näkökulmasta. Luvussa 12, 'Yhteenveto', tehdään johtopäätöksiä tasoloikkapelin toteuttamisesta käsiteltyjen asioiden perusteella.

2. TAUSTAA

Mahdollisimman yleisesti määritellen videopeli on interaktiivinen elektroninen peli [1], jonka kanssa pelaaja on vuorovaikutuksessa käyttäjärajapinnan kautta ja joka puolestaan antaa pelaajalle visuaalisen vasteen näyttölaitetta käyttäen. Näyttölaitteina on käytetty katodisädeputkinäyttöjä jotka olivat oskilloskooppeja, myöhemmin vektorinäyttöjä ja laskentatehon lopulta sallittua rasterinäyttöjä, jotka ovat nykyään käytännössä katsoen ainoita näyttölaitteita videopeleissä sekä yleiskäyttöisissä tietokoneissa. Rasterinäytöissä on siirrytty noin viimeisen kymmenen vuoden aikana analogisesta kuvaputkiteknologiasta TFT-teknologiiaa soveltaviin litteisiin näyttöihin. Ohjainlaitteita on ollut useita erilaisia, mutta kotitietokoneella käytetään usein näppäimistöä ja pelityypistä riippuen myös hiirtä, siinä missä pelikonsoleilla käsissä pidettävä gamepad on suosittu. Videopeleissä on pitkään käytetty musiikkia ja äänitehosteita, joista musiikista on tullut ajan kuluessa enemmän traditionaalista musiikkia muistuttavaa ja ääniefekteistä enemmän luonnossa esiintyvien äänten kuuloisia.

2.1 Historiaa

Varhaisten videopelien teemat liikkuivat sodan, mailalla pelattavien pallopelien ja avaruuden ympärillä, joista ainakin sota ja avaruus kuvastavat hyvin sen aikaista maailmaa. Näistä sotapelit ovat säilyttäneet suosionsa, luultavasti koska sota on aina ajankohtainen aihe. Ensimmäinen tunnettu interaktiivinen visuaalisen vasteen tarjoava elektroninen peli on vuonna 1947 esitelty cathode ray tube amusement device [2], jolla analogisilla ohjaimilla ohjattiin oskilloskoopilla esiintyvää pistettä ja koitettiin rajoitetussa ajassa tulittaa lentokoneita nappia painamalla. Laite oli täysin elektromekaaninen. Vuonna 1950 MIT:n Whirlwind-tietokoneelle kehitetty Bouncing Ball-ohjelma on ensimmäinen graafinen reaaliaikainen tietokoneohjelma [3]. Ensimmäinen varsinainen tietokonepeli on vuonna 1951 Festival of Britain-tapahtumassa esitellylle NIMROD-tietokoneelle tehty NIM [3]. Ensimmäinen graafinen tietokonepeli taas on vuonna 1952 kirjoitettu OXO, jota ajettiin Cambridgen yliopiston EDSAC-tietokoneella. Reaaliaikaisista graafisista tietokonepeleistä ensimmäinen on vuonna 1958 Brookhaven National Laboratoryssa kehitetty Tennis for Two [4]. Peliä ajettiin analogisella Donner Model 30-tietokoneella. Vuonna 1962 MIT:n PDP-1-tietokoneella ajettu Spacewar on ensimmäinen digitaalinen reaaliaikainen tietoko-

nepeli.

Myöhemmin keksittiin, että videopelin pelaamisesta voidaan periä rahaa. Ensimmäinen tunnettu kolikkokäyttöinen peli on vuonna 1971 tehty Galaxy Game [5]. Peliä ajettiin Stanfordin yliopistossa PDP-11/20-tietokoneella. Ensimmäinen massavalmisteinen kolikkopeli on samana vuonna tehty Computer Space [6]. Peli ei nimestään huolimatta ollut varsinaisesti tietokonepeli, sillä sitä ei ollut toteutettu prosessorilla ajettavalla tietokoneohjelmalla. Ensimmäinen varsinaisesti suosittu kolikkopeli on vuonna 1972 julkaistu Pong [7].

Videopeleistä alettiin myöhemmin tehdä kotona pelattavia versioita. Aluksi yhdellä koneella pystyttiin yleensä pelaamaan vain yhtä peliä, mutta myöhemmin julkaistiin konsoleja joihin pelimoduulin vaihtamalla pystyi pelaamaan useita erilaisia pelejä. Myös yleiskäyttöisiä tietokoneita alettiin myymään ihmisten koteihin, ja niilläkin oli mahdolista pelata. Videopelejä pelataan edelleen pelikonsoleilla ja yleiskäyttöisillä tietokoneilla sekä nykyään myös matkapuhelimilla. Kolikkokäyttöiset videopelit näyttäisivät kadonneen lähes kokonaan.

2.2 Tasoloikkagenre

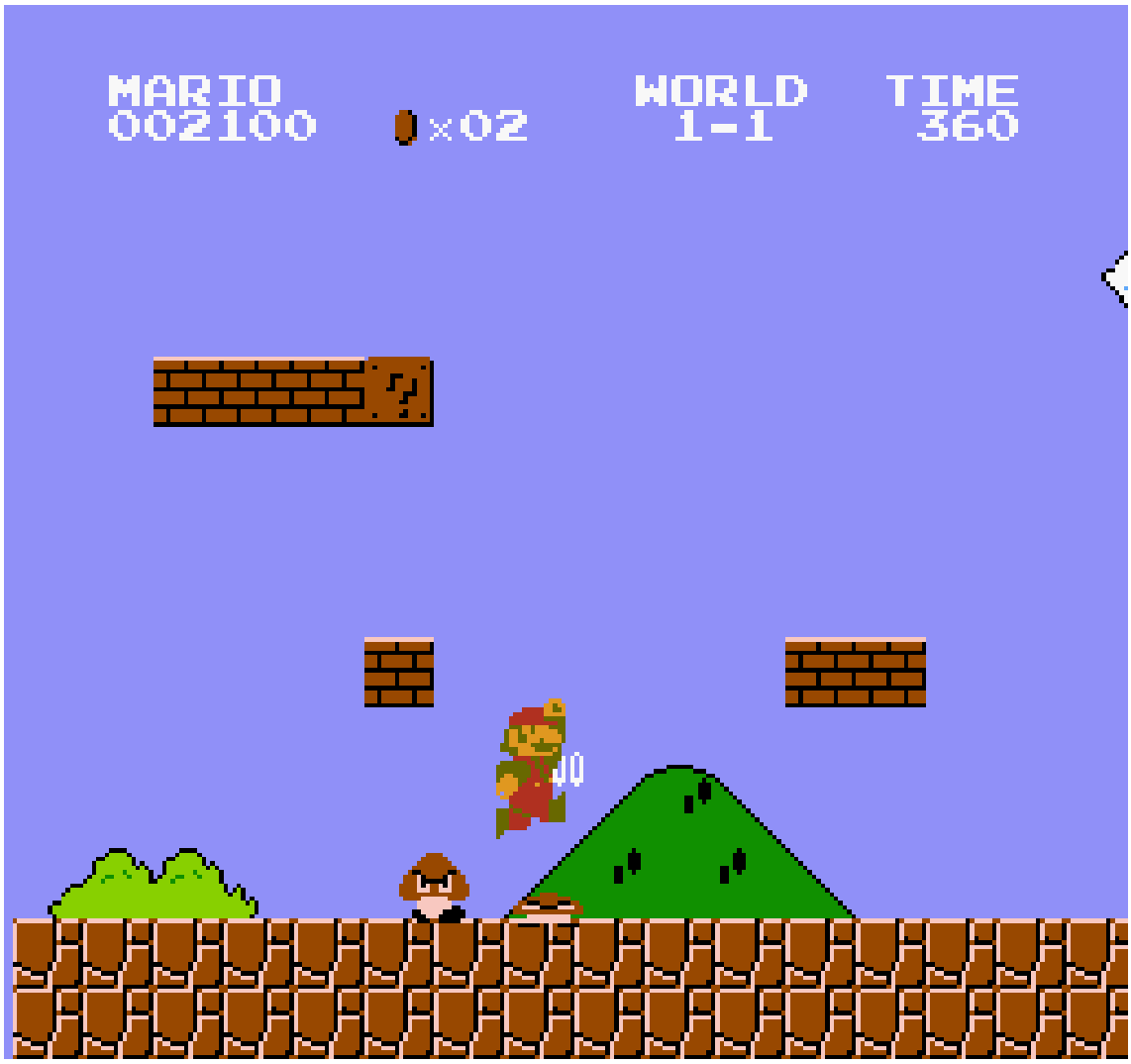
Tasoloikkapeleissä (platform game) pelaaja ohjaa pelihahmoa, joka hyppii tasoilla (platform). Näissä peleissä tärkeät kontrollit ovat suuntanapit ja hyppynappi. Useissa tasoloikkapeleissä on myös käytössä juoksunappi, jonka pohjassa pitäminen saa pelaajan kulkemaan nopeammin ja hyppäämään korkeammalle.

Useissa tasoloikkapeleissä (esim. Super Mario Bros.) pelaaja päihittää vihollisia hyppäämällä niiden päälle ja päämääränä on päästä maaliin. Tyypillisesti pelaaja kuolee tai menettää energiaa jos tämä koskee viholliseen muulla tavalla kuin päälle hyppäämällä. Toisaalta joissain tasoloikkapeleissä vihollisiin ei saa lainkaan koskea.

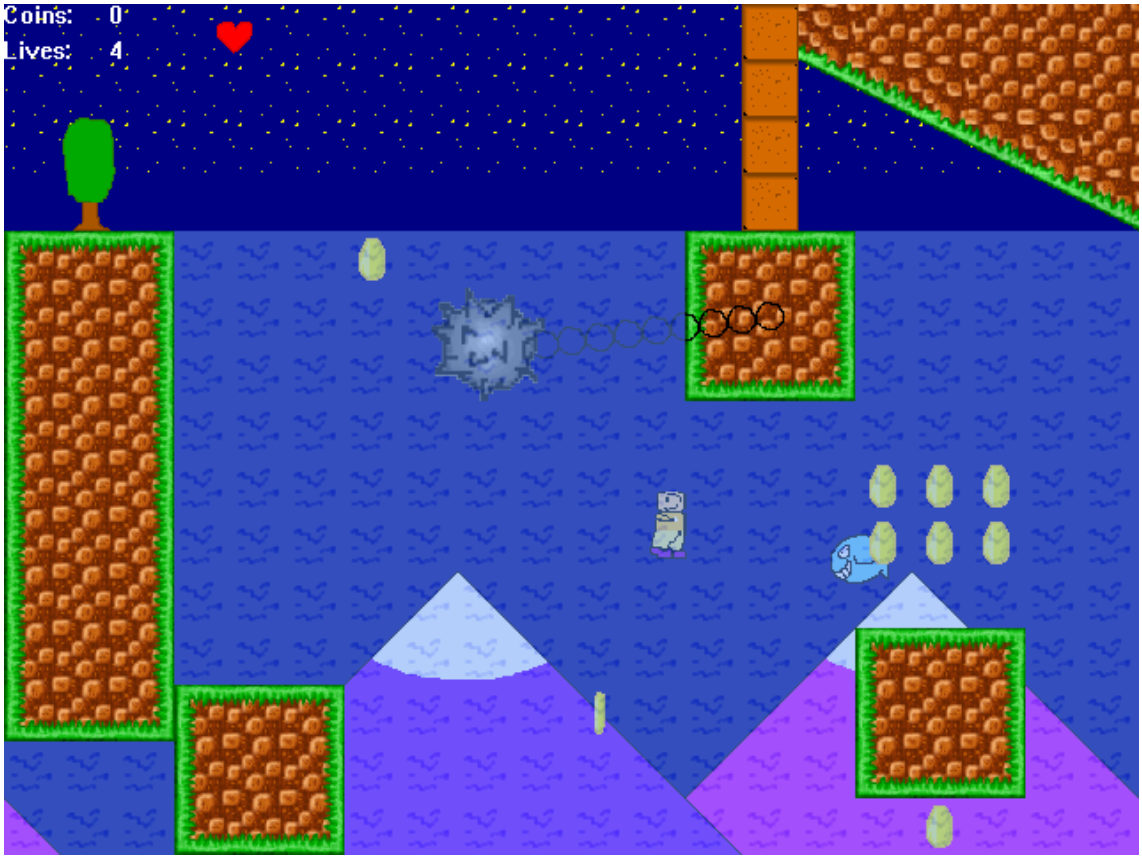
Ensimmäiset tasoloikkapelit sijoittuivat vain yhteen peliruutuun, mutta tekniikan kehittyessä ilmestyi pelejä, joissa maailma vieri eteenpäin (esim. Super Mario Bros.), ja lopulta kaikkiin ilmansuuntiin (esim. Super Mario World ja Sonic the Hedgehog).

Tasoloikkapeleissä on yleensä useita eri kenttiä (level), joiden vaikeus kasvaa pelin edetessä. Kentät täytyy läpäistä joko määrätyssä tai vapaasti valittavassa järjestyksessä. Aina kaikkia kenttiä ei tarvitse läpäistä päästäkseen pelin loppuun, ja jotkut kentät voivat olla niin sanottuja salaisia kenttiä, joihin pääsee vain löytämällä salaisen uloskäynnin.

Kun pelilaitteistolla pystyttiin piirtämään kolmiulotteista polygonipohjaista grafiikkaa reaaliajassa, myös tasoloikkapeleissä alettiin käyttää kolmiulotteista grafiikkaa ja peliympäristöjä. Äärimmäisessä tapauksessa pelaaja pystyi liikkumaan vapaasti mihin tahansa suuntaan maailmassa (esim. Super Mario 64), vaikkakin monissa peleissä liikkuvuus oli rajatumpaa. Myöhemmin kolmiulotteista grafiikkaa on käytetty myös pelimekaniikaltaan (pelimoottorin sisäiseen toimintaan kantaa otta-



Kuva 2.1: Vierivän tasoloikan perikuva, Super Mario Bros. Kuvassa Super Marioksi sienen syötyään kasvanut Mario on juuri hypännyt vihollisen päälle litistäen sen.



Kuva 2.2: The Adventures of Nasse. Kuvassa Nasse uimassa yrittäen samalla väistellä ketjun päässä pyörivää piikkipalloa sekä pelaajasta kiinnostunutta piraijaa, kauempana olevissa erillisissä taustakerroksissa vuoria ja tähtitaivas.

matta) täysin kaksiulotteisissa peleissä.

2.3 The Adventures of Nasse

Työn teknisenä kontribuutiona on toteutettu tasoloikkapeli The Adventures of Nasse, johon tästä eteenpäin viitataan nimellä Nasse. Pelin tärkein esikuva on japanilaisen peliyhtiö Nintendon omalle SNES-pelikonsolilleen kehittämä ja julkaisema Super Mario World. Ylsi Nasse sitten esikuvansa tasolle tai ei, on peleillä paljon yhteistä. Kummankaan taustatarina ei ole kummoinen, Nassessa se on lähes olematon. Molemmissa peleissä maailma koostuu maailmankartalle sijoitetuista kentistä, joissa pelaaja yrittää päästä maaliin kuolematta vihollisia päihittäen. Kummassakin on salaisia kenttiä, joihin pääsee salaisten uloskäyntien kautta, myös tallennus on peleille yhteinen ominaisuus. Lisäksi maaston muodot ovat samankaltaiset. Nasse julkaistiin internetissä heinäkuussa 2010, ja sai myöhemmin samana vuonna kaksi päivitystä sekä jouludemon. Aika näyttää, mitä tulevaisuus tuo Nasselle tullessaan.

3. SUUNNITTELUNÄKÖKOHTIA

Ohjelmointikielen olennaisen toiminnallisuuden ja jonkin grafiikkakirjaston peruskäytön opittuaan on mahdollista pienellä työllä saada aikaiseksi yksinkertaisia demoja ja jopa pelejä. Kuitenkin vähänkin monipuolisempi tasoloikkapeli, joka sisältää useita eri tyyppisiä kenttiä, eri tavoin käyttäytyviä peliobjekteja sekä paljon tietosisältöä, on perusohjelmointitaidon lisäksi ohjelmiston suunnittelusta ja usein myös tietokoneen laitteistotason toiminnasta ymmärrystä vaativa projekti. Valitettavasti useissa tapauksissa aloitteleva 'ohjelmoija' haluaa saada nopeasti aikaiseksi jotain ja alkaa välittömästi kirjoittaa ohjelmakoodia.

Tasoloikkapeliprojekti, jonka maksimissaan kuudes iteraatio laskutavasta riippuen Nasse on, hylättiin ja aloitettiin useita kertoja kokonaan alusta muun muassa puutteellisen suunnittelun vuoksi. Ensimmäisen yrityksen ja julkaistun ohjelman välillä kuluneiden vuosien laskemiseen on käytettävä kahden käden sormia. Ensimmäiseksi iteraatioksi voidaan haluttaessa laskea kenttäeditorin heikohko yritys (loppukesä 2001), johon ei liittynyt varsinaista pelitoiminnallisuutta ja joka hylättiin hyvin nopeasti. Kyseessä oli DirectX:ää ja grafiikan piirtämiseksi siihen kuuluvaa DirectDraw:ta käyttävä Windows-sovellus. Seuraavassa DOS-käyttöjärjestelmälle toteutetussa iteraatiossa (2002-2003) piirto-, ajastin- ja input-rutiinit olivat itse assemblykoodilla toteutettuja. Se sisälsi kehnon editorin lisäksi kehnon pelitoiminnallisuuden vierivällä maastolla. Sitä seurasi (kesä 2003) DirectX-versio Windowsille, joka kuitenkin oli lähestulkoon toiminnallisuudeltaan sama kuin edellinen iteraatio. Piirtorutiinit oli toteutettu siinäkin assemblyllä, DirectDraw:ta käytettäen vain näyttöpuskuriin käsiksi pääsemiseksi. Seuraava iteraatio (syksy 2003 - kesä 2005) muistuttikin paljon enemmän oikeaa peliä. Se sisälsi hierarkkisia tietorakenteita, vihollisia sekä kaltevia pintoja. DirectDraw:ta käytettiin tällä kertaa laitteistokiihdytetyn grafiikan piirtämiseen. Törmäystarkistus aiheutti kuitenkin erityisen paljon hankaluuksia kyseisessä iteraatiossa, sillä sen toimintaa ei ollut määritelty tarpeeksi tarkasti. Tietorakenteet olivat liian puutteellisia siihen, että uusia vihollistyyppejä oltaisiin voitu lisätä vaivattomasti. Seuraavassa iteraatiossa (kesä 2007 - alkuvuosi 2008) kirjastona käytettiin ensimmäistä kertaa SDL:ää. Törmäystarkistus yritettiin toteuttaa paremmin, tällä kertaa matemaattisemmalla analyysilla, mikä ei kuitenkaan toiminut täydellisesti liukulukulaskennan epätarkkuudesta johtuen. Myöhemmin törmäystarkistus vaihdettiin toimimaan yksinkertaisemmin ja varmemmin. Myös kokonaislu-

vut otettiin varmuuden vuoksi käyttöön. Peliobjekteja ei pystytty muokkaamaan tarpeeksi joustavasti, mikä johti uusien tietorakenteiden suunnitteluun ja uuden iteraation (2008-2010) aloittamiseen. Tällä kertaa peli oli paremmin suunniteltu, sillä oli jopa alustava aikataulu ja välitavotteita. Vuoden 2008 lopulla valmistui pelin demoversio ja se sai myös nimensä. Tämän jälkeen pelattavuutta parannettiin, uusia kenttiä rakennettiin, pikkuvikoja korjattiin sekä ääniefektit ja musiikki toteutettiin. Pelin ensimmäinen verio julkaistiin internetissä heinäkuussa 2010.

On lukija sitten minkä tahansa tasoinen ohjelmoija, on tämän hyvä kiinnittää huomiota pelin suunnitteluun korkealla tasolla ennen ohjelmakoodin suurempaa kirjoittamista. Yleisellä tasolla jo valmiiksi pätevä ohjelmoija, joka ohjelmointikielen ja hyvien käytäntöjen osaamisen lisäksi omaa hyvän ongelmanratkaisu- ja päättelykyvyn sekä on nopea oppimaan, voi toteuttaa Nassea monimutkaisuudeltaan vastaavan pelin kuukausien aikana resurssien salliessa, kunhan toteutus on suunniteltu hyvin etukäteen. Hieman osaamattomampi ohjelmoijakin voi toteuttaa pelin arviolta vuodessa huolellisesti suunnittelemalla.

3.1 Tasoloikkapelin maailma

Tasoloikkapeleissä useimmiten kerrallaan läpäistävä ja mahdolliselta maailmankartalta pelattavaksi valittava yksikkö on kenttä. Kentässä voi olla useita tästä lähtien sektoriksi kutsuttavia toisistaan täysin erillisiä alueita joita pelilogiikka käsittelee ja joista näytölle piirretään yksi kerrallaan. Sektoreiden perimmäisenä tarkoituksena on erottaa käytettäviltä grafiikoilta, musiikeilta tai pelitilanteilta poikkeavat alueet. Jokaisella sektorilla on oma koordinaatistonsa, ja pelaaja voi siirtyä yhdestä toiseen sen koordinaatistossa määritettyyn sijaintiin esimerkiksi oven tai vaikkapa putken välityksellä. Sektorin sijoittumista kentällä ei ole määritelty, joten useampien sektorien esittäminen kerrallaan ja mahdollinen siirtyminen sektorista toiseen niiden välisen rajan ylittämällä ei ole mahdollista, ja se myös toimisi niiden tarkoituksen vastaisesti.

Sektori koostuu maastosta, jossa liikutaan, sekä peliobjekteista jotka edustavat esimerkiksi itse pelaajaa ja vihollisia. Maasto on toteutettu usein käyttäen muistissa sijaitsevaa taulukkoa, tiilikarttaa (tilemap), joka ilmaisee millaisista suorakulmion muotoisista ennalta määritellyistä palasista maasto koostuu. Tekniikka on laskennan kannalta erittäin tehokas, sillä halutussa sijainnissa oleva palanen voidaan lukea vakioajassa olettaen muistiviittauksen olevan vakioaikainen operaatio. Myös muistinkulutuksen kannalta tekniikka on suhteellisen tehokas, verrattuna esimerkiksi pikselikarttaan (pixelmap), jossa maasto ilmaistaan pikselitasolla, ja jota käyttämällä ei rajallisen muistin vuoksi voi toteuttaa kovin laajoja maailmoja. Pikselikartta soveltuu paremmin peleihin, joissa tarvitaan pikselitasolla muokattavaa maastoa, kuten esimerkiksi luolalentelyt ja ongelmanratkaisupeli Lemmings, eikä kirjoittaja osaa ni-

Kuva 3.1: Pikselikarttaa käyttävä peli Lemmings. Kuvassa pelaaja on kaivanut pikselitasolla muokattavaan maastoon kuopan josta sopulit kulkiessaan tippuvat alas ja pääsevät maaliin.

metä yhtäkään vierivää pikselikarttaa käyttävää tasoloikkapeliä. Maaston voi kuvata myös matemaattisilla muodoilla kuten monikulmioilla tai käyrillä, jolloin muistinkulutus luultavasti vähentyisi ja muodot pystyttäisiin ilmaisemaan tarkemmin. Tämä on kuitenkin huomattavasti vaikeampi toteuttaa ja vaatii hyvin suunniteltuja tietorakenteita toimiakseen tehokkaasti.

Tiilikartassa käytettävät palaset ovat määritelty tiilijoukossa (tileset), joka valitaan sektorikohtaisesti. Tiilijoukko on valikoima sektorin maaston rakentamiseen käytettäviä palasia. Tällä tavoin minimoidaan tiilikartan muistinkulutus ja jaetaan erilaiset palaset kätevästi ryhmiin. Mikäli yhtä tiilikartan palasta edustaa n-bittinen luku, voidaan samassa tiilikartassa käyttää kerrallaan 2 potenssiin n erilaista palasta. Esimerkiksi tavun, eli 8-bitin kokoisilla luvuilla voidaan esittää 2 potenssiin 8, eli 256 erilaista palasta yhdessä tiilikartassa, joka on usein riittävä määrä. Tiilikarttojen käyttö tuo joustavuutta, mutta lisää kuitenkin yhden epäsuoruuden tason ja vaatii yhden ylimääräisen muistiviittauksen jokaista palasta käsiteltäessä, mikä ei kuitenkaan vaikuta suoritustehoon merkittävästi.

3.2 Tieto-ohjattu ohjelmointi

"Tieto-ohjatun ohjelmoinnin idea on suunnitella ja toteuttaa ohjelmaan tietorakenteita, joiden avulla tietoa käsittelevien käskyjen määrä saadaan pienemmäksi. - Korvataan käskyjä tiedolla." [8]

Useimmissa videopeleissä tärkeänä osana on 'sisältö', jolla tässä tarkoitetaan kaikkea muuta pelin dataa kuin suoritettavia käskyjä. Näin ollen sisältö voi sijaita itse ohjelmassa data-alueella tai erillisissä tiedostoissa. Laajempaa videopeliä toteuttaessa kannattaa sisältö liittää ohjelmakoodiin mahdollisimman vähäisessä määrin. Sisällön merkityksen lisääntyessä tulee hankalaksi ilmaista se ohjelmakoodin avulla ja mikäli sisältöä on toteuttamassa myös ei-ohjelmoija, ei haluta että tämä joutuu joko opettelemaan ohjelmoimaan tai tarvitsee aina ohjelmoijan apua toteuttaessaan jotain uutta. Esimerkiksi erilaisten pelihahmojen ominaisuudet on hyvä ilmaista datatiedostossa eikä ohjelmakoodissa.

Myös samojen vakioarvojen toistuvaa eksplisiittistä kirjoittamista koodiin tulee välttää. Mikäli arvoa halutaan muuttaa, tulee tällöin jokainen vakion arvon ilmaiseva kohta ohjelmakoodissa korvata, mikä on työlästä ja virhealtista. Parempi ratkaisu on sijoittaa vakiot muuttujiin, joihin vakiot sijoitetaan kerran eksplisiittisesti ohjelmakoodissa tai vaihtoehtoisesti ladataan datatiedostosta.

3.3 Automaatio ja työkalut

Ohjelmistokehitystyökalu on sovellus, jota ohjelmistokehittäjät käyttävät ohjelmien kehittämiseen, virheiden jäljittämiseen ohjelmissa, ohjelmien ylläpitoon tai muuten ohjelmien tukemiseen [9]. Tasoloikkapelissä työkaluja käytetään itse ohjelmoimisen lisäksi esimerkiksi grafiikan piirtämiseen ja kenttien luomiseen.

Vaikka automaatio usein lopulta vähentää työmäärää, tarvitsee sen toteuttamisen eteen ensin tehdä työtä. Videopeliä toteuttaessa tarvitsee usein itse pelin lisäksi ohjelmoida sisällön tuottamiseen liittyviä työkaluja. Usein herääkin kysymys, milloin tulee luoda uusi työkalu, voiko käyttää olemassaolevaa ohjelmaa vai onko erillinen työkalu edes välttämätön. Ongelmaa voi lähestyä punnitsemalla työkalun tuomaa etua ja sen toteuttamiseksi tehtävää työtä sekä jo tarjolla olevan työkalun soveltuvuutta projektiin. Esimerkiksi kenttäeditori on laajoja ympäristöjä sisältävässä videopelissä ehdoton työkalu, eikä välttämättä tarjolla ole sopivaa editoria, jolloin se toteutetaan itse. Soveltuvan piirto-ohjelman toteuttaminen taas voi tuntua turhalta, jos osataan käyttää jo olemassaolevaa sovellusta. Joidenkin datatiedostojen luomiseen tavallinen tekstieditori on jo riittävä työkalu.

On otettava huomioon myös, ovatko pelin työkalut tarkoitettu kehittäjien lisäksi myös pelaajien käytettäväksi. Joissain peleissä halutaan että myös pelaajat voivat tehdä esimerkiksi omia kenttiä, jolloin myös kenttäeditori on toimitettava pelin mu-

kana. Jotkut pelit menevät pidemmälle mahdollistaen pelilogiikan ja täten itse pelityypin muuttamisen. Tällöin on useimmiten käytössä niinsanottu skriptikieli, jonka käyttämiseksi tulee pelin sisältää tulkki, joka lukee skriptikielistä koodia ajon aikana ja yhdistää sen kovakoodattuun logiikkaan.

3.4 Olio-ohjelmointi

Oliosuunnittelussa kapseloidun tiedon ja sitä käsittelevien funktioiden joukkoa kutsutaan olioksi. Yksinkertaisimmillaan olio-ohjelmointi on oliosuunnittelun olioiden toteuttamista oliokielellä, kuten C++:lla [10].

Koska C++:n on peliohjelmointikäytössä suosittu kieli, on luonnollista että myös olio-ohjelmointia hyödynnetään peleissä. Monia pelin osia voi hyvin ajatella olioina, joten tietorakenne ja metodit yhdistyvät luokaksi luonnollisesti. Periytyminen on myös hyödyllinen ominaisuus joidenkin komponenttien selvästi laajentaessa jonkin jo olemassaolevan komponentin perustoiminnallisuutta. Jokaista tietotyyppiä varten ei kuitenkaan tarvitse toteuttaa luokkaa vaan joskus perinteinen struct riittää.

Erityisesti pelihahmojen voi ajatella olevan luokkaa edustavia olioita, edustaahan pelihahmoa näytöllä erityinen animaatio ja niillä on tietty toiminnallisuus. Koska kaikki pelihahmot kuuluvat yläkäsitteeseen peliobjekti, edustaen kuitenkin omaa alatyyppiään, eräs luonnollinen ratkaisu on käyttää periytymistä määrittelemällä kantaluokka 'Peliobjekti' ja tämän eri objektityyppejä edustavat aliluokat. Kuitenkin objektityypin sitominen ohjelmointikielen tarjoamaan aliluokkaan tekee ohjelmasta vähemmän tieto-ohjatun. Kaikkien objektityyppien toiminnallisuus täytyy tällöin kirjoittaa pääohjelmaan, eikä toiminnallisuutta voi muokata ilman uudelleen kääntämistä.

Toteutusta voi parantaa erottamalla objektityypin ominaisuuksia kuvaava tieto luokan toteutuksesta. Mikäli halutaan tiedustella objektin jotain tyyppikohtaista ominaisuutta, on parempi kirjoittaa vastaava get-metodi pelkästään kantaluokkaan ja funktion sisällä hakea tieto tyyppikohtaiset ominaisuudet sisältävästä tietorakenteesta aliluokassa toteutettavan virtuaalifunktion sijaan. Näin voidaan pitää objektityypin ominaisuudet erillään ohjelmakoodista, kuitenkin toteuttaen sen käyttäytymisen aliluokan metodeilla.

Pidemmälle tieto-ohjattu ratkaisu on jättää ohjelmointikielen tarjoama periytyminen käyttämättä ja ilmaista myös objektityypin toiminnallisuus datan avulla, tyyppikohtaisia metodeja kirjoittamatta. Data voi olla toiminnallisuuden yksityiskohtaisesti määrittelemään kykenevää skriptikieltä, jonka pelin sisällä tulkitsemiseen tarvitaan aloittelijalle vaikeasti kirjoitettava ohjelmakoodi, tai yksinkertaisemmin luettavaa tietoa, joka voi esimerkiksi ilmaista mitä varsinaisessa ohjelmakoodissa toteutettuja liitettäviä funktioita objektityyppi käyttää tiettyihin toimintoihin. Jälkimmäinen tapa vaatii ohjelmakoodissa runkokoodin objektien toiminnallisuudelle

sekä olennaisesti liitettävien funktioiden toteuttamisen, objektien toiminnallisuuden rajoittuessa toteutettuihin funktiohin. Kuitenkin myös skriptikieltä käyttäessä kannattaa ja todennäköisesti myös tarvitsee kutsua peliohjelmaan kovakoodattuja apufunktioita. Ensinnäkin skriptikielen tulkkaaminen vie enemmän laskentatehoa kuin konekoodin suorittaminen, jolloin aikakriittiset toiminnot on parempi suorittaa kovakoodilla. Toiseksi monimutkaiset rutiinit vaativat parempaa ohjelmointitaitoa, jota välttämättä objektityypin toiminnallisuutta skriptaavalla henkilöllä ei ole. Viimeiseksi, ohjelma voi olla suunniteltu siten ettei pelihahmon skriptistä käsin pääse suoraan käsittelemään pelin muita tietorakenteita.

3.5 Alusta

Alustalla tarkoitetaan laitteistoarkkitehtuuria ja ohjelmistokehystä, jolla ohjelmaa ajetaan. Yleensä tähän kuuluvat myös käyttöjärjestelmä ja ajonaikaiset kirjastot [11].

Peliä suunnitellessa tulee ottaa huomioon millaiselle alustalle/alustoille se toteutetaan. Koska videopelit ovat interaktiivisia ja nykyään useimmiten reaaliaikaisia, ne ovat tiukasti kytköksissä laitteistoon. Täten laitteistoa ei voi hyvän pelikokemuksen luomisessa ajatella abstraktina osana, jolla ei ole niin väliä. Useat pelihistorian merkkipelit on suunniteltu nimenomaan kohdelaitteistoa varten, ja mahdolliset käännökset muille alustoille ovat usein olleet heikohkoja.

Peli on suunniteltava siten, että grafiikka ja äänimaailma vastaavat suunnittelijan näkemystä kohdealustalla, mutta reaaliaikaisissa videopeleissä ehkäpä kuitenkin tärkein ja valitettavan usein laiminlyöty laitteiston osa on ohjausrajapinta. Pelin tulee olla hyvin ohjattavissa kohdealustan ohjaimilla. Esimerkiksi tasoloikkapeleissä vaaditaan usein ohjainlaite johon mahtuvat molemman käden tarvittavat sormet. Usein toinen käsi painaa juoksu- ja hyppynappeja, kun taas toinen ohjaa pelihahmon suuntaa. Konsolipeleissä hahmoa ohjataan gamepadin ristiohjaimella peukaloa käyttäen ja juoksu- sekä hyppynappeja painetaan toisen käden peukalolla. Käytännön syistä yleiskäyttöisillä tietokoneilla paras ohjainlaite myös tasoloikkapeleille on näppäimistö; oikeaoppinen ristiohjain on ristin muotoinen, jolloin sitä ei paineta väli-ilmansuuntiin vahingossa, mutta useimmissa PC:lle valmistetuissa gamepadeissa ristiohjain on väärän muotoinen, jolloin pelihahmo liikkuu väli-ilmansuuntiin ja ohjain on käyttökelvoton. Tietokoneen näppäimistöä käytettäessä on otettava huomioon, että näppäimistö pystyy lähettämään vain rajatun määrän merkkejä kerralla, jolloin käyttäjän painaessa tiettyjä näppäinyhdistelmiä, ei kaikkia merkkejä lähetetä. Perinteisesti ohjattavaa tasoloikkapeliä ei voida toteuttaa matkapuhelimille, sillä niissä on pelien ohjaamista varten oikeastaan vain numeropainikkeet, joita yleensä painetaan molempien käsien peukaloilla, jolloin painikkeet eivät riitä sekä hahmon suunnan valitsemiseen että juoksu -ja hyppynappien painamiseen. Myöskään painik-

Kuva 3.2: Super Nintendon ohjain, hyvän gamepadin malliesimerkki

keiden tuntuma ei ole kovinkaan soveltuva pelaamiseen. Puutteellisen ohjainlaitteen tapauksessa tulee improvisoida ja rajoituksia hyödyksi käyttäen suunnitella peli eri tavalla ohjattavaksi. Sama pätee kosketusnäytöllisiin matkapuhelimiin.

Konsolipeliä kehitettäessä tilanne helpottuu siinä mielessä, että kaikilla käyttäjillä on samanlainen laitteisto, kun taas PC-alustalla tulee ottaa tietyssä määrin huomioon erilaiset kokoonpanot ja asettaa laitteistolle minimivaatimukset. Ennen pelin toteuttamista tulee selvittää, pystyykö käytettävällä alustalla toteuttamaan halutut toiminnot laitteiston avulla, ohjelmallisesti vai ei ollenkaan.

3.6 Pelisilmukka

Suoritettava peliohjelma voidaan useimmiten jakaa alustustoimenpiteisiin, pelisilmukkaan ja pelin sulkemiseen. Pelisilmukka on nimensä mukaisesti silmukkarakenne ohjelmakoodissa, jossa suoritetaan jatkuvasti määrätyssä järjestyksessä pelin toimintaan liittyviä toimenpiteitä. Silmukasta poistutaan vasta ohjelman lopetusehdon täyttyessä. Pelisilmukka sisältää pelin korkean tason toiminnallisuuden, joka voidaan erottaa pelaajalta saadun syötteen lukemiseen ja käsittelyyn, pelilogiikan suorittamiseen, sekä esim. visuaalisen vasteen tuottamiseksi pelaajalle.

3.7 Koodin tasojen erotus

Korkeamman ja alemman tason koodi voidaan erottaa toisistaan eri vaiheissa. Alustariippuvainen koodi voidaan sijoittaa niin sanotun kääreen sisään. Se on välikerros

joka toimii alustan rajapintana pelille tarjoamalla käyttöön tietorakenteita ja funktioita. Etuna välikerroksen käytössä on pelikoodin yksinkertaistuminen, alustakohtaisen koodin muokkaamisen sekä pelin muille alustoille porttaamisen helpottuminen.

Pelin säännöt voidaan erottaa pelityypille ominaisesta toiminnallisuudesta, jota voi ajatella niin sanottuna pelimoottorina. Moottoriin voi ajatella kuuluvan esim. fysiikan mallinnuksen, törmäystarkistuksen ja grafiikan piirtämisen. Fysiikan mallinnus ja grafiikka voivat olla myös toteutettu omana moottorinaan, joka voi olla joko peliä varten erityisesti kehitetty tai valmiina saatu osa.

3.8 Suunnittelumallit

Ohjelmistotuotannossa käytetään usein tunnettuja suunnittelumalleja joko tietoisesti tai tietämättä. Suunnittelumallit perustuvat samantapaisten ratkaisujen toistumiseen luokkien rakenteessa ja niiden yhteistyön suunnittelussa [10]. Myös tasoloikkapeleissä käytetään erilaisia suunnittelumalleja.

Tilakone. Tilakone on järjestelmä, jossa on joukko erillisiä tiloja ja joka toimii siirtymällä näiden tilojen välillä [10]. Useita ohjelman osia voi ajatella tilakoneina, jotka toimivat sen hetkisen tilan mukaisesti ja vaihtavat tilaansa siirtymisehtojen täyttyessä. Tilakonetta voidaan käyttää esimerkiksi kuvaamaan pelin tilaa ja toiminnallisuutta tai vaikkapa vihollisen tekoälyä. Korkean tason ohjelmointikielissä tilakoneen toteuttamista tukevat valintarakenteet, esimerkiksi C++:ssa erityisesti switch-case-monivalintarakenne. Ohjelmassa haaraudutaan tilaa vastaavaan lohkoon, jossa suoritetaan siihen liittyvä toiminnallisuus.

Sovitin. Kääre- tai sovitinsuunnittelumallissa yhdenlaisesta rajapinnasta luokkaan tehdään kyseisen luokan kanssa yhteensopiva [12], jolloin asiakas voi käyttää kohdeluokkaa tietämättä sen rajapintaa. Mallia voi käyttää esimerkiksi sovelluskohtaisen ohjelmakoodin erottamiseen alustakohtaisesta koodista, kuten käyttöjärjestelmäpalveluiden kutsumisesta sekä ulkoisten liitäntöjen käsittelystä, kuten grafiikasta, äänistä ja syötteen lukemisesta.

3.9 Datasisällön jakaminen

Vähänkään laajemmassa pelissä on useimmiten dataa, joka ei kuulu varsinaisesti millekään komponentille, mutta jota useat komponentit kuitenkin tarvitsevat pelin ajamiseen. On siis tehtävä päätös, kuinka tällainen data jaetaan moduulien välillä.

Yksi ratkaisu on tehdä tietosisällöstä globaalia, jolloin jokainen moduuli pystyy sekä lukemaan että muokkaamaan sitä. Ratkaisu on suorituskyvyn sekä toteutuksen yksinkertaisuuden vuoksi tehokas, mutta valta tuo vastuuta; esimerkiksi TTY:n perusohjelmointikursseilla opetetaan pois globaalien muuttujien käytöstä, ja ne ovat

jopa harjoitustöissä kiellettyjä [13]. Globaaleja tietorakenteita käyttäessä ongelmana ovat niin sanotut mahdolliset sivuvaikutukset funktioita suoritettaessa. Mikäli funktio muokkaa globaalia dataa, muuttaa sen suorittaminen ohjelman kokonaistilaa, minkä jälkeen globaalista datasta riippuvaiset funtiot käyttäytyvät eri tavalla. Sivuvaikutusten myötä funktioiden toiminta ei riipu vain niille annettaville parametreista, jolloin niiden testaamisesta ja täten myös virheiden jäljittämisestä tulee huomattavasti hankalampaa. Globaalia dataa on syytä käyttää säästäen ja tehdä itselleen selväksi mitkä funktiot ovat siitä riippuvaisia ja mitkä muokkaavat sitä. Useamman henkilön projektissa globaalii data ja niitä käyttävät funktiot tulisi määritellä tarkasti dokumentaatiossa ongelmien välttämiseksi.

Toinen ratkaisu on välittää jaettava tietosisältö parametreina kutsuttavalle funktiolle. Näin päästään globaalin datan aiheuttamista sivuvaikutuksista, mutta toisaalta funktiokutsuista tulee raskaampia sekä suoritinsykleissä laskettuna parametrien pinoon työntämisen vuoksi että ohjelmakoodin luettavuuden kannalta. Mikäli ohjelman rakenteen syvimmällä oleva funktio tarvitsee korkeimman tason moduulissa olevaa ei-globaalia dataa, täytyy data kuljettaa parametreina jokaisen tason läpi. Aina kun huomataan funktion tarvitsevan ylimääräistä dataa, joudutaan matkalla olevien funktioiden parametrilistaa muuttamaan, jolloin ohjelman ylläpidettävyydestä tulee hankalaa. Mikäli luottamus omaan huolellisuuteen on vahva, monitasoisten sovellusten tapauksessa on suositeltavampaa käyttää ensimmäistä ratkaisua tämän sijaan.

Kehittynein ratkaisu on kuitenkin niin sanottu resurssien rekisteröiminen. Tällöin moduulin tietorakenteeseen sisältyy osoitin haluttuun resurssiin, joka on ilmeisesti asetettava ennen tämän käyttämistä. Osoittimen arvo saadaan rekisteröimällä eli pyytämällä sitä resurssin omistavalta moduulilta tai mahdollisesti toisinpäin, jolloin resurssin omistaja kutsuu sitä käyttävän moduulin arvon asettavaa funktiota. Rekisteröintiratkaisussa on pidettävä huolta, että data pysyy ajan tasalla. Hyvin suunnittelemalla tämä voi toimia itse asiassa automaattisesti, mutta mikäli rekisteröinnissä siirretäänkin kopio resurssista eikä sen osoitinta siihen, tulee kopio päivittää aina resurssin muuttuessa. Ongelma on ehkä hieman yllättäen läsnä joskus myös osoittimia käytettäessä; mikäli resurssia säilytetään myös sen omistavassa moduulissa osoittimen päässä, tapauksessa jossa omistaja jostain syystä asettaakin osoittimen osoittamaan uuteen resurssiin, täytyy resurssia käyttävän moduulin pyytää resurssin osoitin uudelleen. Yleisessä tapauksessa, mikäli on mahdollista että omistava moduuli muuttaa antamansa resurssin, oli se sitten kopio itse datasta tai siihen osoittava osoitin, arvoa, tulee resurssi pyytää uudelleen. Ongelma voidaan kuitenkin kiertää esimerkiksi omistajan puolella osoittimen päässä olevan resurssin tapauksessa antamalla resurssin pyytäjälle resurssin osoittimen osoitin. Yleisessä tapauksessa annetaan resurssin pyytäjälle osoitin siihen tietoalkioon, jota omistaja voi muut-

taa. Tämän ratkaisun käyttäminen oikealla tavalla vaatii epäsuorien viittausten, eli osoittimien käytön perinpohjaista ymmärrystä, joka on usein hankala alue aloitteville ohjelmoijille. Toisaalta mikäli ohjelmoija ei ymmärrä tarpeeksi hyvin osoittimien toimintaa, ei tämän voi olettaa toteuttavan kovin monimutkaisia tehokkaita ja luotettavasti toimivia sovelluksia, ja täten myöskään pelejä ilman valmiita tietorakenteita tarjoavia kirjastoja, kuten esimerkiksi C++:n STL [14].

4. KENTTÄEDITORI

Kenttäeditori on työkalu, jota käytetään pelin kenttien rakentamiseen ja on yksi tärkeimmistä pelin kehittämisessä käytettävistä työkaluista. Tarjolla on valmiita tiilikarttaeditoreita, mutta nämä eivät välttämättä sovi tarkoitukseen, kenttien usein koostuessa muistakin osista kuin tiilikartasta. Työn tarkoituksena on käsitellä pelin toteutusta kokonaisvaltaisesti, ja koska editori on merkittävä työkalu, käsitellään myös sen toteutusta.

4.1 Editorin vaatimukset

Tasoloikkapelissä kenttään kuuluu sekä maasto että peliobjekteja. Täten editorin pitäisi tarjota ainakin toiminnallisuus maaston muokkaamiseen sekä objektien lisäämiseen ja poistamiseen. Lisäksi kentällä voi olla ominaisuuksia, joita tulee voida säätää editorissa. Myös peliobjekteilla voi erikseen säädettäviä ominaisuuksia, jotka ovat objektikohtaisia. Olennaista on myös, että kenttiä pystyy lataamaan ja tallentamaan.

4.2 Toimintaperiaatteet

Editori voi olla osa peliohjelmaa tai erillinen sovellus. Jälkimmäsessä tapauksessa myös itse pelimoottori voi sisältyä editoriin, mikäli halutaan ajaa kenttiä testimielessä editorista käsin. Vaihtoehtoisesti käyttöjärjestelmäkutsulla voidaan editorista käynnistää peliohjelma editoitavan tason testausta varten. Editoimista varten tulee toteuttaa maaston esitystavasta riippuvia tukifunktioita. Tiilikartan tapauksessa näitä ovat esimerkiksi palasten lisääminen karttaan tai poistaminen siitä. Myös objektien lisäämistä ja muokkaamista varten tarvitaan toiminnallisuutta. Mikäli kenttiin kuuluu määritellyillä ehdoilla laukaistavia tapahtumia, tulee myös niiden asettamista varten toteuttaa toiminnallisuus. Tapahtumat voivat olla skriptattuja, ja myös kentässä olevien vihollisten tekoäly voidaan toteuttaa skriptikielellä, jolloin sitä voidaan haluta kirjoittaa editorissa.

4.3 Käyttöliittymä

Ei ole yhtä määrättyä tapaa, kuinka kenttäeditorin käyttöliittymän tulee toimia. Usein päädytään graafiseen ratkaisuun, jossa hyödynnetään sekä näppäimistöä että

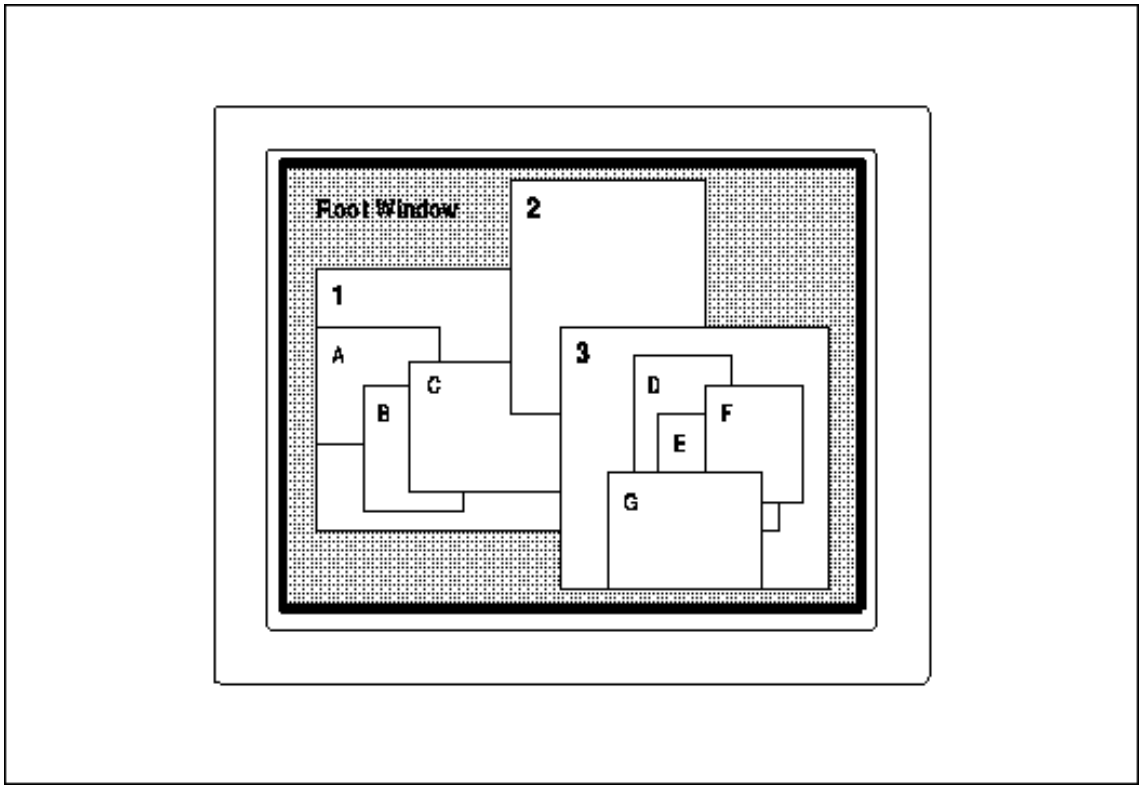
hiirtä. Usein editoriin tarvitaan jonkinlainen ikkunointijärjestelmä joka on toteutettavissa joko käyttöjärjestelmän omalla API:lla, kolmannen osapuolen tarjoamalla ratkaisulla tai täysin itse ohjelmoimalla. Tosin editorin käyttöliittymä on mahdollista toteuttaa ilman kunnollista ikkunointijärjestelmää, karumpia menetelmiä käyttäen. Kuitenkin, mitä enemmän ja monimutkaisemmin dataa editorissa muokataan, sitä tärkeämpi hyvä ikkunointijärjestelmä on.

4.4 Ikkunointi

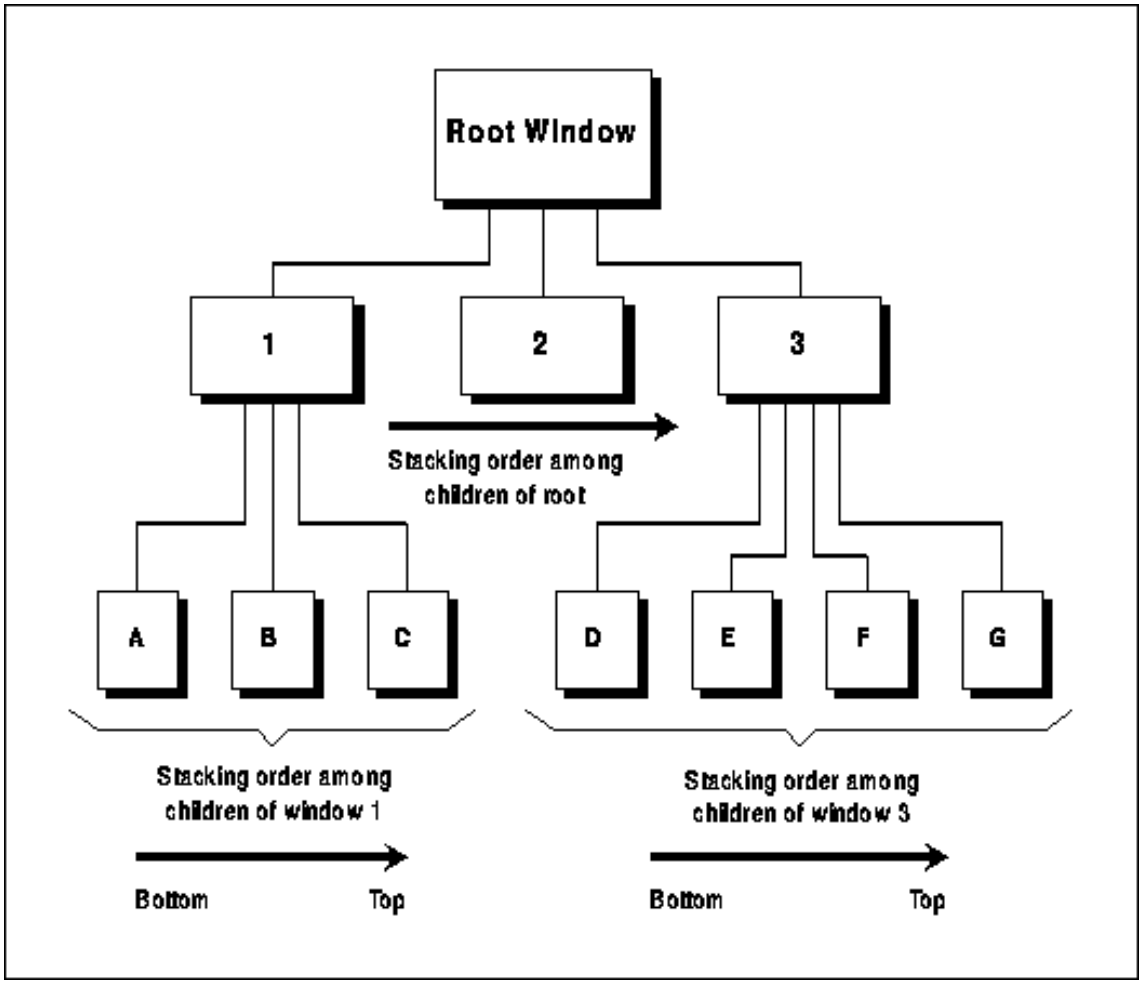
Ikkunajärjestelmän voi ajatella olevan hierarkkinen kokonaisuus, jossa ylemmän tason ikkunat omistavat alemman tason ikkunat. Jokaista järjestelmän näytöllä näkyvää objektia voidaan ajatella ikkunana. Siis esimerkiksi erilaiset painikkeet, tekstikentät ja valikot ovat kaikki ikkunoita, jotka ylemmän tason ikkuna sisältää. Voidaan käyttää olio-ohjelmointia siten, että kaikki erilaiset ikkunatyypit ovat yleisen Ikkuna-luokan aliluokkia, ja jokaisessa Ikkuna-oliossa on tietorakenne, jossa säilytetään ikkunan ali-ikkunoita.

Ikkunointijärjestelmän voi ajatella olevan ohjelman logiikasta irrallinen osa, ja se voi toimia esimerkiksi seuraavalla tavalla: kun pääohjelma haluaa avata näytölle ikkunan, tapahtuu ikkunaolion ja kaikkien sen ali-ikkunoiden luominen pääohjelmassa, jonka jälkeen luotu ikkuna välitetään ikkunointijärjestelmälle. Ikkunan luomisen yhteydessä tulisi määritellä ikkunan ali-ikkunoille pääohjelman puolella sijaitsevat callback-funktiot, joita kutsutaan esimerkiksi kun painiketta painetaan tai tekstikentän arvoa muutetaan. Tällä tavoin ikkunointijärjestelmän ja muun logiikan välillä on yhteys, jolloin ohjelma pystyy toimimaan oikealla tavalla käyttäjän selatessa ikkunavalikkoja ja muokatessa parametrien arvoja.

Jotta järjestelmä toimisi, kutsutaan pääohjelmasta ikkunajärjestelmän käsittelyfunktiota, joka piirtää ikkunat ja tutkii onko painikkeita painettu. Ylimmän tason funktiokutsussa käydään läpi ylimmän tason ikkunat ja kutsutaan niiden käsittelyfunktiota. Ikkunoiden käsittelyfunktiossa taas käydään läpi ikkunoiden ali-ikkunat ja kutsutaan ali-ikkunoille käsittelyfunktiota. Näin saadaan käsiteltyä kaikki ikkunat. Painikkeiden painamisen yhteydessä kutsutaan painikkeeseen sidottua funktiota, jonka suoritus johtaa esim. datan muokkaamiseen, uuden ikkunan luomiseen tai ikkunan sulkemiseen. Koko ohjelman käyttöliittymä on siis mahdollista toteuttaa kirjoittamalla hierarkkinen ikkunointijärjestelmä ja joukko funktioita, jotka vastaavat ohjelman erilaisten ikkunoiden luomisesta ja datan muokkaamisesta callback-funktioina toimien ja ohjelman päätoiminnallisuuden funktiota kutsumalla.



Kuva 4.1: Ikkunointijärjestelmän graafinen näkymä. Kuvassa ikkunoita avattu ikkunoiden päälle hierarkkisesti. Kuva lähteestä [15].



Kuva 4.2: Ikkunointijärjestelmän näkymä puurakenteena. Kuvassa näkyvät ikkunoiden omistussuhteet. Kuva lähteestä [15].

5. LOGIIKKA

Pelilogiikaksi voi laskea määritelmästä riippuen joko kaiken koodin joka käsittelee pelin dataa sekä pelitilanteen että käyttäjältä saadun syötteen perusteella tai koodia joka käsittelee tilannetta erillisen logiikkaan kuulumattoman moottorin/moottorien avulla. Tässä käsitellään vain varsinaisen pelitoiminnan logiikkaa, ei esimerkiksi maailmankarttanäkymän tai pelin päävalikon logiikkaa. Pelitoiminta on vastuussa periaatteessa kaikesta mitä tapahtuu, kun ollaan varsinaisessa pelitilassa. Tähän kuuluu esimerkiksi objektien hallinta, peliobjektien toiminnan kutsuminen, fysiikan ja törmäystarkistuksen toteuttavien funktioiden kutsuminen sekä kameran ohjaus.

5.1 Objektien käsittely

Peliobjektien käsittely on olennainen osa pelitoimintaa. Objektien tulee liikkua ja toimia halutulla tavalla pelimaailmassa, minkä vuoksi tulee toteuttaa logiikka näiden yleiselle hallinnalle sekä yksittäisten objektityyppien toiminnalle. Objektien yleisen hallinnan tulisi toteuttaa toiminnallisuus, joka koskee yleisesti kaikkia pelin objekteja. Peliobjektien sisäinen toiminta taas määrittelee kyseisen objektityypin toiminnan eri tilanteissa.

Kaikista objektien hallinnan toimenpiteistä, paitsi objektien toiminnallisuuden kutsumisesta, voidaan huolehtia joko pelin objektien hallinnasta vastaavassa osassa tai sijoittaa osaksi kunkin peliobjektin sisäistä toimintaa. Esimerkiksi vihollisten liikkumisehto voidaan määrittellä niiden omassa tekoälyssä, jolloin objektien hallinnan ei tarvitse välttämättä erikseen aktivoida tai deaktivoida niitä.

5.1.1 Hallinta korkeammalla tasolla

Korkeamman tason hallinnan tulisi käsitellä tietorakenteita, joissa kaikki pelin objektit sijaitsevat ja käsitellä objekteja samalla tavalla. Ensinnäkin, jotta objektit toteuttaisivat niille määriteltyä toiminnallisuutta, täytyy objektinhallinnan kutsua jokaiselle objektille tyyppikohtaisen logiikan toteuttavaa funktiota. Olio-ohjelmoinnin periytymisen käyttäminen auttaa tässä: jokaista peliobjektia voidaan ajatella kantaluokan jäsenenä ja kutsua vain logiikkafunktiota, jolloin suoritetaan oikean objektityypin logiikka.

Korkeamman tason hallintaan voidaan myös ajatella kuuluvan erilaisten objekteja käsittelevien funktioden kutsumista, vaikkei itse funktioiden objektien hallin-

taan ajatellakaan kuuluvan. Näihin funktiohin kuuluvat esimerkiksi peliobjekteja liikuttavat, niiden törmäyksiä maaston ja toistensa kanssa tarkastelevat sekä niiden animaatioita päivittävät funktiot.

Objekteja saatetaan haluta myös poistaa tietorakenteista tai siirtää tietorakenteiden välillä. Esimerkiksi kun peliobjekti kuolee, saatetaan se poistaa tietorakenteesta, jossa säilytetään eläviä objekteja. Peliobjektia saatetaan kuitenkin tästä huolimatta säilyttää rakenteessa, josta se otetaan uudelleen käyttöön objektin herätessä uudelleen henkiin pelaajan tullessa takaisin samaan sektoriin tai pelaajan kuoltua.

Tasoloikkapeleissä viholliset eivät yleensä lähde vaeltelemaan kentän käynnistyessä vaan alkavat liikkua vasta pelaajan ollessa niitä tarpeeksi lähellä. Jos objektit kulkevat liian kauas ruudusta, ne voivat kadota ja siirtyä myöhemmin takaisin alkusijaintiinsa. Objektien hallinnnan tulee siis pitää huolta objektien aktivoitumisesta. Joidenkin objektien taas saatetaan haluta suorittavan jonkinlaista logiikkaa aina, huolimatta niiden sijainnista pelaajaan nähden. Tällöin hallinnan tulee osata erotella erilaiset objektit.

Jokaiseen peliobjektiin pitää objektien hallinnan toiminnan vuoksi liittää tilamuuttujia esimerkiksi objektin elossa olemiseen ja aktiivisuuteen liittyen. Muuttujat voivat sijaita itse objektiolioissa tai hallinnan omissa tietorakenteissa. Muuttujien tilojen perusteella voidaan päättää, kutsutaanko objekteille esimerkiksi logiikan suorittavaa funktiota, tai jos objekti on kadonnut, tutkia tuleeko se siirtää takaisin alkuperäiselle paikalleen odottamaan aktivoitumista.

5.1.2 Peliobjektin sisäinen toiminta

Peliobjektin sisäiseen toimintaan kuuluvat tyyppikohtaisesti määritellyt asiat, kuten objektin tekoäly. Olio-ohjelmoinnin periytymistä käytettäessä jokaiselle eri objektityypille voidaan kirjoittaa logiikkafunktio, jossa määritellään objektin yleinen toiminta, jota kutsutaan kerran jokaisessa peliruudussa. Voidaan myös toteuttaa erillisiä funktioita, joita kutsutaan erilaisiin tapahtumiin reagoimiseksi. Eräs esimerkki tästä on funktio, jota kutsutaan ja joka määrittelee toiminnan kun objekti kohtaa kuilun reunan.

Objektityyppien toiminta voidaan myös toteuttaa skriptikieltä käyttäen. Tällöin sen sijaan, että joka objektityypille kirjoitettaisiin funktiot pelin ohjelmakoodiin, kirjoitetaankin erillisiin datatiedostoihin skriptit, jotka määrittävät objektien toiminnallisuudet. Skripteille annetaan pääsy objektien/pelin haluttuihin muuttujiin, siten että objektien toiminnallisuuden ohjelmointi skripteillä onnistuu. Skriptikielen käyttäminen vaatii koodin, joka osaa tulkita ja ajaa skriptikielisen tiedoston ja muokata itse pelin puolella olevia muuttujia.

5.1.3 Toiminnallisuuden jaottelu

Osa asiosta, joissa tietyn objektityypin tulee toimia tietyllä tavalla, voidaan toteuttaa joko objektityypin oman logiikan koodina tai yleisessä objektien hallinnassa. Esimerkiksi päätöksessä, mitä tehdä kuilun reunalla, on yksinkertaisesti ajateltuna kaksi vaihtoehtoa, kääntyä vai ei. Toiminnallisuus voidaan tällöin toteuttaa ainakin kolmella tavalla. Ensimmäisessä tavassa, mikäli kyseisen tyypin objektin tulee kääntyä kuilun reunalla, kirjoitetaan objektityypin logiikkaan eksplisiittinen koodi, jossa tarkkaillaan ollaanko tultu kuilun reunalle vai ei, ja vaihdetaan suuntaa mikäli ollaan reunalla. Toisessa tavassa ylemmän tason logiikka pitää huolen jokaisen objektin tarkkailusta ja kutsuu funktiota, joka ilmoittaa objektille reunalle tulosta. Objektityypin funktion toteutuksen perusteella tehdään haluttu toiminto, kuten kääntyminen tai jatketaan matkaa. Kolmannessa tavassa tutkitaan samalla tavalla objektin sijaintia, kuten edellisessä tavassa, mutta sen sijaan että kutsuttaisiin objektityypin funktiota, katsotaan tietorakenteesta, kääntyykö kyseisen tyypin objekti kuilun reunalla vai ei, ja mahdollisesti käännetään objektin suunta.

On syytä kiinnittää huomiota siihen, minkä tason koodin on lupa muokata mitäkin dataa. Esimerkiksi itse peliobjektit luultavasti sisältävät aina tiedon sijainnistaan, mutta niiden ei tulisi koskaan muokata sijaintimuuttujia omassa logiikassaan. Suunnittelijan on itse mietittävä, mitä arvoja objekti saa itse muuttaa. Objektilla voi olla esimerkiksi lupa muuttaa nopeuttansa, ja korkeamman tason funktiolla lupa päivittää sijainti nopeuden perusteella. Suora sijainnin muuttaminen saattaisi rikkoa pelitoiminnallisuuden. Saatetaan myös päätyä ratkaisuun, jossa peliobjekti ei saa suoraan muuttaa edes nopeuttansa vaan pelkästään ilmaista pyrkivänsä johonkin suuntaan, jonka jälkeen fysiikkakoodi laskee objektille nopeuden.

Objektien toiminnan toteuttamiseksi tulee siis tasapainotella sen suhteen, mikä toiminnallisuus toteutetaan enemmän automatisoidusti ylemmän tason hallinnassa ja mikä eksplisiittisellä koodilla objektityypin omassa toteutuksessa. Nyrkkisääntönä voisi pitää, että usein toistuvat kyllä/ei-valinnat kannattaa hoitaa automaattisemmin ja erikoisempi toiminta mahdollisesti ohjelmoimalla se eksplisiittisesti objektityyppiin.

5.2 Kamera

Kameran ohjaus vaikuttaa hyvin paljon pelin pelattavuuteen. Joissain peleissä on tapana yksinkertaisesti lukita kamera pelaajan liikkeisiin ja sijoittaa pelaaja keskelle ruutua, mutta tämä ratkaisu ei ole erityisen hyvä. Parempi ratkaisu on aloittaa kameran korjaaminen ja pelaajan seuraaminen vasta tämän ylitettyä määritetyn rajan ruudulla. Näin nähdään pidemmälle pelaajan menosuuntaan jolloin riski viholliseen törmäämiseen juostessa pienenee. Kamera ei myöskään heilu pelaajan tehdessä äk-

kinäisiä edestakaisia liikkeitä. Menetelmä toimii hyvin vaakatasossa, mutta kameran ohjaus pystytasossa voi vaatia lisätoiminnallisuutta. Esimerkiksi pelaajan hypätessä ruudun yläosassa sijaitsevan maaston osan päälle voidaan haluta kameraa nostaa laskeutuessa. Usein kameran ohjaus ei riipu pelkästään pelaajan koordinaateista vaan myös liikkumiseen käytettävästä metodista.

6. FYSIIKKA

Kaikkien kappaleiden käyttäytymiseen maailmankaikkeudessa pätevät fysiikan lait. Näitä lakeja voidaan hyödyntää videopeleissä realismin lisäämiseksi halutussa määrin. Fysiikan mallinnus jää useimmiten matemaattisen mallin tasolle, sillä todellinen maailma sisältää liikaa yksityiskohtia. Matemaattista mallia voidaan soveltaa vaihtelevalla tarkkuudella. Fysiikan mallinnuksesta on tullut yhä tärkeämpi osa pelejä tekniikan kehittyessä.

Tässä luvussa käsitellään mekaniikkaa, joka on kappaleiden välistä vuorovaikutusta käsittelevä fysiikan alue. Perinteisen tasoloikkapelin fysiikan mallintamisen toteutamiseen riittää ymmärrys klassisesta Newtonin mekaniikasta. Luvussa käydään läpi mekaniikan perusasioita ja kerrotaan, kuinka fysiikan toiminta voidaan ohjelmoida.

6.1 Liike

Objektien liikkuminen on erittäin tärkeää tasoloikkapelissä. Tutkitaan käsitteitä sijainti, aika ja nopeus, ja miten niiden avulla toteutetaan kappaleiden liikkuminen.

6.1.1 Sijainti, s

Sijainti määrittelee kappaleen paikan maailmankaikkeudessa. Sijainti saadaan päättämällä, mikä on origon (sijainti joka merkkaa nollakohtaa) paikka maailmankaikkeudessa ja laskemalla kappaleen ja origon välinen etäisyys. Sijainti ei voi olla millekään kappaleelle absoluuttinen, vaan se on aina suhteellinen valittuun viitepisteeseen. Koska etäisyyden perusyksikkö on metri, on se myös sijainnin perusyksikkö.

$$[s] = m$$

Laskuissa kappaleen sijainnin ilmaisee koordinaattivektori, kaksiulotteisissa tapauksissa vektorissa on 2 lukua ja kolmiulotteisissa tapauksissa 3 lukua. Kaksiulotteisen pelin tapauksessa tarvitaan siis jokaiselle objektille kaksi muuttujaa sijainnin ilmoittamiseksi. Kaksiulotteisissa peleissä origona toimii yleensä kentän vasen yläkulma. Sijainnin x-koordinaatti kasvaa yleensä oikealle mentäessä ja y-koordinaatti alaspäin mentäessä. y-koordinaatti kasvaa nimenomaan näin päin, sillä myös näytön pikselien y-koordinaatit kasvavat alaspäin mentäessä.

Vaikka sijainnin perusyksikkö onkin metri, eivät kaksiulotteisissa peleissä koordinaattimuuttujien arvot yleensä ilmaise objektien sijaintia metreinä vaan pikseleinä. Kun objekti liikkuu yhden pikselin verran johonkin suuntaan, muuttuu koordinaattimuuttujan arvo luvulla 1.

6.1.2 Aika, t

Aika on vaikeammin määriteltävä suure, mutta se erottaa avaruuden erilaiset tilat toisistaan. Ajan perusyksikkö on sekunti.

$$[t] = s$$

Tasoloikkapelissä on tärkeää saada selville tietystä hetkestä kulunut aika tarpeeksi tarkasti. Joka peliruudulla suoritettavassa fysiikkakoodissa täytyy tietää kuinka paljon aikaa on kulunut viime kerrasta, kun fysiikkakoodi suoritettiin. Useat valmiit kirjastot antavat kuluneen ajan millisekunteina, joten mikäli muissa suureissa aika on ilmaistu sekunneissa, tulee muuntaa yksiköt.

6.1.3 Nopeus, v

Nopeus, v, ilmaisee kappaleen sijainnin muutosta Δs ajassa t.

$$v = \Delta s/t$$

$$[v] = [s]/[t] = m/s$$

Nopeuden kaava voidaan kääntää muotoon:

$$\Delta s = v * t$$

Tällä tavoin voidaan laskea kappaleen sijainnin muutos sen nopeuden ja kuluneen ajan perusteella. Fysiikkakoodissa siis kerrotaan kappaleen nopeus ruutujen välissä kuluneella ajalla ja lisätään tulos objektin sijaintimuuttujaan. Mikäli sijaintimuuttujissa yksi yksikkö vastaa yhtä pikseliä, myös nopeudet tulisi ilmaista pikseleinä sekunnissa.

6.2 Kiihtyvä liike ja dynamiikka

Edellä kuvaillulla tavalla voidaan toteuttaa peliin objektien liikkuminen tietyllä nopeudella. Usein halutaan kuitenkin mallintaa kappaleiden nopeuksien muutosta ja vuorovaikutusta.

6.2.1 Kiihtyvyys, a

Kiihtyvyys on kappaleen nopeuden muutos Δv ajassa t.

$$a = \Delta v/t$$

$$[a] = [v]/[t] = m/s^2$$

Kiihtyvyyden kaava voidaan kääntää muotoon:

$$\Delta v = a * t$$

Näin voidaan laskea kappaleen nopeuden muutos kiihtyvyyden ja kuluneen ajan perusteella. Fysiikkakoodissa kappaleelle lasketaan uusi nopeus samalla tavoin kuin ei-kiihtyvän kappaleen tapauksessa laskettiin uusi sijainti. Kiihtyvän kappaleen uuden sijainnin laskemisessa täytyy ottaa huomioon, että sen nopeus muuttuu koko ajan. Vanha kaava ei siis enää päde, vaan koska tiedetään nopeuden muutos, tulee sijainnin muutokseen lisätä nopeuden muutoksen integraali ajan suhteen:

$$\Delta s = 1/2at^2 + vt$$

Fysiikkakoodissa kaavaa käytetään siten, että v merkitsee kappaleen nopeutta ennen ajan t kulumista ja nopeuden kiihtymistä. Se on siis kappaleen alkunopeus ennen fysiikkakoodin suorittamista. Koodissa objektin sijaintiin siis lisätään kaavalla saatava Δs alkuperäisen v:n mukaan ja tämän jälkeen lasketaan kappaleelle uusi nopeus lisäämällä kiihtyvyyskaavalla saatava Δv vanhaan nopeuteen. Uusi nopeus on objektin alkunopeus seuraavalla kerralla kun fysiikkakoodi ajetaan.

6.2.2 Dynamiikka

Dynamiikka on kappaleen liikettä voimien kannalta käsittelevä klassisen mekaniikan osa-alue. Tutkitaan mitä suureita dynamiikkaan liittyy ja miten ne liittyvät toisiinsa.

Massa, m

Massa ilmaisee kappaleen sisältävän materian määrään. Sitä ei tule sekoittaa painoon, jolla itseasiassa tarkoitetaan kappaleeseen kohdistuvaa vetovoimaa, tai toisinpäin. Massan perusyksikkö on kilogramma.

Voima, F

Voima on hankala määritellä, mutta se on kappaleen toiseen kappaleeseen koskettamalla kohdistava suure ja sen yksikkö on Newton, N.

Kiihtyvyys dynamiikan kannalta

Liikkeen kannalta kiihtyvyys määrittelee kappaleen nopeuden muutoksen suhteessa ajan muutokseen. Mekaniikan kannalta kiihtyvyys on riippuvainen kappaleen massasta ja siihen kohdistuvista voimista. Newtonin toinen laki määrittelee kappaleeseen kohdistuvien voimien summan $\sum F$, kappaleen massan m ja kiihtyvyyden välisen suhteen:

$$\sum F = ma$$

Tämä voidaan kääntää muotoon:

$$a = \sum F/m$$

Näin voidaan laskea kappaleen kiihtyvyys, kun tiedetään sen massa ja siihen kohdistuvat voimat. Mitä enemmän massaa kappaleella on, sitä enemmän voimaa tarvitaan sen kiihdyttämiseen.

Pelin fysiikkakoodissa kappaleeseen vaikuttavia voimia voidaan ajatella vektoreina. Nämä vektorit lasketaan yhteen, jolloin saadaan summavektori. Se jaetaan kappaleen massalla, jolloin saadaan kiihtyvyysvektori. Vektorin komponenttien voi jokaisen ajatella vaikuttavan itsenäisesti kappaleen liikkeeseen eri koordinaattiakseleiden suuntaisesti. Siis kun kappaleen kiihtyvyydet eri akseleille on selvitetty, voidaan liikekaavoilla laskea uudet arvot kappaleen sijainnille ja nopeudelle akseli kerrallaan.

Kun pelaaja painaa suuntanappia pelissä, voidaan silloin ajatella että pelaajaobjektiin kohdistuu silloin tietyn suuruinen voima haluttuun suuntaan. Fysiikkakoodin toiminta taas johtaa siihen, että pelaajalle lasketaan nollasta poikkeava kiihtyvyys, jolloin tämän nopeus kasvaa ja sijainti muuttuu.

6.2.3 Gravitaatio

Maailmankaikkeudessa kaikki kappaleet vetävät toisiaan puoleensa niiden massasta riippuvalla voimalla. Perinteisessä tasoloikkapelissä ei kuitenkaan tarvitse huomioida kaikkien kappaleiden vetovoimia, vaan ainoastaan Maan vetovoima. Maan gravitaatiovakio $g = 9,81 m/s^2$. Kun kappaleen massa kerrotaan gravitaatiovakiolla, saadaan tuloksena voima jolla Maa vetää sitä puoleensa. Mitä suurempi kappaleen massa on, sitä suuremmalla voimalla Maa vetää sitä puoleensa. Kuitenkin, mitä

enemmän massaa kappaleella on, sitä suurempi voima tarvitaan sen kiihdyttämiseen. Niinpä jokaiselle kappaleelle pätee sama putoamiskiihtyvyys g.

Kaksiulotteisissa tasoloikkapeleissä, joissa sijainnin yksikkö on pikseli eikä metri, ei ole luultavasti järkeä käyttää suoraan vakiota g. Ensinnäkin yksiköt menevät sekaisin, eikä tasoloikkapeleissä gravitaatio ole yleensä yhtä voimakas kuin Maan pinnalla. Ohjelmoijan tulee itse löytää peliin sopiva gravitaatiovakio. Gravitaatio voidaan toteutettaa siten, että kerrotaan objektin massa gravitaatiovakiolla ja kohdistetaan objektiin tuloksen suuruinen alaspäin osoittava voima.

6.2.4 Liikemäärä, p ja Impulssi, J

Liikemäärä on kappaleen massan ja nopeuden tulo:

$$p = mv$$

$$[p] = [m][v] = kg * m/s$$

Impulssi tarkoittaa kappaleeseen vaikuttavan voiman suuruuden ja vaikutusajan tuloa:

$$J = Ft$$

$$[J] = [F][t] = kg * m/s$$

Huomataan, että liikemäärällä ja impulssilla on sama yksikkö. Impulssin voidaan siis ajatella vastaavan kappaleen liikemäärän muutosta. Erittäin lyhyen ajan (esimerkiksi lyhyemmän kuin kahden peliruudun välisen ajan) vaikuttavaa voiman vaikutusta kappaleen nopeuteen voidaan mallintaa impulssin avulla:

$$\Delta v = J/m$$

Kun tiedetään kappaleeseen vaikuttavan impulssin suuruus, muutetaan kappaleen nopeutta Δv:n verran. Esimerkiksi pelaajan hypätessä ja irtautuessa maasta, voidaan hyppyyn tarvitun voiman vaikutusta ajatella impulssina. Pelaajan y-suuntaiseen nopeuteen lisätään tällöin impulssin suuruus jaettuna pelaajan massalla.

6.2.5 Kitka

Kitka on voima joka vastustaa liikettä tai liikkeen alkamista kahden kappaleen toisiaan koskettavien pintojen välillä. Kitkan laskemiseksi täytyy ensin tietää kappaleen

toiseen kappaleeseen kohdistava normaalivoima. Se on voiman, jolla kappale vaikuttaa toiseen kappaleeseen, kohtisuora komponentti. Esimerkiksi jos laatikkoa työnnetään tasaisen lattian päällä eteenpäin hieman alaspäin painaen, vaikuttaa laatikkoon sillon eteenpäin työntö, alaspäin työntö sekä gravitaatio. Normaalivoima on tällöin alaspäin työnnön ja gravitaation summa.

Kitkakerroin on normaalivoiman ja pinnan suuntaan työnnettävään kappaleeseen päinvastaiseen suuntaan vaikuttavan vastavoiman suhde. Kitkakerroin riippuu molempien kappaleiden materiaaleista. Kun kappale on ensin paikallaan ja sitä yritetään siirtää, siihen vaikuttaa lepokitka, joka saadaan kertomalla normaalivoima materiaaliparin lepokitkakertoimella. Kun kappale on liikkeessä, vaikuttaa siihen liikettä vastustava liukukitka, joka saadaan kertomalla normaalivoima materiaaliparin liukukitkakertoimella. Liukukitkakerroin ei koskaan ole suurempi kuin lepokitkakerroin, eli voima jolla kappale on saatu liikkeelle myös riittää sen pitämiseksi liikkeessä.

Kitkaa käytetään tasoloikkapeleissä esimerkiksi pysäyttämään pelaajan liike, kun tämä ei paina mitään suuntanappia. Liukkaita pintoja voidaan simuloida antamalla niille pienempi kitkakerroin.

6.2.6 Noste

Noste on voima jonka neste tai kaasu kohdistaa siihen upotettuun kappaleeseen. Voima on yhtä suuri kuin kappaleen korvaaman aineen paino. Täytyy siis tietää korvatun aineen tiheys sekä kappaleen tilavuus ja kertoa nämä toisillaan ja gravitaatiovakiolla. Näin saadaan selville voima, jolla kappaletta nostetaan.

Tasoloikkapelissä nostetta mallinnetaan esimerkiksi peliobjektien ollessa veden alla. Niille lasketaan äskeisellä tavalla voima, joka kohdistetaan niihin ylöspäin. Näin simuloidaan sitä, että kappaleet painuvat hitaasti pohjaan, eivätkä putoa nopeasti kuten tekisivät ilmassa.

6.2.7 Mallinnuksen toteuttaminen

Fysiikkakoodia kirjoitettaessa lähtökohtana on, että jokaiseen kappaleeseen kohdistuu voimia jotka kiihdyttävät kappaleen liikettä. Pelaajaan kohdistuvia voimia ovat esimerkiksi gravitaaatio, tämän maanpinnalla gravitaation suuruinen vastavoima ja kitka. Pelaajan liikkumisesta voidaan ajatella, että kävellessään pelaaja kohdistaa itseensä kävelysuuntaan kohdistetun voiman joka liikuttaa tätä.

Koodissa tulisi ensin laskea yhteen kaikki kappaleeseen kohdistuvat voimat ja saada kokonaisvoima. Kokonaisvoiman ja massan avulla selviää kappaleen kiihtyvyys. Aiemmin esitetyillä kaavoilla päivitetään sitten kappaleen sijaintia ja nopeutta. Kappaleisiin kohdistuu myös impulsseja, jotka muuttavat kappaleen nopeutta. Edellä esitellyllä kaavalla voidaan impulssin suuruuden mukaan selvittää, kuinka

paljon kappaleen nopeus muuttuu.

Fysiikan mallinnuksessa aika tulisi jakaa suhteellisen pieniin osiin, sillä peli voi alkaa käyttäytyä väärällä tavalla, jos liian pitkiä harppauksia tehdään tarkastamatta välillä, millaiset voimat vaikuttavat mihinkin kappaleeseen. Mikäli ruudunpäivitys toimii sulavasti ja fysiikkakoodi suoritetaan jokaisella ruudulla, ei ongelmaa synny. Mikäli kuitenkin peli alkaa pätkiä ja Δt kasvaa liian suureksi, voi peli alkaa toimia väärin. Yksi ratkaisu on määrittää kiinteän suuruinen Δt niin muun pelilogiikan kuin myös fysiikan suoritukseen, joiden koodia suoritetaan aina Δt:n välein. Esimerkiksi, jos viimeisessä peliruudun päivityksessä on mennyt aikaa kolme kertaa tämän vakion aika, suoritetaan fysiikka- ja logiikkakoodi kolme kertaa. Tässä oletetaan, että suurin osa laskentatehosta menee pelin grafiikan piirtämiseen, eikä fysiikan ja logiikan suorittamisessa mene niin paljon aikaa etteikö niiden suorittaminen useaan kertaan olisi mahdollista. Muuten pelin alkaessa pätkimään, fysiikan ja logiikan suorittaminen viivästyttäisi seuraavan ruudun valmistumista vielä enemmän, ja jokaisen grafiikkaruudun välissä jouduttaisiin suorittamaan aina vain enemmän fysiikka- ja logiikkakoodia, jolloin peli hidastuisi hidastumistaan.

Kaksiulotteisissa peleissä, joissa hahmojen sijaintia käsitellään pikselin tarkkuudella, saattaa olla hyvä idea ilmaista objektien sijainti kokonaisluvuilla liukulukujen sijaan. Näin esimerkiksi törmäystarkistuksesta tulee selkeämpää, kun hahmot eivät voi olla esimerkiksi puolikkaan pikselin verran toisen kappaleen sisällä, eikä liukulukulaskennan epätarkkuus vaikuta tuloksiin. Kuitenkin, jos hahmo liikkuu niin pienellä nopeudella, ettei se liiku kokonaista pikseliä yhden ruudun aikana, ei se tule liikkumaan lainkaan, ellei sen liikkeen pikselin murto-osista pidetä jollain tapaa kirjaa. Peliobjekteille voidaan käyttää ylimääräisiä muuttujia erottamaan desimaalipilkun eri puolet sijainnin arvoista.

Peleissä, joissa sijaintimuuttujissa luku 1 merkitsee yhtä pikseliä eikä metriä, ei voida käyttää fysiikan metrijärjestelmän yksiköitä suoraan. Yksi mahdollisuus on määritellä ensin montako pikseliä pelin maailmassa yksi metri on, ja jos tämän jälkeen halutaan käyttää metrijärjestelmän yksiköitä esimerkiksi voiman ilmaisemiseen, tulee joko laskea laskut ensin metrijärjestelmää käyttäen ja sitten muuntaa pikseleiksi tai vaihtoehtoisesti muuttaa suureet siten, että yksiköissä metrien sijaan käytetäänkin pikseliä. Mikäli laskuja tehdään suoraan ilman muunnoksia, käytetään vääriä yksiköitä ja täten päädytään vääriin tuloksiin. Toisaalta ei välttämättä tarvitse määrittää, montako pikseliä yksi metri on, vain käyttää järjestelmää jossa metrit on korvattu pikseleillä, ja määrätä suureet sen suuruisiksi että kappaleet käyttäytyvät siten, että pelattavuus toimii halutulla tavalla.

Mikäli fysiikan mallinnuksen ei tarvitse olla erityisen tarkkaa, on mahdollista oikaista ohjelmoinnissa useilla tavoilla. Esimerkiksi mikäli ei jakseta välittää kappaleiden massoista, voidaan niiden kiihtyvyyttä muuttaa suoraan halutulla vakiolla.

Myös nopeutta voidaan muuttaa suoraan vakiolla, esimerkiksi kun pelaaja hyppää tai tähän kohdistuu jokin muu impulssi. Joissain erittäin yksinkertaisissa peleissä ei objekteilla ole erillistä kiihtyvyyttä ollenkaan, vaan niiden nopeudet muuttuvat yhtäkkiä huomattavasti eri suuruisiksi. Tapauksissa, joissa yhden peliruudun logiikkaan ja piirtämiseen menee aina vakion suuruinen aika, voidaan kulunut aika unohtaa kokonaan ja vain lisätä objektien sijaintimuuttujaan joitain vakioita, riippuen niiden menosuunnista.

7. TÖRMÄYSTARKISTUS

Törmäystarkistus on yksi pelilogiikan tärkeimmistä osista. Yksinkertaistetusti törmäystarkistus tarkoittaa, että estetään peliobjektien kulkeminen maaston niitä estävien osien sekä mahdollisesti muiden peliobjektien läpi. Mikäli pelihahmo ei pysähtyisi seiniin ja esteisiin, ei pelin maastolla olisi merkitystä.

Törmäystarkistuksen toteutuksen monimutkaisuus riippuu maaston monimutkaisuudesta ja siitä, kuinka se estää pelihahmojen liikkeitä. Eräässä yksinkertaisimmista tapauksista maastoa kuvataan tiilikartalla, jossa palaset ovat aina neliön muotoisia ja joko kokonaan läpäisemättömiä tai täysin läpäistäviä. Monimutkaisuutta lisäävät maaston neliöstä poikkeavat muodot sekä läpäistävyyden riippuvuus läpäisysijannista ja kulkusuunnasta. Monimutkaisemmassa esimerkissä maasto sisältää erilaisia mäkiä ja jotkut maaston palaset voidaan läpäistä alapuolelta muttei yläpuolelta.

7.1 Rajaus

Törmäystarkistusta suunnitellessa on tärkeää määritellä miten peliobjektit ja maasto ovat rajattu, eli missä kohtaa objektien ja maaston kappaleiden reunat sijaitsevat sekä mistä suunnasta minkäkin reunan voi ylittää.

Usein tasoloikkapeleissä yhden peliobjektin rajaamiseen käytetään yhtä suorakulmiota, jota testataan maaston rajoja vasten. Mikäli peliobjektin muoto ei sovi hyvin suorakulmioon, voidaan toki käyttää useampiakin suorakulmioita määrittelemään objektin rajat tai sitten käyttää erilaisia muotoja, mutta tällöin törmäysten tarkistuksesta tulee monimutkaisempaa.

7.2 Posteriori (diskreetti) tarkistus

Posteriori- eli jälkitarkistuksessa pelin fysiikkaa simuloidaan pienen aikavälin verran, vaikkapa fysiikkaluvussa käsitellyn kiinteän Δt:n verran, jolloin saadaan peliobjektien sijainnit ennen ja jälkeen siirtymän. Tämän jälkeen tutkitaan, ovatko peliobjektit ylittäneet määriteltyjä rajoja, ja siirretään ne arvioituun törmäyskohtaan.

On helppoa tutkia, onko peliobjekti jonkin kielletyn maaston osan sisällä, mutta jo kaksiulotteisessa liikkeessä on vaikempaa tutkia, missä kohtaa varsinainen törmäys on tapahtunut ja mihin sijaintiin peliobjekti on siirrettävä törmäyksen havaitsemisen jälkeen. Jotta vältyttäisiin monimutkaiselta yhtälönratkaisulta, voidaan hyödyntää

törmäystarkastuksen suorituksen tiheydestä seuraavia siirtymien lyhyitä pituuksia ja jakaa peliobjektien siirtymä pääakselien suuntaisiksi komponenteiksi. Ensin voidaan esimerkiksi liikuttaa peliobjektia x-suunnassa ja testata, onko se ylittänyt rajaa, jonka läpi ei pitäisi päästä vaakasuunnassa. Rajan ylityksen tapahtuessa siirretään peliobjekti takaisin rajalle. Tämän jälkeen jatketaan tarkastelua y-suunnassa.

Törmäystarkistus voi myös olla järkevä suorittaa yksi objekti kerrallaan. Kun tutkitaan vaikka kahden peliobjektin törmäystä, annetaan ensimmäisen objektin liikkua pitäen toista objektia alkuperäisessä sijainnissaan, tutkitaan ensimmäisen objektin törmäystä ja korjataan mahdollisesti tätä. Tämän jälkeen ensimmäiselle objektille on saatu lopullinen sijainti, jonka jälkeen voidaan testata toisen objektin törmäystä.

Diskreetin tarkistuksen varjopuolena on sen epätarkkuus: ei saada tietää törmäyksen tarkkaa hetkeä ja sijaintia. On myös mahdollista, että jos vakio Δt on liian suuri, kappaleet saattavat kokonaan ohittaa toisensa, eikä törmäystä havaita.

7.3 Priori (jatkuva) tarkistus

Törmäystarkistus on teoriassa mahdollista toteuttaa täydelllisesti matemaattisesti analysoiden kappaleiden liikettä ja toistensa leikkaamista.

Priorissa eli ennen törmäystä tapahtuvassa tarkistuksessa tiedetään kappaleiden alkusijainnit-, nopeudet sekä näihin vaikuttavat voimat. Näiden perusteella on mahdollista selvittää ajankohta, jolloin kappaleiden törmäys tapahtuu. Näin tiedetään täsmälleen törmäyksen aika sekä kappaleiden sijainnit törmäyshetkellä.

Kuitenkin jatkuva-aikainen törmäystarkistus on vaikea toteuttaa. Tapaus jossa kappaleet ovat rajattu suorakulmioilla, ja kappaleiden kiihtyvyys on vakio, on vielä ratkaistavissa, mutta on esimerkiksi erittäin vaikeaa, ellei jopa mahdotonta, muodostaa törmäysyhtälö tapaukselle, jossa kappaleet ovat mielivaltaisia kolmiulotteisia malleja joihin vaikuttavat vaihtelevan suuruiset voimat, ja jotka vielä kaiken lisäksi pyörivät itsensä ympäri.

7.4 Törmäysvaste

Luvussa on tähän asti kuvattu, kuinka törmäykset havaitaan sekä kuinka estetään objektien liikkuminen kielletyille alueille. Mikäli halutaan realistista käytöstä, pitäisi törmäysten myös vaikuttaa objektien nopeuteen. Tämä voidaan toteuttaa kohdistamalla objekteihin voimia tai impulsseja törmäyksen tapahtuessa. Törmäyskoodiin pitää siis osata lisätä peliobjektiin kohdistuva voima tai impulssi. Tämän jälkeen voima tai impulssi muuttaa objektin nopeutta kun fysiikkakoodi suoritetaan seuraavan kerran.

8. GRAFIIKKA

Videopelien olennaisimpia osia on grafiikka, kuuluuhan se käyttäjältä saatavan syötteen lisäksi videopelin alkuperäiseen määritelmään. Pitkän aikaa tekniikan kehityksessä on keskitytty nimenomaan grafiikan piirtoon.

8.1 Rasterinäyttö

Aluksi grafikan esittämiseen käytettiin oskilloskooppeja, myöhemmin vektorinäyttöjä ja lopulta rasterinäyttöjä. Rasterinäytöllä tarkoitetaan laitetta, jossa kuva esitetään pistematriisilla. Kuva piirretään näytölle asettamalla jokaisen näytöllä olevan pikselin väriarvon halutuksi. Jo kohtalaisen matalallakin resoluutiolla näytöllä olevia pikseleitä on suuri määrä, joten rasterinäytölle piirrettäessä tarvitaan enemmän prosessoritehoa kuin oskilloskoopin tai vektorinäytön ohjauksessa.

8.1.1 Näyttötilat

Näytönohjaimissa on useimmiten valittavissa useampia näyttötiloja, jotka määrittävät pikselien määrän ruudulla ja yhtä pikseliä kohti käytettävien bittien määrän. Joissain järjestelmissä näyttötila myös määrittää millä metodeilla näytölle piirtäminen tapahtuu.

Resoluutio vastaa pikselien määrää ruudulla. Mitä suurempi resoluutio, sitä pienempiä pikselit ovat ja yksityiskohtaisemmin kuva voidaan esittää näytöllä. Näyttölaitteiden kuvakokojen kasvaessa myös resoluution tulee kasvaa samassa suhteessa, jotta pikselien koko pysyisi samana ja kuva yhtä terävänä. Resoluution kasvaessa piirrettävien pikselien määrä kasvaa, jolloin myös laskentatehon tarve ja käytettävän muistin määrä kasvaa.

Bittisyvyys ilmaisee, kuinka monta bittiä käytetään merkitsemään pikselin väriä. Mitä enemmän bittejä, sitä enemmän on yhtäaikaisia mahdollisia värejä ruudulla. Värinäytöissä eri värit saadaan aikaan yhdistämällä punainen, vihreä ja sininen komponentti vaihtelevilla intensiteeteillä. Pienempien bittisyvyyksien tapauksessa yhtä pikseliä kohti käytettävät bitit eivät riitä ilmaisemaan kolmen eri komponentin arvoja tarpeeksi tarkasti monipuolisten värien esittämiseksi. Siksi pienillä bittisyvyyksillä pikselin arvo ei ilmaise itse väriä vaan indeksin jossa varsinainen väri on määritelty paletiksi kutsutussa muistialueessa. Näytöllä yhtä aikaa esiintyvien värien määrä ei lisäänny palettia käyttämällä, mutta värivalikoima kasvaa paletin värin

ilmaisussa käytettävien bittien määrän ollessa yksittäisen pikselin värin ilmaisuun käytettävien bittien määrää suurempi. Paletin väriarvoa muuttamalla myös kyseistä väri-indeksiä käyttävän näytöllä olevan pikselin väri muuttuu automaattisesti. Tätä ominaisuutta käyttämällä voidaan toteuttaa niin sanottuja palettiefektejä, joilla kuvaa voidaan muuttaa ilman näyttöpuskuriin kirjoittamista. Bittisyvyyden kasvaessa pikselin väri voidaan ilmaista suoraan pikselin arvolla. Tällöin eri värikomponentit ilmaistaan luvun eri biteillä. Nykyään usein käytetään 8 bittiä yhden värikomponentin ilmaisemiseen (24-bittiset värit), ja mahdollisesti myös ylimääräistä 8 bittiä (32-bittiset värit) alpha-arvon (läpinäkyvyys) ilmaisemiseen, vaikkakin alpha-arvoa käytetään vain piirron välivaiheissa, ei lopullisessa näytöllä näkyvässä kuvassa. Suurempi bittisyvyys tarkoittaa myös suurempaa piirrettävää datamäärää.

8.1.2 Ruudunpäivitys

Ruudunpäivityksellä voidaan tarkoittaa näytönohjaimen tai grafiikkapiirin näyttölaitteelle lähettämän kuvasignaalin muutosta, näytettävää kuvaa edustavalle muistialueelle kirjoittamista tai mahdollisesti muistialuetta osoittavan osoittimen arvon päivittämistä. Yksinkertaisesti sanottuna ruudunpäivitys tarkoittaa kuitenkin näytöllä olevan kuvan vaihtumista. Jotta käyttäjä ei näkisi piirron eri vaiheita tehdään päivitys yleensä vasta piirron valmistuessa. Väliaikaisen ei-näkyvän muistialueen käyttämistä piirtoa varten kutsutaan kaksoispuskuroinniksi.

Ruudunpäivityksen nopeus riippuu ajasta, joka kuluu yhden peliruudun prosessoimiseen. Suuri osa tästä ajasta kuluu grafiikan piirtämiseen. Mitä suurempi resoluutio ja bittisyvyys ovat käytössä, sitä enemmän piirrettävää dataa on ja hitaampi päivitysnopeus on. Sulavana nopeutena pidetään usein noin 50-60 ruutua sekunnissa, vaikkakin jotkut väittävät erottavansa tätä suuremmat päivitysnopeudet.

8.2 Spritet

Vaikka rasterinäytöt vakiintuivat pian, ei kaikissa rasterinäyttöä käyttävissä pelilaitteissa kuitenkaan ollut muistin korkean hinnan vuoksi varsinaista näytöllä näkyvän pistematriisin säilytykseen käytettävää näyttöpuskuria. Ratkaisuna käytettiin spritejä, jotka ovat laitteiston tukemia grafiikkaobjekteja, muistiin tallennettuja bittikarttoja jotka laitteisto piirtää näytölle haluttuun sijaintiin. Ratkaisu oli hyvä myös CPU:n alhaisen kellotaajuuden vuoksi, sillä kuvien kopioiminen muistissa on datamäärän vuoksi työläs operaatio. Pelikonsolit useimmiten käyttivät laitteistoa grafiikan piirtoon PC-koneiden vasta 1990-luvulla alkaessa yleisesti käyttää laitteistokiihdytettyä grafiikkaa.

8.2.1 Toteutus

Varhaiset grafiikkapiirit muokkasivat näyttölaitteelle lähetettävää signaalia ilman varsinaista näyttömuistia, kuten spritejen piirtäminen alunperin tapahtui. Kuitenkin näyttömuistin yleistyttyä piirto tapahtui muistin sisällön kopioimisella, joka tapahtui spriten rivi kerrallaan. Piirtämisessä tulee ottaa huomioon piirtämättä jätettävät pikselit. Aluksi spritet olivat yksivärisiä kuvia, mutta laitteisto alkoi myöhemmin tukea useampia värejä spriteä kohti. Suhteellisen kauan laitteisto keskittyi vain spritejen piirtämiseen, mutta myöhemmin alettiin tukea erilaisia sprite-efektejä.

8.2.2 Efektit

Laitteiston kehityttyä spriteille voitiin peruspiirron lisäksi käyttää erilaisia efektejä. Näitä olivat skaalaus, rotaatio ja alpha-blending.

Skaalaus. Spriten skaalauksella tarkoitetaan sen koon muuttamista piirrettäessä. Sprite voidaan esimerkiksi piirtää kaksinkertaiseksi näytölle, kuitenkin kokoa voidaan muuttaa pikselinkin tarkkuudella. Kuvan leveyttä ja korkeutta voidaan myös muuttaa eri suhteissa, eli spriten alkuperäisen muodon ei tarvitse säilyä piirrettäessä, minkä ansiosta kuvia voidaan tavallisen suurentamisen tai kutistamisen lisäksi litistää tai venyttää vaaka- tai pystysuunnassa. Skaalausta itse ohjelmallisesti toteuttaessa jää ongelmaksi osata näytteistää alkuperäistä spriteä kohdekuvaan. Pitää siis tietää, moneenko kertaan alkuperäisen kuvan mikäkin pikseli tulee suurentaessa kopioida kohdekuvaan tai koska pikseleitä jätetään piirtämättä kuvaa pienentäessä.

Rotaatio. Spriten rotaatiolla tarkoitetaan sen pyörittämistä itsensä ympäri. On täysin toteutuksesta riippuvaista millä tarkkuudella spriteä voi pyörittää, mutta rotaatio tapahtuu yleensä hyvin sulavasti, jolloin esimerkiksi vähintään yhden asteen tarkkuus olisi suotavaa. Rotaatiota itse ohjelmallisesti toteutettaessa täytyy osata laskea, mistä kohtaa spriteä kohdekuvaan otetaan mikäkin pikseli. Rotaation lähtökohta onkin, että piirretään käännetty suorakulmio näytölle, ja selvitetään mistä kohtaa alkuperäistä spriteä sen mikäkin pikseli on peräisin, eikä se, että alkuperäisestä spritestä käydään läpi jokainen pikseli ja käännetään se kohdesijaintiin.

Alpha-blending. Spriten alpha-blendingilla tarkoitetaan spriten piirtämistä osittain läpinäkyvänä, esimerkiksi piirto voidaan tehdä siten että taustalla oleva grafiikka näkyy piirron jälkeen puoliksi yhtä vahvana kuin ennen piirtoa ja piirretty sprite näkyy puolliksi yhtä vahvana kuin normaalisti. Toteutuksesta riippuu kuinka tarkasti taustan ja sen päälle piirrettävän spriten vahvuuksien suhteen voi säätää.

8.3 Animaatio

Animaatiolla tarkoitetaan tässä peliobjekteja edustavien kuvien liikettä. Tämä toteutetaan vaihtamalla piirrettävää kuvaa ajan kuluessa. Mitä nopeammin kuvaa

vaihdetaan ja vähemmän uusi kuva eroaa edellisestä kuvasta sitä sulavampaa animaatio on, kuitenkin jopa vain kahta kuvaa käyttämällä voidaan saada tehokas vaikutelma hahmon liikkeestä. Animaation piirtämiseksi vaaditaan tietorakenne, jossa ilmaistaan mistä kuvista animaatio koostuu. Spriten määritelmästä riippuen tietorakenne voi sisältää osoittimen kaikki animaatioruudut sisältävään spriteen sekä animaatioruutujen koordinaatit tai vaihtoehtoisesti osoittimet jokaiseen spriteen, josta animaatio koostuu. Yhden peliobjektin animaation vaiheesta tulee pitää objektikohtaisesti kirjaa ellei haluta kaikkien samaa animaatiota käyttävien peliobjektien olevan aina samassa vaiheessa. Usein animaatiossa vain osa kuvasta vaihtuu. Tällöin voidaan muistin säästämiseksi käyttää monimutkaisempaa tietorakennetta, jossa yhtä animaatioruutua kohden määritellään useampia kuvia joista animaatio koostuu. Näin useissa animaatioruuduissa esiintyvä kuvan osaa ei tarvitse sijoittaa muistiin kuin vain yhden kerran.

8.4 Laitteistokiihdytetty grafiikka

Laitteistokiihdytetyllä grafiikalla tarkoitetaan piirtämisen suorittamista laitteistoeikä ohjelmakooditasolla. Etuna ovat paitsi keskusyksikölle muihin tehtäviin jäävät suoritussyklit, myös piirron suurempi nopeus laitteistotasolla. Jotta voisi ymmärtää, miksi ohjelmallisesti ei voida piirtää grafiikkaa yhtä nopeasti, on ymmärrettävä ero mikroprosessorilla ajettavan ohjelman ja digitaalisen logiikan välillä. Mikroprosessori lukee muistista käskyjä, jotka se tulkitsee ja suorittaa. Käskyt ovat useimmiten muistista lukemista, muistiin kirjoittamista sekä aritmeettisloogisia operaatioita. Tällä toiminnallisuudella pystytään toteuttamaan joustavasti erilaisia tietokoneohjelmia, mutta tiettyyn tarkoitukseen erikoistunut laitteistolla toteutettu logiikka pystyy suorittamaan tehtävän nopeammin. Esimerkiksi grafiikkalaitteiston ei tarvitse kuvaa näytölle kopioidessaan jokaisen piirrettävän pikselin kohdalla prosessorin tavoin noutaa, tulkita ja suorittaa jokaista tehtävään liittyvää ohjelmakäskyä, joihin usein kuuluu pikselin arvon lukeminen muistista, arvon kirjoittaminen toiseen muistipaikkaan, lähde- ja kohdeosoittimien kasvattaminen, mahdollisen silmukkamuuttujan arvon vähennys, vertailu nollaan ja hyppykäsky. Ohjelmakäskyjä vastaavat operaatiot tapahtuvat kyllä myös laitteistototeutuksessa, mutta digitaalisessa logiikassa järjestyksessä suoritettavien konekäskyjen sijaan on rinnakkaisesti toimivia komponentteja, jotka ovat toisiinsa yhteydessä. Koko järjestelmän maksimikellotaajuus riippuu niinsanotusta hitaimmasta polusta, eikä suoraan rinnakkain toimivien komponenttien määrästä. Myös mikroprosessori on digitaalinen logiikka, jossa on rinnakkain toimivia komponentteja. Tätä ominaisuutta hyödyntäen prosessoreissa on käytetty niinsanottua liukuhihnoitusta eli useampien ohjelmakäskyjen suorittamista lomittain toiminnan näin nopeutuessa. Kyseisellä tekniikalla ei kuitenkaan päästä grafiikkalaitteiston nopeuteen.

8.5 Kaksiulotteinen grafiikka 3D-laitteistolla

Useimmiten ohjelmoija haluaa pystyä käyttämään laitteistokiihdytettyä grafiikan piirtoa. Uudet laitteet on kuitenkin useimmiten tarkoitettu kolmiulotteisen grafiikan piirtoon, mikä tarkoittaa että ne piirtävät teksturoituja kolmioita. Tätä toiminnallisuutta pystyy käyttäämään spritejen piirtoon, mutta tekstuurikoordinaatit täytyy määritellä liukulukuina ja varoa pyöristysvirheitä. Laitteiston tarjoamaa piirtoa voidaan käyttää laitteistokiihdytettyihin sprite-efekteihin, kuten skaalaukseen, rotaatioon ja läpinäkyvyyteen.

8.6 Vieritys

Veritystä käyttävissä peleissä vieritys voidaan toteuttaa laitteistolla tai ohjelmallisesti. Ohjelmallisesti piirtäessä määritellään kamerakoordinaatit, jotka ilmaisevat kameran sijainnin. Piirrettävien objektien koordinaateista vähennetään kamerakoordinaatit, jolloin objektit kuvautuvat näytölle kameran mukaan. Kuvan ulkopuolelle jäävät objektit ja maasto jätetään luonnollisesti piirtämättä.

Kolmiulotteisen vaikutelman aikaansaamiseksi kaksiulotteisissa peleissä usein käytetään eri nopeudella vieriviä kerroksia, jotka ovat joko maaston edessä tai takana. Kerroksen vierimisnopeuden suhde maastokerrokseen saadaan sen etäisyydestä. Mitä syvemmällä kerros on, sitä hitaammin se vierii. Usein edessä olevat kerrokset peittävät osia taaempana olevista kerroksista, mikä tarkoittaa että ilman optimointia suoritetaan turhaa piirtämistä. Pahimmassa tapauksessa n kerrosta käytettäessä piirretään n-kertainen määrä vaadittuun määrään verrattuna. Kerroksien muodostuessa tiilikartoista on yksinkertaista selvittää mitkä edessä olevan tiilikartan tiilet osuvat taaempana olevan tiilikartan minkäkin tiilen päälle. Jos eri kerrosten tiilikartat koostuvat saman kokoisista suorakulmion muotoisista tiilistä, tarvitsee yhtä tiiltä kohti testata karttojen keskenäisestä vaiheesta riippuen 1-4 edessä olevan tiilikartan tiiltä. Mikäli ne peittävät taakse jäävän tiilen kokonaan, voidaan tiili jättää piirtämättä. Se peittääkö tiili taustan riippuu siitä, ovatko tiiltä edustavassa kuvassa kaikki pikselit piirrettäviä vai onko mukana kokonaan tai osittain läpinäkyviä pikseleitä. Tämä joko analysoidaan alustuksen aikana ohjelmallisesti tai ilmoitetaan erikseen. Asiaa ei siis tarvitse selvittää itse piirtämisen aikana, mikä tekisi menetelmästä täysin käyttökelvottoman, vaan päätös taaemman tiilen piirtämisestä tehdään lukemalla edessä olevien tiilten arvot ja tarkastamalla taulukosta näiden peittävyys.

9. ÄÄNET JA MUSIIKKI

Videopeleissä äänimaailmalla on ehkä jopa graafista ilmettä suurempi merkitys, sillä musiikki vaikuttaa vahvasti ihmisen tunnetiloihin. Pelin äänimaailman merkitys ja toteutus ovat muuttuneet suorittimien nopeutuessa, laitteiston kehittyessä ja tallennuskapasiteetin kasvaessa. Ensimmäiset pelit olivat äänettömiä, jonka jälkeen alettiin hyödyntää äänigeneraattoreita, siirtyen tästä primitiiviseen ja myöhemmin kehittyneempään synteesiin. Tämän jälkeen siirryttin käyttämään nauhoitettuja ääniä, myöhemmin kokonaan nauhoitettuja kappaleita, siirryttäen interaktiivisuuden aikaansaamiseksi vähemmän ennaltamäärättyihin ratkaisuihin.

9.1 Digitaalinen ääni

Ääni on ilmanpaineen muutoksen värähtelyä, joka tapahtuu kuultavalla taajuusalueella. Sitä voi ajatella aaltona, jossa amplitudi merkitsee voimakkuutta ja taajuus korkeutta. Digitaalisella äänellä tarkoitetaan numeromuodossa muistiin näytteistettyä jännitettä joka vahvistettuna liikuttaa ilmaa liikuttavia kaiuttimia. Näytteistys tapahtuu AD-muuntimella ja muunnos analogiseksi tapahtuu DA-muuntimella. Näytteistettävän signaalin jännite on jatkuva ajan funktio. Jännitteestä poimitaan näytteenottotaajuuden määrämän ajan välein näytearvoja, jotka kvantisoidaan käytetyn lukutarkkuuden mukaan. Jotta alkuperäisen jännitteen sisältämät taajuudet pystyttäisiin rakentamaan uudelleen, tulee Nyquistin näytteenottoteoreeman mukaan näytteenottotaajuuden olla vähintään kaksinkertainen jännitteen suurimpaan taajuuskomponenttiin verrattuna [16]. Ihmisen kuuloalueen yläraja on noin 20 kilohertsiä, joten kaikkien kuultavien taajuuksien näytteistämiseksi näytteenottotaajuuden on oltava noin 40 kilohertsiä. Myös näytettä kohti käytettävällä bittimäärällä on vaikutus äänen laatuun, sillä bittimäärän laskiessa esitystarkkuus pienenee ja kvantisointivirheestä aiheutuva näytteistetyssä signaalissa oleva kohina kasvaa. Audio-CD-levyissä on päädytty 44.1 kHz näytteenottotaajuuteen, 16 bitin eli 2 tavun kokoisiin näytteisiin sekä 2-kanavaiseen stereoääneen. Täten sekunti CD-laatuista stereoääntä vie muistia $44.1kHz * 1s * 2 * 2$ tavua eli 176400 tavua, joka on huomattava osuus tallennusmedian sekä varsinkin keskusmuistin kapasiteetista, minkä vuoksi CD-laatuiseen nauhoitettuun musiikkiin ei vähemmän yllättäen pystytty videopeleissä siirtymään ennen CD-ROM-aikakautta.

9.2 Historiaa

Ensimmäisissä ääntä hyödyntävissä peleissä toteutus oli äänigeneraattori joka ei ollut ohjelmoitava, eiväthän videopelitkään olleet aina mikroprosessorilla suoritettavia ohjelmia. Konekoodilla ohjelmoitujen pelien myötä pelilaitteisiin sisällytettiin ohjelmoitava äänigeneraattori, joka yleensä pystyi tuottamaan muutamaa erilaista ääntä samanaikaisesti. Generaattorin äänilähteenä on yleensä oskillaattori, joka värähtelee tuottaen useimmiten jotain muutamasta perusaaltomuodosta valittavaa aaltoa halutulla taajuudella. Perusaaltomuodoilla tarkoitetaan helposti generoitavia aaltoja, kuten pulssi-, kolmio-, saha- ja siniaalto, joista kaikki paitsi siniaalto sisältävät harmonisia ylätaajuuksia. Siniaalto on kaikkien aaltojen peruselementti, jonka eri taajuuksisia, -vaiheisisia ja -voimakkuuksisia sarjoja summaamalla voidaan teoriassa muodostaa mikä tahansa ääniaalto. Usein generaattorin äänilähteenä oli käytössä myös kohina. Oskillaattoreiden generoimien äänten voimakkuutta säädeltiin ADRS-verhokäyrällä. Näin saatiin aikaan yksinkertaisia ääniefektejä, melodioita ja bassolinjoja. Kuitenkin erilaisilla keinoilla voitiin näilläkin piireillä toistaa näytteistettyjä ääniä vaihtelevalla menestyksellä. MOS Technologyn SID-piiri (Sound Interface Device) oli tällaisista äänipiireistä kehittynein. Se tarjosi käyttöön suodattimen jolla sointiväriä pystyttiin muokkaamaan perusaaltomuodoista poikkeavaksi sekä rengasmodulaation, jolla saatiin aikaan kellomaisia ja metallisia ääniä. Myöhemmissä laitteissa alettiin hyödyntää taajuusmodulaatiosynteesiä (englanniksi Frequency Modulation), äänen taajuuden moduloimista toisella signaalilla. Sointiväriä muokkaamalla saatiin jossain määrin paremmin jäljiteltyä oikean soittimen sointiväriä.

Tallennuskapasiteetin kasvaessa siirryttiin synteesistä ääninäytteiden käyttöön ääniefekteissä ja musiikissa. Synteesin ja ääninäytteiden välimuoto on myös videopeleissä käytettävä wavetable-synteesi, jossa äänilähteenä käytetään perusaaltomuotojen sijasta nauhoitettua ääninäytettä. CD-ROM-teknologian myötä pelien tallennusmedioiden muistikapasiteetti kasvoi huomattavasti, ja kokonaisia valmiiksi nauhoitettuja kappaleita pystyttiin käyttämään pelien musiikkina. Tästä on sittemmin siirrytty silmukkapohjaisiin ratkaisuihin, jotka tarjoavat suurempaa joustavuutta musiikin hallintaan pelin aikana.

9.3 Äänen soittaminen

Äänijärjestelmää toteuttaessa on mahdollista käyttää valmiiksi tarjolla olevia ratkaisuja. Usein nämä helpottavat työtä huomattavasti, mutta voivat joskus olla riittämättömiä tarkoitukseen. Ohjelmoijan toteuttaessa itse oman äänijärjestelmänsä tulee ottaa huomioon muutamia seikkoja. Soitettava ääni on kirjoitettava äänikortin puskuriin ennen soittamista. Puskurin muisti vastaa usein alle sekuntia soitettavaa ääntä. Puskuriin on siis kirjoitettava usein ja vieläpä ajoissa, jottei soitettavaan ää-

neen tule tyhjiä kohtia jotka aiheuttavat pätkimistä, joka voi häiritä huomattavasti enemmän kuin tahmea ruudunpäivitysnopeus. Puskurin täyttävän rutiinin täytyy siis olla tarpeeksi nopea, ja ääntä on tuotettava vähintään samalla tahdilla kuin sitä toistetaan. Usein rutiini asetetaan kutsuttavaksi niin sanottuna callback-funktiona, jota kutsutaan kun puskurin täyttäminen on ajankohtaista. Mitä lyhyempi puskuri, sitä lyhyemmässä ajassa pelissä soitettavat ääniefektit voidaan soittaa äänten tapahtumisesta pelissä, mutta sitä helpommin voidaan viivästyä puskurin täyttämisestä.

9.3.1 Uudelleennäytteistys

Ääninäytteen soittaminen alkuperäisestä poikkeavalla taajuudella onnistuu äänikortin soittotaajuutta säätämällä, mutta taajuutta ei välttämättä pysty asettamaan halutulla tarkkuudella eikä ratkaisu useamman eri taajudella soitettavan äänen tapauksessa toimi laitteen sisältäessä vain yhden soittotaajuuden. Ainoa ratkaisu on käyttää uudelleennäytteistystä, jossa alkuperäinen ääninäyte kirjoitetaan äänipuskuriin eri pituisena. Äänen hidastamiseksi se tulee kirjoittaa venytettynä ja nopeuttamiseksi se tulee kirjoittaa tiivistettynä.

Uudelleennäytteistys voi olla toteutettu laitteistotasolla, mutta joskus se joudutaan toteuttamaan ohjelmallisesti. Kuten grafiikassa, voidaan uudelleennäytteistyksessä käyttää Bresenhamin algoritmia kertolaskujen poistamiseksi. Voidaan kuitenkin haluta käyttää kehittyneempää metodia, joka käyttää suodatusta.

9.3.2 Kromaattinen sävelasteikko

Länsimaisessa musiikissa käytettävässä kromaattisessa sävelasteikossa oktaavi, väli jossa sävelten taajuus kaksinkertaistuu on jaettu kahteentoista osaan, joita kutsutaan puolisävelaskeliksi. Jokaisen sävelen ja sitä puolisävelaskeleen ylempänä olevan sävelen taajuuksien suhde on vakio. Kun tiedetään että tämä vakio potenssiin kaksitoista on kaksi, voidaan päätellä että vakio on luvun kaksi kahdestoista juuri. Tämän suhteen avulla voidaan laskea annetun alkusävelen taajuuden perusteella muiden asteikon sävelten taajuudet, joita käyttämällä voidaan seuraavaksi käsiteltävän uudelleennäytteistyksen avulla soittaa melodioita.

9.3.3 Miksaus

Miksauksella tarkoitetaan useamman äänen yhdistämistä yhdeksi ääniaalloksi. Joissain äänikorteissa on useita kanavia, ja miksaus tapahtuu laitteiston sisällä. Usein kuitenkin kanavia on vain kaksi, yksi kummallekin stereojärjestelmän kaiuttimelle

tai muuten liian vähän, jolloin ohjelmoijan on toteutettava miksaus ohjelmallisesti. Onneksi miksaus on periaatteeltaan hyvin yksinkertaista, ääniaaltojen summaamista. Äänipuskurin lukuformaatti kannattaa asettaa etumerkilliseksi sekä näytteet tallentaa samassa formaatissa. Tällöin ääniaallon huiput ovat joko positiivisia tai negatiivisia, jolloin yhteenlasku tuottaa oikean tuloksen ilman tarvetta välivaiheille.

Miksauksessa on huomioitava äänipuskurin arvoalue. Summatessa ei saa tapahtua ylivuotoja, muuten ääni kuulostaa vääristyneeltä. Äänet on tapana näytteistää siten, että ne käyttävät mahdollisimman suurta osaa arvoalueesta parhaan laadun saamiseksi. Näytteiden arvoalue vastaa myös usein äänipuskurin arvoaluetta. Tällöini kahden äänen summaaminen voi tuottaa arvoalueen maksimiin verrattuna pahimmillaan kaksinkertaisen arvon. Ongelma ratkeaa kertomalla ääniaallot ennen summaamista kertoimilla jotka ovat pienempiä kuin 1. Jos halutaan varmistua ettei ylivuotoja tapahdu, tulee yleisessä tapauksessa äänten kertoimien summan olla korkeintaan 1. Kuitenkin lopullisen miksatun äänen maksimikohta on riippuvainen summattavien äänten vaiheesta ja amplitudista. Sääntöä ei tarvitse aina noudattaa, vaan esimerkiksi musiikin miksaamisessa tulee tietää kappaleen amplitudin huippukohta ja säätää kertoimia sen mukaan.

9.3.4 Sekvensoitu ja nauhoitettu musiikki

Pelin musiikki voidaan toteuttaa sekvensoimalla, missä kappaleen tapahtumat tallennetaan nuottitasolla sekvensseriin, joka on joko fyysinen laite, tietokoneohjelma tai sen osa. Sekvensoinnista on käytetty hieman harhaanjohtavasti ainakin rumpurytmien luomiseen liittyen termiä ohjelmointi, vaikka se tapahtuisikin soittamalla rytmi reaaliajassa sekvensseriin. Tämä johtaa juurensa vanhimpiin ohjelmoitaviin rumpukoneisiin, joissa käyttäjä asetti rytmin nappeja painamalla kuudestoistaosanuotin tarkkuudella. Toisaalta aina digitaalista tekniikkaa käytettäessä ajoitukseltaan tarkinkin sekvensseri kvantisoi nuotin aika- ja kaikki muut mahdolliset arvot käytettävän lukutarkkuuden mukaiseksi, joten termi on siinä mielessä asianmukainen.

Varhaisen videopelimusiikin voi sanoa olevan sekvensoitua, sillä rytmi ja mahdollinen melodia on sijoitettu sille varatulle muistialueelle, vaikka se onkin kovakoodattu peliohjelman konekoodiin ilman erillisen sekvensserin käyttöä. Peliohjelma sisältää muistialuetta lukevan ja äänten soittamisesta huolehtivan rutiinin, jota voidaan vaihtoehtoisesti kutsua keskeytysten avulla tai vaikkapa pelisilmukan tietyssä vaiheessa. Varhaisissa videopeleissä ruudun prosessointiin käytettävä aika oli lukittu näyttölaitteen elektronitykin pystysuuntaiseen ruudun pyyhkäisyyn, jolloin rutiinia kutsuttiin lähes tasavälein. Myös pelin nopeus ja musiikin tempo olivat lineaarisesti riippuvaisia näyttölaitteen kuvan päivitysnopeudesta, mistä johtuen sama videopeli toimi eri nopeuksilla eri TV-standardeja käyttävillä laitteiston versioilla. Nykype-

lit toimivat laitteistolla jossa on äänipuskuri, johon soitettavan äänen aaltomuoto kirjoitetaan ja jonka sisällölle äänipiiri tekee DA-muunnoksen, eli muuttaa sen digitaalisista ääninäytteistä jatkuva-aikaiseksi jännitteeksi. Äänipuskuria käyttävissä laitteissa voidaan soitettavat äänet ajoittaa yksittäisen ääninäytteen tarkkuudella, esim. CD-standardia vastaavassa äänessä 1/44.1 kHz eli noin 23 mikrosekunnin tarkkuudella, kun taas äänen laukaisun kytkeminen kerran peliruudussa tapahtuvaan rutiiniin jonka kutsumista ennen saatetaan pelisilmukan vaiheiden järjestyksestä riippuen suorittaa vaihteleva määrä prosessointia mahdollisesti suhteellisen hitaalla prosessorilla voi aiheuttaa musiikin rytmin jonkinasteista huojuntaa.

Nauhoitettu ääni tarkoittaa itse ulkoisesta lähteestä saatavan äänen tallentamista, esimerkiksi analogisessa tapauksessa magneettinauhalle tai vinyylilevylle ja digitaalisessa tapauksessa laserlevylle, kiintolevylle ja videopeliä lopulta ajettaessa tietokoneen keskusmuistiin tai mahdollisesti äänikortin omaan muistiin. Etuna nauhoitetuissa synteesiin verrattuna on mahdollisuus nauhoittaa oikeita soittimia, jolloin saadaan oikea sointiväri. Nauhoitettu ääni voi olla esimerkiksi jollain soittimella soitettu yksittäinen nuotti, jolloin vain ääntä toistuvasti eri taajuuksille uudelleennäytteistämällä voidaan muodostaa melodioita. Haittapuolena synteesiin verrattuna on, ettei yksittäistä ääninäytettä käyttäen pystytä jäljittelemään useimpien soittimien soittoa realistisesti.

Nauhoitettu musiikki taas on yksinkertaisesti musikaalisia elementtejä kuten rytmiä ja/tai melodiaa sisältävää nauhoitettua ääntä. Digitaaliset musiikkikappaleet vievät pakkaamattomassa muodossa yleensä laadusta ja pituudesta riippuen useita kymmeniä megatavuja, joten niitä ei yleensä säilytetä kokonaisina keskusmuistissa eikä varsinkaan äänikortin muistissa, vaan käytetään niin sanottua virtausta (streaming), joko suoraan optisesta asemasta äänikortille tai lukemalla tiedostoa lyhyissä osissa keskusmuistiin ja kirjoittamalla äänikortin puskuriin.

Pelin äänitoteutusta suunitellessa tulee miettiä, missä vaiheessa luoda ääntä ajonaikaisesti ja missä vaiheessa käyttää nauhoitettua ääntä. Ääniefektit, musiikissa käytettävät soittimet sekä itse musiikki voivat olla ajonaikaisesti luotavia tai nauhoitettuja. Esimerkiksi itse musiikki voi olla sekvensoitua, jolloin lopullinen miksattu ääniaalto luodaan ohjelman ajon aikana, kuitenkin samalla käyttäen soittimissa nauhoitettuja ääninäytteitä. Synteesillä tai näytteistämisellä ei välttämättä saada tyydyttävästi jäljiteltyä kaikkia akustisia tai sähköisesti vahvistettuja soittimia, osittain näytteiden joustamattomuuden sekä realistisen äänisynteesin vaativan laskentatehon ja myös suurelta osin koska muusikoiden yleisimpänä kontrollerina käyttämät koskettimistot ovat kykenemättömiä ilmaisemaan hyvin muiden kuin koskettimistoihin pohjautuvien instrumenttien soittoa sekä jo 1980-luvulla kehitetyn digitaalisten musiikkilaitteiden väliseen kommunikointiin käytetyn MIDI-standardin hitaan tiedonsiirtonopeuden vuoksi. Tällöin saatetaan käyttää nauhoitettuja silmu-

koita tai musikaalisia fraaseja tiettyjen soittimien kohdalla. Kummassakin ratkaisussa on etunsa ja haittansa: näytteistetty ääni vie tilaa muistista eikä ole joustavasti muokattavissa, kun taas äänisynteesi ja myös melodioiden soittaminen ääninäytteitä uudelleennäytteistämällä kuluttavat laskentatehoa. Tällä hetkellä useimmat korkean profiilin pelit soittavat musiikkia pohjautuen vaihtelevan mittaisiin nauhoitettuihin ääninäytteisiin, jolloin musiikki voi olla rajoitetussa määrin interaktiivista kuitenkaan viemättä liikaa laitteistotehoa. Kenties tulevaisuudessa laskentatehoa annetaan enemmän musiikin ja äänten soittamista varten, jolloin musiikki voidaan luoda kokonaan ajonaikaisesti nuottipohjaisesti, jolloin interaktiivisuus saadaan maksimoitua.

10. INPUT

Videopelien alkuperäiseen määritelmään kuuluu visuaalisen vasteen lisäksi käyttäjän antama syöte ja tähän reagointi. Tässä luvussa käsitellään erilaisia input-laitteita ja syötteen käsittelyä.

10.1 Input-laitteet

Input-laitteita on ollut videopelien historiassa monia erilaisia, joidenkin unohduttua kun taas muutamat ratkaisut ovat säilyttäneet suosionsa. Viime vuosina videopelivalmistajat ovat keskittyneet liiketunnistukseen ja kosketusnäyttöihin.

Näppäimistö. Yleiskäyttöisillä tietokoneilla pelattavissa peleissä alunperin mekaaniseen kirjoituskoneeseen perustuva näppäimistö on hiiren ohella yksi suosituimmista ratkaisuista, sillä jopa nykyaikaiset graafiset käyttöjärjestelmät ovat käytännössä mahdottomia käyttää tehokkaasti ilman tekstin syöttöön soveltuvaa laitetta. Näppäimistö koostuu useimmiten reilusta sadasta näppäimestä, joiden pohjaan painamisen ja vapauttamisen sen sisäinen elektroniikka havaitsee ja lähettää siitä tiedon keskusyksikölle.

Hiiri. Hiiri on yleiskäyttöisissä tietokoneissa graafisten käyttöliittymien myötä yleistynyt input-laite, joka on myöhemmin saavuttanut suosiota myös videopeleissä ja korvannut käytännössä näppäimistön täydellisesti kameran kääntämisessä ensimmäisen persoonan peleissä. Hiiressä on painike, yleensä useampia, mahdollisesti kierrettävä rullla joka toimii myös painikkeena sekä sen alustalla liikuttamisen havaitseva elektroniikka. Hiiri lähettää keskusyksikölle tiedon painikkeen painamisen (klikkaaminen) ja vapauttamisen lisäksi sen liikuttamisesta alustalla. Liikedataan kuuluvat x- ja y-koordinaatit, jotka ilmaisevat hiiren sijainnin muutosta, ei absoluuttista sijaintia.

Joystick. Joystick on alunperin hissien ja ilmailussa siivekkeiden ohjaamiseen käytetty laite. Sähköisen 2-akselisen joystickin keksi ja patentoi vuonna 1926 C. B. Mirick Yhdysvaltain laivaston tutkimuslaboratoriossa laitteiden kauko-ohjaamiseen [17]. Videopeleissä ensimmäiset joystickit olivat alkuperäisen sähköisen joystickin lailla analogisia, joissa tikkua vääntämällä ohjataan pelihahmoa haluttuun suuntaan. Joystickissa on myös usein nappeja joita käytetään esimerkiksi hyppäämiseen tai tulittamiseen. Jossain vaiheessa joystickeista tuli analogisten laitteiden sijaan päällä/pois-kytkimiä käyttäviä ohjauslaitteita. Nämä olivat sen aikaisiin videopelei-

hin tarkkuutensa vuoksi paremmin soveltuvia laitteita, mutta myöhemmin yleistyvissä lentosimulaattoreissa paras ohjausväline oli analoginen joystick.

Gamepad. Gamepadit ovat molemmissa käsissä pidettäviä ohjaimia, joissa on sekä ristiohjain että eri tarkoituksiin käytettäviä painikkeita. Painikkeet ovat perinteisesti olleet joko digitaalisia kytkimiä jotka ovat joko painettuja tai vapautettuja. Myöhemmin gamepadeissa on alettu käyttää analogisia painikkeita sekä peukaloilla käytettäviä analogisia joystickin tapaisia 'tatteja'. Konsoleilla gamepad on tasoloikkiin parhaiten soveltuva ohjainlaite, mutta yleiskäyttöisille tietokoneille valmistetut gamepadit eivät valitettavasti ole käytännössä kelvollisia.

Liikeohjaus. Liiketunnistus perustuu siihen, että käyttäjän ohjain lähettää signaalia esim. näyttölaitteen lähellä olevalle sensorille, jolloin selviää ohjaimen etäisyys sensorista. Jotta ohjaimen tarkempi sijainti saataisiin selville, täytyy sensorissa olla useampi piste joka on yhteydessä ohjaimeen. Kun ohjaimen sijainti saadaan selville tarpeeksi pienin väliajoin, voidaan ohjelmallisesti selvittää pelaajan ohjaimella tekemät liikkeet. Liiketunnistuksella voidaan esimerkiksi simuloida aseella tähtäämistä tai tunnistaa nopeat huitaisut, jolloin pelihahmo voi esimerkiksi lyödä miekalla.

Kosketusnäyttö. Kosketusnäyttö on varsinkin mobiililaitteissa yleistynyt ohjausrajapinta. Näyttö tunnistaa ilmeisesti siihen kohdistuvan paineen perusteella kosketuksen, joka voidaan tulkita esimerkiksi kursorin liikuttamiseksi tai napin painallukseksi. Kosketusnäytössä pelihahmon liikutus perustuu niin sanottuun viivojen vetämiseen, joka on esimerkiksi myös Angry Birds-mobiilipelin ohjauksen perusta. Kosketusnäytöllä ohjaus toimii jossain määrin ylhäältä kuvatuissa peleissä, mutta sivusta kuvattuun tasoloikkapelin pelaamiseen se ei välttämättä sovellu. Myöskään vaativa tietojenkäsittely ei luonnistu kosketusnäyttöä käyttämällä.

10.2 Syötteen lukeminen ja käsittely

Input-laitteilta saatavaa tiladataa tulee käsitellä eri tavoin laitekohtaisesti. Käsittelyyn vaikuttaa myös, onko kyseessä laitteen tilan muutosta koskevaa dataa vai laitteen kokonaistilan ilmaisevaa tietorakennetta. Koska reaaliaikaiset videopelit prosessoivat logiikkaa ja piirtävät grafiikkaa usein ruudun kerrallaan pelisilmukassa, on luontevaa käsitellä input-dataa myös ruutukohtaisesti. Tällöin input-laitteilla ajatellaan olevan yhden peliruudun aikana muuttumaton tila, vaikkei näin todellisuudessa olekaan.

Käyttäjän muuttaessa input-laitteen tilaa, lähettää laite keskusyksikölle tilan muutoksesta kertovan viestin. Tästä aiheutuu laitteistokeskeytys, joka käsitellään laitteistosta, mahdollisesta käyttöjärjestelmästä tai käytettävästä kirjastosta riippuen suhteellisen pienellä viiveellä. Usein voidaan käyttää valmista kirjastoa tai käyttöjärjestelmän palveluita laitteen tilan lukemiseen, mutta vielä käyttöjärjestelmälle DOS toteutetuissa peleissä oman keskeytysrutiinin kirjoittaminen ei ollut

harvinaista.

Keskeytyskäsittelijässä yleensä laitteen tilaa ilmaiseva tietorakenne päivitetään. Laitteen tilaa voidaan kysellä tietorakenteesta periaatteessa koska vain, mutta jos halutaan laitteen tilan näkyvän ruudun aikana muuttumattomana, luetaan kerran ruudussa pelisilmukan määrätyssä vaiheessa laitteen tila tietorakenteeseen, jota ei muuteta enää myöhemmin saman ruudun aikana. Ruutukohtainen tila saadaan tätä tietorakennetta lukemalla. Luonteva vaihe ruutukohtaisen tilan lukemiselle on pelisilmukan alussa ennen pelilogiikkaa.

Näppäimistöltä tekstimerkkejä luettaessa täytyy tietää milloin näppäintä on painettu, ja haluttaessa käyttää automaattista merkintoistoa, pitää kirjaa myös näppäimen tilasta. Jotta voitaisiin selvittää, onko näppäin painettu pohjaan juuri halutulla hetkellä voidaan joko tarkastella onko kyseisellä ruudulla tullut keskeytyskäsittelijälle viesti näppäimen painamisesta tai vaihtoehtoisesti säilyttää tietoa näppäimistön edellisestä kokonaistilasta, ja tutkia onko näppäin pohjassa tällä hetkellä muttei edellisellä. Myös näppäimen vapauttaminen voidaan tutkia keskeytyskäsittelijälle lähetetyn vapautusviestin perusteella tai vastaavasti verrata kokonaistilaa edelliseen kokonaistilaan. Gamepadien, joystickien ja hiirten painikkeiden tilan tarkastelu tapahtuu samalla tavalla.

Jos laite lähettää viestinä jonkin parametrin erotuksen edelliseen vaiheeseen, kuten esimerkiksi hiiri lähettää tiedon sijaintinsa muutoksesta, tulee yksinkertaisesti summata erotus muistissa säilytettävään vanhaan koordinaattiin, mahdollisesti herkkyyden määrittävää kerrointa käyttäen.

Analogisten parametrien käsittely riippuu siitä, halutaanko ohjauksen olevan oikeasti analogista vai ei. Todellisesti analogisessa tapauksessa pelihahmon pitäisi liikkua tällaisen parametrin arvon mukaisella nopeudella, eli hahmo liikkuu hitaasti kun analogista säädintä väännetään kevyesti, ja säätimen ollessa ääriasennossa hahmo juoksee maksiminopeudella. Analogiselle ohjaimelle määritellään yleensä myös niin sanottu kuollut alue, joka vastaa säätimen vapauttamisesta huolimatta laitteen lähettämien arvojen aluetta. Ei-todellisesti-analogisessa tapauksessa analogista ohjainta halutaan käyttää digitaalisesti, jolloin säätimelle määritetään raja-arvo, jonka ulommalla puolella oleva asento vastaa digitaalisen säätimen painamista ja sisemmällä puolella vapauttamista. Tällainen ratkaisu ei ole kovinkaan tarkka, vaan ohjauksen ollessa digitaalista on ihanteellista käyttää myös digitaalista ohjainta, kuten ristiohjainta.

Laitteilta saatavaa syötettä voidaan abstrahoida. Koska samassa pelissä voi olla mahdollisuus käyttää erilaisia ohjainlaitteita ja laitteille voi olla erilaisia näppäinkonfiguraatiotia, kannattaa laitteiden tiladatan ja pelilogiikan välille toteuttaa abstraktiokerros, joka muuntaa laitteen tilan muutoksen pelihahmoa liikuttaviksi komennoiksi. Tällöin laitteiden tilaa tutkittaessa yhdistetään eri painikkeet ja sää-

timet konfiguraatiotietojen perusteella eri komentoihin. Pelilogiikka käsittelee pelihahmoa ohjatessaan eksplisiittisesti määriteltyjen painikkeiden ja säädinten tilojen mukaan komentoihin liitettyjä tilatietoja. Tutkitaan esimerkiksi, onko saatu yleinen hyppykäsky sen sijaan, että tutkittaisiin onko gamepadin a-nappia painettu. Näin pelilogiikan ohjelmakoodista tulee abstraktimpaa ja helpommin ylläpidettävää.

11. THE ADVENTURES OF NASSE

The Adventures of Nasse on kirjoittajan toteuttama tasoloikkapeli, jonka pohjalta tätä diplomityötä alettiin alunperin kirjoittaa. Työn jokaisessa luvussa on asioita, joita Nassen toteutuksessa on täytynyt ottaa huomioon.

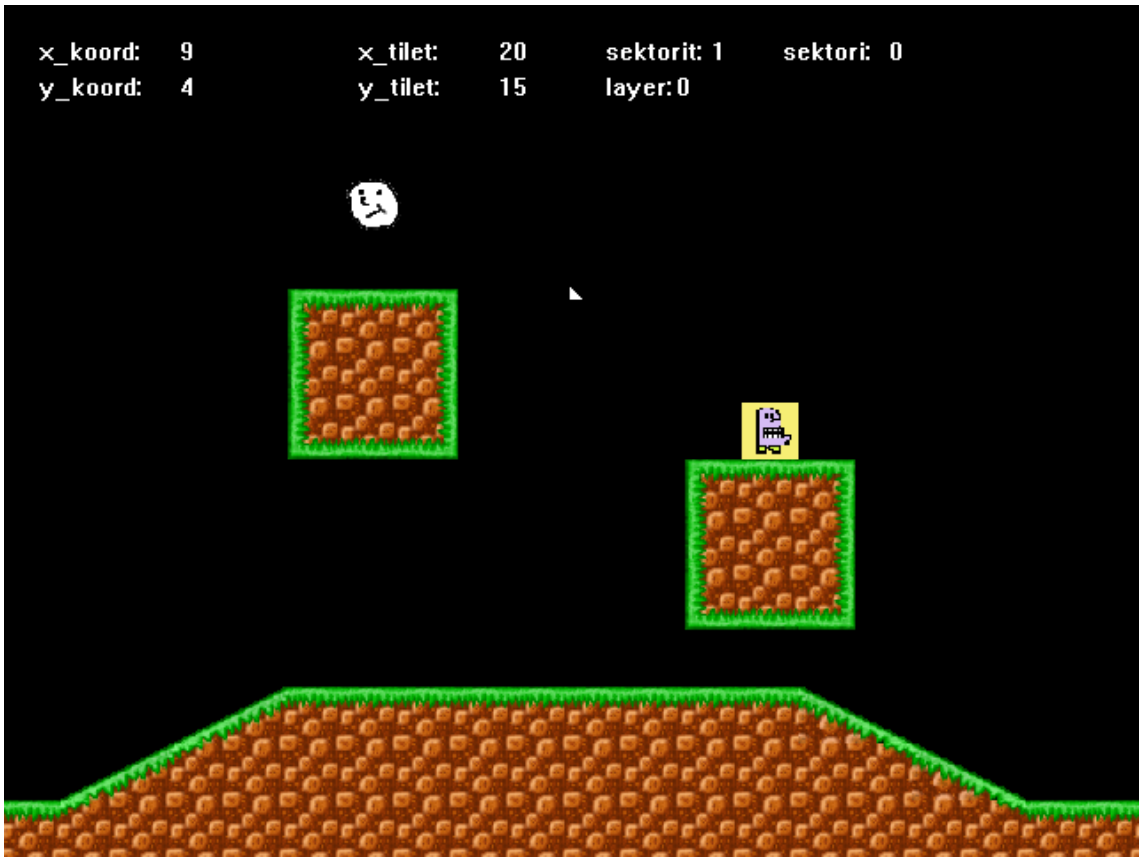
11.1 Suunnittelu

Nassessa maasto on kuvattu tiilikartalla, jossa on erilaisia palasia. Neliön muotoisten palasten lisäksi käytetään myös vinoja palasia, joista voi muodostaa erilaisia mäkiä, mikä tekee törmäystarkistuksen ohjelmoimisesta huomattavasti hankalampaa. Nasse toimii hyvinkin tieto-ohjatusti, ja kovakoodattu informaatio on pyritty minimoimaan ohjelmakoodissa. Suurin osa informaatiosta ladataan tietopankkiin pelin käynnistyessä konfiguraatiotiedostojen sisältöjen perusteella. Esimerkiksi jokaiselle objektityypille osataan liittää animaatio konfiguraatiotiedostojen perusteella.

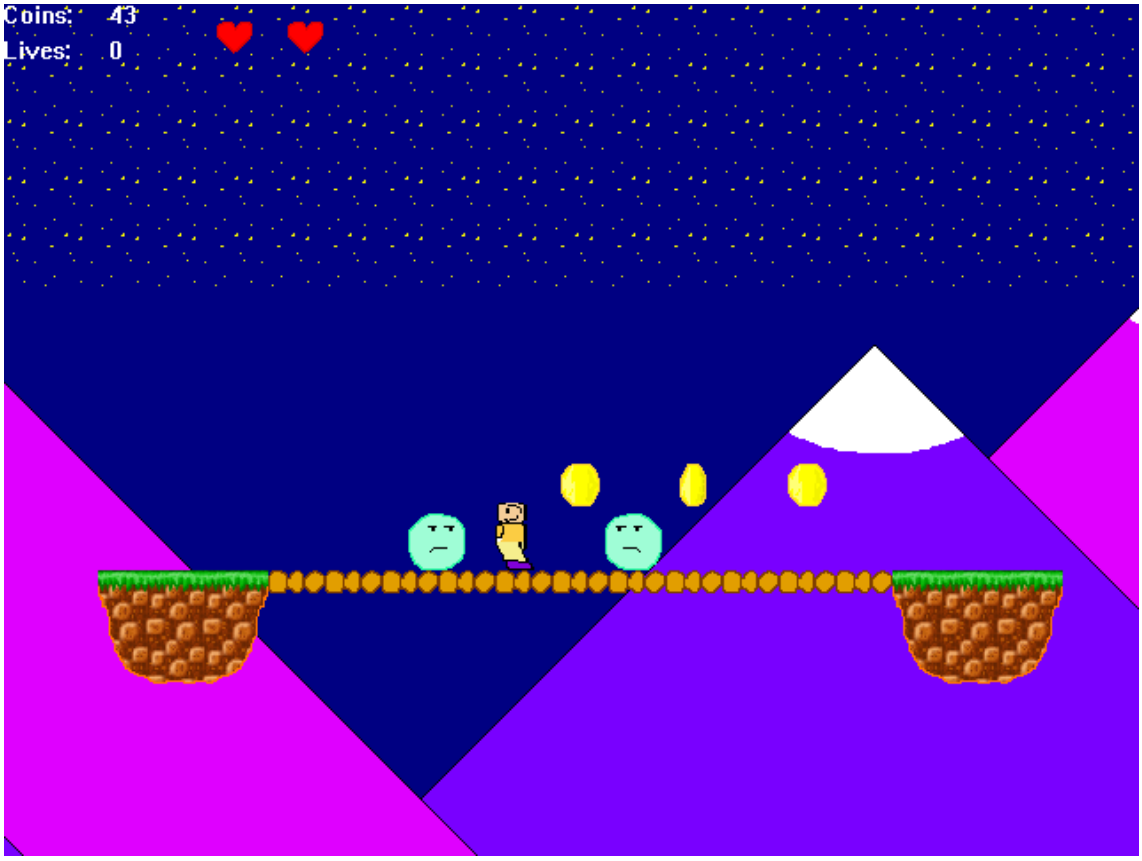
Olio-ohjelmointia on hyödynnetty siten, että jokainen pelihahmo edustaa kantaluokkaa 'peliobjekti' mutta eri aliluokille on omat toteutuksensa, jotka huolehtivat kunkin peliobjektityypin tekoälystä ja käyttäytymisestä. Eri tasoilla oleva ohjelmakoodi on pyritty erottamaan toisistaan esimerkiksi siten, että pelilogiikasta ja ulkoisista liitynnöistä huolehtivat koodinpätkät on erotettu toisistaan kääreen avulla. Kääre siis toimii rajapintana, jota pelikoodi käyttää, jottei pelilogiikassa tarvisi välittää laitteistotason yksityiskohdista.

11.2 Kenttäeditori

Nassea varten ei ole kirjoitettu kenttäeditorin lisäksi juuri muita työkaluja kuin muutama tiedostomuunnin. Kenttäeditori ei sisällä muita ominaisuuksia, kuin mitä ehdottomasti tarvitaan kenttien luomiseen. Tiilikarttaa voi editoida sekä vihollisia voi luoda, muokata ja poistaa, ja kentän voi ladata ja tallentaa. Kenttää voi myös testata suoraan kenttäeditorista käsin. Kenttäeditorissa on myös yritetty toteuttaa jossakin mielessä ikkunointia, mutta siinä ei noudateta mitään työssä esiteltyjä ikkunoinnin perusteita, ja lopputulos on karu.



Kuva 11.1: Kuva Nassen kenttäeditorista. Kuvassa näkyy tekstiinformaatiota kentästä, kentän tiilikartta, pelaajan alkupistettä merkkaava pään kuva sekä vihollinen.

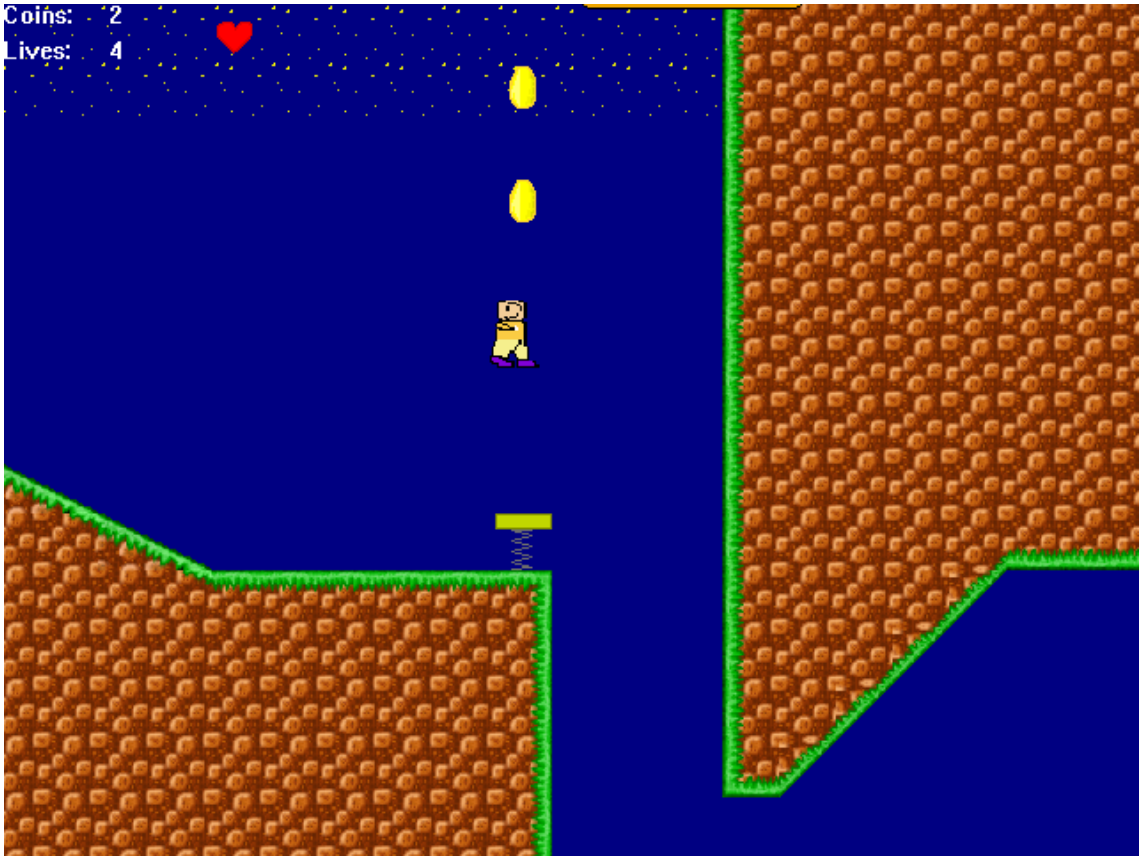


Kuva 11.2: Kuvassa kaksi vihollista seuraavat pelaajaa. Tämä toiminnallisuus on toteutettu peliobjektikohtaisessa logiikkafunktiossa.

11.3 Logiikka

Varsinaisen pelitoiminnan logiikka on jaettu Nassessa ylemmän tason logiikkaan ja peliobjektikohtaiseen logiikkaan. Ylemmän tason logiikassa huolehditaan peliobjektien käsittelystä kuten niiden aktivoitumisesta ja niiden siirtelemisestä eri tietorakenteiden välillä. Ylemmän tason logiikassa käydään myös läpi tietorakenteessa olevat peliobjektit ja kutsutaan jokaiselle sen tekoälyn toteuttavaa funktiota. Ylemmän tason logiikassa kutsutaan myös funktiota, jotka toteuttavat fysiikan mallinnuksen ja törmäystarkistuksen.

Jokaisella peliobjektilla on matalan tason toiminnallisuus, joka toteuttaa objektikohtaisen toiminnallisuuden. Tähän toiminnallisuuteen kuuluu esimerkiksi logiikkafunktio, joka määrittelee peliobjektin tekoälyn. Osa objektikohtaisesta toiminnallisuudesta on kaikille eri objektityypeille sama, mutta osa on toteutettu jokaiselle objektityypille erikseen omanlaisekseen. Esimerkiksi lähes jokaisella objektityypillä on oma tekoälyfunktio, mutta lähes kaikkien objektityyppien objektit piirretään käyttäen samaa piirtofunktiota.



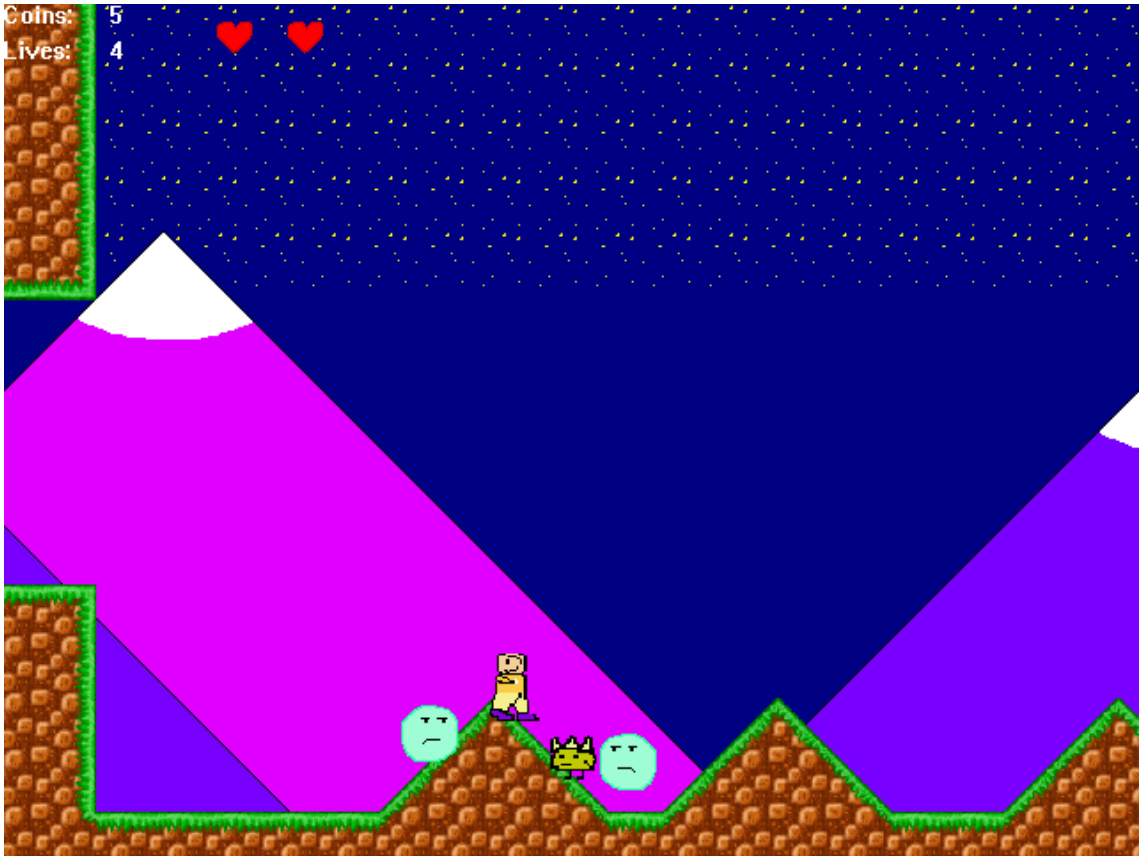
Kuva 11.3: Ponnahduslaudan päälle hyppääminen auttaa pelaajaa hyppäämään korkeammalle. Pelaajan y-suuntaista nopeutta muutetaan riippumatta tämän massasta.

11.4 Fysiikka

Nassen fysiikan mallinnuksessa on oikaistu huomattavasti työssä esitetystä tavasta. Ensinnäkään käytössä ei ole aikavakiota, joka lisättäisiin kertolaskuihin, vaan objekteilla on vain nopeudet, joiden mukaan niitä siirretään eteenpäin kerran joka animaatioruudussa. Pelin fysiikka on ohjelmoitu tietylle ruudunpäivitysnopeudelle ja aikavakion puutteen vuoksi ruudunpäivitysnopeuden muutos jälkikäteen muuttaisi suoraan pelin nopeutta. Luultavasti varhaiset tasoloikkapelit on ohjelmoitu samalla tavalla tietäen, että television kuva päivittyy 50 tai 60 kertaa sekunnissa, ja objektien siirtymävakiot on säädetty sen mukaan. Nassessa ei käytetä myöskään kaavaa, jossa kappaleen massa, siihen vaikuttavat voimat ja sen kiihtyvyys ovat suhteessa toisiinsa. Jos jokin voima vaikuttaa kappaleeseen Nassessa, muutetaan vain kappaleen nopeutta jollain tietyllä vakiolla.

11.5 Törmäystarkistus

Törmäystarkistus on toteutettu Nassessa posteriori-menetelmällä, jossa kappaleella on ensin alkusijainti, ja sille lasketaan fysiikan mallinnuksen perusteella uusi kohde-



Kuva 11.4: Törmäystarkistus pitää pelaajan mäen pinnalla.

sijainti, johon kappale pyrkii liikkumaan. Siirtymä jaetaan x- ja y-suuntaisiin vaiheisiin ja näissä testataan, onko kappale liikkunut jonkin maaastonpalasen sellaisen rajan yli, jota se ei saisi kyseisestä suunnasta lähestyessään ylittää. Laittoman ylityksen tapauksessa siirretään kappale takaisin rajalle.

Nassessa törmäystarkistusta tekevät monimutkaisemmaksi palaset, jotka voi ylittää vain tietystä suunnasta sekä vinot mäkipalaset, joita pitkin pelaaja voi liikkua.

11.6 Grafiikka

Nassessa grafiikka piirretään käyttäen SDL-kirjastoa, jonka tärkein palvelu tässä tapauksessa on spritejen piirtämiseen käytettävä funktio. Nassessa ei käytetä mitään muita erikoistehosteita spritejen piirrossa kuin alpha-blendingiä, jolla saadaan aikaan läpinäkyvät vesialueet. Nassessa on useimpien tasoloikkapelien tapaan vierivä grafiikka ja monta eri grafiikkakerrosta. Taaempien grafiikkakerrosten piiloon jäävät palaset osataan jättää piirtämättä, mikä nopeuttaa grafiikan piirtoa.

11.7 Äänet

Nassen äänimaailma on toteutettu käyttäen FMod-kirjastoa, jonka avulla voi soittaa äänitehosteita ja musiikkia. Musiikki on toteutettu tracker-ohjelmalla, jossa erilaisia ääninäytteitä käytetään soittimina. Soittamalla ääninäytteitä eri taajuuksilla saadaan aikaan melodioita. Kirjasto soittaa musiikin siten, että se soitetaan lennossa. Tämä tarkoittaa, että seuraavaksi soiva musiikin kohta kirjoitetaan äänipuskuriin valmiiksi hetkeä ennen sen soimista. Tracker-musiikin käyttäminen säästää huomattavasti levytilaa sekä muistia, mutta soittorutiini vie jonkin verran prosessoritehoa. Valmiin kirjaston käyttäminen musiikin soittoa varten säästää huomattavasti aikaa, sillä soittorutiinin kirjoittaminen on aikaa vievä tehtävä.

11.8 Input

Syötelaitteiden lukeminen on Nassessa toteutettu käyttäen SDL-kirjastoa, joka tarjoaa funktiot näppäimistön, hiiren ja gamepadin tilan lukemiselle. Jokaisen syötelaitteen tila luetaan kerran peliruudussa ja sitä käytetään koko ruudun ajan yksinkertaisuuden vuoksi. Syötelaitteet abstrahoidaan siten, että on olemassa tietorakenne, jossa kerrotaan mikä nappi milläkin laitteella vastaa mitäkin toimintoa. Esimerkiksi, jos halutaan tietää, onko pelaaja painanut hyppynappia, tarkistetaan asia abstraktion toteuttavasta tietorakenteesta, eikä testata onko näppäimistöstä painettu vaikkapa ctrl-näppäintä tai gamepadista B-nappia.

12. YHTEENVETO

Työn lukujen sisällön perusteella huomataan, ettei tasoloikkapelin toteuttaminen ole niin yksinkertaista kuin voisi aluksi ajatella. Jokaisessa eri osa-alueessa on omat haasteensa, ja ensimmäistä kertaa peliä toteuttavalle onkin paljon ongelmia ratkaistavaksi.

Pelin toteuttajalla olisi hyvä olla jonkinlaista kokemusta ohjelmien suunnittelusta, sillä ilman hyvää rakennetta ohjelman toteutuksessa tulee varmasti vastaan ongelmia, jos ohjelma on yhtään monimutkaisempi. Ohjelmoijan pitää myös pystyä päättämään, mitkä pelin toteuttamiseen tarvittavat työkalut hän ohjelmoi itse ja mihin asioihin löytyy jo valmis hyvä ratkaisu. Esimerkiksi kenttäeditoria toteuttaessa tulee miettiä, millainen käyttöliittymä riittää editorin käyttämiseen, sillä esimerkiksi ikkunointijärjestelmän toteuttaminen voi olla hyvinkin monimutkainen tehtävä. Pelin ydintoiminnasta huolehtiva logiikkakaan ei ole täysin itsestään selvä asia. Jonkinlainen järjestelmä täytyy myös toteuttaa, jotta peliobjektien fysiikan mallinnus toimisi oikein. Tasoloikkapelissä vaadittavat fysiikan lait eivät kuitenkaan onneksi ole kovin monimutkaisia. Törmäystarkistus voi toteuttamistavasta riippuen olla hyvin yksinkertainen tai erittäin monimutkainen. Monimutkaisuutta lisäävät erikoiset maaston tai objektien muodot sekä matemaattisen analyysin, eli priorin tarkistuksen käyttö. Törmäystarkistuksesta tulee myös tehdä erittäin varmasti toimiva, sillä peliobjektien ei haluta missään tapauksessa kulkevan kiellettyjen rajojen läpi. Myös pelin grafiikan ja äänimaailman toteutus sekä käyttäjältä tulevan syötteen lukeminen muodostavat omat haasteensa.

Voidaankin todeta, ettei tasoloikkapelin toteuttamiseen ole yhtä kaikkiin tapauksiin sopivaa menetelmää, vaan toteuttajan on monessa eri vaiheessa punnittava erilaisten vaihtoehtojen välillä ja valittava niistä omaan peliinsä sopiva. Pelin toteuttaminen ei tule hyvästäkään suunnittelusta huolimatta olemaan missään tapauksessa helppoa, sillä reaaliaikainen videopeli on usein suhteellisen monimutkainen järjestelmä, jossa on lukemattomia erilaisia yksityiskohtia jotka tulee ottaa huomioon. Tasoloikkapelin toteutus ei silti missään tapauksessa ole mahdoton tehtävä yhdellekään henkilölle, mutta vaatii työtä ja ymmärrystä monista eri osajärjestelmistä sekä hyvää päättely- ja ongelmanratkaisukykyä.

LÄHTEET

[1] Video game (Wikipedia), http://en.wikipedia.org/wiki/Video_game, Ladattu 6.10.2013

[2] Cathode ray tube amusement device (Wikipedia), http://en.wikipedia.org/wiki/Cathode_ray_tube_amusement_device, Ladattu 30.9.2013

[3] David Morris, Birth of the Video Game: Bouncing Ball to Tic-Tac-Toe, http://videogamestimeline.blogspot.fi/2012/08/bouncing-ball-to-tic-tac-toe.html, Ladattu 30.9.2013

[4] Tennis for Two (Wikipedia), http://en.wikipedia.org/wiki/Tennis_for_Two, Ladattu 30.9.2013

[5] Galaxy Game (Wikipedia), http://en.wikipedia.org/wiki/Galaxy_Game, Ladattu 30.9.2013

[6] Computer Space (Wikipedia), http://en.wikipedia.org/wiki/Computer_Space, Ladattu 30.9.2013

[7] Pong (Wikipedia), http://en.wikipedia.org/wiki/Pong, Ladattu 30.9.2013

[8] Essi Lahtinen, TTY:n Ohjelmointi 2 e-kurssin luentomateriaali, http://www.cs.tut.fi/~ohj2e/materiaali/pruju.pdf, Ladattu 12.11.2013, s.6

[9] Programming tool (Wikipedia), http://en.wikipedia.org/wiki/Programming_tool, Ladattu 14.11.2013

[10] Haikala ja Märijärvi, Ohjelmistotuotanto. Talentum (2006) s.137, 352-353, 365

[11] Computing platform (Wikipedia), http://en.wikipedia.org/wiki/Computing_platform, Ladattu 14.11.2013

[12] Adapter pattern (Wikipedia), http://en.wikipedia.org/wiki/Adapter_pattern, Ladattu 12.11.2013

[13] TTY:n Ohjelmointi 1 e-kurssin tehtävien yleisohje, http://www.cs.tut.fi/~ohj1e/tehtavienyleisohje/, Ladattu 13.5.2013

[14] Standard Template Library (Wikipedia), http://en.wikipedia.org/wiki/Standard_Template_Library, Ladattu 25.11.2013

[15] Adrian Nye, Volume One: Xlib Programming Manual, http://menehune.opt.wfu.edu/Kokua/Irix_6.5.21_doc_cd/usr/share/Insight/library/

SGI_bookshelves/SGI_Developer/books/XLib_PG/sgi_html/ch02.html, Ladattu 25.11.2013

[16] Nyquist-Shannon sampling theorem (Wikipedia), http://en.wikipedia.org/wiki/Nyquist-Shannon_sampling_theorem, Ladattu 6.10.2013

[17] Joystick (Wikipedia), http://en.wikipedia.org/wiki/Joystick, Ladattu 6.10.2013