Jevgeni Lensu

GameMaker-pelimoottorin rajoitteiden kiertäminen

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tietotekniikka

Opinnäytetyö

20.11.2017

Metropolia

Tekijä(t) Otsikko Sivumäärä Aika	Jevgeni Lensu GameMaker-pelimoottorin rajoitteiden kiertäminen 56 sivua + 1 liitettä 20.11.2017
Tutkinto	Insinööri (AMK)
Koulutusohjelma	Tietotekniikka
Suuntautumisvaihtoehto	Ohjelmistotuotanto
Ohjaaja(t)	lehtori Miikka Mäki-Uuro

Insinöörityön tavoitteena oli tutkia, miten paljon on mahdollista kiertää työkalun rajoitteita, jos tuntee kyseisen työkalun perusteellisesti ja katsoo ongelmia luovasti. Tässä tapauksessa työkaluna toimi GameMaker-pelimoottori, jolla kilpailijoihin verrattuna on hyvin mitättömät graafiset ominaisuudet. Graafisen vaatimattomuuden vastapainoksi GameMaker tarjoaa kuitenkin huippuluokan joustavuuden ja kehityksen ketteryyden. Tämä tekee siitä erittäin suositun etenkin pienten kehittäjien keskuudessa.

Työ lähestyy tilannetta vahvasti insinöörityöntekijän henkilökohtaisen kokemuksen kautta. Ensimmäinen luku on omistettu tietokonepelien historialle. Eniten tässä kohtaa keskitytään pelien varhaishistoriaan ja insinöörityöntekijän omiin kokemuksiin pelienkehityksen parissa. Tämä on tehty havainnollistamaan, miten alusta alkaen tietokonepeleissä oli kyse siitä, miten työkaluja luovasti sovellettiin siihen, mihin ne eivät olleet tarkoitettu.

Tämän jälkeen tutustutaan GameMaker-pelimoottorin taustoihin, vahvuuksiin ja heikkouksiin, sekä sen hyödyntämään GML-skriptikieleen. Lisäksi työssä nopeasti esitellään muut työkalut, joiden avulla oli mahdollista luoda grafiikka, joka ei olisi muuten mahdollista GameMaker-ympäristössä.

Seuraava luku käy perusteellisesti läpi kaikki ne tekniset ratkaisut ja innovaatiot, joita työssä on hyödynnetty rajoitteiden kiertämiseksi. Luvun alussa tutustutaan konsepteihin, jotka ovat ominaisia valtaosalle työn ratkaisuista, ja lopuksi käydään läpi, miten näitä konsepteja on sovellettu käytännössä.

Viimeisessä varsinaisessa luvussa tutustumme yksinkertaiseen panssarivaunupelin demoon, jossa insinöörityöntekijä on yrittänyt yhdistää kaikki keksimänsä ratkaisut kokonaisuudeksi.

Tuloksen perusteella voitiin päätellä, että GameMaker on edelleen varteenotettava pelinkehitysalusta. Insinöörityöntekijä otti tavoitteekseen viimeistellä demon julkaisukelpoiseksi peliksi.

Avainsanat	2D, 3D, pelimoottori, GameMaker, valaistus

Metropolia

Author(s) Title Number of Pages Date	Jevgeni Lensu Circumventing Constraints of GameMaker Game Engine 56 pages + 1 appendices 20 November 2017
Degree	Bachelor of Engineering
Degree Programme	Information and Communication Technology
Specialisation option	Software Engineering
Instructor(s)	Miikka Mäki-Uuro, Senior Lecturer

The goal of this study was to examine the possibilities of circumventing the constraints of a tool with a creative outlook and deep knowledge of the tool's working principles. As an example, this study the uses GameMaker game engine which is equipped with rather humble graphical capabilities when compared to many other game engines. On the other hand, GameMaker offers unparalleled flexibility and agility. This makes it very popular among small indie-developers.

The study strongly examines the case through the personal experience of the author. The study starts by introducing the history of video games. For the most part, it concentrates on the early history of the game industry, and the author's personal experience in the industry. This is meant to illustrate how from the very beginning game making was always about using unsuited tools in a creative way.

The study also presents a short history of GameMaker, its strengths and weaknesses, and the GML-scripting language which it uses. Also, the study briefly introduces other tools that made it possible to create graphics that would be impossible with only GameMaker.

Next the study concentrates on all technical solutions and innovations used to surpass limitations, e.g. some universal concepts used in most of the study's solutions and how those solutions were implemented in practice.

Lastly, the study introduces a simple tank game demo which combines most of the solutions discussed into one.

Based on the results of the study it can be concluded that GameMaker is still a noteworthy game engine. The author decided to try to continue developing the demo into a complete game.

Keywords	2D, 3D, game, engine, GameMaker, lighting

Sisällys

1 Johdanto 1

2 GameMaker: Studio 1.4 -pelimoottori 7

2.1 Lyhyt esittely 7
2.2 GameMaker: Studio 1.4 -moottorin vahvuudet 10
2.3 GameMaker: Studio 1.4 -moottorin heikkoudet 11
2.4 GML-skriptikieli 13

3 Muut käytetyt työkalut 16

3.1 NewTek Lightwave 10 16
3.2 Adobe After Effects CS5.5 17

4 Työn hyödyntämät teknologiat 18

4.1 Yleiset teknologiat 19
4.1.1 Line Collision -funktio 19
4.1.2 Surface-muuttujat 24
4.2 Pseudokolmiulotteisen grafiikan luonti 27
4.2.1 Poikkileikkauspiirto 27
4.2.2 Pseudopolygonaaliset mallit 30
4.2.3 Esirenderöinti 34
4.3 Valaistustekniikoiden simulointi 37
4.3.1 Heijastukset 38
4.3.2 Epäsuora valaistus 40
4.3.3 Varjot 47

5 Pelidemo 52

5.1 Idea 52
5.2 Pelin suunnittelu 53
5.2.1 Pelihahmot 53
5.2.2 Pelihahmojen ohjaus 53
5.2.3 Kentät 54

6 Yhteenveto 55

Metropolia

7 Lähteet 56

Liitteet

Liite 1. Hello World -ohjelman luonti GameMaker-ympäristössä

Liite 2. Pelidemon pelikuvavideo

1 Johdanto

Kysymykseen siitä, mikä on maailman ensimmäinen tietokonepeli, ei ole täysin yksiselitteistä vastausta. Riippuen siitä, miten tietokonepelin määrittelee, moni pääsee kilpailemaan ensimmäisen tietokonepelin tittelistä. Alempana olevasta kuvasta näkee yhden tunnetuimmista varhaisista tietokonepeleistä. Se on vuonna 1950 Canadian National Exhibition -tapahtumassa esitelty Bertie the Brain -ristinollapeli, jossa ihminen pääsi ottamaan yhteen tekoälyn kanssa [1].

Kuva 1: Valokuva Bertie the Brainistä näyttelyssä

Vaikka Bertie the Brain oli peli, joka toimi tietokoneen voimalla, sen ensisijainen tarkoitus oli esitellä Rogers Majestic -radioputkivalmistajan innovaatioita. Tämän lisäksi, koska ristinolla on täysin vuoropohjainen peli, voidaan ajatella, että kyseessä oli enemmänkin oikean asian virtuaalikopio, joka ei tuonut mukanaan käytännössä mitään uutta.

Vuonna 1958 Brookhavenin kansallislaboratorion työntekijä William Higinbotham huomasi, että heidän vuosittain järjestettävät tiedemessut ovat hyvin tylsiä. Higinbotham päätti tehdä asialle jotain. Laboratoriossa oli käytössä Donner Model 30 -analoginen tietokone, ja siihen ohjekirja, jossa kerrottiin muun muassa, miten voi simuloida pallon pomppimista.

Higinbotham keksi, että yhdistämällä kirjassa esitetyt esimerkit, tulokseksi voi saada yksikertaisen tennispelin. Muutamassa päivässä näistä ideoissa syntyi valmis ja toimiva peli nimeltä Tennis for Two. Peli osoittautui erittäin isoksi hitiksi sen vuoden näyttelyssä. Ihmiset jonottivat pitkään saadakseen kokeilla tätä keksintöä. Sen, miltä tämä peli näytti, voi nähdä alempana olevasta kuvasta.

Kuva 2: Jälleenluotu versio Tennis fot Two -pelistä

Sen lisäksi, että Tennis for Two oli lähtökohtaisesti luotu viihdyttämään, eikä esitelemään vain tieteellisiä saavutuksia, reaaliaikaisena pelinä se myös tarjosi jotain sellaista, mitä oikeassa maailmassa ei olisi mahdollista [2.] Ja tästä syystä koen, että Tennis for Two on erinomainen ehdokas ensimmäisen tietokonepelin tittelin saajaksi.

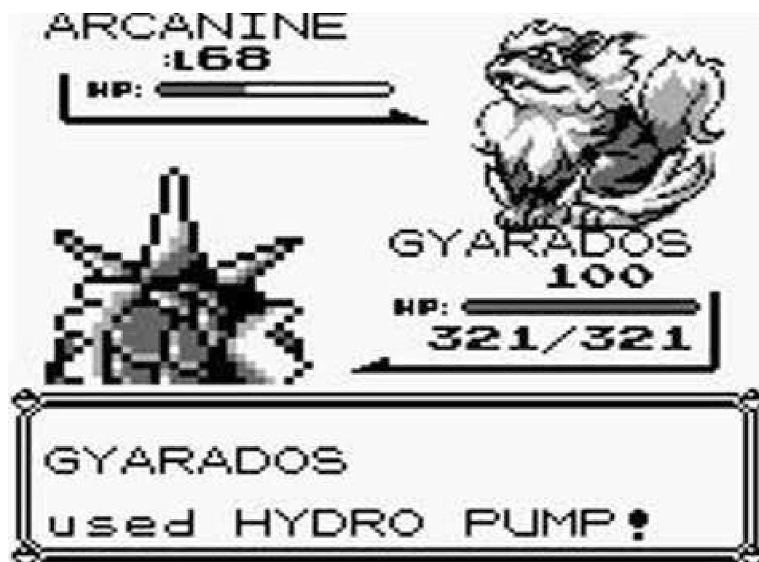
Tietokonepeliteollisuus on kulkenut pitkän matkan. Muutamaa harha-askelta lukuunottamatta, kuten vuoden 1983 videopelilamaa [3], tietokonepeliteollisuus on kasvanut vuodesta vuoteen. Tällä hetkellä 1,8 miljardia ihmisiä, noin neljännes koko maailman väestöstä, viettää aikaa tietokonepelien parissa [4]. Jopa Suomessa peliteollisuus on miljardibisnestä [5].

Ensimmäistä kertaa törmäsin tietokonepeleihin 90-luvun alussa, noin viiden vuoden ikäisenä. Menimme käymään minun äitini kaverilla, ja tämän kaverin pojalla oli Dendy, Venäjän markkinoille suunniteltu, Taiwan-valmisteinen, Nes-kloonikonsoli. En päässyt

sitä silloin pelaamaan, mutta sekin pieni määrä osoitti minulle, miten paljon hauskaa tekemistä tietokonepelit tarjoavat.
Oman ensimmäisen konsolini, aidon käytetyn Nesin, sain noin vuotta myöhemmin vuonna 1995, kun muutimme perheeni kanssa Suomeen. Pelasin niin paljon kuin vanhemmat sallivat. Silloin tällöin piirtelin myös paperille ideoita omista peleistä, mutta missään vaiheessa en yrittänyt tehdä peliä itse. Tietokoneet tuntuivat silloin minusta erittäin vaikeilta laitteilta.

Vuonna 1999 Suomen televisioihin rantautui Pokemon-animaatiosarja, joka nousi erittäin kovaan suosioon. Myös minä seurasin kovalla innolla sitä, ja koulun välitunneilla kävin kiivaaita keskusteluita siitä, mikä pokemon on paras. Kun saimme tietää, että Pokemonista on olemassa myös tietokonepeli Game Boy Color käsikonsolille, sitä oli pakko päästä kokeilemaan.

Koulun rikkaimmat lapset saivat itselleen kopion Pokemon-pelistä ja kertoivat kaikille, miten mahtava se oli. Minun odotukseni nousivat erittäin korkealle, ja kun vihdoinkin pääsin testaamaan oikeaa peliä Veikon Kone -liikkeessä, petyin paljon. Pokemonien taistelut, jotka olivat animaatiosarjan parhaita kohtia, oli toteutettu vuoropohjaisena roolipelinä. Kuvankaappauksen Pokemon-pelistä voi nähdä alempana. Tasohyppelyiden suurkuluttajana, se tuntui minusta todella tylsältä. Minä päätin, että pystyn parempaan. Minä teen oman Pokemon-pelin.



Kuva 3: Kuvankaappaus Pokemon Blue -pelistä Gama Boy -konsolilla [6]

Seuraavana päivänä marssin kirjastoon, ja lainasin Jesse Libertyn kirjoittaman Opeta itsellesi C++-ohjelmointi-kirjan. Rehellisesti sanoen, en tiennyt edes, mikä on C++, mutta kirjan paksuus vakuutti, että kyseessä olisi hyvä kirja. Kuukautta myöhemmin palautin kirjan takaisin oppimatta ohjelmoimaan yhtään, sillä paremman ohjeistuksen puutteessa kirjoitin kaikki kirjan esimerkit WordPadilla, ja ihmettelin pitkään, miksi mitään ei tapahdu.

Siihen aikaan Mikrobitin, nykyisen MB-lehden, nettisivulla julkaistiin päivittäisn päivän peli ja ohjelma. Valtaosan ajasta kävin hakemassa sieltä tuoreimman ilmaisen pelin enkä välittänyt ohjelmista. Välillä kuitenkin vastaan tuli ohjelma, joka herätti minussa kiinnostuksen. Yksi tällaisista ohjelmista oli GameMaker 6.0, jota mainostettiin työkaluksi, jolla jokainen pystyisi tekemään pelin ilman ohjelmointitaitoa.

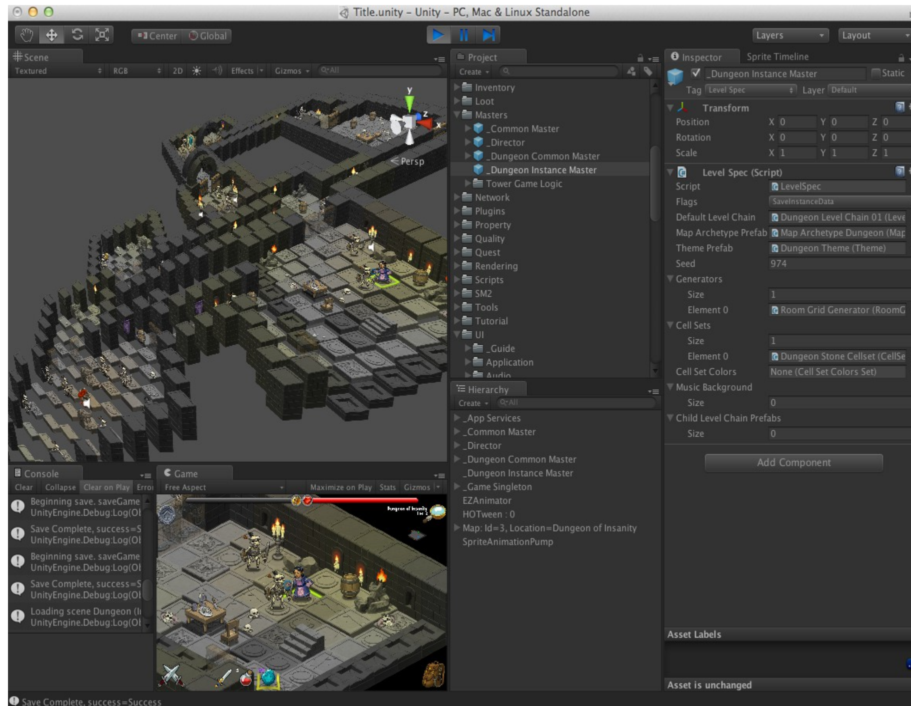
Englanninkielisenä ohjelman GameMaker kuitenkin osoittautui hivenen haastavaksi, ja silloin peliurani suurimmaksi saavutukseksi jäi Ritari Ässä -peli, joka oli käytännössä GameMakerin mukana oleva esimerkkipeli, jossa vaihdoin grafiikat ja musiikit,

Yläasteella tapasin kuitenkin yhden pojan, joka oli käyttänyt GameMakeria pitkään, ja hän selitti minulle, miten sillä tehdään pelejä. Siitä alkaen, jo viidentoista vuoden ajan olen harrastanut pelien kehitystä GameMaker-alustalla.

Viidentoista aikana paljon on ehtinyt tapahtua pelialalla. Markkinoilla on ehtinyt vaihtua kaksi sukupolvea pelikonsoleita. Älypuhelimet toivat pelit ennennäkemättömän yleisön ulottuville. Uusien laitteiden myötä myös pelimoottorit ja kehitystyökalut ovat muuttuneet rajusti.

GameMaker on päivittynyt tasaiseen tahtiin, ja on jopa ehtinyt vaihtaa nimensä GameMaker: Studioksi, mutta mitään isompia muutoksia ei tapahtunut. GameMakeriin toteutettiin 3D-grafiikan tuki, mutta se on niin alkeellista ja vaikeakäyttöistä, että sen käytössä on hyvin vähän järkeä. GameMaker sai varjostimien tuen, mutta niiden käyttö rikkoo moottorin luontaisen ergonomian, sillä kyseessä on suora OpenGL-toteutus. GameMaker vihdoinkin sai fysiikkamoottorin, mutta ne ovat niin yleisiä nykyään, että sellaisen puute olisi erittäin outoa. GameMaker on ensisijaisesti edelleen ketterä 2D-pelimoottori pienten pelien luontiin.

Tämän päivän suosituin pelimoottori indie-puolella on Unity, jonka käyttöliittymän voi nähdä kuvasta 4. Unity implementoi jatkuvasti uusia tenologioita, ja tukee niin 2D- kuin 3D-pelien luontia. Mikäli sitä vertaa ominaisuuksiltaan suoraan GameMakeriin, niin puhtaasti kaksiulotteisena moottorina jälkimmäinen alkaa vaikuttaamaan täysin hyödyttömältä.

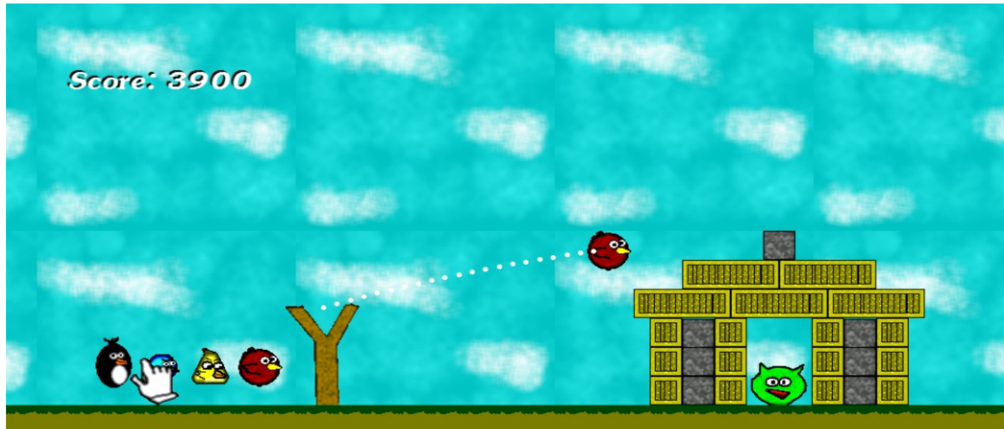


Kuva 4: Unity 3D -pelimoottorin käyttöliittymä [7].

Minä en kuitenkaan suostu jättämään GameMakeria kahdesta syystä. Ensimmäinen syy on se, että vaikka olen kokeillut lukuisia nykyisiä pelimoottoreita, en ole löytänyt yhtäkään moottoria, joka olisi pystynyt kilpailemaan GameMakerin kanssa ketteryydessä. Osaavissa käsissä, jo alle tunnissa, voi syntyä täysin toimiva pikkupelin prototyyppi. Kaikki muut moottorit, joissa on enemmän ominaisuuksia kuin GameMakerissa, tuntuvat kömpelöiltä ja ylimonimutkaisilta.

Minulla löytyy esimerkki omasta elämästäni. Opiskellessani Suomen Liikemiesten Kauppaopistossa erään tunnin alussa keskustelimme pelialasta, ja puheeksi nousi Angry Birds -peli, jota itse en pitänyt kummoisena pelinä mekaniikaltaan. Eräs oppilas sanoi minulle, että en pystyisi tekemään samanlaista. Tästä syntyi toverillinen vedonlyönti, jonka seurauksena samaisen oppitunnin aikan asensin koulun koneelle GameMaker 8.1 -ohjelmiston ja loihdin toimivan Angry Birds -kloonin nimeltään

Vihaiset tiput. Prototyyppi sisälsi käytännössä kaikki ominaisuudet, jotka olivat alkuperäisessä versiossa mukaan lukien grafiikat ja musiikit. Ainoa suuri ero oli fysiikkamallinnuksen puute, mikä johtui siitä, että siinä vaiheessa GameMakerissa ei vielä ollut natiivia fysiikkamoottoria. Alempana on esitetty kuvankaappaus Vihaiset tiput -pelistä.



Kuva 5: Kuvankaappaus Vihaiset tiput -pelistä

Toinen syy, miksi edelleen pysyn GameMakerissa kiinni, on minun laiskuuteni ja haluttomuus opiskella mitään uutta ilman sen suurempaa syytä. Pelinkehitys on minulle ensisijaisesti hauskaa ajanvietettä. Niin kauan, kun en tee sitä työkseni, koen, että sen pitäisi olla rentouttavaa. Paradoksaalisesti GameMakerin rajoitusten kiertäminen ja luovien ratkaisujen keksiminen tuntuu minusta paljon hauskemmalta kuin opetella joku tehokkaampi työkalu, ja ratkaista ongelmat hetkessä.

Nyt, kun ajankohtaiseksi tuli insinöörityön kirjoittaminen, mieleeni tuli, että voisi olla kiinnostavaa kirjoittaa työ, jonka aiheena olisi tutkia, miten paljon on mahdollista kiertää työkalun rajoitteita, jos tietää, mitä tekee. Peliala on lähtenyt liikkeelle siitä, että pelejä syntyi työkaluilla, jotka eivät ole tarkoitettu pelien luontia varten. Tämä lopputyö toimisi siis samalla myös eräänlaisena kunnianosoituksena niille ajoille.

Otin tavoitteekseni tutkia, miten on mahdollista simuloida visuaalisia tekniikoita, joita GameMaker ei luontaisesti tue. Tavoitteenani on tutkia ja luoda prototyypit oikean maailman ilmiöille, joita perinteisesti pystytään simuloimaan vain kolmiulotteisen mallinnuksen avulla. Tämän jälkeen minulla on tarkoituksena yhdistää näiden prototyyppien aikaansannokset, ja luoda niistä peliprototyyppi, joka toimisi sulavasti tavallisella nykyaikaisella tietokoneella.

2 GameMaker: Studio 1.4 -pelimoottori

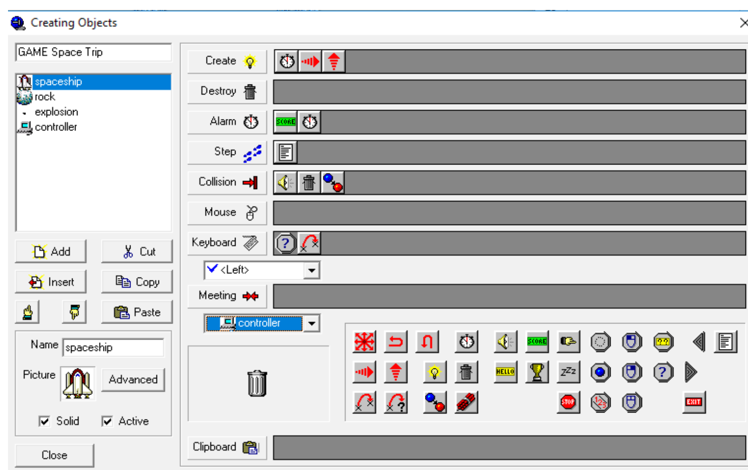
Tässä luvussa tutustutaan GameMaker-pelimoottorin taustoihin, vahvuuksiin ja heikkouksiin. Luvun lopussa käydään perusteet GameMakerin käyttämästä GML-skriptikielestä.

2.1 Lyhyt esittely

GameMaker on 2D-pelimoottori, jonka ensimmäinen versio julkaistiin vuonna 1999. Sen historia alkoi 90-luvun loppupuolella, kun alankomaalainen tietojenkäsittelytieteilijä ja peliohjelmoinnin opettaja Mark Overmars (s. 1958) [8] halusi opettaa lapsilleen tietojenkäsittelyä ja päätti kirjoittaa tätä tarkoitusta varten opetuskäyttöön tarkoitetun sovelluksen. [9.] Lapset eivät kuitenkaan innostuneet ideasta.

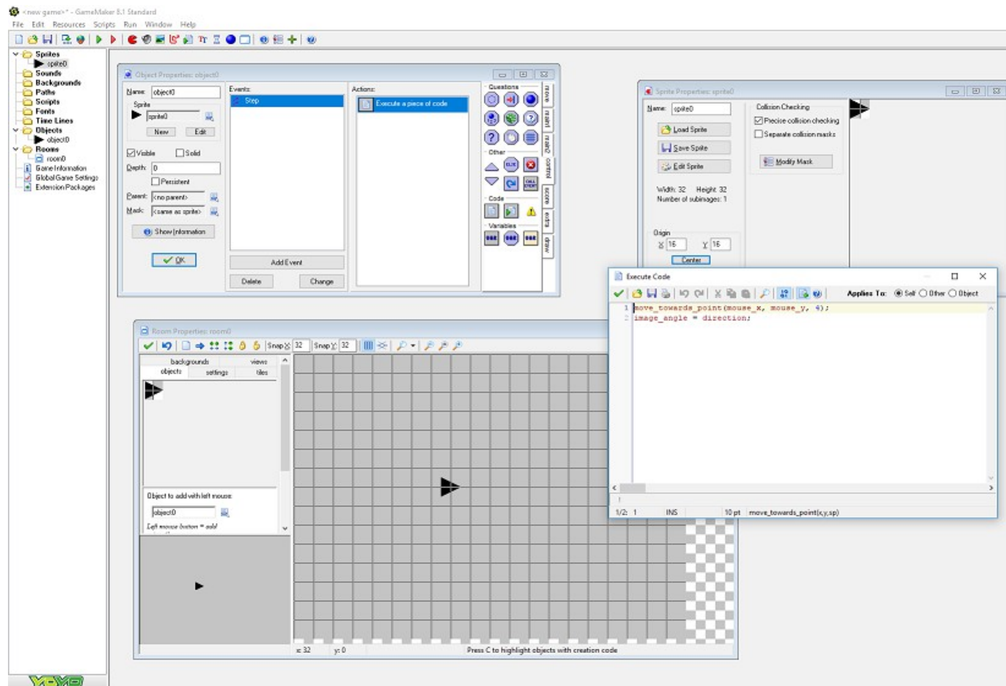
Tästä huolimatta Overmars jatkoi projektiaan, ja lopputuloksena syntyi Animo-niminen animaatiotyökalu. Animo ei ollut saatavilla julkisesti, eikä siitä edes näytä löytyvän yhtäkään kuvankaappausta Internetistä.

Animon pohjalta Overmars jatkokehitti GameMaker 1.1, josta tuli ensimmäinen julkaistu GameMakerin versio. Internetistä on mahdollista löytää näitä vanhoja GameMakerin versioita [10]. Mielenkiinnosta latasin version 1.4 (kuva 6) ja kokeilin sitä hetken. Monessa mielessä se näytti erittäin tutulta, mutta käyttöliittymä ja toiminnallisuus olivat niin vanhentuneita, että sillä en uskaltaisi lähteä tekemään yhtään mitään. Esimerkiksi tässä versiossa ei pystynyt viemään ulos exe-tiedostoja, vaan sillä luodut pelit olivat vaatineet toimiakseen, että koneella olisi GameMaker.



Kuva 6: Kuvankaappaus GameMaker 1.4 -ohjelmistosta

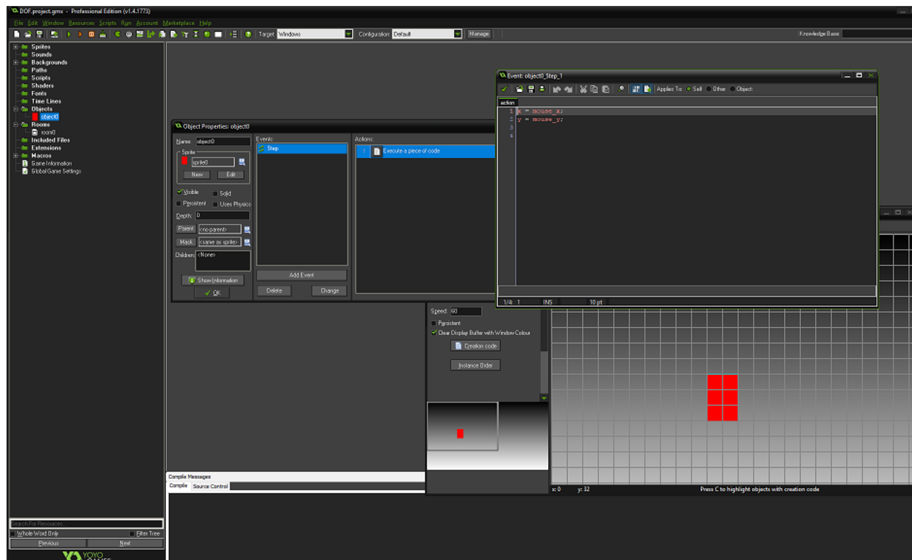
GameMakerin suosion kasvaessa ohjelman kehitys jatkui. Vuonna 2007 Overmars aloitti yhteistyön skotlantilaisen YoYo Games Ltd.n kanssa, koska ei omasta mielestään kyennyt omistamaan asialle kylliksi aikaa yliopistovirkansa vuoksi. [11.] Tuloksena syntyi GameMaker 7 ja myöhemmin GameMaker 8.1 (kuva 7).



Kuva 7: GameMaker 8.1 johon on avattu alle viidessä minuutissa tehty peli, jossa musta kolmio liikkuu kohti hiirtä.

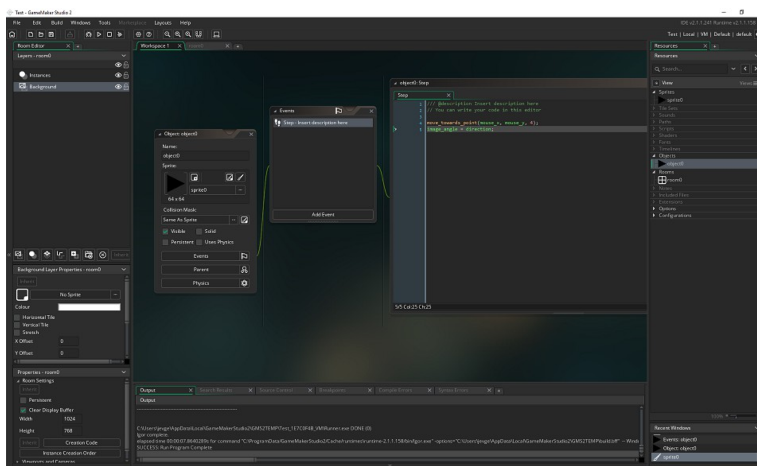
Jatkokehityksen myötä ohjelman nimi vaihtui GameMaker: Studioksi (kuva 8), eli GM:Ssksi. GM:Sn lisäsi pakettiin mahdollisuuden kehittää pelejä myös konsoleille ja älypuhelimille, mutta näistä ominaisuuksista pyydettiin peräti sata dollaria per kappale, ja monelta ne jäivät hankkimatta. GMS lisäsi mukaan kuitenkin myös kauan odotetut fysiikat.

Huolimatta siitä, että GMS on paljon edistyneempi kuin GameMaker 8.1, käytän säännöllisesti myös 8.1:tä,tämä johtune siitä, että GMS vaatii toimiakseen projektia, mutta 8.1:ssä pystyy lähtemään kehittämään ja koodaamaan suoraan lennosta. Tämä on erittäin hyödyllistä, kun tarkoituksena ei ole tehdä mitään sen isompaa, vaan ainoastaan testata mielen juolahtanutta koodin pätkää.



Kuva 8: GameMaker: Studio 1.4 ja sen käyttöliittymä

Pitkään aikaan GMS oli versiossa 1.4, kunnes marraskuussa 2016 alkoi GameMaker Studio 2:n avoin beta [12]. Tällä hetkellä GameMaker Studio 2 on virallisesti julkaistu. GMS2 koki niin isoja muutoksia käyttöliittymässä, että tekijät ovat päättäneet tehdä siitä erillisen sovelluksen (kuva 9). Tämä tarkoittaa sitä, että GMS 1.4:n käyttäjät eivät saaneet ilmaista päivitystä tuoreimpaan versioon, kuten aikaisemmin on ollut tapana.



Kuva 9: GameMaker Studio 2:n käyttöliittymä

GMS2 muutti käyttöliittymänsä node-pohjaiseksi ja panosti kovasti animaatiotyökaluihin. GMS2 on luvattu olevan paljon stabiilimpi kuin edeltäjänsä.

GMS 1.4:n jatkokehitys on virallisesti päättynyt 31. elokuuta 2017 [13]. Henkilökohtaisesti en kuitenkaan aio päivittää uusimpaan versioon. En koe, että se tarjoaa minulle mitään sellaista, mitä tarvitsen, varsinkin kun ottaa huomioon sen hintavuuden. Lisäksi koen, että vanhentuneen version käyttö sopii paremmin tämän lopputyön teemaan.

2.2 GameMaker: Studio 1.4 -moottorin vahvuudet

Kuten on jo useampaan kertaan sanottu, Game Makerin suurin vahvuus on ketteryys, jolla sillä kehittäminen sujuu. Poiketen puhtaista ohjelmointikielisistä pelinkehitysratkaisuista, joissa pelkästään peli-ikkunan luonti vaatii kymmeniä riviä koodia, ja jokainen luokka sekä olio koostuu useammasta tiedostosta, GameMakerissa nämä asiat on piilotettu taustalle ja automatisoitu, eikä käyttäjän tarvitse kirjoittaa mitään ”turhaa”.

Asiaa on havainnollistettu liitteessä 1, jossa on esitetty yksinkertainen ohjeistus, miten GameMaker: Studio 1.4 -moottorilla on mahdollista luoda Hello World -ohjelma.

Huolimatta hieman vanhanaikaisesta käyttöliittymästään GameMaker on erittäin looginen, mikä tekee siitä erittäin helpon ja nopean oppia. Ohjelman mukana tulee kattava ohjekirja, jossa tutustutaan ohjelman mahdollisuuksiin, mutta mikäli ymmärtää alustan logiikan, valtaosan perusasioista pystyy päättelemään kokeilemalla.

Hyvänä esimerkkinä GameMakerin ketteryydestä voisin mainita tarinan omasta elämästäni. Muutama vuosi sitten erään pelinkehityskerhon seurassa kävin Grand Cru -pelitalon tiloissa Sörnäisissä. Huomasin, että valtaosalla koneista oli asennettu niin Unity kuin GM:S. Kysyin henkilökunnalta asiasta, ja he kertoivat, että vaikka Unity on heidän käyttämänsä pelimoottori, he käyttävät GameMakeria konseptien testaamiseen, koska se on niin ketterä.

Viimeinen hyvä asia GameMakerissa on laaja alustojen tuki. Sillä pystyy kehittämään niin yleisimmille käyttöjärjestelmille, älypuhelimille kuin Playstation- ja Xbox -konsoleille. Totuuden nimissä on sanottava, että nämä ulosvientimoduulit ovat muuttuneet melko kalliiksi versio 2:n myötä, mutta mikäli puhutaan yrityspuolesta, hinnat ovat edelleen kilpailukykyisiä. Lista alustoista, jotka tukevat GameMaker-pelimoottorilla tehtyjä pelejä, löytyy kuvasta 10.

	Trial	Desktop	Web	UWP	Mobile	PS4	Xbox One	Ultimate
Unlimited Resources		✓	✓	✓	✓	✓	✓	✓
Expert Features		✓	✓	✓	✓	✓	✓	✓
Target Platform(s)	TEST only	Windows, Mac, Ubuntu	HTML5	Microsoft UWP	Android, iOS	PS4	Xbox One	All Platforms
GameMaker: Studio 1.4 Professional Access		✓	✓	✓	✓	✓	✓	✓
Marketplace		✓	✓	✓	✓	✓	✓	✓
Support	GMC	✓	✓	✓	✓	✓	✓	✓
License Type	Permanent	Permanent	Permanent	Permanent	Permanent	12 Month	12 Month	12 Month
	DOWNLOAD	$99.99	$149.99	$399.99	$399.99	$799.99	$799.99	$1500.00

Kuva 10: Valmistajan sivulla mainitut tuetut alustat

2.3 GameMaker: Studio 1.4 -moottorin heikkoudet

Se, mitä GameMaker voittaa ketteryydessä, se häviää modernien teknologioiden tuen heikkoudessa. Mark Overmarsilta on kysytty monesti, voisiko hän luoda GameMaker 3D:n, joka olisi kuin GameMaker, mutta suunniteltu 3D-pelien luontiin. Tähän hän on aina vastannut, että mikäli sellaisia ominaisuuksia lähtisi GameMakeriin lisäämään, siitä tulisi turhan monimutkainen geneeriseksi työkaluksi [14]. GameMakerin nykyinen julkaisija, YoYo Games, on nähtävästi asiasta samaa mieltä.

Haluttomuus monimutkaistaa GameMakeria teki alustasta rajoittuneen. Ominaisuudet, jotka muodostuivat muilla alustoilla vakioiksi, tulivat GameMakeriin viiveellä. 2D-peleissä valtaosa visuaalisesta loisteesta on kiinni siitä, miten hyvin taiteilija osaa piirtää. 3D-peleissä pelkkä laadukas 3D-malli ei riitä, vaan tarvitaan myös moottori, joka osaa renderöidä sen näytölle ja valaista sen uskottavasti. Koska GameMakerin 3D-tuki on erittäin alkeellista, jopa parhaat sillä tehdyt 3D-pelit hädin tuskin ylittävät Playstation 1- tai Nintendo 64 -konsolien grafiikat. Asiaa voi havainnollistaa vertailemalla kuvaa 11 ja kuvaa 12.

Kuva 11: Kuvankaappaus Sakis Rogkasin Mythology -pelistä vuodelta 2011. Tämä on yksi parhaimmannäköisistä GameMakerilla tehdyistä 3D-peleistä [15].

Kuva 12: Kuvankaappaus God of War 2 -pelistä vuodelta 2007 [16].

GameMakeria kritisoidaan monesti siitä, että se ei ole ”oikeaa ohjelmointia” edes skriptauspuolella, koska GameMakerin skriptikieli tekee monet asiat täysin eri tavalla kuin valtaosa kielistä. Tämän väitetään opettavan huonoja ohjelmointitapoja, mutta itse olen asiasta eri mieltä. Kuten nähdään myöhemmin, GameMakerissa voi ohjelmoida hyvin monella tavalla.

Olen törmännyt mielipiteeseen, että GameMakerin käyttöliittymä on jäykkä ja vanhentunut, mutta tämä on minusta erittäin subjektiivista. Esimerkiksi omasta mielestäni GMS2 on paljon sekavampi käytössä.

Yksi asia, joka ei todellakaan ole subjektiivista, on GameMakerin heikko kyky hyödyntää laitteistoa. Mikäli ei halua opetella OpenGL-varjostimien käyttöä, GameMaker tulee pyörittämään koko pelin käyttäen käytännössä pelkkää prosessoria välittämättä näytönohjaimesta.

Esimerkiksi, minun tietokoneeni, jossa on 16 gigatavua muistia ja Intel Core I7-6700HQ -prosessori hädin tuskin pääsi 28 kuvaan sekunnissa, kun sen piti piirtää 60 kertaa sekunnissa 32 * 21 -kokoisen staattisen ruudukko. Toden nimissä on mainittava, että tuollainen ratkaisu on huono jopa sellaisessa kielessä, joka on luotu ottamaan kaiken irti koneen tehoista, koska siinä piirretään jatkuvasti uusiksi asioita, joita ei ole tarvetta piirtää. Kuitenkin on outoa, että GameMakerin optimointi on niin heikkoa, että se ei jaksa suoriutua hyvin vaatimattomista tehtävästä.

YoYo Games on kylläkin yrittänyt korjata heikon resurssioptimoinnin esittelemällä YoYo Compilerin eli YYCn. Kyseessä on GameMaker: Studion lisäosa, joka ottaa GameMakerilla luodun pelin, ja kääntää GameMakerin omat skriptit C++-kieleksi, mikä nostaa suorituskykyä moninkertaisesti. YYC maksoi alun perin 299 dollaria, mutta myöhemmin siitä tuli ilmainen. Se vaatii toimiakseen Visual Studio 2012. YYC hieman monimutkaistaa GameMakerin käyttöä, koska toimiakseen se vaatii käyttäjältä joidenkin ohjelmointikonventioiden noudattamista, jotka muuten eivät ole tarpeellisia GameMakerissa. Lisäksi se poistaa GameMakerin omat virheilmoitukset käytöstä, joten bugien etsiminen käy vaikeammaksi. Onneksi mikään ei estä suorittamasta debuggaus GameMakerin omalla kääntäjällä, ja viedä debugattu peli ulos YYC:n avulla.

Valitettavasti en saanut YYC:tä toimimaan, koska VS2012:ta on todella vaikea löytää enää mistään.

2.4 GML-skriptikieli

GameMaker on suunniteltu niin, että sillä pystyy tekemään pelin hyödyntäen pelkästään yksinkertaista drag-and-drop-käyttöliittymää. Jossain vaiheessa kuitenkin tulee drag-and-drop-puolen rajallisuus. Esimerkiksi ainoa toistorakenne on

GameMakerin oma repeat(toistojen määrä)-funktio, joka ei perustu ehtoihin vaan toistojen määrään. Piirtofunktiosta on saatavilla vain ympyrät, neliöt ja viivat. Johtuen siitä, että drag-and-drop-komponentit ovat melko kookkaita, pienikin ”koodi” vie visuaalisesti paljon turhaa tilaa.

Avuksi tulee GameMakerin oma skriptikieli GML, eli GameMaker Language. Kyseessä on erittäin vapaaseen syntaksiin perustuva, heikosti tyypitetty kieli, joka avaa GameMakerin koko potentiaalin. Kaikissa tulevissa esimerkeissä hyödynnetään ainoastaan sitä.

Ensiksi on hyvä mainita pari pientä GML:n ominaisuutta, jotka saattavat vaikuttaa oudoilta ihmisistä, jotka ovat tottuneet monimutkaisimpiin kieliin. Ensinnäkin GML:ssä ei ole pakko käyttää puolipistettä. Toiseksi GML:ssä ei ole pakko käyttää sulkeita ehto- ja toistolauseissa, pois lukien for-rakenne. Kolmas outous on se, että ehtorakenteiden kohdalla se hyväksyy, että ==, että = vertailuoperaattoriksi. Jos kuitenkin tavoitteena on hyödyntää YoYo Compileria, nämä ratkaisut on hyvä unohtaa, sillä ne eivät toimi C++:n kanssa. Alempanan on havainnollistettu esitetty muutama GML-skriptille ominainen lause.

```
draw_text(100, 100, ”teksti”) //Toimii
draw_text(100, 200, ”teksti”) ; //Toimii

if (mouse_x = 100)
    draw_text(100, 300, ”teksti”) //Toimii

if mouse_x = 100
    draw_text(100, 400, ”teksti”) //Toimii

if mouse_x == 100
    draw_text(100, 500, ”teksti”) //Toimii
```

Esimerkkikoodi 1: Esimerkkejä GML-kielen hyväksymistä syntakseista.

GameMaker ja GML ovat erittäin uniikkeja, mikäli puhutaan olio-ohjelmoinnista. GameMakerin luokat (objektit) tukevat muuttujia, mutta niistä puuttuvat metodit sanan perinteisessä mielessä. Kaikki koodi, joka on kirjoitettu objektien tapahtumiin, on kertakäyttöistä, eikä sitä voi käyttää uusiksi myöhemmin.

Olisi outoa, jos GameMakerissa ei olisi millaistakaan mahdollisuutta uudelleenkäyttää koodia, ja aina kun olisi tarve, että joku instanssi suorittaisi ryhmän identtisiä käskyjä, ne joutusi kovakoodaamaan aina uudestaan jokaiseen kohtaan. GameMakerissa tämä ratkeaa skriptien avulla.

Skripti-tiedostoon kirjoitetaan komentoja. Jokaista skriptiä kannattaa ajatella omana metodinaan. Poiketen perinteisestä ratkaisusta, jossa kakki metodit on sidottu tiettyihin luokkiin, GameMakerissa kaikki skriptit ovat kaikkien objekti-luokkien käytössä. Käyttäjän vastuulle jää varmistaa, että skripti ei sisällä viittauksia muuttujiin, joita ei sitä kutsuvalla objektilla ole. Mikäli on tarvetta, skriptejä voi ajatella toiminnoista koostuvana kokonaisuutena, jota voi kutsua tarpeen tullen.

Alempana on esitetty skripti, joka ottaa objektin pisteet ja palauttaa sen kaksinkertaisena.

```
pisteet = atrgument0;
return pisteet * 2;
```

Esimerkkikoodi 2: Parametreja vastaanottava ja arvon palauttava metodimainen skripti.

Nyt aina, kun on tarve, että instanssi tuplaisi pistemääräänsä, on mahdollista suorittaa pisteet = script_execute(script0, pisteet); -komento. Script_execute -komento vaatii parametrikseen skriptin nimen, ja mikäli skriptissä on käytetty argument-muuttujia, yhtä paljon parametreja kuin skriptistä löytyy argument-muuttujia. Yhdessä skriptissä voi olla korkeintaan 16 argumenttia. GameMaker ei tunne ylikuormitusta, joten jokaista tapausta varten pitää luoda oma skriptinsä.

Mikäli on tarvetta, että joku instanssi pakottaa toisen instanssin suorittamaan jonkun skriptin, pitää käyttää with-rakennetta. With-rakenne kehottaa sulkeiden sisällä olevaa instanssia tai kaikkia objektin instansseja suorittamaan, mitä ikinä aaltosulkeiden välissä on. Rakennetta on havainnollistettu alempana.

```
with (object0)
    script_execute(script0, pisteet);
```

Esimerkkikoodi 3: GML-skriptikielen with-rakenne.

GMLssä muuttujia on kolmea tyyppiä: tavallinen, global ja var. Muuttuja voidaan alustaa ilman mitään erikoismerkkiä tai sanaa. Kirjoittamalla ”muuttuja = 2” luodaan muuttuja-niminen muuttuja, jossa tällä hetkellä sijaitsee arvo 2. Tämä muuuttuja on kiinni tietyssä objektin instanssissa. Mikäli muuttujan nimen eteen lisätään global., esimerkiksi global.muuttuja, muuttujasta tulee globaali, eli se on kaikille pelin instansseille yhteinen. Mikäli muuttujan nimen eteen kirjoitetaan var, kuten ”var muuttuja = 2”, niin saadaan aikaan väliaikainen muuttuja, joka pyyhkiytyy koneen muistista heti, kun sille ei ole enää käyttöä, vapauttaen näin resursseja. Var-alkuiset muuttujat sopivat erinomaisesti esimerkiksi, kun pitää säilyttää välituloksia monimutkaisen laskun aikana.

Viimeinen mainitsemisen arvoinen asia on se, että GMLssä voi luoda instansseja ilman sitä, että antaa niille nimen. Komennot ”instance_create(x,y,object);” ja ”olio = instance_create(x,y,object);” ovat molemmat täysin oikeaoppisia. Tämä ominaisuus ei ole uniikki GML-skriptille, ja löytyy myös esimerkiksi C++. Poiketen kuitenkin C++:sta, käyttäjällä on myös mahdollisuus viitata nimettömiin instansseihin tarpeen tullen. Tämä onnistuu esimerkiksi id-muuttujalla, joka generoidaan automaattisesti jokaiselle instanssille, tai funktioilla, jotka hakevat tiettyjä ehtoja täyttäviä instansseja. Esimerkiksi instance_nearest(mouse_x, mouse_y, object0) -funktio palauttaa object0-luokan instanssin, joka on lähimpänä hiirtä.

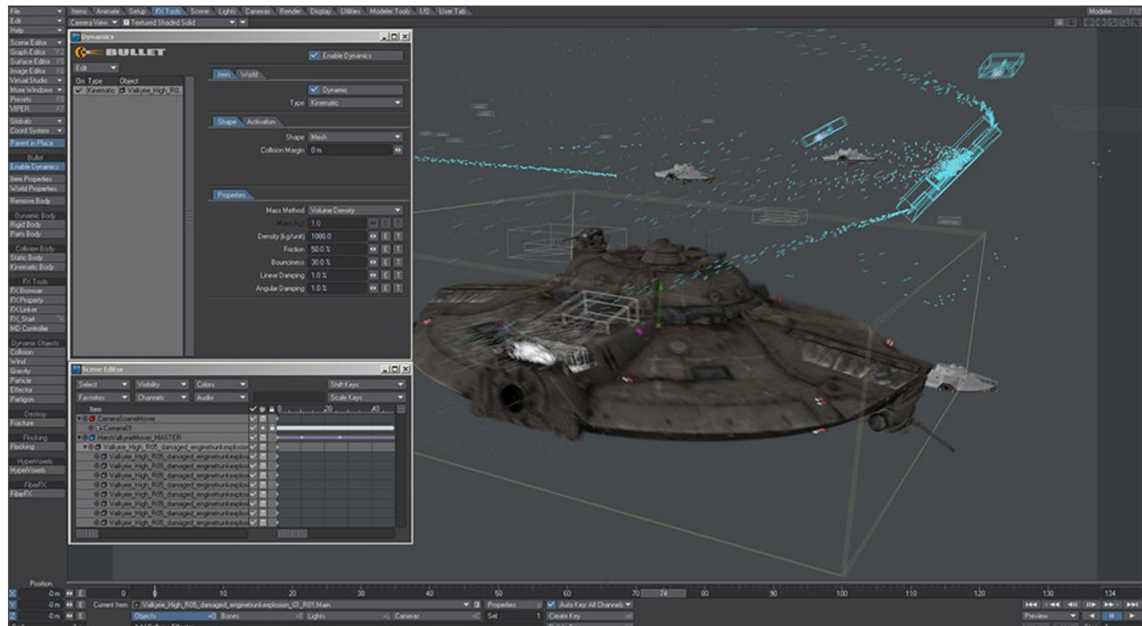
3 Muut käytetyt työkalut

Tässä luvussa tutustutaan muihin työkaluihin, joita on käytetty GameMaker-pelimoottorin lisäksi, työssä esitettyjen graafisten tekniikoiden simuloimisessa.

3.1 NewTek Lightwave 10

Johtuen siitä, että lopputyöni käsittelee sitä, miten 2D-pelimootorilla voisi luoda visuaalisesti miellyttävää pseudokolmiulotteista kuvaa, tarvitsen jonkun työkalun, jolla voisi luoda kolmiulotteista grafiikkaa, jonka voisi ”huijaten” saada GameMakerin sisään.

Minun valitsemani ohjelmisto on NewTekin Lightwave 3D, joka on etenkin televisiopuolella käytetty 3D-animaatioon tarkoitettu sovellus. Käytän versiota 10, joka on ilmestynyt vuonna 2010. Kuvankaappaus tästä sovelluksesta löytyy kuvassa 13.



Kuva 13 Lightwave 10 -käyttölittymä, jossa on auki kohtaus Iron Sky -elokuvasta [17].

Ligtwavella on muutama suuri ero verrattuna kilpailijoihinsa. Graafisten ikonien sijaan kaikki napit on varustettu tekstillä. Eli sen sijaan, että napissa olisi kuva, jossa on kolme eri suuntiin lähtevää nuolta, siinä vain lukee Move. Omasta mielestäni tuollainen lähestymistapa on tervetullut, koska tuolloin ei tarvitse pähkäillä, mitä nappi tekee, vaan teksti kertoo sen suoraan.

Toinen Lightwaven omalaatuisuus perustuu siihen, että siinä mallinnus- ja animointipuoli on erotettu omiksi sovelluksikseen. Omasta mielestäni tämä tekee ohjelmasta helpommin hallittavan, koska samaan käyttöliittymään ei ole tarvetta tunkea kaikkea toiminnallisuutta.

Tässä työssä Lightwavea tullaan käyttämään ensisijaisesti poikkileikkauspiirto- ja esirenderöintiratkaisuissa, joista puhutaan lisää luvussa 4.2.

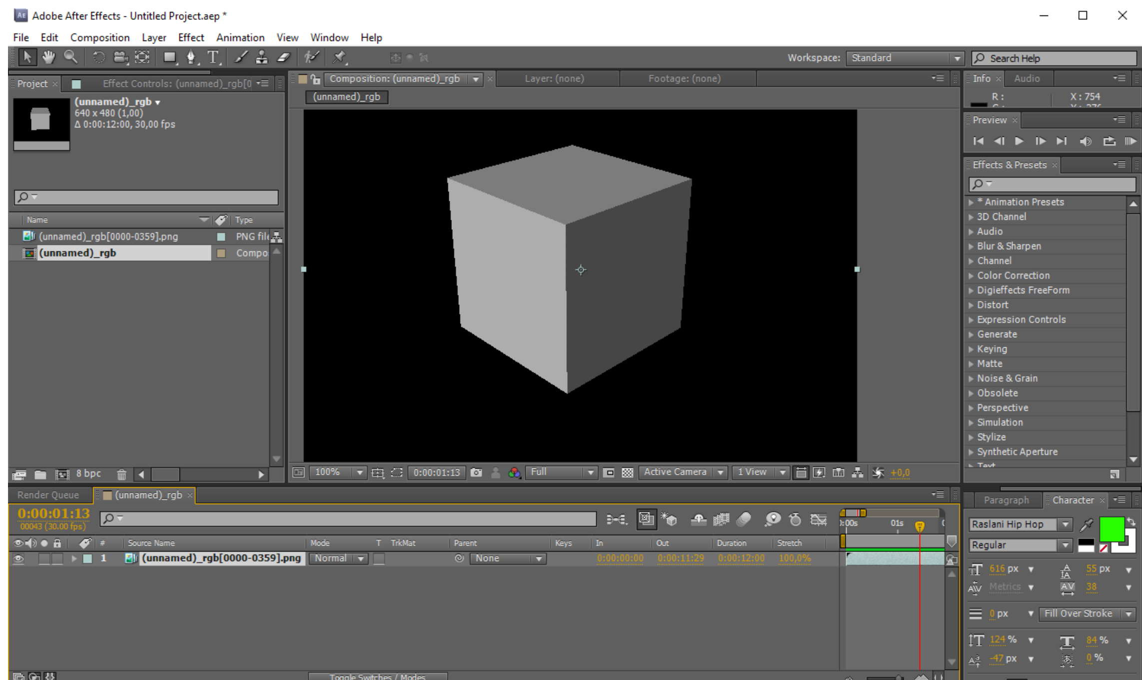
3.2 Adobe After Effects CS5.5

Lightwaven avulla pystyy renderöimään ulos monenlaista kuvaa, mutta monesti kuvaa on tarve jatkokehittää. Esimerkiksi peliä varten Lightwavesta kuvan saadaan ulos kuva, jossa on punainen pallo, mutta pelissä on tarvetta myös muuten identtiselle kuvalle, mutta jossa pallo on vihreä. Sen sijaan, että on tarvetta muokata Lightwavessa olevan pallon pintaa vihreäksi, ja renderöimämään uutta kuvaa, on mahdollista värimääritellä

vanha kuva. Näin tulokseksi saadaan molemmat kuvat, mutta pallon malli jää muuttumattomaksi

Tähän tarkoitukseen minun on tarkoitusta hyödyntää Adoben AfterEffects CS5.5 -sovellusta. Tällä versiolla on ikää seitsemän vuotta, mutta sekin on täysin käyttökelpoinen.

AfterEffects on Photoshop-ohjelmiston kehittäjien tuote, jota voisi kuvailla videopuolen Photoshopiksi. After Effectsin käyttöliittymä näkyy kuvassa 14.



Kuva 14: After Effects CS5.5, jossa on avattu esirenderöity pyörivä kuutio, jota pystyy nyt jatkotyöstämään.

After Effects on hyvin monipuolinen työkalu, jota voi hyödyntää monenlaisessa graafisessa käsittelyssä. Tästä syystä, en mainitse erityisemmin, missä sitä on käytetty, ellei kyseessä ole joku poikkeuksellinen tapaus.

4 Työn hyödyntämät teknologiat

Tässä luvussa tutustutaan GameMakerin sisäänrakennettuihin työkaluihin, ja siihen, miten niiden luovalla käytöllä voi saada aikaan vaikutelman, että GameMakerilla pystyy

tekemään asioita, joihin todellisuudessa se ei ole suunniteltu, kuten esimerkiksi dynaaminen valaistus.

4.1 Yleiset teknologiat

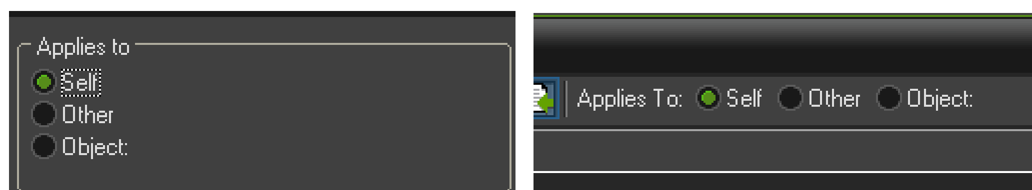
Osiossa tutustutaan GameMakerin toimintoihin ja työkaluihin, joiden varaan rakentuvat tulevassa luvussa esitetyt konkreettiset esimerkit.

4.1.1 Line Collision -funktio

Kun liikutaan drag-and-drop-puolella, ainoa tapa tarkistaa objektien törmäystä on hyödyntää objektikohtaista Collision-tapahtumaa. Esimerkiksi, jos on tavoitteena, että luodin osuessaan seinään, kummankin pitää tuhoutua, ja seinä-objektille on luotava Collision-tapahtuman luoti-objektin kanssa. Kun tämä tapahtuma toteutuu, GameMaker suorittaa instance_destroy(); -komennon, joka tuhoaa sen laukaisseen instanssin. Tuloksena kuitenkin tuhoutuu pelkästään seinäinstanssi, koska suoritettu tuhoutumiskomento ei liity mitenkään luoti-instanssiin.

Vastaavasti ohjelmassa pitää luoda identtinen Collision-tapahtuma luotiobjektille, jossa se määritellään tuhottavaksi törmätessä seinäobjektin instanssin. Tämä monimutkaistaa pelin hallintaa, koska nyt joudutaan jakaamaan kahteen paikkaan koodia, joka työskentelee saman prosessin parissa.

Kehittäjätkin ovat varmasti ajatelleet samaa, ja siksi jokaisella skriptillä ja drag-and-drop-ikonilla on Applies to -valikko, josta voi valita, koskeeko kyseinen komento tätä kyseistä instanssia (Self), toista tapahtumaan osallistuvaa instanssia (Other) vai jotain muuta objekti-luokkaa (Object). Asiaa on havainnollistettu kuvassa 15.



Kuva 15: Applies to -valikko drag-and-drop- ja skripti-puolella

Other-määrite on siitä erikoinen, että se toimii pelkästään Collision-tapahtumien yhteydessä. Mikäli sen valitsee missä tahansa muussa yhteydessä, se tulee toimimaan Self-määritteen lailla. Object-määrite on puolestaan erittäin epätarkka ja kattaa aivan

kaikki valitun luokan instanssit. Ilman lisäkoodin luomista ei esimerkiksi voi määrittää, että tapahtuman seurauksena vain ne objektin instanssit, jotka ovat 100 pikselin etäisyydellä tietystä pisteestä, suorittavat halutun toiminnon. Olen myös aina epäillyt Object-määritteen tarpeellisuutta, sillä omasta mielestäni kaikkien objektien tapahtumat kannattaa pitää kyseessä olevassa objektissa, eikä viitata niihin ristiin mistä sattuu.

GameMaker antaa mahdollisuuden valita objektien törmäyslaatikot hyvin vapaasti. Törmäyslaatikko voi olla suorakulmion, vinoneliön, ellipsin tai objektin spriten, eli grafiikan muotoinen. GameMaker pystyy myös hyödyntämään maskeja, jolloin törmäyslaatikko on tarkalleen jonkun spriten muotoinen, mutta kyseessä on aivan eri sprite kuin se, jonka objekti piirtää näytölle.

Peruskäytössä nämä törmäyksiä käsittelyä varten tehdyt työkalut riittävät käytännössä kaikkeen. Moni aloittelija jopa käyttää niitä liian innokkaasti ja ratkaisee niillä asioita, joihin olisi parempi ratkaisu. Esimerkiksi, sen sijaan, että aloittelija kirjoittaisi koodia, jossa tarkistetaan, että objektin x- ja y-koordinaatit ovat isompia kuin 0, mutta pienempiä kuin huoneen levys ja korkeus, hän laittaa kentän reunoille seinäobjekteja ja tarkistaa sijainnin törmäyksellä.

Kuten missä tahansa muussa yhteydessä GameMakerin koodipuolella, törmäyksien toiminnallisuutta pystyy laajentamaan käyttäen skriptauspuolta. Kaikki törmäystä käsittelevät funktiot alkavat sanalla collision_, kuten esimerkiksi collision_line(). Nämä funktiot ottavat kaikki hieman erilaisen määrän parametreja, mutta rakenne on kaikilla sama. Ensiksi tulevat parametrit, jotka vastaavat törmäysalueen geometriasta, kuten koordinaatit ja säde. Näiden perään tulee kolme parametria: obj, prec ja notme.

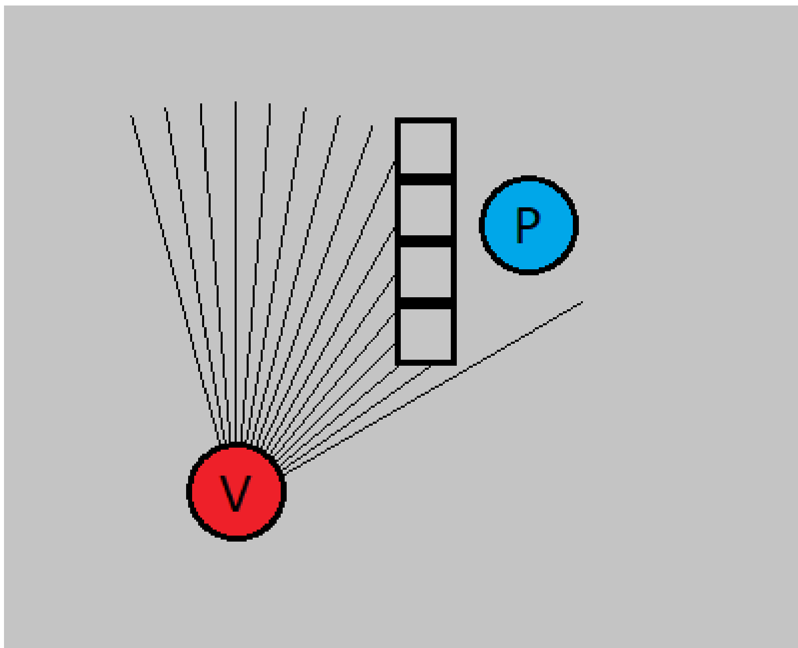
Obj-parametri kertoo, minkä objektin törmäystä seurataan. Sinne voi laittaa niin kokonaisen objekti-luokan kuin tietyin instanssin tai all-valitsimen, joka tarkoittaa sitä, että seurataan kaikkien objektien törmäilyjä. Prec-parametri, eri precicion, vastaanottaa totuusarvon, joka GMLssä voi olla joko true, false, 1 tai 0. Tämän parametrin ollessa totta törmäystä tarkistetaan objektin spriten muodon mukaan eikä neliskulmaisella laatikolla. Notme-parametri on myös totuusarvo, jolla voi poistaa funktiota suorittava instanssi tarkistuksesta, jos se muuten täyttää obj-parametrin. Seuraavalla sivulla on esitetty törmäyksen tarkistus collision_circle -funktiolla, joka kattaa pikselintarkasti kaikki instanssit, pois lukien se, joka funktiota suorittaa.

```
if (collision_circle(x, y, 64, all, 1, 1)) {
    //Suoritettavat komennot
}
```

Esimerkkikoodi 4: GML-kielen collision_circle-funktion yksinkertaistettu toteutus.

Jos GameMakerin drag-and-drop puolen törmäykset ovat itsessään noin tehokkaita ja monikäyttöisiä, mihin tarvitaan skriptipuolen antamia mahdollisuuksia? Ensimmäisenä mieleen tulee se, että tällä tavalla on mahdollista luoda objekteille useampia törmäyslaatikoita, jotka on tarkoitettu kattamaan erilaisia tilanteita. Tämän työn kannalta kiinnostavampi käyttötarkoitus on ray casting, eli säteenheittoteknologian toteutus collision_line-funktion avulla.

Säteenheitto on perinteisesti 3D-grafiikassa käytetty tekniikka, jossa kamerasta lähtee säteitä, jotka törmäävät matkalla oleviin esteisiin ja lyhenevät tarpeen mukaan. Tämän avulla on mahdollista saada selville kameran ja tietyn pisteen välisen etäisyyden, ja sen tiedon perusteella onnistuu mm. valonlähteiden simulointi ja varjojen piirtäminen. Eikä kyseessä edes tarvitse olla piirtämiseen liittyvä asia – täysin samoilla funktioilla on mahdollista toteuttaa esimerkiksi näkökenttiä pelihahmoille, joita voi sitten hyödyntää tekoälyssä. Yksinkertainen säteenheittoon perustuva näkökenttä on havainollistettu kuvassa 16. Tarvitaan keino, jolla voidaan lyhentää sädettä.



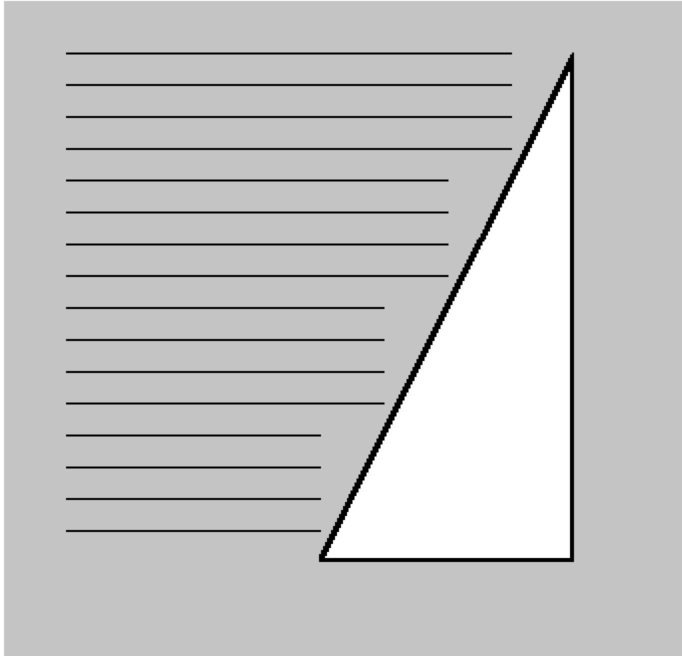
Kuva 16: Pelaaja ei osu vihollisen näkökenttään, koska seinät ovat tiellä.

Absoluuttisesti tarkin tapa lyhentää säteitä pituusyksikkö kerrallaan. Kuvitellaan, että on olemassa säde, jonka pituus on 256 pikseliä. Kun säde törmää esteeseen, ohjelma alkaa vähentää säteestä pikseli kerrallaan. Tuollainen lähestymistapa hyvässä tapauksessa tarvitsee yhden tarkistuksen, ja huonoimmassa säteen pituuden verran. Keskimäärin ratkaisu tulee tarvitsemaan puolen säteen pituudesta verran tarkistuksia. Mikäli tarkistettavana on vain yksi säde, tämän ei pitäisi viedä koneelta erityisemmin tehoja, mutta jos tarkistettavia säteitä on useimpia, niin ohjelma käy todella nopeasti raskaaksi. Esimerkki siitä, miltä tuollainen pikseli kerrallaan vähentävä, skripti voisi näyttää, on esitetty alempana

```
pituus =256;
while (collision_line(mouse_x, mouse_y, mouse_x + pituus,
       mouse_y, all, 1, 0)) {
    pituus -= 1;
}
```

Esimerkkikoodi 5: Pikseli kerrallaan lyhentävä pituuden etsintäskripti.

On mahdollista pienentää koneen rasitusta lyhentämällä sädettä kerrallaan pituusyksiköllä, joka on suurempi kuin 1. Esimerkiksi, vähentämällä 16 pikselin verran kerrallaan päästään tilanteeseen, jossa suurin määrä toistoja tulee olemaan 256 / 16, eli 16, ja keskimääräinen 8. Tämä on luonnollisesti paljon helpompaa koneen kannalta, mutta tuloksen tarkkuus kärsii, ja lopputuloksesta tulee sahamainen, koska huolimatta siitä, miten pieni lyhennys olisi tarpeen, ohjelma vähentää aina tasan tarkkaan 16 pikseliä. Sahamaisuuden voi nähdä kuvasta 17.



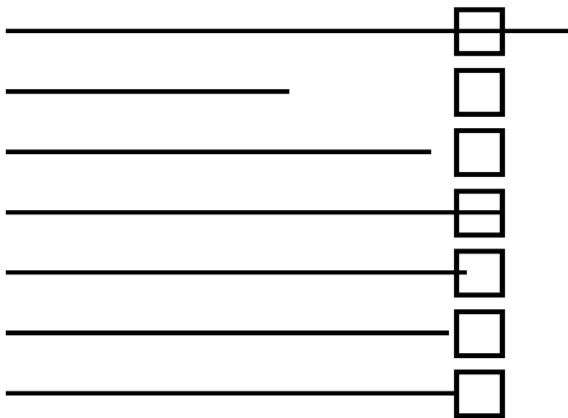
Kuva 17: Liian suuren pituusyksikön aiheuttama laskostuminen

Toistaiseksi työssä on käyty läpi raskas tarkka ja nopea epätarkka ratkaisu. Kumpikaan näistä ratkaisuista ei käy, jos on tarvetta ratkaisulle, joka olisi sekä tarkka että nopea. Ratkaisu tilanteeseen löytyy soveltamalla puolitus- eli binäärihakua. Alustetaan väliarvomuuttuja, jonka alkuarvoksi asetetaan puolet säteen pituudesta, ja tarkistetaan, jos säde osuu johonkin. Mikäli säde osuu, siitä lyhennetään väliarvomuuttujan verran, ja itse väliarvomuuttuja jaetaan kahtia. Tehdään uusi tarkistus. Riippuen siitä, törmääkö säde vai, siihen joko lisätään tai vähennetään siitä väliarvomuuttujan verran. Jokaisen kierroksen päätteeksi väliarvomuuttuja puolitetaan. Tätä jatketaan niin kauan, kunnes väliarvomuuttuja on 1. Alempana on esitetty eräs tämän algoritmin mahdollinen toteutus, ja sen toimintaa havainnollistaa kuva (kuva 18).

```
//Parametrit
sade = argument0;
suunta = argument1;
puolittaja = sade;

if (!collision_line(x,y, x+lengthdir_x(sade,
suunta),y+lengthdir_y(sade, suunta),all,1,1))
  return sateenPituus;

repeat (log2(sade) + 1) {
```

```
if (collision_line(x,y, x+lengthdir_x(sade,
suunta),y+lengthdir_y(sade, suunta),all,1,1))
   sade -= puolittaja;
else
   sade += puolittaja;
puolittaja /= 2;
}

return sade;
```

Esimerkkikoodi 6: Graafisen puolitushaun toteutus GML-skriptikielellä.



Kuva 18: Visuaalinen kuvaus algoritmin vaiheista

Poiketen aikaisemmista ratkaisuista, jossa jokaisen säteen kohdalla tarkistusten määrä vaihteli, tässä se on vaikio. Mikäli säteen matkalla on yksikin este, pituuden etsiminen tulee tarvitsemaan 1 + $\log_2$(säteen pituus) tarkistusta. Esimerkiksi 256 pikselin säteessä tämä luku tulee olemaan 9, koska $\log_2(256)$ on 8.

4.1.2 Surface-muuttujat

Kuten on luvussa 2.3 sanottu, GameMaker ei osaa hyödyntää sellaisenaan näytönohjaimia nimeksikään. Tämä tarkoittaa, että on absoluuttisen tärkeää minimoida turhat toistot piirtämisen yhteydessä.

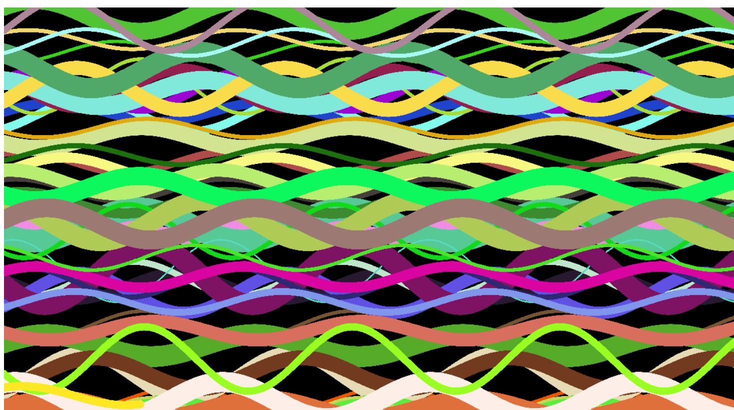
Luvussa 2.3 puhuttiin siitä, miten GameMakerilla tehty peli ylirasittaa tehokkaan tietokoneen, ihan vaan sillä, että se joutuu piirtämään 32 * 21 kokoista staattista

ruudukkoa. Olisi paljon tehokkaampaa, jos olisi mahdollisuus muodostaa ensiksi taulukosta kokoanaisuus, jolloin 672 piirtofunktion kutsun sijaan tarvitsisimme vain yhden.

Lisäksi on kysymys, mitä tehdä, jos tavoitteena on luoda ohjelma, joka piirtää näytölle uuden staattisen ympyrän siihen kohtaan, mihin käyttäjä klikkasi. On mahdollista esimerkiksi luoda aina uusi instanssi jokaisen klikkauksen yhteydessä, mutta tämä luo runsaasti instansseja,jotka vievät turhaa tilaa muistista. Vaihtoehtoisesti käyttäjä voi tehdä objektiin taulukon, johon tallentuu jokaisen klikkauksen koordinaatti. Myöhemmin Draw-tapahtumassa tämä taulukko käydään toistorakenteella läpi, ja näytölle piirretään kaikki ympyrät. Tässä tapauksessa kuitenkin päädytään siihen tilanteeseen, jossa tietokone joutuu suorittamaan joka sekunti satoja piirtofunktioita.

Apua löytyy surfaceista eli pinnoista. Kyseessä on GameMakerin eräänlainen vastine Javan Gaphics-luokalle. Pinnat ovat kuitenkin käytännössä vain muuttujia. Sen sijaan, että asioita piirretään suoraan ikkunaan, ne piirretään ensiksi pinnalle, ja sitten pinta piirretään ikkunaan. Kuvassa 19 on havainnollistettu pintojen avulla piirretty joukko erivärisiä siniaaltoja

Tämä vapauttaa laskentatehoja, sillä enää koneen ei tarvitse piirtää kaikkia asioita erikseen, vaan ne kaikki menevät samaan pakettiin. Poiketen monesta muusta GameMakerin ominaisuudesta, pinnat ovat puhtaasti skriptipuolella, eikä niille löydy mitään toteutusta drag-and-drop-puolella.



Kuva 19: Pinnan avulla toteutettu pieni ohjelma, joka generoi näytölle modernia taidetta, joka koostuu satunnaisvärisistä ja paksuisista siniaalloista.

Pintojen käyttö koostuu kolmesta vaiheesta. Aluksi pinta pitää luoda. Tämä onnistuu nimi = surface_create(leveys, korkeus); -komennolla. On tärkeää muistaa antaa pinnalle nimi, koska muuten viittaaminen siihen käy mahdottomaksi. On myös suotavaa olla luomatta isompia pintoja kuin on tarve, sillä pinnat vievät paljon muistia.

Poiketen tavallisesta piirtämisestä, joka tapahtuu aina Draw-tapahtumasta käsin, pintoihin voi piirtää missä tahansa tapahtumassa. Ennen kuin on mahdollista piirtää pintaan, se pitää asettaa se kohteeksi. Tämä tehdään komennolla surface_set_target(nimi);. Kun pinnalle on piirretty kaikki, mitä olisi tarkoitus piirtää, siitä pitää poistua komennolla surface_reset_target();. Kun piirretään pintaan, on tärkeää muistaa, että pinnat elävät omien koordinaattien mukaan. Ei ole väliä, missä kohtaa pintaa käsittelevä instanssi sijaitsee, pinnan 0,0-koordinaatti on absoluuttisesti kiinni vasemmassa ylänurkassa. Alempana on esitetty esimerkki, jossa on pinta, joka täytetään ensiksi punaisella ja sitten pinnan pisteeseen (50,50) piirretään sininen ympyrä.

```
surface_set_target(pinta);

draw_clear(c_red);
draw_set_colour(c_blue);
draw_circle(50, 50, 20, 1):

surface_reset_target();
```

Esimerkkikoodi 7: Piirtäminen pinnalle GML-skriptikielellä.

Vihdoinkin, jotta pinta näkyisi, se pitää piirtää Draw-tapahtumassa draw_surface(nimi, x, y); -komennolla.

Pinnat ovat niin sanottuja epävakaita eli volatiileja muuttujia. Tämä tarkoittaa sitä, että ohjelma saattaa poistaa muuttujan muistista viemästä turhaa tilaa, mikäli sen mielestä kyseistä muuttujaa ei enää tarvita. Kaksi yleisintä syytä ovat ohjelman kokonäyttötilan kytkeminen ja pois kytkeminen sekä ikkunan minimointi. Mikäli pinta katoaa muistista, eikä tilanteeseen ei ole varauduttu, ohjelma kaatuu.

Estääkseen kaatumisen aina kun koodissa viitataan pintaan pitää muistaa tarkistaa sen olemassaolo surface_exists(pinnan nimi) -funktiolla. Mikäli muuttujaa ei ole, se pitää alustaa uudestaan surface_create()-funtiolla.

Muistista poistetun pinnan sisältö katoaa olemattomiin. Vaikka ohjelma palauttaa pinnan, sen sisältöä ei saa takaisin. Mikäli kyseessä oli pinta, joka sisälsi staattista dataa, joka ei muutu, kuten vaikkapa vakioarvoinen taulukko, sen voi piirtää sen uusiksi, eikä kukaan huomaa mitään eroa.

Samaa lähestymistapaa voi hyödyntää pinnoissa, joiden sisältö piirretään joka hetki uusiksi välittämättä siitä, mikä oli siinä aikaisemmin. Tällaisiin tapauksiin voi lukea esimerkiksi varjot, joista puhutaan luvussa 4.3.3.

Ongelmia kuitenkin tulee, jos kyseessä on pinta, jonka ulkonäkö kehittyy dynaamisesti. Esimerkiksi pinnan avulla simuloidaan lunta, johon jää jalanjälkiä. Kun pinta luodaan uusiksi, kaikki olemassa olevat jalanjäljet katoaa. Hyvässä tapauksessa tämä tulee näyttämään hölmöltä, mutta huonossa tämä tulee tuhomaan pelin. Onneksi on olemassa mahdollisuus kopioida pinnan sisällön puskuriin, ja hakea sen sieltä tarpeen tullen buffer_set_surface() ja buffer_get_surface() -funktioiden avulla. Puskurit ovat itsessään melko massiivinen aihe, joten siihen ei paneuduta tällä kertaa sen enempää.

4.2 Pseudokolmiulotteisen grafiikan luonti

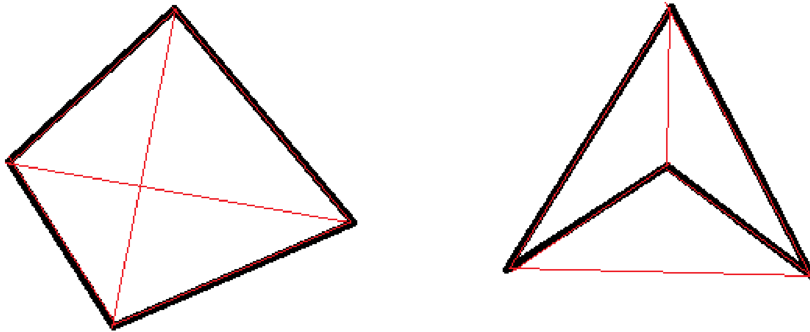
Tässä luvussa tutustutaan pseudokolmiulotteisen grafiikan luontiin. Pseudokomiulotteinen grafiikka tarkoittaa kaksiulotteista grafiikkaa, jossa tavalla tai toisella on saatu aikaan illuusio oikeasta kolmiulotteisuudesta.

4.2.1 Poikkileikkauspiirto

Yleisin tapa luoda ja tallentaa 3D-malleja on käyttää monikulmioita eli polygoneja. Polygoni on ryhmä pisteitä, jotka sijaitsevat kolmiulotteisessa avaruudessa ja joiden väliin on venytetty taso. Perinteisesti polygoni on joko neljän pisteen väliin venytetty nelikulmio tai vielä yleisemmin kolmesta pisteestä koostuva kolmio.

Polygonissa voi olla koostua enemmänkin kuin neljästä pisteestä, jolloin siitä käytetään yleisesti termiä ngon, mutta niiden käyttöä ei suositella, koska kaikki 3D-sovellukset eivät ymmärrä niitä.

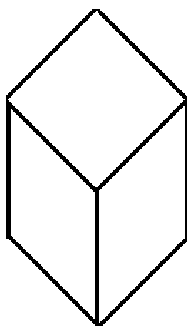
Ngonien suurin haaste on konveksi-periaatteen noudattaminen. Konveksi muoto on sellainen, jonka jokaisesta muodon nurkasta voi vetää suoran viivan niin, että viiva ei joudu poistumaan muodon ulkopuolelle. Asiaa on havainollistettu kuvassa 20.



Kuva 20: Esimerkki konveksi (Vasen) ja ei-konveksista, eli konkaavista polygonista (Oikea)

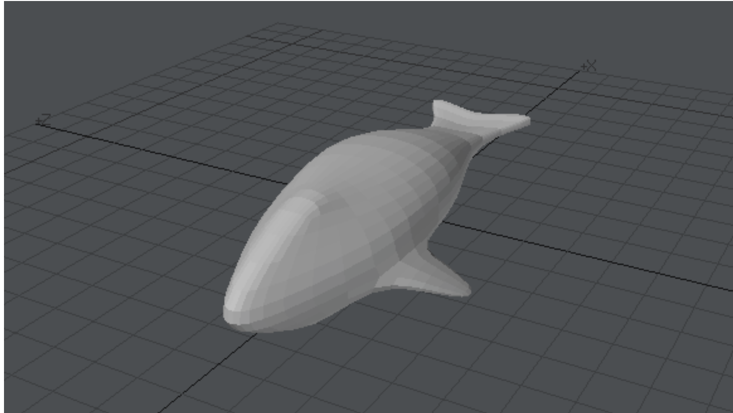
Käytännössä puhtaasti kaksiulotteisena moottorina GameMaker ei ymmärrä kolmiulotteisia polygonaaliseja malleja erityisemmin. Päätin tutkia, onko olemassa mitään mahdollisuutta simuloida monimutkaisia kolmiulotteisia muotoja 2D-ympäristössä.

Kuulin kerran, että kaksiulotteinen kuva on käytännössä poikkileikkauskerros kolmiulotteisesta objektista. Tämä herätti minussa idean. Jos kolmiulotteisesta mallista otetaan tasaisin välin poikkileikkausnäytteitä, ja piirretään ne päällekkäin niin, että jokainen seuraava kerros on hieman siirtynyt Y-akselilla, tulokseksi saadaan mukautetulla sotilasperspektiivillä (kuva 21) piirretty malli. Idea on hieman sama kuin magneettikuvauksessa. Nimesin tämän tekniikan poikkileikkauspiirroksi.



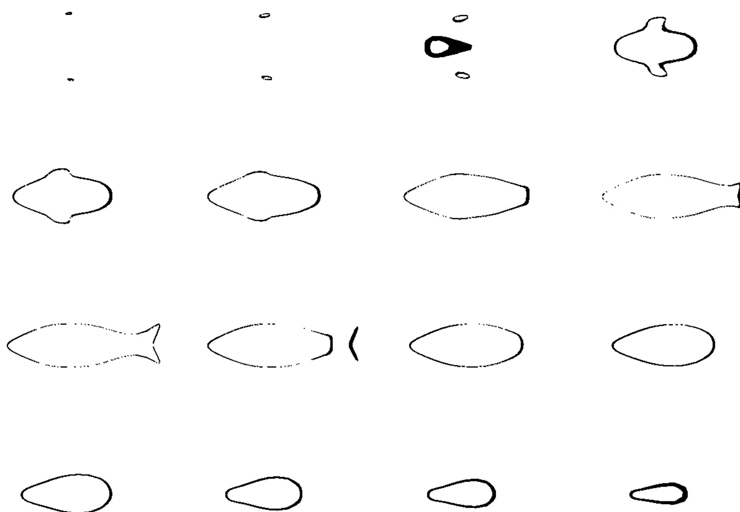
Kuva 21: Esimerkki sotilasperspektiivillä piirretystä suorakulmaisesta särmiöstä.

Tutustutaan kyseiseen ratkaisuun konkreettisen ratkaisun avulla. Käännetään kuvassa 22 oleva yksinkertainen valasta esittävä 3D-malli poikkileikkauspiirtomalliksi.



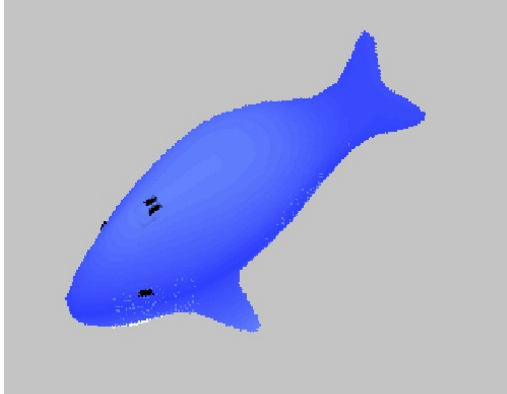
Kuva 22: Yksinkertainen valasta esittävä polygonaalinen malli

Suoritetaan poikkileikkauksiin jako. Tämä on melko manuaalinen prosessi, joka vaatii tarkkuutta. Käyttäen animoituja tekstuureja sain Lightwavesta ulos kutakuinkin selkeät poikkileikkauksen siluetit. Tekniikka on kaukana täydellisestä, eikä esimerkiksi toimi hyvin niissä kohdissa, jossa mallin seinät ovat Y-akselin suuntaisia. Silmämääräisesti arvioin, että 32 näytettä on sopiva määrä kyseisen mallin tapauksessa. Osa näistä kerroksista on havainollistettu kuvassa 23.



Kuva 23: Parilliset kerrokset kyseisestä muunnosprosessista. Tilan säästämiseksi parittomat kerrokset jätetty pois tästä kuvasta.

Tämän jälkeen silutetit pitää värittää niin kuin haluamme ja piirtää kaikki kerrokset päällekkäin toistolauseella. Tuloksena saadaan kolmiulotteisen vaikutelman luovan, parallaksin-perustuvat optisen illuusion, joka pystyy pyörimään Y-akselinsa ympäri (kuva 24).

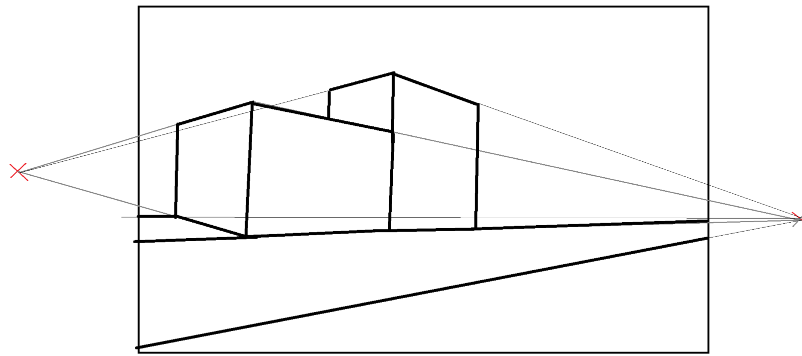


Kuva 24: Lopullinen tulos.

Still-kuvana kyseinen tekniikka ei näytä kummoiselta, mutta liikkeessä tulos on erittäin miellyttävän oloinen. Ratkaisun huonoiksi puoliksi voi laskea sen, että käännösprosessi on tällä hetkellä pitkälti manuaalinen, ja objekteissa ei voi olla animaatiota.

4.2.2 Pseudopolygonaaliset mallit

Kolmiulotteisessa kuvassa on kyse perspektiivistä. On mahdollista piirtää niin, että kuvasta välittyy syvyys, ja on mahdollistaa luoda sellaisia kolmiulotteisia malleja, jotka tuntuvat litteiltä. Riippuen, mistä perspektiivistä piirtää, aikaan saa hyvinkin erilaisia tuloksia, mutta idea on aina siinä, että kuvan samansuuntaiset viivat pyrkivät kohti samaa pistettä. Esimerkiksi kahden pisteen perspektiivin tapauksessa nämä pisteet sijaitsevat kuvan laidoilla (kuva 25) . Yhdenmukaisuuden nimissä pohditaan, miten voisi toteuttaa luvussa 4.2.1 esitellyn, sovelletun sotilasperspektiivin GameMakersita käsin.



Kuva 25: Esimerkki kahden pisteen perspektiivistä

Klassisen sotilasperspektiivin tunnistaa siitä, että siinä X- ja Y-akseleiden välissä on 45 asteen kulma, mikä tarkoittaa, että XZ-akseleilla oleva taso on piirretty ilman vääristymiä. Tämä taso on myös käännetty 45 asteella. Y-akselilla olevat viivat piirretään pystysuoraan sen pituisina kuin ne on oikeasti [18].

Sovellettu, eli minun käyttämä sotilasperspektiivi, eroaa hieman perinteisestä. Koska objektit voivat pyöriä Y-akselinsa ympäri, unohdetaan, että XZ-tason pitää olla käännetty 45 asteen kulmaan. Tässä yhteydessä tämä kulma saa olla sitä, mitä tilanne kulloinkin vaatii. Myös se, että Y-akselilla olevat viivat on piirretty 1:1 mittakaavalla on valinnaista, sillä kokeilujen seurauksena tulin tulokseen, että 3:4-suhde näyttää esteettisesti miellyttävämmältä. Tuloksena saadaan perspektiivi, joka on isometrisen- ja sotilasperspektiivin risteytys.

Klassisessa sotilasperspektiivissä riittää, että ohjelma selvittää Pythagoran lauseen avulla nelikulmion lävistäjän pituuden, ja sen tiedon avulla on mahdollista piirtää kaikki pisteet. Kuitenkin, koska pyörivä kuvio voi olla missä kulmassa tahansa, kulmien sijainnin joutuu selvittämään trigonometristen funktioiden avulla.

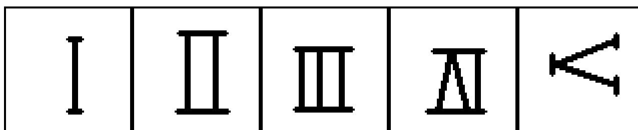
GameMaker tuntee kaikki yleisimmät trigonometriset funktiot: niin tavalliset, kuin käänteiset. Vaikka niiden käyttö on helppoa, niiden avulla kulmien koordinaattien selvittäminen vaatii matemaattisten kaavojen kirjoittamista, mikä voi tehdä koodista monimutkaisemman lukea. Onneksi GameMakerissa on lendir_x()- ja lendir_y(), funktiot, jotka kertovat, mikä piste sijaitsee halutussa suunnassa, halutun etäisyyden päässä. Nämä funktiot ottavat parametrikseen etäisyyden ja suunnan. Lisäksi, koska

oletuksilla funktiot laskevat kaikki uudet koordinaatit pisteestä (0,0), yleensä tulokseen pitää lisätä nykyisen objektin X- ja Y-muuttujien nykyiset arvot.

Trigonometriset funktiot yleensä tuottavat desimaaliarvoisia vastauksia, joten kaikki tulokset olisi hyvä pyöristää, jotta vältytään rasteroinnista johtuvasta laskostumisesta, eli aliasingista. Henkilökohtaisesti käytän floor()-funktiota, mutta myös ceil()-funktio toimii ihan yhtä hyvin,

Kun tarpeelliset pisteet on laskettu, pitää venyttää niiden väliin tasot, jotka luodaan spriteistä. Laatikon piirtämistä varten tarkoitetun esimerkkispriten näkee kuvasta 26.

GML sisältää lukuisia ns. edistysneitä piirtofunktiota, joiden avulla voi muokata näytölle piirrettävien spritejen ominaisuuksia. Edistyneisllä funktioilla voidaan mm. säätää läpinäkyvyys, koko ja väri. Tämän esimerkin kannalta kiinnostava on draw_sprite_pos()-funktio, joka venyttää annetun spriten, annettujen neljän pisteen väliin.



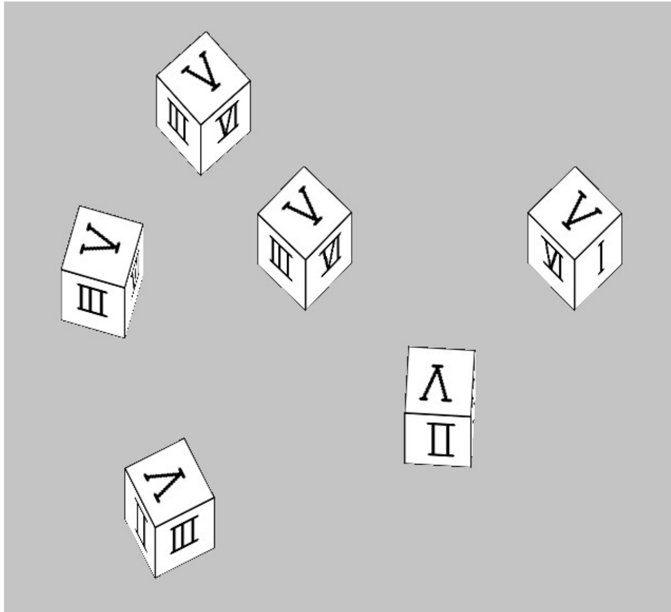
Kuva 26: Tyypillinen sprite, josta tuleva pseudopolygonaalinen kuutio tulee koostumaan. Vaikka kuutiolla on kuusi tahko, kuvia tarvitaan vain viisi, koska viimeistä ei ikinä näy.

On muistettava, että kun laatikko pyörii Y-akselinsa ympäri, katsoja kykenee näkemään siitä korkeintaan kolme tahkoa, joista päällimmäinen näkyy aina, eikä siihen tarvitse kiinnittää huomiota. Kirjoitetaan muutama ehto-lause, joilla valitaan, mitkä tahot piirretään missäkin kulmissa. GameMaker ei osaa hyödyntää pyörityksessä modulaarista aritmetiikkaa, eli vaikka 270 astetta ja -90 astetta on visuaalisesti identtisiä, GameMaker tulkitsee ne eri kulmista. Etenkin tämä tuottaa ongelmia, kun joudutaan käsittelemään alueita, joissa ylittyy 360 asteen kulma. Kuten seuraavasta esimerkkikoodista näkee, viimeisen kulman joutuu tarkistamaan kahdessa osassa, 270 – 360 ja 0 -90.

```
//Kuution kulmien kordinaatit
ylaoikea_x = x + floor(lengthdir_x(45, direction + 45));
ylaoikea_y = y + floor(lengthdir_y(45, direction + 45));
ylavasen_x = x + floor(lengthdir_x(45, direction + 135));
ylavasen_y = y + floor(lengthdir_y(45, direction + 135));
alavasen_x = x + floor(lengthdir_x(45, direction + 225));
alavasen_y = y + floor(lengthdir_y(45, direction + 225));
alaoikea_x = x + floor(lengthdir_x(45, direction + 315));
alaoikea_y = y + floor(lengthdir_y(45, direction + 315));

//Kuution tahkojen piirto
if (direction >= 0 and direction < 180)
          draw_sprite_pos(kuutio, 1, ylavasen_x,
          ylavasen_y, alavasen_x, alavasen_y, alavasen_x,
          alavasen_y - 64, ylavasen_x, ylavasen_y - 64, 1);

if (direction >= 90 and direction < 270)
          draw_sprite_pos(kuutio, 0, ylaoikea_x,
          ylaoikea_y, ylavasen_x, ylavasen_y, ylavasen_x,
          ylavasen_y - 64, ylaoikea_x, ylaoikea_y - 64, 1);

if (direction >= 180 and direction < 360)
          draw_sprite_pos(kuutio, 3, alaoikea_x,
          alaoikea_y, ylaoikea_x, ylaoikea_y, ylaoikea_x,
          ylaoikea_y - 64, alaoikea_x, alaoikea_y - 64, 1);

if ((direction >= 270 and direction < 360) or (direction >=
0 and direction < 90))
          draw_sprite_pos(kuutio, 2, alavasen_x,
          alavasen_y, alaoikea_x, alaoikea_y, alaoikea_x,
          alaoikea_y - 64, alavasen_x, alavasen_y - 64, 1);

//Piirrä kuution ylätahko
draw_sprite_pos(kuutio, 4, ylaoikea_x, ylaoikea_y - 64,
ylavasen_x, ylavasen_y - 64,alavasen_x, alavasen_y - 64,
alaoikea_x, alaoikea_y - 64, 1);
```

Esimerkkikoodi 8: Kuution piirtäminen GML-skriptikielellä.

Lopputuloksena saadaan kuutio, jossa on viisi selkeästi tunnistettavaa tahkoa ja joka pystyy pyörittämään vapaasti Y-akselin ympäri. Havainnollistaakseen, että jokainen kuutio on uniikki instanssi, tein pienen ohjelman, joka käskee kaikkia ruudulla olevia kuutioita katsomaan kohti pistettä, jossa hiiren pitäisi olla (kuva 27).



Kuva 27: Kohti samaa pistettä osoittavat kuutiot.

Mikäli jaksaa nähdä vaivaa ja piirtää spritet, joissa on hienoa grafiikkaa, voi saada erittäin vahvan kolmiulotteisen vaikutelman tekevät objektit. Ratkaisulla on kuitenkin huonoja puoliaan. Kuten myös poikkileikkauspiirrossa tässäkin pyöriminen on rajoitettu Y-akseliin. Lisäksi, vaikka on mahdollista luoda muotoja, jotka ovat paljon monimutkaisempia kuin kuutio, oikean piirtojärjetyksen löytäminen tulee viemään aikansa.

4.2.3 Esirenderöinti

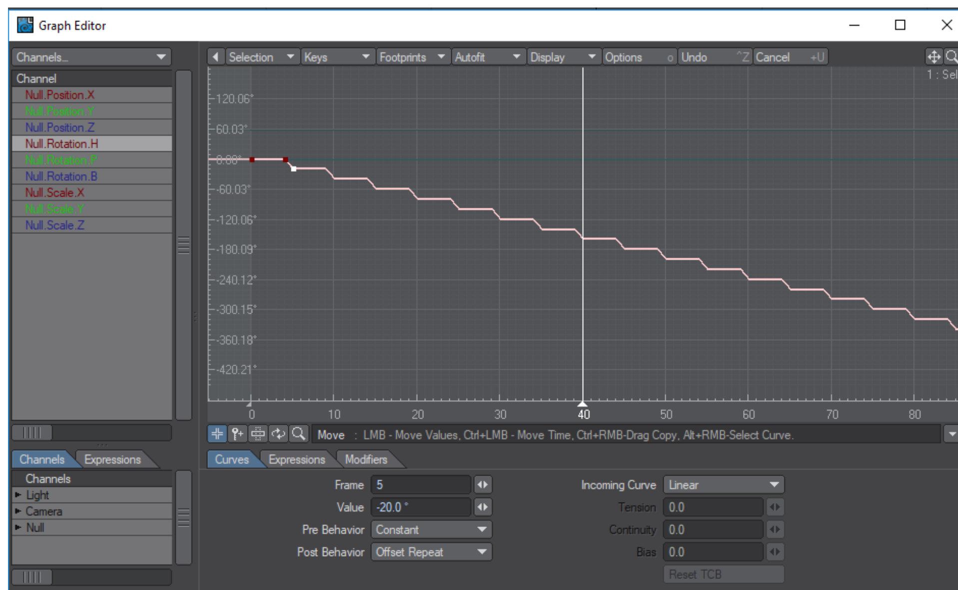
Esirenderöinti on tekniikka, jossa näytölle luotavaa kuvaa ei renderöidä reaaliajassa, vaan kaikki laskut on jo suoritettu joskus aikaisemmin, ja lopputulos tallennettu helposti toistettavaan muotoon, joka ei vie tehoja juuri yhtään. Esirenderöintiä voi hyödyntää osittain, jolloin esimerkiksi tausta on esirenderöity, mutta edessä olevat objektit on laskettu reaaliajassa tai kokonaan, jolloin kaikki, mikä kuvassa näkyy, on esirenderöityä. Esimerkkinä tunnetusta esirenderöidystä pelistä toimii Donkey Kong Country, joka on esitetty kuvassa 28.

Kuva 28: Kuvankaappaus Donkey Kong Country -pelistä SNES-konsolilla, jossa kaikki grafiikka on esirenderöity [19]

Esirenderöinti on todennäköisesti vanhin tapa käyttää monimutkaista kolmiulotteista grafiikkaa siellä, missä se ei ole tuettua. Sen avulla voi saada ihmeellisiä tuloksia aikaan, mutta sen käyttö vaatii tarkkaa suunnittelua. Jokainen esirenderöity kuva vie oman tilansa kovalevyltä ja muistista, joten mitään ei kannata renderöidä, jos sille ei ole käyttöä.

Kuvitellaan, että tavoitteena on piirtää näytölle esirenderöinnin avulla noppa, joka vapaasti pyörisi Y-akselinsa ympäri. Oletetaan, että tavoitteena on lopputulos, jossa voisi pyörittää noppaa mahdollisimman sulavasti. Kuten on jo sanottu, mitä sulavamman lopputuoksen haluaa, sitä tiheämpää näytteenottoa tarvitaan. Pelataan varman päälle ja otetaan kuva yhden asteen tiheydellä. Lopputuloksena syntyy 360 kuvaa. Tämän ei pitäisi olla haaste millekään nykyiselle kuluttajatason koneelle.

Lightwaven avulla pystyy automatisoimaan prosessia. Käyttäjän ei tarvitse kuin animoida yksi sykli ja kertoa ohjelmalle, että sen pitää jatkaa samaan malliin. Kuitenkin, jokaisen syklin jälkeen, kameraa on käännettävä tietyllä määrällä asteita. Tämä onnistuu Graph Editor -työkalulla, jolla pystyy hallitsemaan animaatiota diagrammien avulla (kuva 29).



Kuva 29: Graph Editorilla luotu animaatio, jossa kohde kääntyy 20 astetta Y-akselilla viiden askelen välein

Monimutkaistetaan tehtävää. Kuvitellaan, että saman nopan pitää pyöriä myös X-akselinsa ympäri. Oletetaan, että myös X-akselin ympäri tapahtuvan liikkeen on oltava yhtä sulavaa kuin Y-akseni ympäri tapahtuva, joten siitäkin otetaan näytteitä yhden asteen tiheydellä.

Aina kun noppa kääntyy yhdellä asteella Y-akselilla, Lightwave jää odottamaan, että kuutio tekisi täyden kierroksen X-akselillaan. 360 astetta Y-akselilla kertaa 360 astetta Y-akselilla tuottaa 129600 kuvaa. Älykkäällä kuvien vaiheittaisella latauksella kyseisen ratkaisun ehkä saisi toimimaan, mutta epäilemättä tämä alkaa käymään jo erittäin raskaaksi.

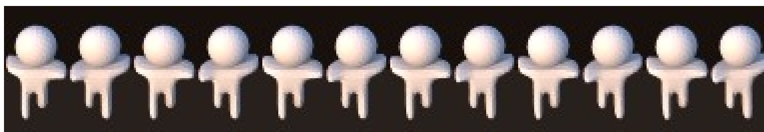
Mikäli mukaan otetaan vielä pyöriminen Z-akselin ympäri samalla laadulla, niin tuloksena olisi 46656000 kuvan massiivinen kokonaisuus. Esimerkiksi käyttämäni Lightwave 10 ei edes osaa luoda noin pitkiä aikajanoja, joilla tuollaista kuvien määrää voisi animoida.

On myös tärkeää muistaa, että kyseinen noppa on staattinen objekti, joka ei sisällä mitään animaatiota. Jos nopalla olisi jotain muuta animaatiota kuin pyöriminen jokaisen asteen jälkeen, jokaisella akselilla, joutuisi esirenderöimään myös tämän animaation. Rehellisesti sanoen tässä liikutaan jo sellaisessa esirenderöinnin määrässä, että on

pakko siirtyä aitoon kolmiulotteiseen grafiikkaan. Tässä valossa teen johtopäätöksen, että esirenderöinti soveltuu parhaiten tilanteisiin, jossa objekti pyörii korkeintaan yhden akselin ympäri.

Tässä vaiheessa ongelmaksi koostuu, miten saada GameMakerin piirtämään oikean spriten kuvan milloinkin. Jos kyseessä on 360 kuvan sprite, jossa ei ole kävelyanimaatiota, voidaan hyödyntää image_single-sisäänrakennettua muuttujaa, jolla pystyy valitsemaan spritestä tietyn kuva. Käytetään komentoa image_single = direction;.

Mikäli kyseessä on animaatiolla varustettu sprite, myös animaation joutuu ottamaan huomioon. Tässä tapauksessa image_singlen joudutaan asettamaan kahdessa osassa: suunta, johon objekti osoittaa, ja kyseisen animaatiosyklin kuva. Kuvassa 30 on esitetty katkelma siitä, millaisista kuvista koostu esirenderöity 3D-malli, joka voi liikkua 360 suuntaan, kahden kuvan pituisen animaation avulla. Todellisuudessa lopputulos on 720 kuvan kokoinen, ja vie noin 5 megabitiiä kovalevytilaa.



Kuva 30: Katkelma esirenderöidystä yksinkertaisen tikku-ukkon 3D-mallista.

4.3 Valaistustekniikoiden simulointi

Tässä luvussa tutustutaan erilaisten valon ominaispiirteiden, kuten varjojen ja heijastusten simuloimiseen.

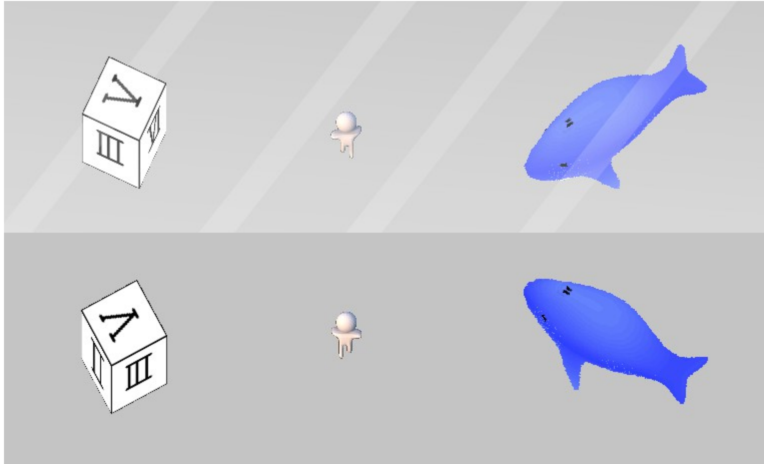
4.3.1 Heijastukset

Kuten huomataan myöhemmin luvussa 4.3.2, käytännössä kaikki pinnat heijastavat valoa tavalla tai toisella. Sitä ilmiötä, jota arkikielessä mielletään heijastukseksi, jossa yksi esine näkyy toisen esineen pinnassa, sanotaan kokonaisheijastukseksi.

Kun tavoitteena on fotorealismi, kolmiulotteisessa grafiikassa heijastuksia lasketaan ray tracingin, eli säteen seurannan avulla. Säteen seuranta on monimutkainen ja laskennallisesti raskas tekniikka, jota harva nykykone jaksaa laskea reaaliajassa, etenkin, jos tavoitteena on laadukas kuva. Tästä syystä valtaosan ajasta, kun puhutaan heijastuksista tietokonepeleistä, puhumme valeheijastuksista.

Tekniikka, jota tullaan hyödyntämään, on käytössä niin kaksiulotteisissa, kuin kolmiulotteisissa peleissä, ja se perustuu siihen, että pelikentästä luodaan peilikopio, joka pistetään peilin taakse. Tämä luonnollisesti vaatii kaksi kertaa enemmän komentoja suorituksessa, mutta se on silti murto-osa siitä, mitä oikea säteen seuranta vaatisi.

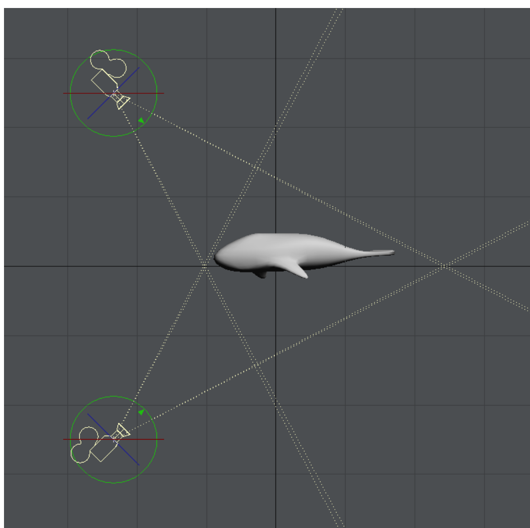
Tässä kaksiulotteisessa pelissä heijastus voi tapahtua joko seinille, jotka ovat kohtisuoraan kohti kameraa, tai lattialle. Seinälle heijastuminen voi simuloida esimerkiksi peiliä, kun taas lattialle heijastuminen voi olla esimerkiksi jääpintaa. Seinälle heijastaminen ei vaadi juuri mitään erikoisjärjestelyjä, joten sitä ei käydä yhtään sen tarkemmin läpi. Ainoa, mitä pitää tehdä, on kääntää objekti Y-akselilla vastapäivää objektin suunnan verran. Eli näin esimerkiksi, jos objekti osoittaa suuntaan 23, niin heijastus osoittaa suuntaan -23 astetta. Seinästä heijastumista on havannollistettu kuvassa 31.



Kuva 31: Peilaava seinä, josta heijastuu kaikki kolme esiteltyä objekti-tyyppiä. Jotta peili tuntuisi peililtä, sen päälle on piirretty puoliläpinäky kuva, joka simuloi sitä, miten peili heijastaa häikäisyjä. Heijastus on toteutettu pinnan avulla.

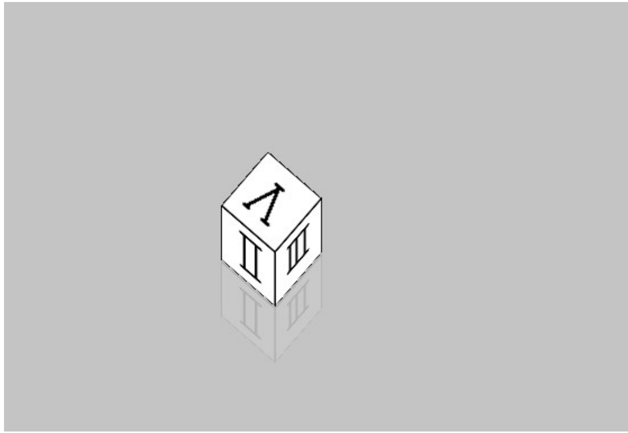
Idealtaan lattialle heijastuminen ei paljoa eroa seinään heijastumisesta. Nyt kuitenkaan ei enää riitä, että objekti käännetään vastakkaiseen suuntaan, koska lattiaheijastuksessa on aivan eri perspektiivi. Jokainen grafiikkatyyppi vaatii hieman erilaisen lähestymistavan.

Esirenderöidyt objektit ovat ikävimpiä lattiaheijastuksen kannalta. Koska perspektiivi on leivottu kuviin, ainoa mahdollisuus toteuttaa lattiaheijastus on renderöidä se omaksi tiedostokseen. Esirenderöinnissä käyttäen toista kameraa, joka on alkuperäisen kameran peilikuva (kuva 32).



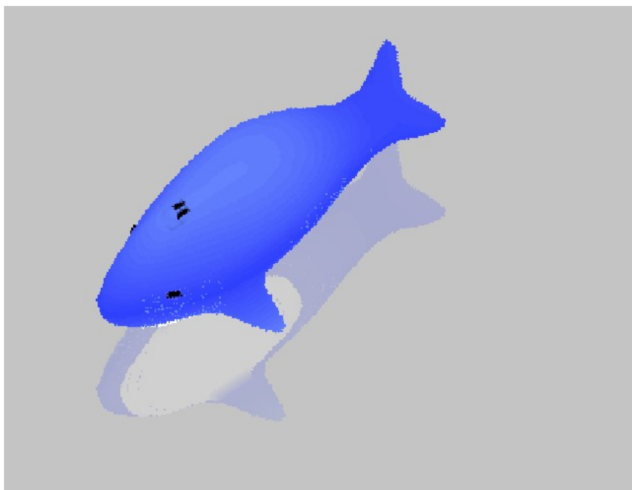
Kuva 32: Kameroiden sijainti, kun on tarve renderöidä myös lattiaheijastus

Pseudopolygonaaliset mallit ovat vaivattomimpia lattiaheijastuksen suhteen. Pitää vain ottaa sivulla 32 näkyvä koodi ja muuttaa sitä siten, että joka kohdassa, jossa korkeus-muuttuja vähennetään, nyt se päinvastoin lisätään. Pseudopolygonaalisen mallin lattiaheijastuksen voi nähdä kuvasta 33.



Kuva 33: Pseudopolygonaalisen mallin lattiaheijastus

Poikkileikkausmalleissa heijastus on tekniseltä toteutukseltaan hyvin samanlainen kuin pseudopolygonaalisissa malleissa, mutta muutos vaatii, että kaikki kerrokset piirretään käänteisessä järjestyksessä. Esimerkki tästä tekniikasta näkyy kuvassa 34.

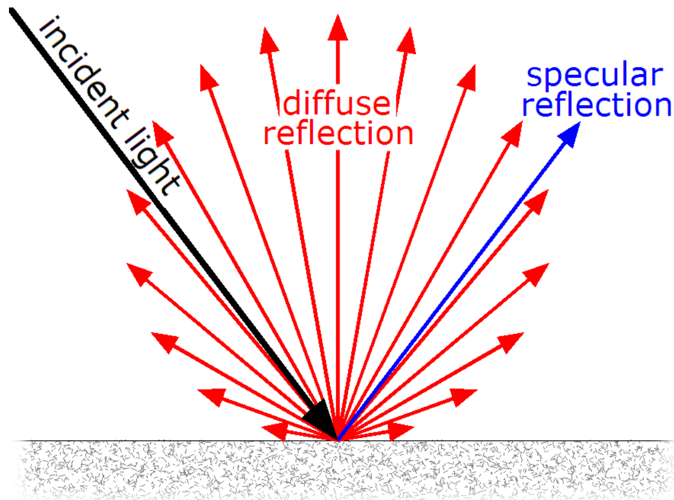


Kuva 34: Poikkileikkausmallin lattiaheijastus

4.3.2 Epäsuora valaistus

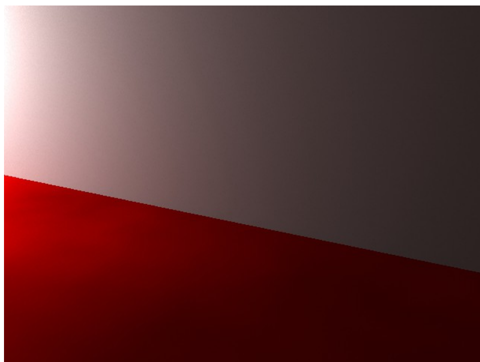
Ihminen tarvitsee nähdäkseen valoa. Valonsäteet lähtevät valonlähteestä ja kimpoavat pinnasta ihmisen silmään. Valonsäde ei kuitenkaan heijastu sellaisenaan vaan hajoaa

eri suuntiin pienempinä valonsäteinä (kuva 35). Tätä ilmiötä sanotaan diffuusi heijastukseksi [20]. Kuten saattaa jo arvata, tällä tavalla valtaosa valosta heijastuu lähellä oleviin pintoihin, ja prosessi jatkuu niin kauan, kunnes valon energia loppuu.



Kuva 35: Diffuusi heijastus [21]

Jokaisella pinnalla on olemassa valon aallonpituuksia, joita se absorboi, ja sellaisia, joita se heijastaa. Käytännössä tämä tarkoittaa sitä, että pinnasta heijastunut valo muuttuu pinnan väriseksi, ja näin valaisee uudella värillään kaikki seuraavat pinnat (kuva 36).



Kuva 36: Epäsuoraa valaistuta havainnollistava renderöity kuva. Valkoinen valo kimpoaa punaisesta lattiasta, ja valaisee valkoisen seinän punertavalla värillä

Koska epäsuora valaistus perustuu myös säteiden kimpoamiseen, sillä on hyvin paljon samaa edellisessä kappaleessa esitetyn heijastuksen kanssa. Epäsuora valaistus on kuitenkin vielä raskaampi; jos heijastuksessa jokainen säde pysyy yhtenä säteenä,

tässä jokainen säde hajoaa moneksi säteeksi, jotka puolestaan taas hajoavat moneksi säteeksi. Mitä tarkempaa ja realistisempaa tulosta haetaan, sitä useamman kerran säteiden annetaan heijastua ja hajota. Laskennan määrä kasvaa rekursiivisesti jokaisella kierroksella.

Epäsuora valaistus on niin raskasta, että jopa siellä, missä on mahdollisuus esirenderöidä, kuten elokuvissa, sitä on yritetty välttää ja ainoastaan luoda illuusiota siitä. Esimerksiksi animaatiostudio Pixar, joka julkaisi ensimmäisen täyspitkän elokuvansa, Toy Story, vuonna 1995, otti epäsuoran valaistuksen käyttöön vasta vuonna 2003 elokuvassa Autot. [22]

Tuntien GameMakerin heikon 3D-tuen ja olemattoman kyvyn optimoida resursseja, olisi turhaa odottaa, että se pystyisi natiivisti toteuttamaan epäsuoran valaistuksen. Minun piti keksiä tapa, jolla saisin aikaan visuaalisesti miellyttävä simuloitu epäsuora valaistus, joka ei kuitenkaan ylikuormittaisi muuta peliä.

Ratkaisu löytyi lightwrap-tekniikasta, jota käytetään yleisesti elokuva- ja televisiopuolella green screenissa ja muissa vastaavissa tekniikoissa. Tekniikkaa on havainollistettu kuvassa 37. Kun tarkoituksena on yhdistää tausta, ja kohde, jotka on kuvattu eri paikoissa, vaikka kuvien värit saataisiin täsmättyä identtisiksi, lopputulos silti näyttää hieman luonnottomalta, koska taustan valaistus ei millään tapaa vaikuta kohteeseen. Lightwrapissa luodaan etualan muotoinen ja taustan värinen aura, joka liitetään kohteeseen. Tämä luo illuusion siitä, että tausta valaisee kohdetta reunoilta.

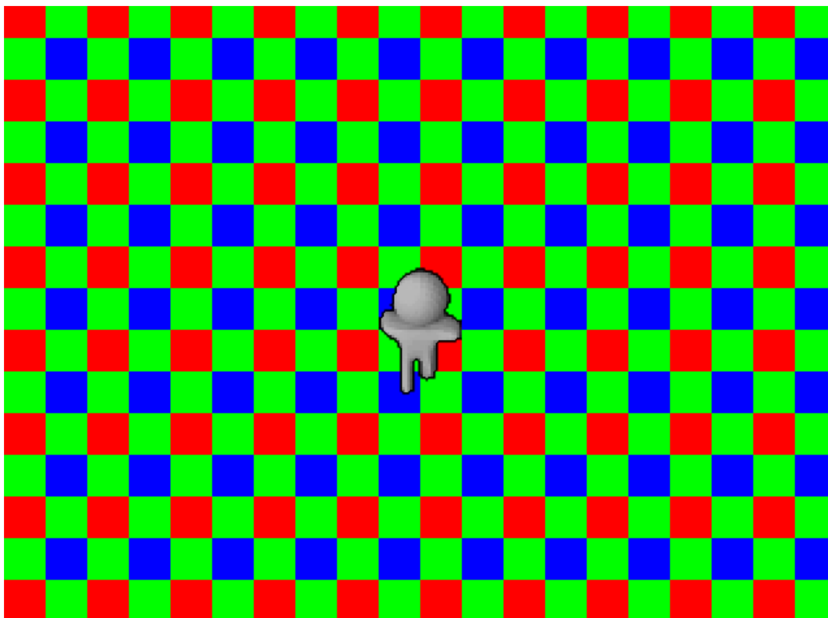
Kuva 37: Esimerkki green screenista, jossa on hyödynnetty lightwrap-tekniikkaa. Kuvassa on esitetty kaikki komponentit erikseen ja yhdessä. Viimeisessä kuvassa,

jossa näkyy kokonaiuus, miehen värejä on korjailtu, jotta kokonaisuus näyttäisi luonnollisemmalta [23; 24].

Oikeassa maailmassa valoa tulee yleensä hyvin monesta suunnasta. Jos tavoitteeksi otetaan simuloida kaikkia niitä, luultavasti ajaudutaan tilanteeseen, missä joutusi tekemään runsaasti työtä, joka vain nimellisesti parantaisi lopputulosta, mutta söisi liikaa tehoa. Tämän työn tarkoitukseen riittää, että simuloidaan vain kirkkainta valoa ja vain niihin pintoihin, jotka selkeästi näkyvät.

Kirkkain ja suurin valonlähde, joka maapallolta löytyy, on aurinko. Auringonvalo tulee ylhäältä, joten valtaosan ajasta se heijastuu maasta. Valtaosan ajasta pelaaja näkee kaikki objektit edestä. Toimiakseen ohjelma vaatii ratkaisun, miten saada maassa olevat sävyt ”heijastettua” objektien etualalle.

Käydään vaihe vaiheelta läpi, miten tämä tulee käytännössä toimimaan. Alempana on esitetty kuva, jossa vaaleanharmaa esirenderöity hahmo seisoo Bayer-kuvioisella lattialla. On helppoa huomata, että lattia ei heijastu mitenkään hahmoon.



Kuva 38: Epäsuoran valaistuksen toteuttavan esimerkin lähtöasema.

Tässä hyödynnetään blend modeja eli sekoitustiloja. Sekoitustilat kertovat ohjelmalle, miten päällekkäin olevat pikselit piirretään. Esimerkiksi, jos käytössä on Add (Lisäys) -tila, ohjelma ottaa päällekkäin olevien pikselien RGB-arvot, ja laskee ne yhteen. Näin

esimerkksi, jos päällekkäin on kaksi pikseliä, joista ensimmäinen on vihreä (RGB-arvo 0, 255, 0), sekä sininen (RGB-arvo 0, 0, 255), ja ne piirretään päällekkäin, niin tulokseksi saadaan syaanin värinen pikseli (RGB-arvo 0,255, 255).

Oletuksena kaikki kerrokset piirretään normal-sekoitustilaa, joka GameMakerissa tunnetaan nimellä bm_normal. Tässä tilassa pikselit piirretään päällekkäin niin, että päällimmäinen korvaa alimmaisen. Sekoitustila vaihdetaan käyttäen draw_set_blend_mode(tyyppi) -komentoa. GameMakeriin on sisäänrakennettu lukuisia sekoitustiloja, jotka löytyvät bm_ -alkuisissa vakioissa. Mikäli sisäänrakennetut sekoitustilat loppuvat, on mahdollista luoda myös omia sekoitustiloja hyödyntäen draw_set_blend_mode_ext(tyyppi1, tyyppi2) -komentoa.

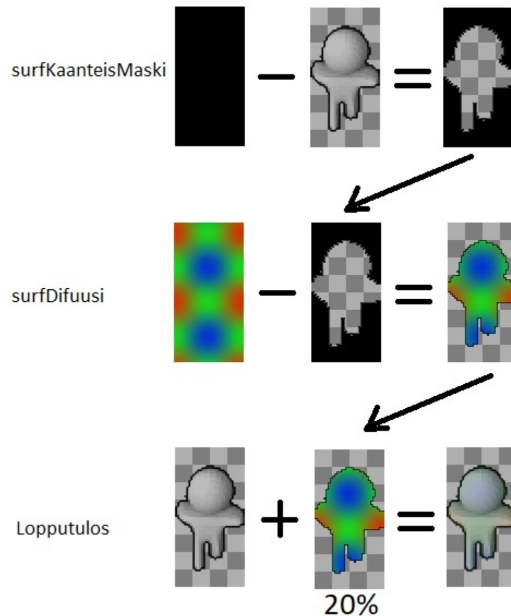
Sen jälkeen, kun toiminto, jossa tarvittiin sekoitustilaa, on suoritettu, pitää muistaa palauttaa sekoitustila normaaliksi. Muussa tapauksessa ohjelma tulee piirtämään kaiken valitulla sekoitustilalla myös jatkossa.

Poiketen valtaosasta kuvanmuokkaussovelluksista, GameMakerin sekoitustilat toimivat hieman omalaatuisesti. Jokaisella pikselillä on RGB-arvon lisäksi A-arvo, eli alpha, eli läpinäkyvyys. Tämä mahdollistaa sen, että sekoitustilojen avulla voidaan tehdä myös ns. maskeja. Alempana esitetään bm_substract -sekoitustilaa hyödyntävä esimerkki, jossa koko ruutu täytetään mustalla värillä, ja siitä leikataan 100 pikselin säteen omaava ympyrä.

```
draw_clear(c_black);
draw_set_blend_mode(bm_subtract);
draw_circle(100, 100, 100, 0);
draw_set_bland_mode(bm_normal);
```

Esimerkkikoodi 9: GML-skriptikeielen bm-subtract sekoitustilan käyttö.

Tekniikassa on tarkoitus leikata taustasta hahmon muotoinen pala ja piirtää se hahmon päälle niin, että hahmo saa taustalla olevia sävyjä itseensä. Asiaa on havainnollisetettu kuvassa 39.



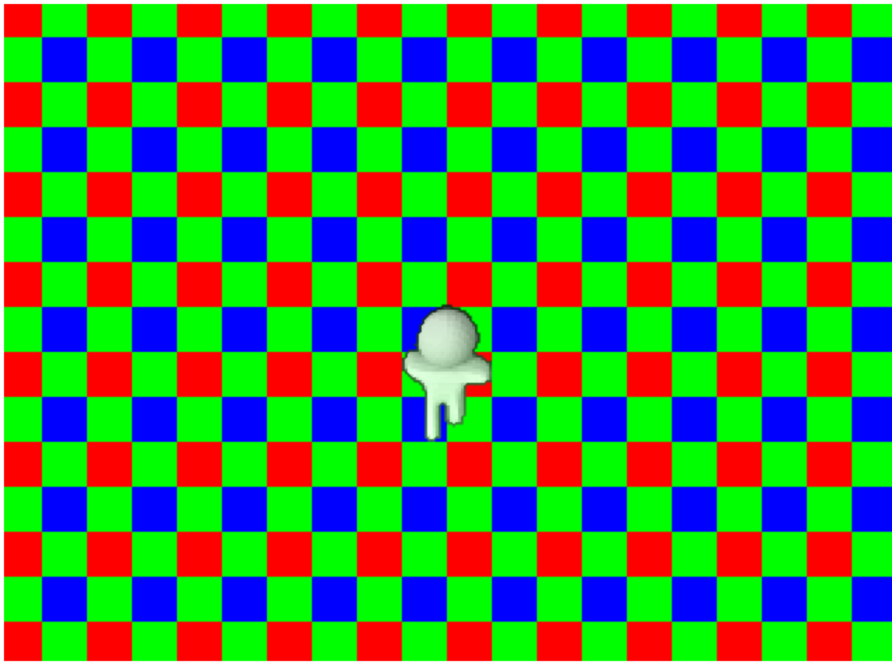
Kuva 39: Simuloidun epäsuoran valaistuksen muodostuminen kerroksista. Tehty kuvanmuokkausohjelmalla, mutta periaate on identtinen.

Jokaiseen objektiin, johon halutaan, että epäsuora valaistus kohdistuu, luodaan alkuun kaksi pintaa, jotka nimetään surfDifuusiksi ja surfKaanteisMaskiksi. Niiden kooksi laitetaan sama kuin mikä on muun objektin koko. SurfDifuusi tulee sisältämään epäsuoran valaistuksen.

Koska GameMakerissa ei ole sekoitustilaa, joka automaattisesti leikkaisi pois kaiken, paitsi seuraavaksi piirrettävän asian muodon, joudutaan tekemään ensiksi negatiivisen maskin (surfKaanteisMaski), josta vähennämme halutun muodon. Sitten vähennämme tämän negatiivisen maskin lopullisesta kuvasta.

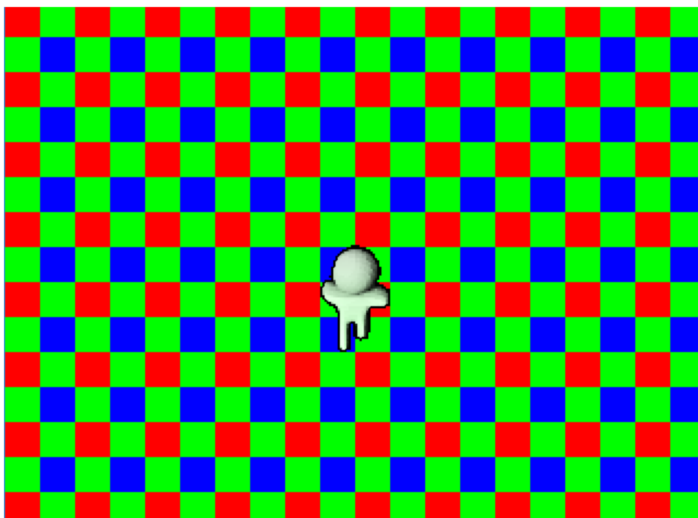
Tavoitteena on, että epäsuora valaistus tehoaa ensisijaisesti objektin alalaitaan. Tiedetään, että koska bm_add-sekoitustila laskee arvoja yhteen, jos päällimmäisenä oleva kerros on musta, mitään ei tapahdu, koska musta on RGB-arvoltaan (0,0,0). Täten, kun piirretään surfDifuusi-kerrosta, halutaan, että sen yläreunat olisivat mustia. Tämän voi toteuttaa monella tapaa, mutta itse päädyin hyödyntämään draw_rectangle_colour() -komentoa, jolla pystyy piirtämään neliön, jolla kaikki reunat ovat erivärisiä.

Lopputulokseksi saadaan kuva 40.



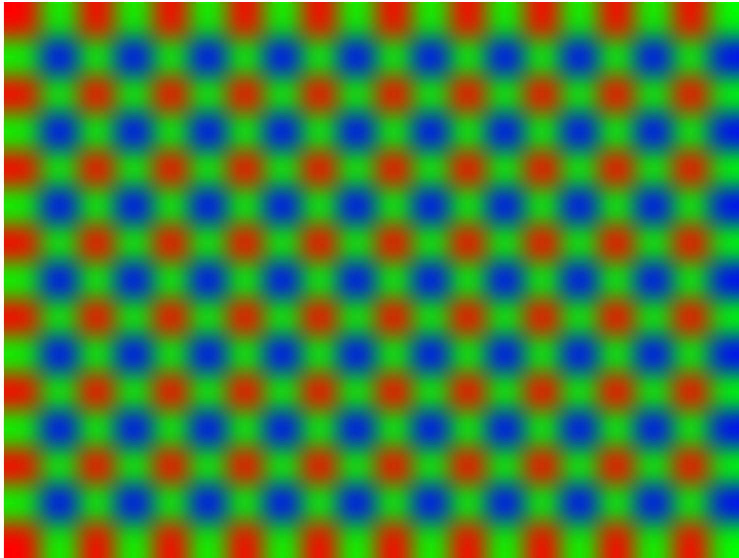
Kuva 40: Epäsuoran valaistuksen toteuttavan esimerkin lopputulos

Lopputulos on hyvännäköinen, mutta tarkka katsoja saattaa huomata, että bm_add -sekoitustilan käyttö aiheutti sen, että myös hahmon ääriviivat, joiden voisi olettaa olevan mustia, vaalenivat (Kuva 41). Kompensoidakseen tämän, piirretään ääriviivat uusiksi. Ääriviivat on toteutettu irrottamalla kuvan tummat osat omaksi tiedostokseen After Effects -ohjelmistossa.



Kuva 41: Epäsuoran valaistuksen toteuttavan esimerkin lopputulos, jossa on uudelleenpiirretyt ääriviivat.

Kun lähtee hakemaan väriä taustasta, kannattaa käyttää sitä varten erikseen luotua taustakuvaa, joka on pehmennetty (kuva 42). Tämä auttaa välttämään tilanteen, jossa esimerkiksi keskellä punaista taustaa on yksi vihreä pikseli, josta otetaan värinäyte ja lopputuloksena saadaan kuva, joka on valaistu sen ainoan pikselin värillä.



Kuva 42: Esimerkki pehmennetystä taustakuvasta, josta väriarvo haetaan.

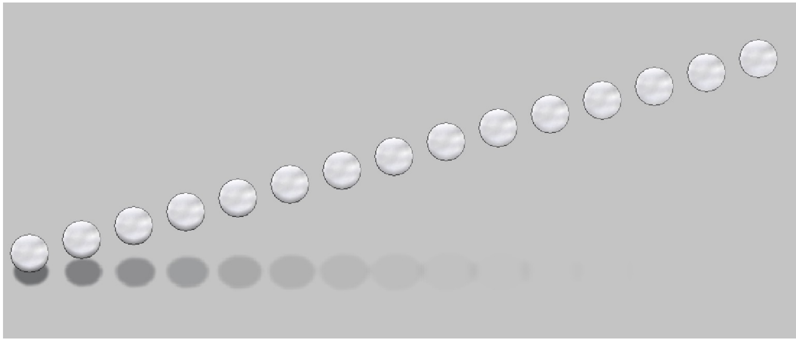
Epäsuoran valaistuksen tekniikkaa voi käyttää niin paikallaan seisovien kuin liikkuvienkin objektien kanssa. Tekniikka on hieman raskas, joten mikäli kyseessä on staattinen objekti, joka ei liiku, sen epäsuora valaistus kannattaa piirtää heti alussa, ja olla koskematta siihen.

4.3.3 Varjot

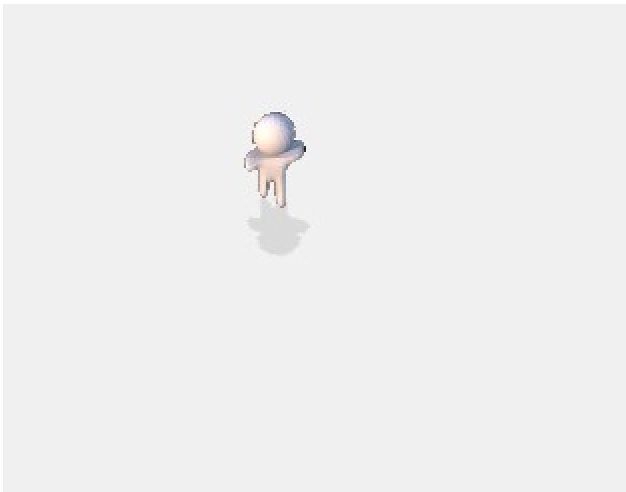
Varjot ovat käytännössä valon puutteen seurausta. Varjot ovat niin yleisiä meidän elämässä, että niiden puute tekee kuvasta miltei aina epärealistisen. Toki on olemassa tilanteita, joissa valo tulee vain ja ainoastaan erittäin pehmeistä valonlähteistä, mikä aiheuttaa sen, että valoa tulee kaikista suunnista tasaisesti, jolloin varjot eivät ole selkeitä, mutta ne ovat silti olemassa. Tutustutaan muutamaan keinoon luoda varjoja GameMakerissa.

Aluksi tutustutaan, miten voi hyödyntää varjomaskeja. Kuten myös heijastukset ja epäsuora valaistus, aidot varjot perustuvat säteenseurantaan, joten etenkin vanhoissa peleissä oli tapana tehdä valevarjot. Varjomaski on hyvin klassinen tapa luoda valevarjoja. Parhaiten se toimii silloin, kun tavoitteena on simuloida varjoja, jotka syntyvät globaalista valonlähteestä, kuten auringosta.

Varjomaskien idea on siinä, että varjot esirenderöidään. Sen sijaan, että ohjelma seuraisi säteistä, miten ja mihin suuntaan varjon pitäisi tippua, varjo piirretään valmiina olevasta kuvasta. Tällä tavalla voidaan piirtää niin niitä varjoja, jotka syntyvät valonlähteestä, joka on suoraan pään yläpuolellla (kuva 43) kuin jossain muuallakin (kuva 44).

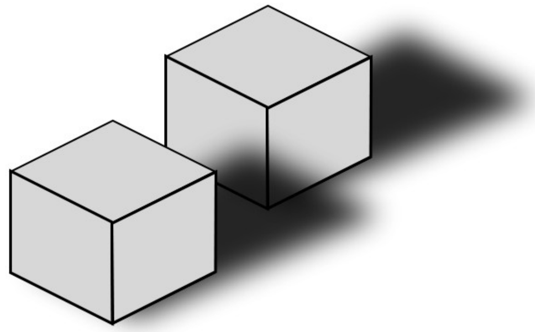


Kuva 43: Esimerkki varjomaskeista, jotka syntyvät valonlähteestä, joka on suoraan yläpuolella. On mahdollista myös säätää varjojen voimakkuutta. Tässä tapauksessa, mitä kauempana maasta pallo on, sitä pehmeämmän ja laajemman varjon se muodostaa. Huomioidaan, että varjoja muodostuu niin maahan kuin itse palloonkin



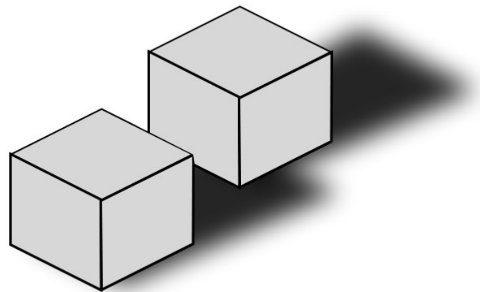
Kuva 44: Esimerkki varjomaskista, jossa valo tulee jostain kaukaisuudesta horisontin yläpuolelta. Kuvasta ei sitä näy, mutta varjo kääntyy poispäin ”valonlähteestä”.

Varjomaskeilla on olemassa yksi puute: ne eivät osaa sopeutua automaattisesti vastaantulevien objektien muotoihin. Havainnollistetaan tilannetta kuvalla 45.



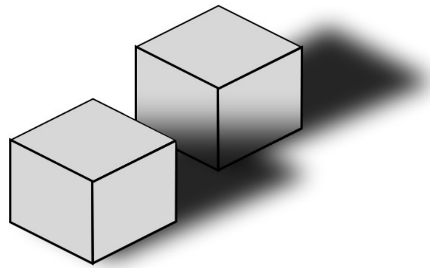
Kuva 45: Edessä olevan kuution varjo tippuu sellaisenaan taaimmaisen kuution päälle. Joskus tämä saattaa toimia, mutta yleensä ei.

On mahdollista tehdä niin, että varjomaskit piirretään omaan pintaan, mikä piirretään aina alimmaiseksi. Tämä kuitenkin johtaa siihen, että varjo ei vaikuta mitenkään taustalla olevaan objektiin, kuten kuvassa 46.



Kuva 46: Edessä oleva kuutio ei heitä enää väärännäköistä varjoa taaimmaiseen laatikkoon. Ongelma on siinä, että se ei heitä enää mitään varjoa taaimmaiseen laatikkoon, ja syntyy tunne, että laatikot leijuvat.

Kuitenkin se, että varjot piirretään omalle kerrokselle, mahdollisti sen, että nyt on mahdollista johdattaa siitä pystyvarjot. Metodiikka on aivan samanlainen kuin epäsuorassa valaistuksessa. Valon sijaan piirretään varjoja. Lopputulos näkyy kuvassa 47.

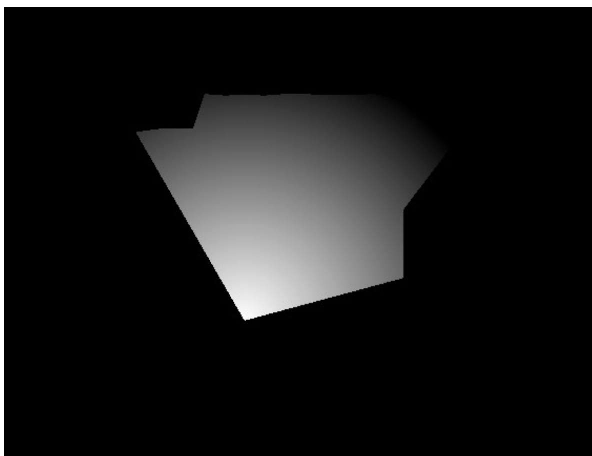


Kuva 47: Kutakuinkin realistisesti käyttäytyvät valevarjot.

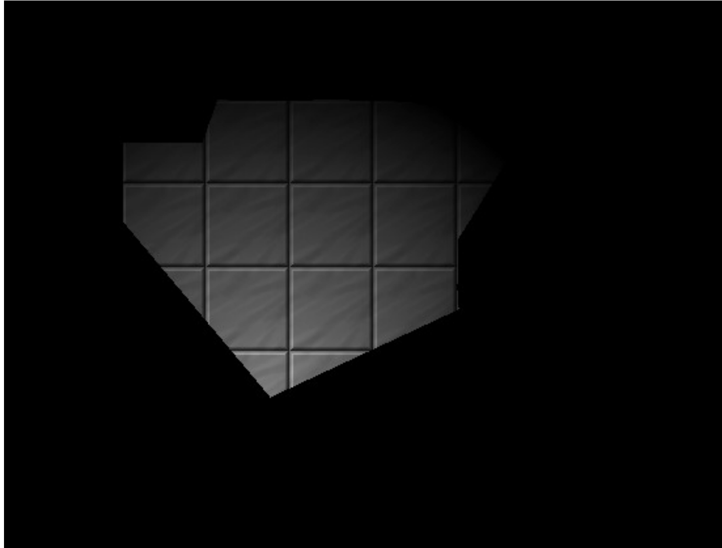
Mitä jos kuitenkin vastaan tulee tilanne, jossa valoa on vähän, ja valtaosa kuvasta on varjon peitossa?

Varjot ovat paljon helpompia laskea kuin heijastukset, ja varsinkin epäsuora valaistus. Tästä johtuen on mahdollista toteuttaa yksinkertainen, kaksiulotteinen säteenjäljitys hyödyntäen luvussa 4.1.1 esitettyjä metodeja.

Koska jokaisen säteen erillinen piirtäminen olisi laskennallisesti raskasta, lasketaan vain muutama säde ja piirretään niiden väliin polygoneja. Mitä pidempi polygoni syntyy, sitä tummempi sen loppu tulee olemaan. Tämä simuloi sitä, miten valo heikkenee matkan varrella. Nämä säteet piirretään omalle pinnalle (kuva 48). Kun tämä pinta piirretään multiply-sekoitustilassa, tulokseksi saadaan ylhäältä kuvattu näkymä, jossa ovat realistiset varjot (kuva 49).

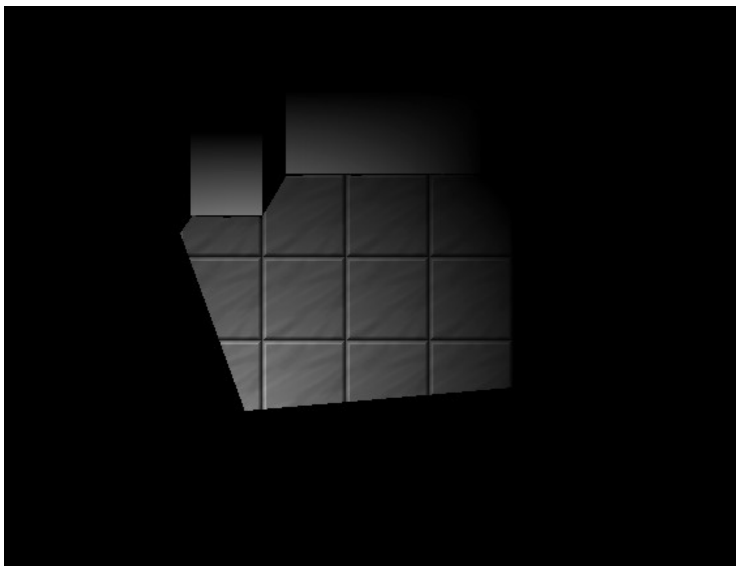


Kuva 48: ”Valokartta”, joka kertoo, mitkä kohdat tulevat olemaan varjossa.



Kuva 49: Valokartalla korjattu näkymä.

Luonnollisesti tässäkin tekniikassa voidaan varjostaa seiniä samalla tavalla kuten aikaisemmin. Lopputuloksena saadaan kuvan 50 mukainen näkymä, jota voi miltei kutsua realistiseksi.



Kuva 50: Kaksiulotteisella säteenjäljityksellä varustettu näkymä, jossa varjot muodostuvat myös seinälle.

5 Pelidemo

Tässä luvussa tutustutaan pieneen pelidemoon, joka on laadittu hyödyntäen luvussa 4 esitettyjä ratakaisuja.

5.1 Idea

Koska työ lähtökohtaisesti perustuu siihen, että jollekin vanhalle yritetään antaa uusi elämä, järkevältä tuntu, että myös lopputuloksena syntynyt peli viittaa menneisyyteen ja nostalgiaan.

Kun olin ala-asteella, varsinkin ensimmäisellä luokalla, yksi harrastuksista, joita meidän luokan pojat harrastivat, oli kerääntyä porukalla saman A4-arkin ympäri ja piirtää siihen toiminnallisia töherryksiä kertoen tarinaa. Yhtenä päivänä joku saattoi piirtää auton ja sanoa, että tällä autolla ryhmä rosvoja vie varastettua kultaa. Vastauksena siihen, joku saattoi piirtää poliisiautoa, ja kertoa, että poliisipartio lähtee perään. Kolmas poika saattoi isoilla punaisilla ja oransseilla tusseilla sotkea poliisiauton mössöksi, koska rikollisilla oli sinko, ja auto räjähti.

Lopputuloksena syntyi aina melkoinen tussien, värikynien ja liitujen sekamelska, jota oli mahdotonta ymmärtää, ellei ollut seuraamassa taideteoksen syntyä alusta alkaen. Poikkeuksetta lopputulos oli aina myös erittäin mielikuvituksekas, ja mielenkiintoinen tutkia.

Yllä mainittujen ryhmäpiirustuksen henkeä olisi hyvin vaikeaa välittää missään pelissä täydellisesti, mutta ajattelin, että paperille piirretyn näköinen peli voisi olla esteettisesti miellyttävä, ja hauska konsepti.

Peligenreksi valitsin niinsanotun twin-stick-shooterin, eli pelin, jossa voi liikkua ja ampua 360-asteen suunnissa. Syynä tähän on se, että juuri tuollaiset pelit olivat suosiossa ala-asteella.

Teknisiltä määrityksiltään ainoat vaatimukset pelille ovat, että sen pitää näyttää esteettisesti miellyttävältä, ja pyöriä sulavasti 720p-resoluutiolla 60 kuvaa sekunnissa, koska tämä on absoluuttinen minimi, jonka jälkeen peliä voidaan sanoa moderniksi.

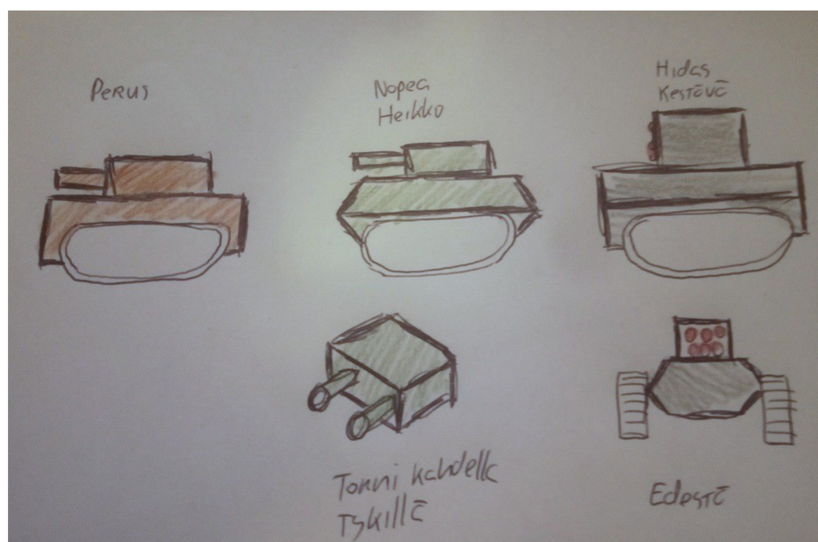
5.2 Pelin suunnittelu

Tässä luvussa tutustutaan pelidemon taidepuoleen.

5.2.1 Pelihahmot

Ensimmäinen resurssien rajallisuudesta seurannut ratkaisu oli tehdä kaikista pelin objekteista kulkuneuvoja ja robotteja, sillä niitä on helpompaa piirtää. Alkuperäinen idea oli tehdä pelin hahmoista 80-luvun elokuvien tyylisiä, rambomaisia supersotilaita, mutta omat piirtotaidot osoittautuivat liian rajallisiksi, enkä löytänyt mistään graafikkoa.

Alkuperäisessä suunnitelmassa oli luoda peliin kolme pelattavaa panssarivaunua, mutta ajan puutteen vuoksi päätin tehdä vain yhden. Alkuperäinen konseptitaide kolmesta suunnittelusta panssarivaunista löytyy alempana olevasta kuvasta numero 51.



Kuva 51: Panssarivaunujen konseptitaide

Panssarivaunu on toteutettu yhdistämällä erilaisia piirtotekniikoita. Telaketjut on esirenderöity, mutta itse runko on pseudopolygonaalinen malli. Tykki puolestaan on toteutettu poikkileikkauspiirrolla.

5.2.2 Pelihahmojen ohjaus

Koska pelattavat hahmot ovat panssarivaunuja, ohjattavuudessa piti tuntua raskaus, mutta kokonaisuuden piti olla silti intuitiivinen ja helppo ohjata. Tuloksena syntyi skripti,

joka osaa päätellä, mihin suuntaan panssarivaunun runko katsoo, ja sen mukaan valitsee, kannattaako kääntyä myötä vai vastapäivän, jos suunta muuttuu.

Immersion parantamiseksi panssarivaunun ohjaus on toteutuett Xbox 360 -ohjaimella ja analogisilla tateilla.

Parantaakseen pelin objektin vuorovaikutusta objekteissa oli tarkoitus käyttää Box2D -fysiikoita, jotka olisivat mahdollistaneet realistisemmat törmäykset, mutta ajan puutteen vuoksi ideasta luovuttiin.

5.2.3 Kentät

Johtuen siitä, että GameMakerissa on huono resurssien optimointi, tuli päätetyksi, että yhdessäkään kentässä ei saa olla kaikkia monimutkaisia piirtotekniikoita.

Ainut piirtotekniikka, joka kulkee kaikkien kenttien läpi, on epäsuora valaistus. Säteenjäjitysvarjot ja heijastukset ovat kuitenkin liian raskaita samanaikaiseen luotettavaan toteutukseen, joten tasoissa aina käytetään vain jompaa kumpaa.

Lopputloksena syntyi kolme kenttää. Ensimmäinen on jääkenttä, jossa esitellään lattiaheijastukset ja surfacejen avulla lumeen jäävät jäljet (kuva 52). Toinen on yökenttä, joka sijoittuu asfalttipinnalle, ja sisältää yksinkertaisen tulidynamiikan toteutuksen. Viimeinen kenttä on ruohokenttä, jossa ei ole mitään erikoista, mutta hieman enemmän vihollisia.

Kuva 52: Kuvankaappaus ruohokentältä.

6 Yhteenveto

Työn aikana on kehitetty ja hyödynnetty lukuisia tekniikoita, joilla graafisesti yksinkertainen moottori saatiin näyttämään paremmalta. Valtaosa näistä tekniikoista liittyi valaistuksen simuloitiin ja keskittyi varjoihin ja heijastuksiin.

Lopullista demoa tehdessä joutui jatkuvasti miettimään, kuinka monta mitäkin objektia voi laittaa kentällä. Tavoitteena oli saada tilanne, jossa peli ei hidastelisi, mutta olisi kuitenkin dynaamisen tuntuinen.

Kokonaisuudessaan voidaan kuitenkin sanoa, että lopputyö oli onnistunut kokonaisvaltaisesti. Tarkoituksena oli tutkia, miten paljon GameMakeria pystyy venyttämään, ja asiassa on myös pääosin onnistuttu. Jotkut konseptit jouduin hylkäämään ennen kuin edes pääsin kirjoittamaan niistä, koska ne vaativat aivan liikaa laskentatehoja, mutta kaikki, mikä raportissa mainitaan, toimii niin kuin pitää.

Tehdessäni työtä, halusin saada vastauksen, olisiko minun sittenkin korkea aika vaihtaa Unityyn. Tulin siihen lopputulokseen, että vaikka Unity automatisoisi monia prosesseja, joita jouduin simuiloimaan tässä, niin silti tulen pysymään GameMakerissa. Syynä tähän on se, että koen, että hauskin osa pelienkehityksessä on keksiä luovia ratkaisuja.

Mikä tulee lopulliseen pelidemoon, niin koen, että se onnistui niin hyvin, että saatan jatkaa sen kehittämistä koulun ulkopuolella. Mielessä olisi ainakin äänipuolen lisäys, lisätä kenttiä ja pelimoodeja, kuten mahdollinen moninpeli. Mikäli jatkokehitys toteutuu, olisi tärkeää saada mukaan parempi graafikko. Linkki videoon, jossa näkyy raportissa kuvattu pelidemo löytyy liitteestä 2.

7 Lähteet

Bateman, Chris. 2014. Meet Bertie the Brain, the world's first arcade game, built in Toronto. Verkkoaineisto. Spacing Maagazine. <http://spacing.ca/toronto/2014/08/13/meet-bertie-brain-worlds-first-arcade-game-built-toronto/>. Luettu 30.10.2017.

The First Video Game?. Verkkoaineisto. Brookhaven National Laboratory. <https://www.bnl.gov/about/history/firstvideo.php>. Luettu 30.10.2017.

Great Video Game Crash of 1983. Verkkoaineisto. Bugsplat. <https://www.bugsplat.com/great-video-game-crash-1983>. Luettu 30.10.2017.

Essential Facts about The Computer and Video Game Industry. 2015. Verkkoaineisto. <http://www.theesa.com/wp-content/uploads/2015/04/ESA-Essential-Facts-2015.pdf>. Luettu 30.10.2017.

Suvanto, Soili. 2014. Peliteollisuus on jo miljardibisnestä Suomessa. Verkkoaineisto. Yle Uutiset. <https://yle.fi/uutiset/3-7169571> 3.4.2014. Luettu 30.10.2017.

Pokemon Blue. 1996. Tietokonepeli. Game Freak.

Day, Collin. 2014. Unity 3D. Verkkoaineisto. Spellbound Studios. <https://spellbindstudios.com/2014/09/30/unity-3d/> 30.9.2014. Luettu 30.10.2017.

Overmars, Mark. 2010. Curriculum Vitae. <https://web.archive.org/web/20111002054918/http://people.cs.uu.nl/markov/curric.pdf>. 22.6.2010. Luettu 30.10.2017.

Mark Overmars talks about Game Maker's past and future. 2009. Verkkoaineisto. GameMaker Blog. <http://gamemakerblog.com/2009/11/19/mark-overmars-talks-about-game-makers-past-and-future/>19.11.2009. Luettu 30.10.2017.

Tibbetrs, Ben. Verkkoaineisto. <http://bentibbetts.net/animo/>. Luettu 30.10.2017.

Overmars, Mark. 2007. Yoyo Games, Our new company. Verkkoaineisto. Wayback Machine. <https://web.archive.org/web/20070203064832/http://forums.gamemaker.nl/index.php?showtopic=271847>. 26.1.2007. Luettu 30.10.2017.

Spalding, Shaun. 2016. Introducing GameMaker Studio 2. Verkkoaineisto. Yoyo Games. <https://www.yoyogames.com/blog/397/introducing-gamemaker-studio-2>. 2.11.2016. Luettu 6.11.2017.

Smart, Gavin. 2017. GameMaker Studio 1.4 Sunset. Verkkoaineisto. Yoyo Games. <https://www.yoyogames.com/blog/436/gamemaker-studio-1-4-sunset>. 8.6.2016. Luettu 6.11.2017.

Mark Overmars talks about Game Maker's past and future. 2009. Verkkoaineisto. GameMaker Blog. <http://gamemakerblog.com/2009/11/19/mark-overmars-talks-about-game-makers-past-and-future/>. 19.11.2009. Luettu 6.11.2017.

Rogkas, Sakis, 2011. Mythology. Verkkoaineisto. Games 25. <http://sakis25games.blogspot.fi/2011/06/mythology.html>. 27.6.2011. Luettu 6.11.2017.

God of War 2. 2007. Tietokonepeli. SIE Santa Monica Studio.

"Iron Sky" Conquers the World, One Theater at a Time. 2012. Verkkoaineisto. NewTek. <https://www.lightwave3d.com/news/article/iron-sky-conquering-the-world-one-theater-at-a-time/>. 17.4.2012. Luettu 6.11.2017.

Axonometric projections - a technical overview. 2011. Compu Phase. Verkkoaineisto. <https://www.compuphase.com/axometr.htm>. Päivitetty 23.5.2017. Luettu 6.11.2017.

Donkey Kong Contry. 1994. Tietokonepeli. Rareware.

OpenGL/Sanasto. Verkkoaineisto. <https://fi.wikibooks.org/wiki/OpenGL/Sanasto>. Luettu 6.11.2017.

Diffuse Reflection. Kuva. <https://en.wikipedia.org/wiki/Diffuse_reflection#/media/File:Lambert2.gif>. By GianniG46 - Own work, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=11902338.

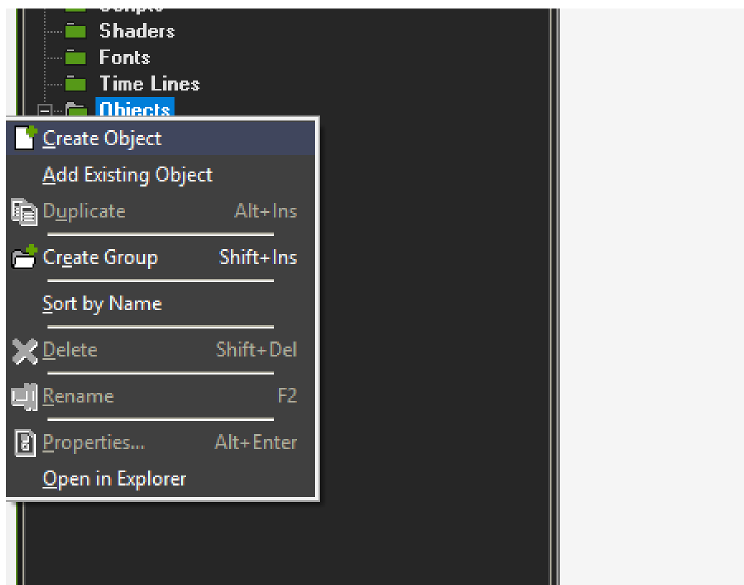
Christensen, Per; Fong, Julian; Laur, David; Batali, Dana. 2006. Ray Tracing for the Movie 'Cars'. Verkkoaineisto. Pixar. <https://graphics.pixar.com/library/RayTracingCars/paper.pdf>. 20.9.2006. Luettu 6.11.2017.

Green screen -materiaali. <http://tubularinsights.com/wp-content/uploads/2013/09/key-green-screen.jpg>.

Tarumaa. 2017. Valokuva.

Hello World -ohjelman luonti GameMaker-ympäristössä

Esimerkiksi halumme luoda ikkunan, joka kirjoittaa hiiren kohdalle perinteikkään Hello World! -tekstin. Tätä varten käyttäjän ei tarvitse kuin luoda peliprojekti, johon luodaan yksi huone (Room), joka on GameMakerin ikkunan perusyksikkö, ja yksi objekti (Object), joka on GameMakerin vastine luokalle. Kumpikin onnistuu klikkaamalla vastaavaa kansioita hiiren oikealla näppäimellä käyttöliittymän vasemmalla puolella ja valitsemalla pudotusvalikosta Create object/room (kuva 51).

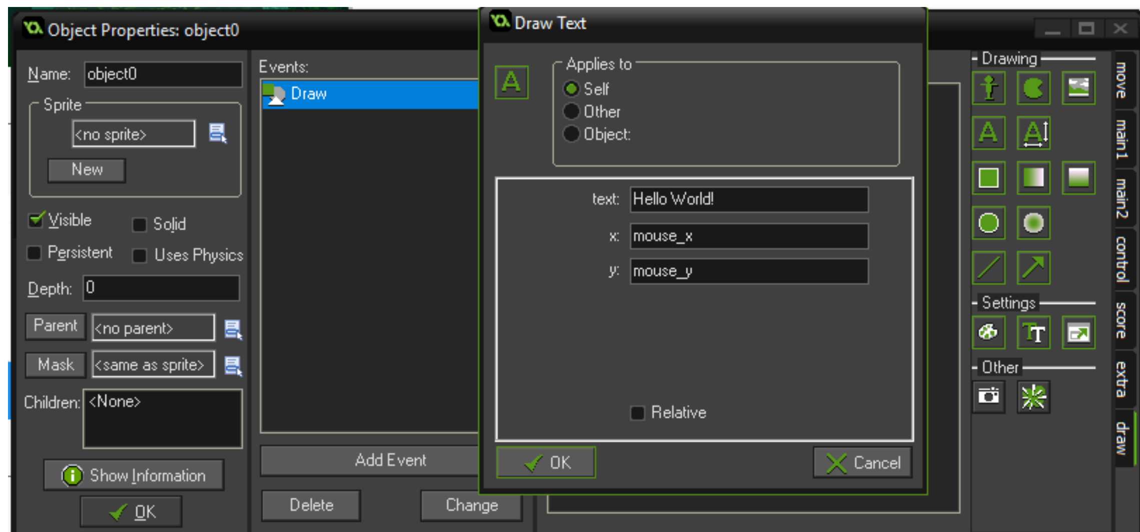


Kuva 51: Uuden objektin lisäys

Tämän jälkeen päästään "ohjelmoimaan". GameMakerissa on sekä drag-and-drop-puoli, että skriptauspuoli. Jälkimmäinen on luonnollisesti monipuolisempi, mutta ensimmäiselläkin pärjää pitkälle.

Avataan object0-objekti klikkaamalla sitä kahdesti. Avautuu Object Properties -ikkuna, jossa päästään syöttämään ohjeita. GameMaker on tapahtumapohjainen mootori. Klikataan Add Event -nappia, ja valitaan Draw-tapahtuma, joka vastaa luonnollisesti näytölle piirtämisestä.

Tämän jälkeen siirrytään draw-välilehteen, ja klikataan Draw Text -ikonia. Tämä avaa Draw Text -komennon parametri-ikkunan, johon text kohtaan syötetään haluttu viesti, ja x ja y kenttiin syötetään mouse_x ja mouse_y, jotka ovat sisäänrakennettuja read-only muuttujia, joissa sijaitsevat hiiren koordinaatit ikkunassa. Klikataan ok. (kuva 52)



Kuva 52: Object Properteies -näkymä ja Draw Text -parametri-ikkuna

Tämän jälkeen avataan aikaisemmin luotu huone, joka on nimetty oletuksellisesti room0. Avautuneesta Room Properties -ikkunasta valitaan objects-välilehti. Object to add with left mouse -tekstin alla sijaitsevasa valikosta valitaan object0, ja klikataan oikealla olevaa ruudukkoa, joka esittää meidän huonetta. Ruudukko on tällä hetkellä tyhjä, mutta klikkauksen jälkeen, klikkauksen kohtaan ilmestyy sininen pallo, jossa on punainen kysymysmerkki. Tämä on object0-objektin instanssi, eli eräänlainen olion vastine. GameMaker merkitsee kaikki objektit, joilla ei ole oletusspriteä, sinisenä pallona. Tämä pallo ei tule näkymään, kun ajamme ohjelman.

Nyt, kun käyttäjä painaa F5, GameMaker kääntää ja ajaa äskettäin luodun pelin. Tämä peli ei välttämättä ole vaikuttava itsessään, mutta ottaen huomioon, miten nopeasti se syntyi, se on erittäin hyvä näyte GameMaker-alustan ketteryydestä.

Pelidemon pelikuvavideo

Jotta myös niillä, jotka eivät olleet läsnä pelidemon esittelytilaisuudessa, olisi mahdollisuus nähdä peli liikkeessä, nauhoitin pienen pelidemon, jossa näkyy pelin kulku ja visuaalinen ilme.

Linkki trailerin Youtube-osoitteeseen:
https://www.youtube.com/watch?v=birpFYIUcJI