OULUN AMMATTIKORKEAKOULU

GAMEMAKER: STUDION KÄYTTÖ PROSEDURAALISESTI GENEROIDUSSA 2D-VIDEOPELISSÄ

Karl Sartorisio
Opinnäytetyö
Syksy 2016
Tietojenkäsittelyn koulutusohjelma
Oulun ammattikorkeakoulu

TIIVISTELMÄ

Oulun ammattikorkeakoulu
Tietojenkäsittely, Web-sovelluskehitys

Tekijä: Karl Sartorisio
Opinnäytetyön nimi: GameMaker: Studion käyttö proseduraalisesti generoidussa 2D-pelissä
Työn ohjaaja: Teppo Räisänen
Työn valmistumislukukausi ja -vuosi: Syksy 2016 Sivumäärä: 32

Tämän opinnäytetyön toimeksiantajana on Super God Oy. Työn tarkoituksena oli ohjelmoida yrityksen ensimmäistä peliä, proseduraalisesti generoitua lähitaistelupainotteista 2D-luolastotasoloikkapeliä Riptalea. Roolinani oli työryhmän pääohjelmoija ja työnkuvana oli täyspäiväinen peliohjelmointi. Työnkuvani kattoi muunmuassa pelin suunnitelman mukaisten mekaniikkojen toteutus GameMaker: Studiolla, vanhan koodin refaktorointia ja osallistuin myös pelin suunnitteluun. Opinnäytetyön kirjoitushetkellä peli ei ole vielä valmis, mutta se on suunniteltu julkaistavaksi vuoden 2017 ensimmäisen neljänneksen aikana julkaisualusta Steamissa.

Kehittämiini mekaniikkoihin kuului laajalti suunnittelu ja toteutus proseduraalisesta kenttägeneroinnista, kustomoidusta objektipohjaisesta partikkelijärjestelmästä, tekstilaatikko -ja kauppamekaniikoista, pelihahmon dynaamisesta huivista sekä pelihahmon hyökkäyksistä. Tämän lisäksi olen refaktoroinut olemassa olevaa osikseen varsin kömpelösti toteutettua koodia. Olen myös luonut peliin graafisen HUDin, mekaniikan kuvan täristämiselle, luonut useita partikkeliefektejä GameMaker: Studion omalla partikkelijärjestelmällä ja omakehittämällä järjestelmälläni. Lisäksi olen hionut jo olemassa olleita ominaisuuksia kuten pelihahmon liikkumista, fysiikkaominaisuuksia sekä vihollishahmojen käyttäytymistä.

Lähes kaikki laatimani mekaniikat ja ominaisuudet olen itse suunnitellut päättelyn ja kokeilun perusteella, eli ne on tehty sen mukaan, mikä on todettu toimivaksi ratkaisuksi pelin toiminnan ja ohjelman toimivuuden kannalta. Ainoina ulkoisina referensseinä olen käyttänyt GameMaker: Studion dokumentointia opetellakseni ky. ohjelmiston kykenevyydet ja YouTube-videoa, jonka pohjalta on laadittu tekstilaatikkomekaniikka.

Asiasanat: GameMaker: Studio, peliohjelmointi, proseduraalinen generointi

ABSTRACT

Oulu University of Applied Sciences
Degree Programme in Business Information Systems, Web Development

---

Author: Karl Sartorisio
Title of thesis: Usage of GameMaker: Studio In A Procedurally Generated 2D-Video Game
Supervisor: Teppo Räisänen
Term and year when the thesis was submitted: Autumn 2017 Number of pages: 32

The client of this thesis is Super God Oy. The purpose of this thesis is to develop the client's first game Riptale, a melee-oriented 2D-dungeon crawling game with procedurally generated content. As a part of the development team, my role was the lead programmer and my job description was full-time video game programming. The job description includes development of game mechanics using GameMaker: Studio based on game design, refactoring of older code and participating in designing the game. At the time of writing this thesis the game hasn't been completed yet, but it is planned to be released on publishing platform Steam during the first quarter of year 2017

The development of mechanics includes the planning and implementation of procedural level generation, customized object-based particle system, text box and shop mechanics and player character's attack mechanics. It also includes code refactoring on existing clumsily coded mechanics, creation of graphic HUD, a mechanic for screen shake effect, creation of a variety of particle effects using GameMaker: Studio's own particle system alongside the one I created. I also have done some polishing on other existing mechanics such as player character's movement, physics and enemy character behavior.

Almost all the mechanics I developed have been planned using rationalization and trial and error, thus they have been developed on basis of what works in practice, what works in gameplay's terms and overall good programming practice. The only outside references I have used are GameMaker: Studio's documentation for learning the program's possibilities and a YouTube-video, which introduced a basis for text box mechanics.

Keywords: GameMaker: Studio, game programming, procedural generation

SISÄLLYS

1 JOHDANTO

Tässä opinnäytetyössä tutkitaan GameMaker: Studio -pelinkehitysohjelmiston soveltuvuutta käytännönläheisesti kaupallisen 2D-pelin kehityksessä yritysympäristössä. Työ on tehty yhdessä toimeksiantajan, Super God Oy:n, henkilöstön kanssa yrityksen ensimmäisen tuotteen, 2D-peli Riptalen, tuottamiseksi. Opinnäytetyössä käydään läpi GameMaker: Studion käyttöliittymän käyttöä, ohjelmiston käyttämää GML -ohjelmointikieltä ja näillä kehitettyjä mekaaniikkoja, joita Riptaleen on tuotettu. Lisäksi perehdytään proseduraaliseen kenttägenerointiin, jonka ympärille Riptale on rakennettu.

Riptale on lähitaistelupainotteinen 2D-videopeli, jonka pelimaailma ja kentät kehitetään lennosta proseduraalisen generoinnin avulla. Pelin graafinen ilme on musta-valkoinen, joskin tehostevärinä käytetään punaista. Peli on suunniteltu hieman varttuneemmalle pelaajakunnalle pelin väkivaltaisuuden ja veren määrän takia. Pelissä ohjataan Kumo -nimistä samurai-hahmoja, joka matkaa läpi luolastojen tuhoten vihamielisiä lohikäärmeitä ja lohikäärmeiden ympärille luodun kultin seuraajia. Hahmo aloittaa maan pinnalta ja matkaa yhä syvemmälle maan sisään pelin edetessä. Pelaaja ei voi koskaan edetä ylöspäin eikä jo-vierailluissa huoneissa voi enää käydä, jos ne sijoittuvat nykyisen huoneen yläpuolelle.

Työosuus on tehty projektipohjaisena yritysympäristössä osana työryhmää. Työryhmässä roolini oli projektin pääohjelmoija. Lähdin työryhmään mukaan Super God Oy:n edustajan Juha Keräsen ehdotuksesta. Olimme Juhan kanssa tuttuja jo entuudestaan opiskelun tiimoilta. Tämän opinnäytetyön olen kuitenkin laatinut yksin, eli muita opiskelijoita tämän opinnäytetyön tekoon ei liity. Käytin keskimäärin 35 tuntia viikossa pelin ohjelmointiin.

2 TYÖVAIHEET

2.1 PROJEKTIIN TUTUSTUMINEN

Projektityöskentely aloitettiin minun osaltani tutustumalla Riptale-projektiin. Pelin kehitys oltiin jo aloitettu ennen liittämistäni työryhmään. Työryhmä oli osallistunut Game Jam -tapahtumaan, johon Riptalen ensimmäinen konseptidemo luotiin. Tapahtuman jälkeen työryhmä päätti alkaa työstämään Riptalesta varsinaista peliä.

Minua edeltävät ohjelmoijat olivat kehittäneet pelin pohjalle pelimoottorin, joka nimettiin Kyyrma Engineksi ohjelmoijan nimen mukaan. Kyyrma Engineen kuului perusominaisuudet ja mekaniikat muunmuassa hahmojen liikuttamiseen ja hyökkäyksiin. Moottoria alettiin sitemmin kehittämään

2.2 GAMEMAKER: STUDIOON TUTUSTUMINEN

Ennen tätä projektia minulla ei ollut laisinkaan kokemusta GameMakerin käytöstä, joten ennen kuin pystyin aloittamaan varsinaisen työskentelyni, täytyi ohjelmiston perustoiminnot opetella. Verraten aikaisempiin kokemuksiini pelinkehitysalustoista GameMaker osoittautui nopeasti yksinkertaiseksi käyttää jopa hämmennykseen asti. Olin työskennellyt edeltävän vuoden aikana Unity3D-alustan parissa, joka moniulottuvuuteen olin jo ehtinyt tottua. Ohjelmiston käyttöliittymän ihmettelyyn käytin aikaa muutaman viikon luoden yksinkertaisia asioita kuten esimerkiksi pienen lohikäärmeviholisen peruskäyttäytymisen ja hyökkäysmenetelmän.

2.3 TYÖSKENTELY

Työryhmän toimintaan ja työkaluihin tutustuttuani voin aloittaa työskentelyn pelin parissa. Työskentely toimi pääsääntöisesti kokopäivätyönä Super God Oy:n toimitiloissa. Projektin etenemistä hallittiin scrum-menetelmällä. Tehtävistä pidettiin kirjaa käyttämällä HackNPlan-internetsivustoa, jonka projektinhallintatyökalut erikoistuu lähinnä pelinkehitykseen. Sivusto on vielä beta-vaiheessa, mutta on osoittautunut päteväksi työkaluksi.

HackNPlanin käyttö helpotti työmäärän suunnittelua huomattavasti. Sen työkalujen avulla tehtävät on helppo pilkkoa pieniin osiin, jolloin projektin seuranta ja työntehtävien suunnittelu sujuu helposti. Osille voi antaa tärkeysarvon ja arvoin siihen käytettävästä ajasta. Työskentelimme kahden viikon sprinteissä, uusi sprintti aloitettiin aina maanantaisin.

Työryhmämme osallistui Oulu Game Day -pelinkehitystapahtumaan, jossa esittelimme pelin sen hetkistä demoversiota standilla. Ihmiset saivat käydä kokeilemassa ja tarkastelemassa sen hetkistä tuotosta ja antaa suullista palautetta. Pidimme demossa myös pienen kilpailun, jossa vihollisia tappamalla pystyi keräämään rahaa. Eniten rahaa kerännyt voitti itselleen Super God Oy:n nimellä varustetun t-paidan.

Pelin kehitys jatkuu opinnäytetyöni jälkeen aina sen valmistumiseen asti. Pelin valmistuminen ja julkaisu on arvioitu vuoden 2017 ensimmäiselle neljännekselle.

3 MEKANIIKAT

3.1 PELIN PERUSMEKANIIKAT

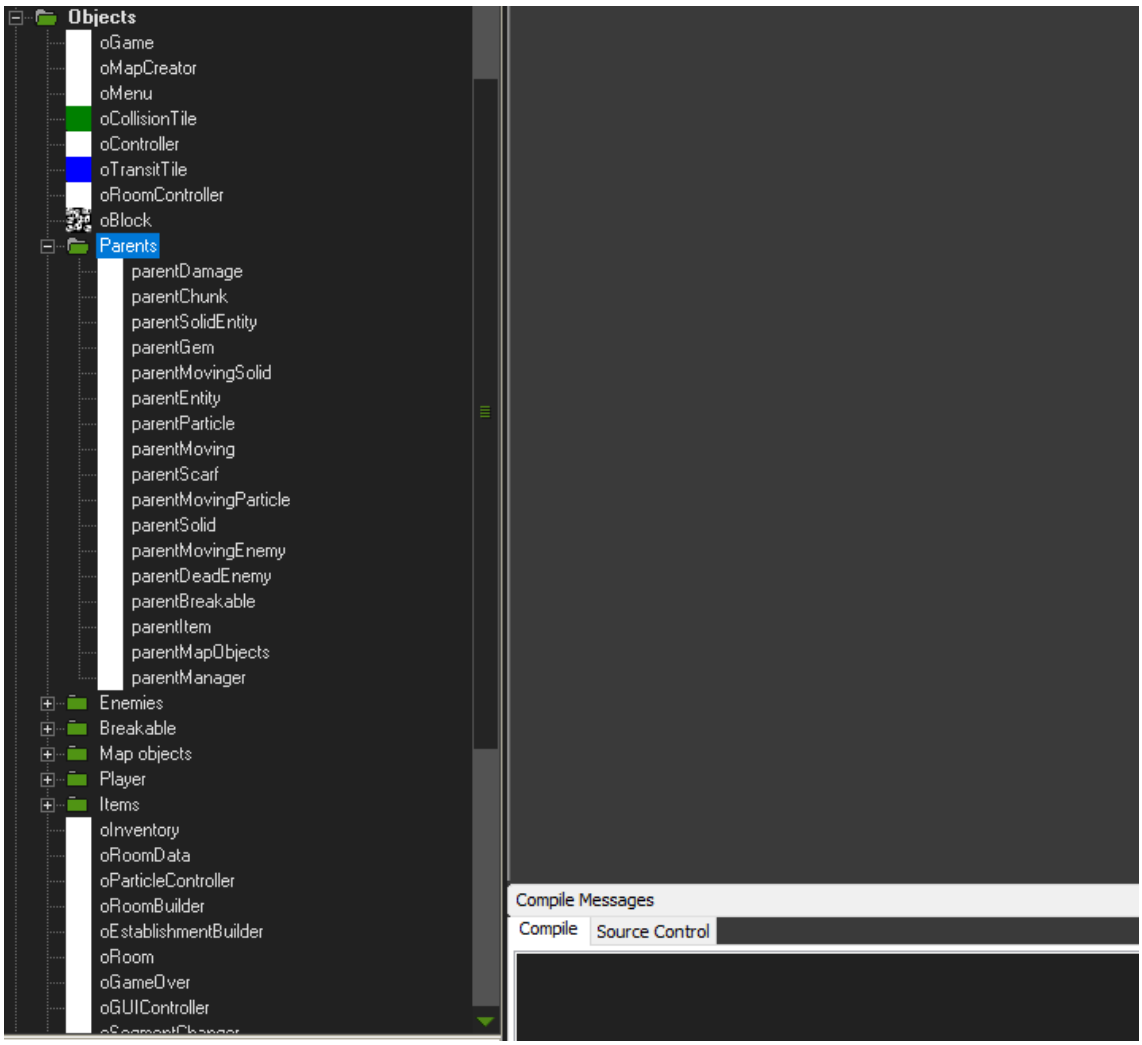
3.1.1 PELIN FLOW

Peli on suunniteltu pelattavaksi sivulta päin, ns. sidescroller-muodossa. Pelaaja ohjastaa Kumo-nimistä samurai-hahmoa, jonka tehtävä on edetä syvälle maan uumeniin ja tuhota lohikäärmeet, jotka tekevät tuhojansa. Matkalla tulee vastaan lohikäärmeisiin ja niiden ympärillä pyörivään kulttiin liittyviä vihollisia, joita pelaaja tappaa käyttäen miekkaa aseenaan. Pelin on tarkoitus olla lyhyt mutta vaikea ja jokainen pelikerta on suunniteltu olemaan erilainen, sillä huoneet ja niiden sisällöt generoidaan proseduraalisesti.

Pelin taistelun ideana on näyttävät, mobiilit hyökkäykset. Pelaajalla on käytössään jalokiviä, joiden perustalla hyökkäykset suoritetaan. Pelin alussa on vain tilaa vain kolmelle jalokivelle, mutta lisää tilaa uusille kiville on mahdollista saada pelin edetessä. Pelaaja suorittaa hyökkäyksen painamalla hyökkäysnappia, jolloin pelihahmo suorittaa hyökkäyksen. Hyökkäyksen perustana oleva jalokivi menee käyttökelvottomaksi ns. cooldownille reilun puolen sekunnin ajaksi hyökkäyksen loputtua. Hyökkäykset on mahdollista suorittaa yhdistelmänä, ns. combona käyttämällä ne peräkkäin. Koska hyökkäyksiä ei voi käyttää jatkuvasti, on pelaajan suunniteltava tekemisensä huolella selvitäkseen huoneesta toiseen. (Chen, 2006)

3.1.2 PERUSMEKANIIKKOJEN TOIMINNOT

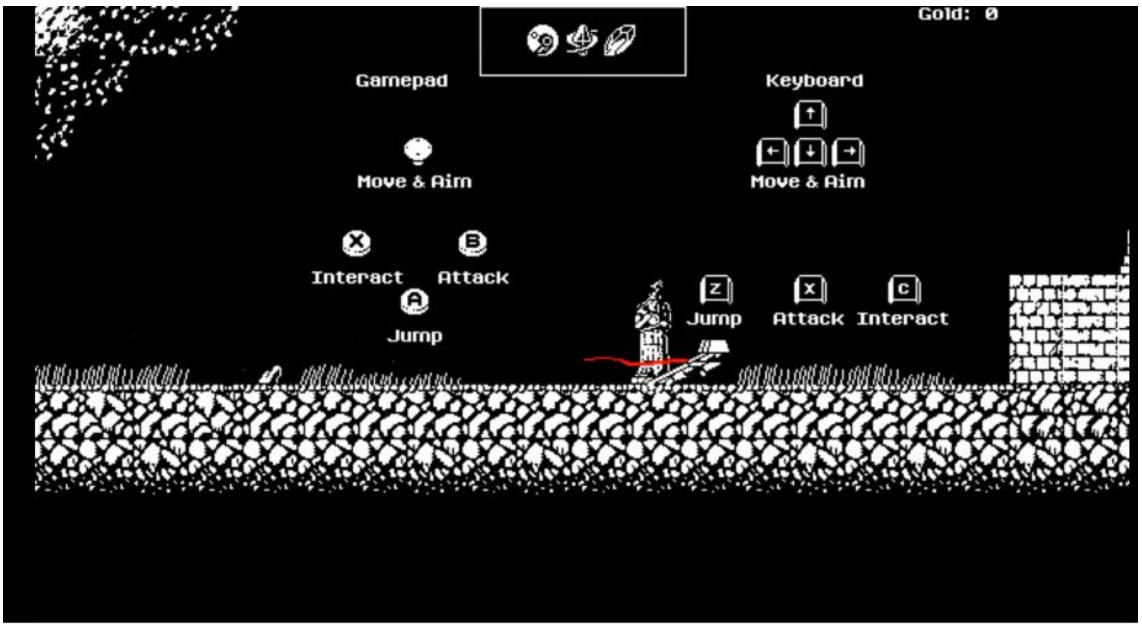
Ennen liittymistäni työryhmään peliin oli kehitetty valmiiksi perustoiminnot. Hahmoa kykeni liikuttamaan ruudulla molemmille sivuille ja peliin oli tehty painovoimamekaniikka joten hahmolla pystyi hyppimään ja tippumaan alas. Muutamia vihollisia oli tehty jo valmiiksi ja niille oli annettu peruskäyttäytymisensä suunnittelun mukaan. Viholliset liikkuivat ja jotkut kävivät pelaajan kimppuun tämän ollessa tarpeeksi lähellä.



KUVA 1: Lista parenttiobjekteista

Kaikkien objektien perustalla on jonkinlainen parenttiobjekti, joista peritään arvoja objetin tarkoitusperän mukaan. Lähes kaikkien parenttien ja muiden objektien pohjana on parentEntity, poikkeuksena mm. parentParticle, joka toimii pohjana fysiikkapohjaisille partikkeliobjekteille. Moni muu parenttiobjekti perii parentEntityn, mm. parentMoving, parentSolid ja parentMovingSolid, joilla on kaikilla omat tarkoituksensa. ParentMovingia käyttää pohjanaan pelaajan hahmo sekä vihollishahmot, parentSolidia käyttävät pohjanaan kaikki seinäobjektit ja parentMovingSolidia on tarkoitus käyttää liikkuvissa alustoissa, joita tällä hetkellä pelissä ei ole. Kuvassa 1 on lista tällä hetkellä käytössä olevista parenteista. (HeartBeast, 2014)

Yksikään parentti tai objekti ei hoida objektien liikuttamista itse. Tähän on suunniteltu oma hallintaobjekti oController. OControllerin tehtäviin kuuluu mm. pelin inputtien käsittely, yleiset debug-asiat, ruudun skaalaus sekä kaikkien liikkuvien objektien liikuttaminen. Tämä on tehty siksi, että kaikki liikuttaminen hoidettaisiin yhden kanavan kautta, rasittaen vähemmän prosessorin säikeitä ja täten vähentäen tietokoneen rasitusta.



KUVA 2: Tutorial panel pelin alussa

3.2 PROSEDURAALINEN KENTTÄGENEROINTI

3.2.1 PROSEDURAALISEN GENEROINNIN MÄÄRITELMÄ

Proseduraalisella kenttägeneroinnilla tarkoitetaan dynaamista datan luontia ja manipulointia asetettujen sääntöjen mukaisesti. Joskus proseduraalisesta generoinnista puhutaan virheellisesti satunnaisena generointina. Näiden kahden ero on se, että proseduraalisessa generoinnissa on erilaisia sääntöjä, jonka mukaan materiaalia luodaan. Esimerkiksi jos halutaan luoda pelihahmo, proseduraalisessa generoinnissa määritellään, mitkä osat hahmossa pitää olla ja mitä osia annettujen parametrien mukaan voi antaa. Jos vaikka halutaan luoda fantasiapelissä velhohahmo proseduraalisesti, se voidaan määritellä näyttämään ihmiseltä eli sille on annettu pää, vartalo,

kaksi kättä ja kaksi jalkaa. Velho voi olla fyysisesti liian heikko, eli sille ei voi antaa raskasta panssaria, mutta sille voi antaa vaikkapa kaavun tai nahkaliivin.

Satunnaisgenerointi voisi antaa käytännössä ihan mitä tahansa. Hahmolle voi ilmestyä kolme päätä, yksi käsi, neljä jalkaa ja kymmenen keskivartaloa. Vaikka se olisi määritetty velhoksi, generointi antaisi sille haarniskan, nahkahousut, kaasunaamarin ja neljä miekkaa, koska miksipäs ei, generointi toimii satunnaisesti. Toki perussäännöt on oltava, että generointi voidaan suorittaa, mutta ohjelmaa ei kiinnosta montako päätä tai jäätynyttä kanankoipea velholle saa antaa.

Maailmojen ja pelikenttien luonnissa generointimallien erot näkyvät varsin selvästi. Esimerkkinä Minecraft-tyylinen maailman generointi: Satunnaisgeneraattori luo mitä sattuu rakennelmia, kaikki materiaalit ovat sekaisin eikä lopputulos vastaa luonnollista maailmaa ollenkaan. Proseduraalisessa generoinnissa puolestaan voidaan määrittää biomeja, kuinka suuria biomit ovat, mitä kasveja, mineraaleja ja maaperää niihin kuuluu, minkä kiven lähettyvillä löytyy mitäkin mineraaleja, minkälaiset oliot ja eläimet niissä kulkevat, mitkä olosuhdevaatimukset ovat ja niin edelleen.

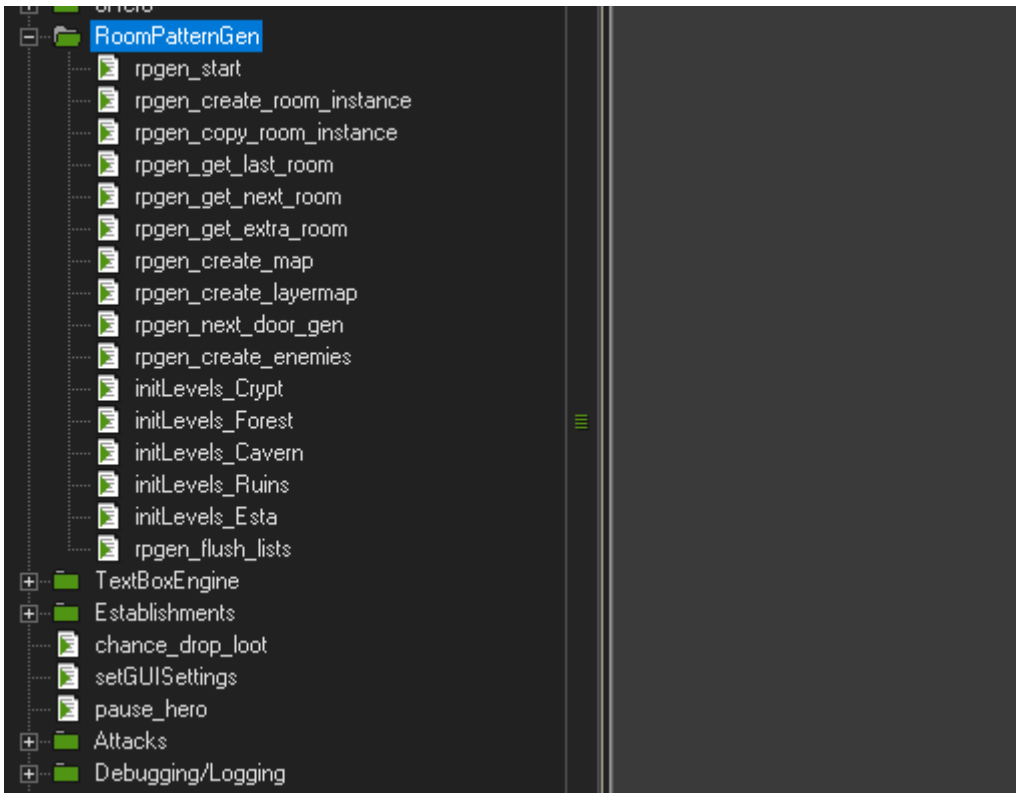
Proseduraaliselle generoinnille ei ole vain yhtä tai kahta tapaa toimia. Sen hienous on nimenomaan siinä, että voit muokata sääntöjä mielin määrin. Voit laatia lisää ja muuttaa jo keksittyjä sääntöjä mielensä mukaan, kunhan ne seuraavat syntaksia ja ohjelmointilogiikkaa. Sääntöjen määrä ja sisältö muuttuu aina sitä mukaa, minkälaista lopputulosta halutaan. (Shaker, Togelius, Nelson, 2017.)

3.2.2 PROSEDURAALISEN GENEROINNIN PERUSIDEA RIPTALESSA

Projektin yksi tärkeimmistä mekaniikoista ja pelin flow:n määrittävä ominaisuus on proseduraalinen kenttägenerointi. Generaattorin tehtävä on luoda pelimaailman huonejärjestys ja kasata huoneet lennosta pelin käynnissäolon aikana. Huoneille täytyi järjestää oviaukot ja miettiä, miten pelaajan hahmo kuljetetaan huoneesta toiseen ja joissain tapauksissa takaisin edelliseen. Mahdollisia oviaukkoja huoneilla on 1-3 per sivu, riippuen huoneen koosta. Oviaukkojen valinnalle piti keksiä jollain muotoa järkevät säännöt, esim. oviaukot eivät voi olla samalla sivulla, eivätkä ne saa olla samassa nurkassa vierekkäin.

Huoneet oli luotu valmiiksi käyttämällä erillistä kenttäeditointiohjelmaa Tiled:iä, joskin huoneet eivät Tiledin omassa formaatissa suoraan toimineet, vaan ne piti ensin konvertoida binääritiedostoiksi. Tähän löytyi GamePhasen kehittämä valmis työkalu Tiled to Binary Converter. Vaikkei työkalu ei käännä ihan kaikkea dataa, oli dataa tarpeeksi. Binääritiedostot liitettii projektiin importtaamalla ne GameMakerin omilla työkaluilla. Pelkästään binääritiedostot eivät riitä huoneiden rakentamiseen, vaan ne listattiin käyttämällä JSON-skriptiä (JavaScript Object Notation) GameMakerin käyttämiin mappitiedostoihin, jotka taas listattiin erilliseen listaan, josta tiedot saa noudettua helposti indeksin avulla. JSONin avulla pystyttiin helposti määrittelemään oviaukkojen määrä kullekkin sivulle. (Norrgård, 2016)

Generaattori noudataa suunniteltua pelimaailman rakennetta. Pelissä on neljä eri segmenttiä, eli osiota, joilla on kaikilla oma teemansa, graafinen ilmeensä ja niissä tulee vastaan erilaisia vihollisia. Kullakin segmentillä on kolme leveliä, eli tasoa, jotka ovat ilmestymisjärjestyksessä ”upper”, ”lower” ja ”final”. Uuden segmentin alkaessa on aina levelinä ”upper”, josta sitten pelin edetessä laskeudutaan ”lower”:iin ja lopuksi ”final”:iin. Jos segmentti ei ole viimeinen, ”final”:n jälkeen segmentti vaihtuu ja leveliksi vaihtuu ”upper”. Jokaisella kerralla, kun level vaihtuu, generaattori ajetaan sen hetkisillä tiedoilla ja siten luodaan uusi setti huoneita, jotka listataan ilmestymisjärjestykseen. Generoinnin ajettua peli siirtää pelihahmon listan ensimmäiseen huoneeseen ja peli jatkuu.



KUVA 3: Generaattorissa käytettäviä skriptejä.

Vanhat huoneet on samalla tarkoitus poistaa, mutta siinä missä GameMaker osaa luoda dynaamisesti uusia huoneita, niiden poistaminen on mahdotonta. Tämä johtaa siihen, että peli alkaa käyttämään yhä enemmän ja enemmän välimuistia, alan termein peli vuotaa muistia. Esimerkiksi seitsemän huoneen luonnilla peli varaa heti noin 4 megatavua lisämuistia, jota ei pysty vapauttamaan. Peli on itsessään melko kevyt ohjelma, opinnäytetyön kirjoitushetkellä peli vie noin 40 megatavua välimuistia sen alkaessa. Normaalilla pelaamisella lyhyt pelisessio ei tuota 100 megatavua enempää lisälastia, mikä ei sinänsä ole paljoa välimuistia enää nykyaikana mutta yhtään pidemmillä pelisessioilla voi tietysti nostaa lukemia melko hurjiksi. Muistin käyttö on siis tällä hetkellä kohtuutonta.

Muistin käytön ongelmalle voi löytyä ratkaisu, jonka yhtä mallia olen kokeillutkin jokseenkin järkevin tuloksin, mutta ei täysin toimivasti. Ideana on pelin käynnistyshetkellä luoda setti huoneita ja kun seuraavan kerran ajetaan generaattori, ohjelma pyyhkisi vanhoista huoneista tiedot pois, ajaisi uudet tiedot ja peli jatkuisi normaalisti.

GameMakerin huoneiden mekaaniikat tuottavat tässä kohtaa omanlaisiaan ongelmia. Yksi suurimmista huoneiden harmeista on ”persistent”-arvo. Kyseessä on boolean-arvo, joka päättää

säilyttääkö huone tietonsa sen jälkeen kun pelaaja käy ja poistuu sieltä, vai alkaako huone joka kerta alusta. Pelin tämänhetkisen flow:n takia ”persistent” on laitettu päälle. Tämä aiheuttaa sen, ettei huoneen tietoja voi muuttaa ns. etänä, eli jos et ole huoneen sisällä ja haluat tuhota tai muokata sen tietoja, GameMakerin omat funktiot eivät enää toimi. ”Persistent”:iä ei voi myöskään muokata etää, vaan sen voi muokata ainoastaan huoneen ollessa käytössä.

Tämä johtaa siihen, ettei huoneiden dataa voi muokata esimerkiksi pelkällä for-loopilla generoinnin sisällä, vaan huoneiden täydellinen flushaus ja uudelleenkäyttö täytyy suorittaa jotenkin muutoin. Olen suunnitellut metodia, jonka laatiminen on sen verran työläs, etten ole tässä vaiheessa vielä muulta kehitykseltä ehtinyt sitä testata. Pelin restarttaus eli uudelleenkäynnistys (jos vaikka pelihahmo kuolee) täytyy muuttaa manuaaliseksi GameMakerin oman skriptin sijaan, joka käynnistää koko ohjelman uusiksi. Manuaalisen uudelleenaloituksen aikana ohjelma loisi persistentin pyyhkimisobjektin ja aukaisisi ensimmäisen huoneen listalta. Objekti sitten muuttaisi huoneen arvot sisältä käsin ja käskisi ohjelman aukaista seuraava huone. Looppi jatkuisi, kunnes huoneet olisi pyyhitty ja persistenssit muutettu

GameMakerin tutoriaaleihin ja manuaaleihin perehdyttyäni totean metodin toimivan ainakin teoriassa, niillä tiedoilla ainakin mitä löysin. Manuaalit ovat toki sen verran suppeat huoneiden luontiin liittyviän asioiden osalta, että joudun ottamaan kaiken ns. suolan kanssa, kunnes voin todeta toimivaksi käytännössä.

3.2.3 GENEROINNIN KULKU

Generointi aloitetaan käskemällä peli menemään huoneeseen nimeltä ”rmGenerate”. Huoneen aloitus kutsuu skriptiä rpgen_start(<parametrit>). Parametreiksi annetaan järjestyksessä seuraavat:

1. Huoneiden lukumäärä (integer). Montako huonetta luodaan yhteensä. Tähän ei lasketa aloitushuonetta eikä lopetushuonetta, vaan ne tehdään automaattisesti tilanteen mukaan.
2. Huoneiden lukumäärä aloitushuoneen ja lopetushuoneen välillä (integer). Täytyy olla pienempi tai yhtä suuri kuin edellinen parametri. Tämän ja edellisen parametrin erotus määrää montako extrahuonetta luodaan (establishment, salaiset huoneet jne).

3. Segmentti (string). Pitää olla joko ”crypt”, ”caverns”, ”forest” tai ”ruins”. Määrittää perin osion.
4. Leveli (string). Pitää olla joko ”upper”, ”lower” tai ”final”. Määrittää segmentin tason.

Skriptin alussa luodaan varsin suuri määrä erilaisia listoja, jotka tulevat käsittelemään eri huoneiden dataa, lähes kaikkien listojen indeksi viittaa samaan huoneeseen. Listat sisältävät mm. binääritiedoston nimen, järjestyksen grid-taulukossa, johon huoneet asetetaan ilmestymisjärjestyksestä ja jonka perustalla oviaukot avataan. Osa listoista näkyy kuvassa 4.

```
*/
// More lists

global.room_list_maps = ds_list_create();
global.room_list_maps_esta = ds_list_create();
global.room_list_index = ds_list_create();
global.room_bins_esta = ds_list_create();
global.room_bins = ds_list_create(); // Stores binary files in this for room creation purposes.
                                     // They must be added in the same order as global.room_list_total
                                     // so that they may be fetched correctly

global.room_list_esta_enter = ds_list_create();
global.room_order_esta = ds_list_create();  // Stores a list of establishment rooms
global.room_order_rand = ds_list_create();  // Stores a list of rooms in order which player can advance through
global.room_grid_x = ds_list_create();      // Stores grid x-vector data
global.room_grid_y = ds_list_create();      // Stores grid y-vector data
global.room_grid_esta_x = ds_list_create();
global.room_grid_esta_y = ds_list_create();
global.room_grid = ds_grid_create(50, 50);  // Grid for aiding room placement structure

// Other
global.tileset_Main = bgTilesetStone1;
global.tileset_Establishment = establishmentTileset;
global.levelPath = "";
global.roomIndex = 0; //Used to navigate between rooms later on
global.estaIndex = 0;
global.currentRoomType = "normal";
global.currentLevel = level;
global.currentArea = area;

```

KUVA 4: Osa datanhallintalistoista

Listojen luonnin jälkeen katsotaan switch-casella, mikä segmentti on haluttu generoida. Ensimmäisenä se ajaa segmentin mukaan jonkun initLevels-skripteistä, joka palauttaa huoneiden mahdollisten oviaukkojen määrän ja binääritiedoston nimen listattuna mappeihin. Seuraavaksi päätetään mitä tilesettiä ohjelma tulee käyttämään huoneita piirtäessä. Lopuksi valitaan kansiopathi, jotta ohjelma osaisi myöhemmin lukea oikeat binääritiedostot. Katso kuva 5.

```
03 switch(area)
04 {
05     case "crypt":
06 
07         //show_debug_message("Loading crypt_levels...");
08         lobal.room_level_data = initLevels_Crypt();
09         global.tileset_Main = bgTilesetStone0;
10         global.levelPath = "crypt_levels\";
11         //show_debug_message("Done");
12 
13         break;
14 
15     case "cavern":
16 
17         //show_debug_message("Loading cavern_levels...");
18         global.room_level_data = initLevels_Cavern();
19         global.tileset_Main = bgTilesetStone2;
20         global.levelPath = "cavern_levels\";
21         //show_debug_message("Done");
22 
23         break;
24 
25     case "forest":
26 
27         //show_debug_message("Loading forest_levels...");
28         global.room_level_data = initLevels_Forest();
29         global.tileset_Main = bgTilesetStone1;
30         global.levelPath = "forest_levels\";
31         //show_debug_message("Done");
32 
33         break;
34 
35     case "ruins":
36 
37         //show_debug_message("Loading ruins_levels...");
38         global.room_level_data = initLevels_Ruins();
39         global.tileset_Main = bgTilesetStone3;
40         global.levelPath = "ruins_levels\";
41         //show_debug_message("Done");
42 
43         break;
```

KUVA 5: segmentin switch-case

Seuraavassa osiossa generaattori järjestelee juuri saaneensa huoneiden tiedot listoihin, joista tiedot on jatkossa helpompi hakea. Lisäksi ohjelma käy läpi löytyykö tietyt erityishuoneet mapeista. Löytäessään ne merkitään erillisiin globaaleihin muuttujiin myöhempää käyttöä varten. Näitä huoneita ovat tutoriaalihuone, pomovastushuone sekä ns. segway-huone, jota käytetään seuraavan leveliosion generointiin.

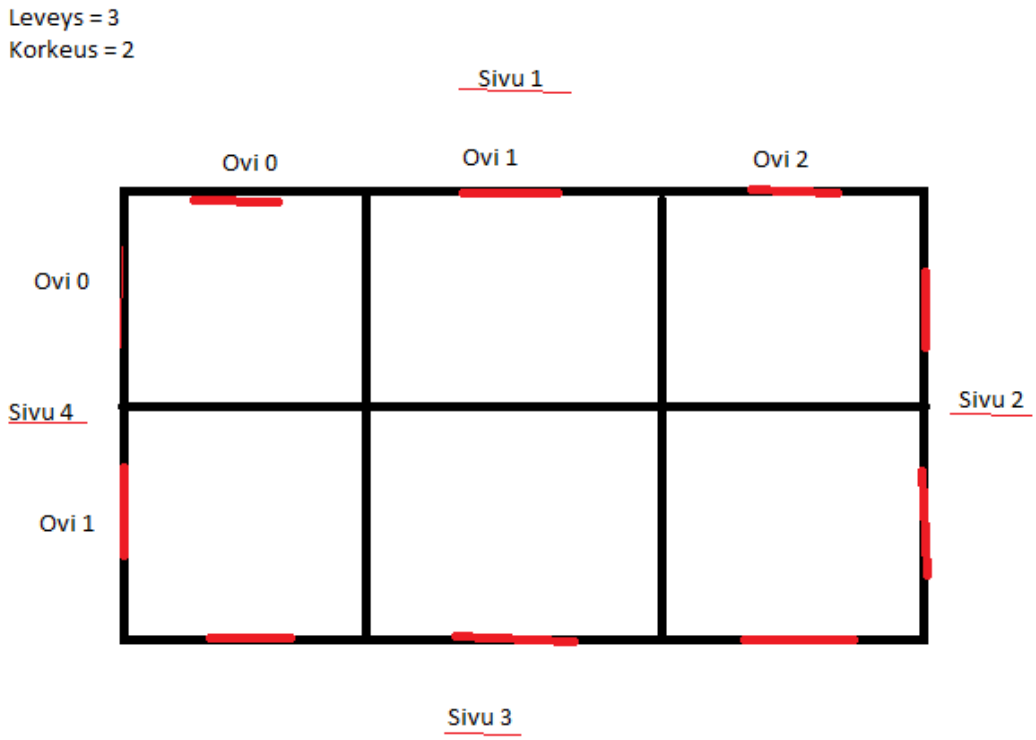
Seuraavaksi generaattori katsoo tarvittavan huonemäärän ja arpoo juuri saaduista huonetiedoista huoneita satunnaisessa järjestyksessä, lukuunottamatta erityishuoneita, joille on omat sääntönsä. Nekin lisätään huonelistaan tässä vaiheessa generaattorille syötettyjen parametrien mukaan. Generaattori myös katsoo, ettei nämä erityishuoneet päädy vahingossa listaan, jos niiden ei ole tarkoitus olla juuri siinä kohti listaa, mitä kohtaa arvotaan.

Sitten äsken arvotut huoneet lisätään gridiin järjestykseen. Aloittava huone laitetaan gridin keskiosaan ylälaitaan, josta seuraava huone lisätään aina alapuolelle. Tämän jälkeen huoneet lisätään joko vasemmalle, oikealle tai alas. Tätä jatketaan kunnes tulee arvotun listan viimeinen

huone, joka on aina edeltävän huoneen alapuolella. Tämä tehdään siksi, ettei yläpuolella olevaan huoneeseen voi koskaan palata takaisin pelin aikana. Lisäksi lisätään listoihin minkäkin huoneen koordinaatit gridillä, että oviaukot voidaan päättää myöhemmin.

Tähän väliin generaattori luo tarvittavan määrän uusia huone-instansseja. Pohjana näille huoneille toimii huonepohja, jolle on annettu tarvittavat tiedot käsin valmiiksi, ettei niitä tarvisi erikseen asettaa enää koodissa. Huoneet listataan, niinkuin kaikki muutkin tiedot, huoneiden ilmestymisjärjestyksen mukaan.

Tässä vaiheessa huoneiden tiedot alkavat olla tallessa, mutta ne pitää vielä asettaa mappeihin helppoa tiedon hakua varten. Generaattori luo mapin, johon sen hetkisen indeksinumeron tiedot kaikista muista listoista lisätään. Lisäksi tässä välissä arvotaan oviaukkojen paikat. Huoneet on suunnitelut siten, että niissä on samalla vektorilla saman verran oviaukkoja, esim jos ylälaidassa on kaksi mahdollista oviaukkoa, myös alalaidassa on kaksi. Nämä merkitään leveys- ja korkeusarvoina mappiin. Generaattori lukee nämä arvot ja päättää annettujen sääntöjen mukaisesti mitkä oviaukot tulee aukaista. Jokaiseen mappiin tulee arvot ”enterDoor”, ”exitDoor” ja ”extraDoor”. Ensin maininnu on se oviaukko, josta saavutaan huoneeseen, seuraava on poistumisovi ja viimeinen on extrahuoneita varten, kuten kauppoja ja kasinoja varten.



KUVA 6: huoneiden sivujen ja oviaukkojen numerointi

Kuvasta 6 voi tarkastella huoneiden laitojen ja oviaukkojen numerointitapaa. Nämä numerot yhdistetään kaksinumeroiseksi luvuksi, josta ohjelma tunnistaa myöhemmin oviaukon. Esimerkiksi alhaalla oikealla oleva oviaukko on arvoltaan 21. Vasen alhaalla on 41. Vasen ylhäällä on 10. Vastoin normaaleja ohjelmointikäytäntöjä on sivujen numeroinnista jätetty nolla pois. Päätös on tehty sillä perusteella, että koodissa olisi voinut ilmetä ongelmia asettaessa kaksinumeroista lukua, joka alkaa nollalla. On hyvinkin mahdollista, että ohjelma olisi osannut tulkita sen oikein, mutta pelasin varman päälle tässä tapauksessa.

Sivut määritellään grid-koordinaattien myötä. Tämän jälkeen päätetään oviaukot. Näille oviaukoille on omat sääntönsä, käytännössä luotu isolla if-else hirviöllä. Näissä säännöissä on muunmuassa, ettei oviaukot voi olla samassa kulmassa. Esimerkkiksi jos pelaaja saapuu sivulta 2 alimmasta aukosta, ei oviaukko voi olla heti jalkojen juuressa sivulla 3 viimeisessä aukossa.

```
repeat(ins_repeat)
{
    var room_t = 0;
    var inst = ds_list_find_value(global.room_list_instance, ii);
    var roomData = ds_list_find_value(global.room_list_maps, ii);

    var nId = ds_list_find_value(global.room_order_rand, ii);
    var roomData = rpgen_create_map(nId);

    // Add a data storage and add some values
    //var newroomData = room_instance_add(inst, 0, 0, oRoomData);
    var r_data = ds_list_find_value(global.room_level_data, ds_list_find_value(global.room_order_rand, ii));
    roomData[? "roomId"] = ii;
    roomData[? "roomListType"] = "normal";
    roomData[? "roomName"] = "rpgen_room_" + string(ii);
    roomData[? "roomType"] = ds_list_find_value(global.room_order_rand, ii)
    roomData[? "roomFileName"] = ds_list_find_value(global.room_bins, roomData[? "roomType"]);
    //show_debug_message("Bin filename: " + string(roomData[? "roomFileName"]));
    roomData[? "doorWidth"] = r_data[? "width"];
    roomData[? "doorHeight"] = r_data[? "height"];

    var door_open = 1;

    if (ii > 0)
    {
        door_open += 1;
    }

    roomData[? "openDoors"] = door_open;
    roomData[? "enterDoor"] = rpgen_get_last_room(ii, r_data[? "width"], r_data[? "height"]);
    roomData[? "exitDoor"] = rpgen_get_next_room(ii, r_data[? "width"], r_data[? "height"], roomData[? "enterDoor"]);
    roomData[? "extraDoor"] = rpgen_get_extra_room(ii, room_t, r_data[? "width"], r_data[? "height"]);

    //show_debug_message("enter: " + string(roomData[? "enterDoor"]));
    //show_debug_message("exit: " + string(roomData[? "exitDoor"]));

    ds_list_add(global.room_list_maps, roomData);
    // Bunch of data has been set

    room_instance_add(inst, 0, 1, oRoomBuilder);
```

KUVA 7: huoneen tietojen lisäys yhteen mappiin

Oviaukkojen päättämisen jälkeen huone-instanssiin heitetään objekti oRoomBuilder, jonka tehtävä on ajaa huoneen piirtämiseen liittyvät skriptit silloin, kun huoneeseen siirrytään. Objektin create-eventissä haetaan sen huoneen indeksin perusteella huoneen tiedot. Sen jälkeen ajetaan skripti, joka muuntaa huoneen binääritiedoston GameMakerille luettavaan muotoon, käytännössä kyseessä on listoista ja mapeista koostuva laaja tietostruktuuri, joka on sisäytettynä mappiin. Sitten mapin tiedot luetaan, muut generoinnissa luodut tiedot otetaan mukaan ja näiden tietojen perusteella ohjelma piirtää huoneen. Kun huone on piirretty, ohjelma käy vielä kertaalleen läpi huoneen tilejen arvot. Tiettyjen tilejen kohdalle luodaan ”oCollisionTile”-objekti, jonka tehtävä on estää läpipääsy, jolloin se kohta huoneesta toimii seinänä tai lattiana. Oviaukot piirretään samalla huoneisiin aikaisemmin pääteltyjen numeroarvojen perusteella.

Jotta oviaukoista voisi oikeasti kulkea, generaattori luo oviaukolle kahdelle ensimmäiselle tyhjälle tilelle objektit nimeltään ”oTransitTile”. Nämä objektit listataan globaaliin listaan, jotta niihin pääsee helposti käsiksi. Näiden objektien tehtävänä on toimia ns. maamerkkinä ohjelmalle ja hallita huoneissa kulkua. Kun pelaaja lähestyy oviaukkoa, hän koskee tietämättään ”oTransitTile”:jä, jolloin ohjelma hakee niihin tallennetut arvot, jonka mukaan pelaaja matkustaa

huoneesta toiseen. Esimerkiksi pelaja poistuu huoneesta käyttämällä poistumisoviaukkoa, jolloin ohjelma tietää, että mennään listassa seuraavaan huoneeseen. Seuraavan huoneen luonnin jälkeen ohjelma hakee siitä huoneesta vastaavan oviaukon, johon pelaajan pitää ilmestyä. Sen oviaukon ”oTransitTile”:jen huonekoordinaattien keskiarvosta lasketaan kohta, johon pelaajan hahmo siirretään. Tämä tapahtuu myös silloinkin, kun halutaan palata takaisin edelliseen huoneeseen (sikäli kun se ei ole yläpuolella), ohjelma osaa hakea oikeat ”oTransitTile”:t ja laskee niiden perusteella uuden koordinaatin, johon pelaaja siirretään.

Mainittakoon, että tässä generaattorissa binääritiedoston kääntäminen on tehty Tiled to Binary Converter-apuohjelman luojan tekemillä skripteillä, joskin niihin on tehty rankalla kädellä lisäyksiä jälkeenpäin.

3.3 PELIHAHMON HYÖKKÄYKSET

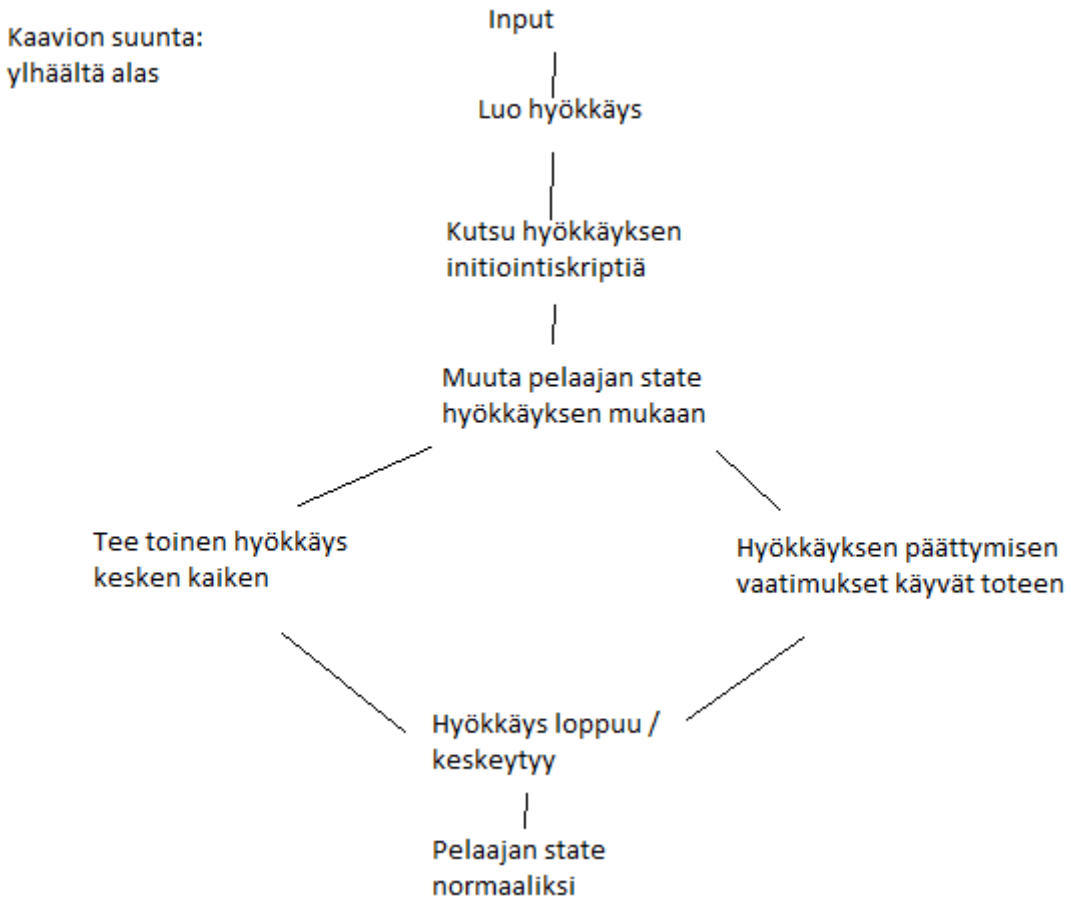
Pelaajan hahmolla ei ollut tähän mennessä kuin kolme eri hyökkäystä eli Straight Attack, Double Side Attack ja Circle Attack. Jottei pelistä tulisi liian monotoninen, teimme suunnitelmia liudalle uusia hyökkäyksiä. Näistä suunnitelmista lopulta toteutimme viisi, Trident Attackin, Triple Attackin, Homing Attackin, Saw Attackin sekä Bulldoze Attackin. Hyökkäyksiä voi tarkastella taulukosta 1.

Hyökkäyksen nimi	Toiminto
Straight Attack	Sinkoaa pelaajan hahmon määrittelemäänsä suuntaan nopealla vauhdilla ja luo hyökkäysobjektin ollessaan tarpeeksi lähellä vihollista tai seinää. Pitkä lentoetäisyys.
Double Side Attack	Sinkoaa pelaajan Straight Attackin lailla, mutta luo hyökkäyksiä vuorotellen molemmille puolelle pelihahmoa lentomatkan ajan. Pitkä lentoetäisyys.
Circle Attack	Hahmon kohdalle ilmestyy hahmoa suurempi hyökkäysanimaatio joka pyörähtää ympäri pelaajaa, luoden pelaajan ympärille vihollisia vahingoittavan

	ympyräefektin. Ei lentoa.
Trident Attack	Sinkoaa pelaajan hahmon määrittelemäänsä suuntaan nopealla vauhdilla ja luo kolme eri hyökkäystä, yhden suoraan menosuuntaan, kaksi muuta noin 15 astetta ylä- ja alapuolelle. Lyhyt lentoetäisyys
Triple Attack	Sinkoaa pelaajan hahmon määrittelemäänsä suuntaan nopealla vauhdilla ja luo hyökkäysobjektin ollessaan tarpeeksi lähellä vihollista tai seinää. Tämän jälkeen hahmo toistaa hyökkäyksen kaksi kertaa uudestaan, mutta vain erittäin lyhyellä lentomatkalla. Pitkä lentoetäisyys.
Homing Attack	Sinkoaa pelaajan hahmon automaattisesti kohti lähintä vihollista ja luo Circle Attackin tyylisen hyökkäysobjektin hahmon kohdalle ollessaan tarpeeksi lähellä. Keskipitkä lentomatka.
Saw Attack	Sinkoaa pelaajan hahmon pelaajan määrittelemästä suunnasta 45 astetta ylöspäin lyhyen matkaa ja luo hyökkäyksen. Tämän jälkeen hyökkäys toistetaan useamman kerran ylös ja alas, tehden siksakkikuviota.
Bulldoze Attack	Luo pelaajan kohdalle hyökkäysobjektin ja sinkoaa pelaajan hahmon määrittelemäänsä suuntaan nopealla vauhdilla hyökkäysobjektin seuratessa pelaajaa, luoden vaikutelma jatkuvista miekaniskuista. Hyökkäys loppuu kun pelaaja törmää seinään tai on matkustanut tarpeeksi pitkän matkan. Keskipitkä lentomatka.

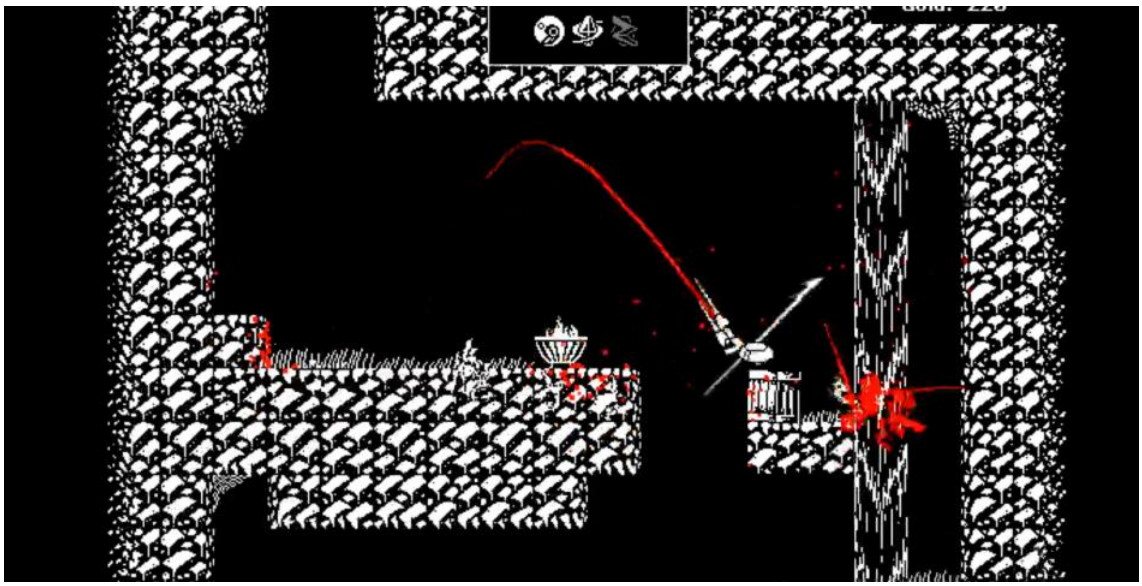
TAULUKKO 1: HYÖKKÄYKSET

Alkuun hyökkäykset toimivat vain ja ainoastaan oikealle ja vasemmalle, mutta myöhemmin ne piti muokata ottamaan useampia suuntia nuolinäppäinten tai padiohjaimen tattien mukaan. Hyökkäykset toimivat kuvan 8 esittämällä periaatteella.



Kuva 8: Hyökkäystoimintojen kaavio

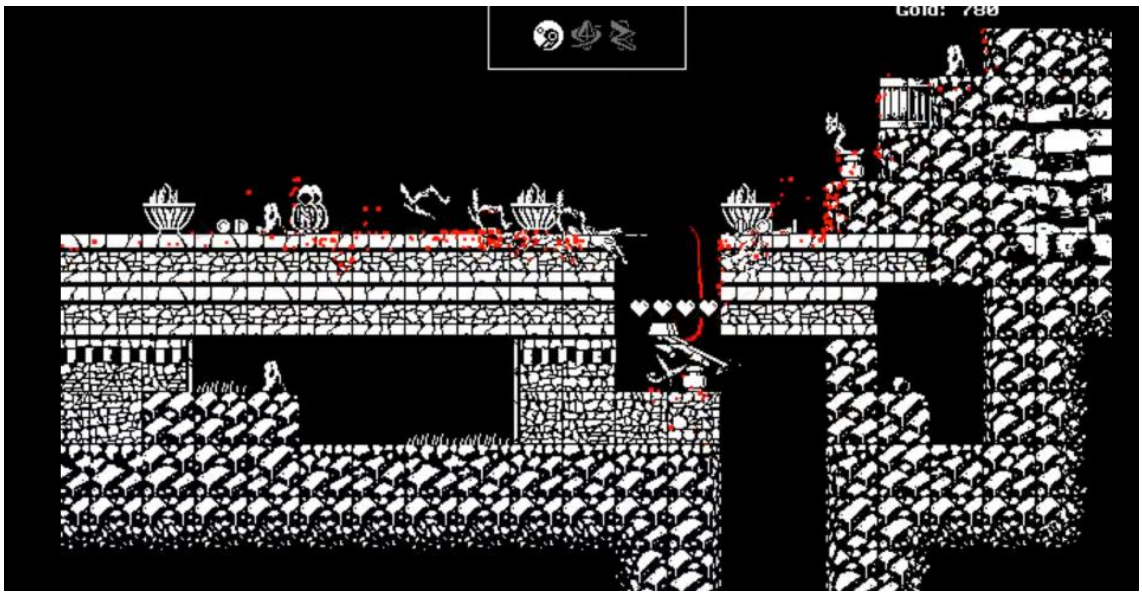
Pelihahmolla kutsutaan skriptiä, joka alustaa hyökkäyksen, mm. laskemalla sille kulkusuunnan, antamalla nopeusarvot, muuttamalla hahmon tilan ja niin edelleen. Seuraavalla framella pelihahmon tarkastaessa tilaa huomataan, että hyökkäys on käynnissä, jolloin ajetaan hyökkäyksen toimintoskripti. Skriptissä käytetään aiemmin alustettuja tietoja. Hyökkäyksen loppuessa hahmon tila muutetaan takaisin normaaliksi. Esimerkki hyökkäyksestä kuvassa 9.



KUVA 9: Pelaaja suorittamassa Saw Attack -hyökkäystä.

3.4 KUSTOMOITU PARTIKKELIJÄRJESTELMÄ

Peliin piti sada veriefektejä, jotka vuorovaikuttaisivat muiden objektien kanssa esim. jäisivät kiinni seiniin. Tällaista ominaisuutta GameMakerista ei vakiona löytynyt, joten se piti kehittää itse. Siinä missä GameMakerin oma partikkelijärjestelmä toimii täysin graafisella tasolla, uusi järjestelmä suunniteltiin objektipohjaiseksi. Järjestelmä suunniteltiin muistuttamaan käytössä monin osin GameMakerin omaa järjestelmää, eli efektien luonti tapahtui samalla kaavalla: luodaan partikkeliefektille tietorakenne, jolle syötetään arvoja ja efektiä sitten kutsutaan muualla koodissa kun sitä halutaan käyttää, jolloin partikkelijärjestelmä luo syötettyjen tietojen perusteella graafisia efektejä. Veriefekti kuvassa 10. (Shiffman, 2012)

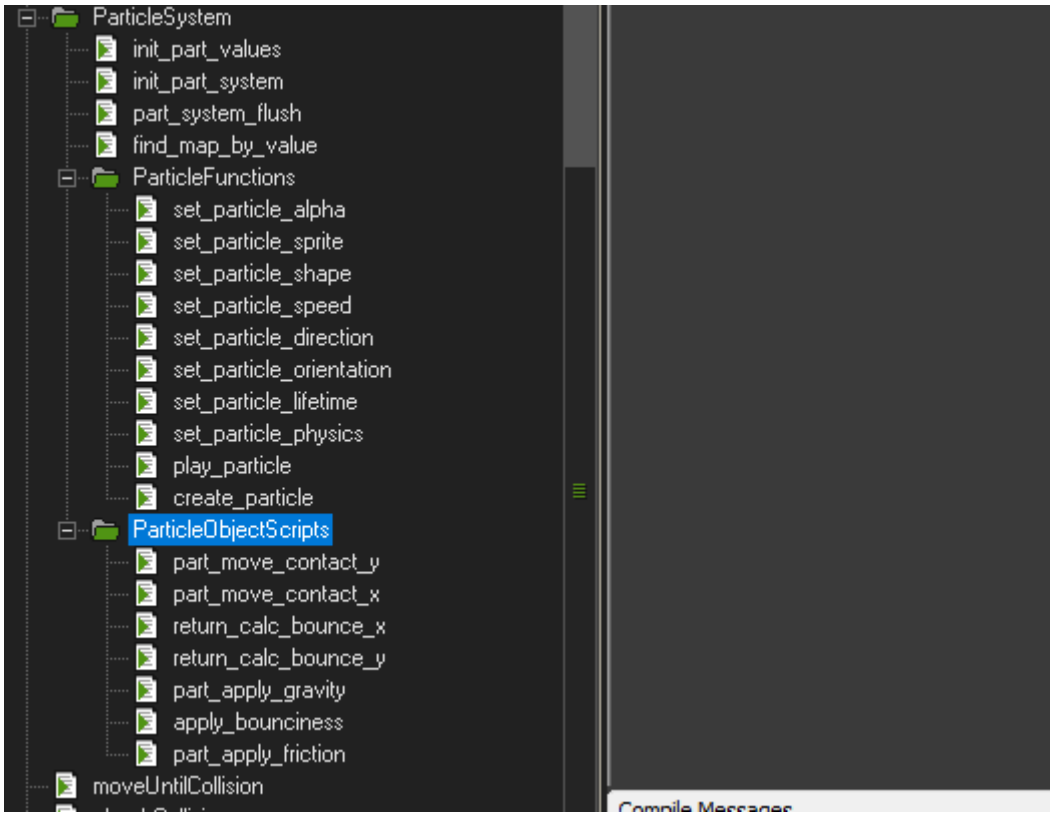


KUVA 10: Seinille ja lattialle roiskunutta verta

Toteutin oman järjestelmäni toimimaan seuraavasti:

oParticleObject	- Pohjaobjekti, josta löytyy oleelliset toiminnaallisuudet liikkeelle, sen muutoksille jne.
oController	- Luomamme pelimoottorin hallintaobjekti, jonka tärkeimpänä tehtävä on käsitellä inputit sekä liikuttaa kaikkia liikutettavia objekteja, kuten oParticleObjecteja.
init_particle_system();	- Luo listat efektien tietomapeille sekä luotaville objekteille
create_particle(”efekti0”);	- Luo mapin nimellä “efekti0”;
set_particle_sprite(”efekti0”, ...);	- Syöttää mappiin käytettävän spriten ja sen animointiarvot
set_particle_speed(”efekti0”, ...);	- Syöttää mappiin nopeuden min ja max, lisäys/frame jne

set_particle_alpha(”efekti0”, ...);	- Syöttää mappiin alphan eli piirtämisen näkyvyysasteen
set_particle_physics(”efekti0”, ...);	- Syöttää mappiin halutut käytettävät fysiikkaelementit
set_particle_lifetime(”efekti0”, ...);	- Syöttää mappiin min/max sekunttiajat, jonka efekti on olemassa
set_particle_orientation("efekti0”,..);	- Syöttää mappiin efektin piirtosuunnan ja mahd. pyörimisen
play_particle(”efekti0”,...);	- Luo syötetyn määrän partikkeliobjekteja ja syöttää niille annetun mapin mukaisesti tiedot, jotta ne voisivat toimia halutusti. Objektit alkavat sitten toimia niinkuin niiden pitää.



KUVA 11: Partikkelijärjestelmän skriptit

Kuten huomata voi, toimivat funktion hyvin samanlaisella kaavalla kuin GameMakerin omassa järjestelmässä. Yksityiskohdissa,funktioiden nimissä ja toki toiminnoissa on eroja. Menetelmäni on todettu toimivaksi testauksien kautta. Lista skripteistä kuvassa 11.

Objektipohjaisen partikkelijärjestelmän huonona puolena on sen raskuus. Jos halutaan paljon verta ruudulle, tarkoittaa se sitä, että jokaisen pienenkin punaisen pikselin takana on yksi objekti. Kun kaikille näille objekteille on annettu fysiikkaominaisuudet ja tehtävät, kuormitus on kova. Tätä on pyritty ratkaisemaan neljällä eri tavalla:

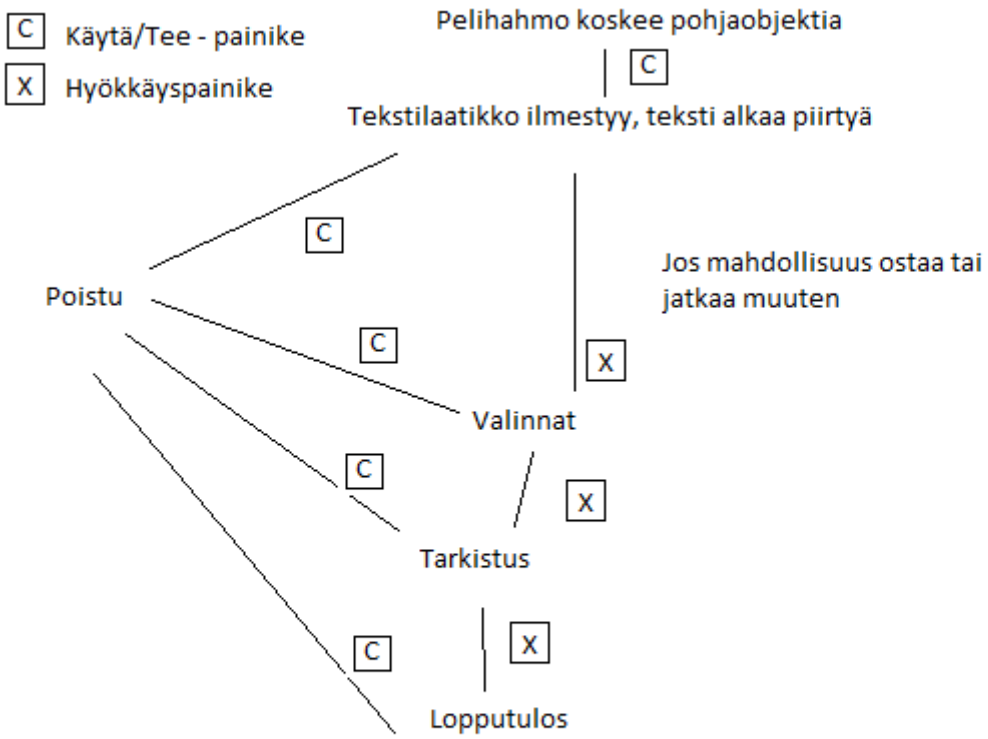
1. Rajoitetaan luotavien partikkelien määrää. Partikkelien luontiin on laitettu rajoitin, joka tarkistaa jo olemassaolevien partikkeliobjektien määrän. Jos katto on tullut vastaan, poistetaan listan vanhin olemassaoleva objekti. Täten uudet partikkelit pääsevät liikenteeseen ja lähtöosa efektistä ei näytä vajavaiselta. Menetelmä on osottautunut varsin toimivaksi, joskin liikkeessä olevien objektien määrä voi olla vajavainen. Tätä toki lopputuotteessa kykenee itse säätämään sen mukaan, miten itse haluaa ja kuinka paljon tehoa omassa tietokoneessa on.
2. Kun partikkeli pysähtyy, lakataan sen fysiikoita päivittämästä. Tällöin ainoa, mitä päivitetään, on partikkelin spriten piirto. Tämän on todettu laskevan rasitusta melko tehokkaasti, joskin tapauskohtaisesti.
3. Objekteille on annettu lyhyt elämänkaari. Objekteille annetaan muutamien sekunttien elinaika. Elinaikaa voi säätää mielitekonsa mukaan, tällä hetkellä veriefektille on annettu kolmesta seitsemään sekuntia elinaikaa.
4. Paikataan efektin näkyvyyttä GameMakerin omalla partikkelijärjestelmällä. Samalla kun kutsutaan omaa järjestelmääni, kutsutaan myös GameMakerin järjestelmää. Olemme tehneet ison liudan sprite-animaatiopohjaisia veriefektejä toimimaan GameMakerin oman järjestelmän kautta. Näin on helppoa ja kevyttä luoda isot määrät verta ruudulle vihollisten kuollessa, vaikkeivät kaikki niistä seiniin kiinni jääkkään.

3.5 TEKSTILAATIKKOMEKANIIKAT

Peliin on suunniteltu liuta erilaisia extrahuoneita, establishmentteja, joista löytyy mm. jalokivikauppa, kasino, kotieläintarha, kirjasto ja niin edelleen. Näissä huoneissa pitää pystyä keskustelemaan erilaisten hahmojen kanssa, lukemaan kirjaa tai ostamaan timantti, joten peliin tarvittii tekstilaatikkomekaniikat. Opinnäytetyön kirjoitushetkellä mekaniikka on varsin hiomaton, mutta toimii. Itse tekstin piirtäminen tässä mekaniikassa pohjautuu YouTube-käyttäjä HeartBeast:n luomassa videossa ”Game Maker Tutorial: RPG Text Box Example” esitettyyn menetelmään. Lähes kaikki muut osat tästä mekaniikasta on suunniteltu itse.

Mekaniikka toimii seuraavalla idealla:

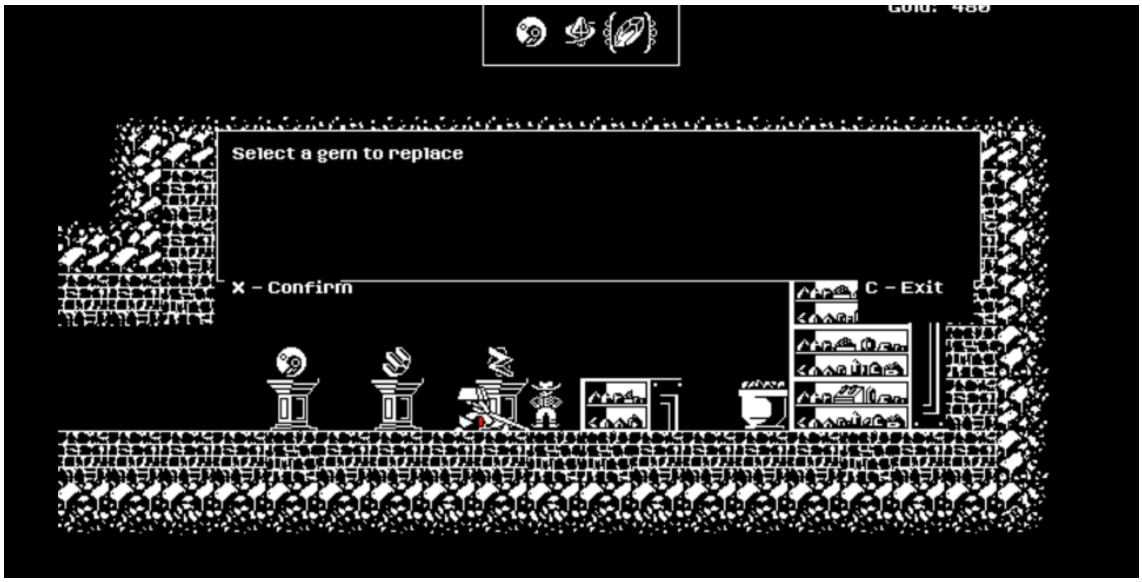
oTextObject	- Pohjaobjekti, jolle on annettu state-pohjainen käyttäytyminen ja muut tarvittavat menetelmät.
texteng_create(x, y, text...);	- Alustaa tekstilaatikon ja sille syötettävän tekstin
texteng_draw_box();	- Piirtää alustetun laatikon
texteng_draw_text();	- Piirtää alustetun tekstin
texteng_draw_button(x, y, text...);	- Piirtää pienemmän laatikon ja siihen tekstin. Ei piirrä toiminnallista nappia, vaan esimerkiksi pienen kyltin, jossa lukee ”C – Cancel” eli ilmoituksen, että c-painiketta painaessa tekstilaatikko tuhoutuu.



KUVA 12: tekstilaatikon toimintokartta

Kuvasta 12 voi huomata, että vuorovaikutus tekstiobjektin kanssa aloitetaan Käytä-painikkeella, mutta jatkossa tehtävät valinnat tehdään hyökkäyspainikkeella. Syy tähän on se, ettei pelaaja voi vahingossa ostaa jotain mitä ei halua sen takia, että hän painoi toistuvasti samaa painiketta toimiakseen objektin kanssa. Sen sijaan kauppa sulkeutuu aiheuttamatta haittaa pelisessiolle.

Tekstin piirtäminen itsessään on suunniteltu seuraavasti: tekstilaatikolle annetaan koordinaatit, mistä piirto aloitetaan. Laatikko piirretään sitten määrätyn kokoiseksi peliruudun koon mukaan keskelle ruutua. Tekstin kirjoitus aloitetaan ja teksti piirtyy kirjain kirjaimelta. Jos teksti on liian pitkä määritellylle laatikolle, siirretään viimeinen sana seuraavalle riville ja piirtämistä jatketaan siitä eteenpäin. Näin jatketaan kunnes teksti on piirretty. (HeartBeast, 2014)

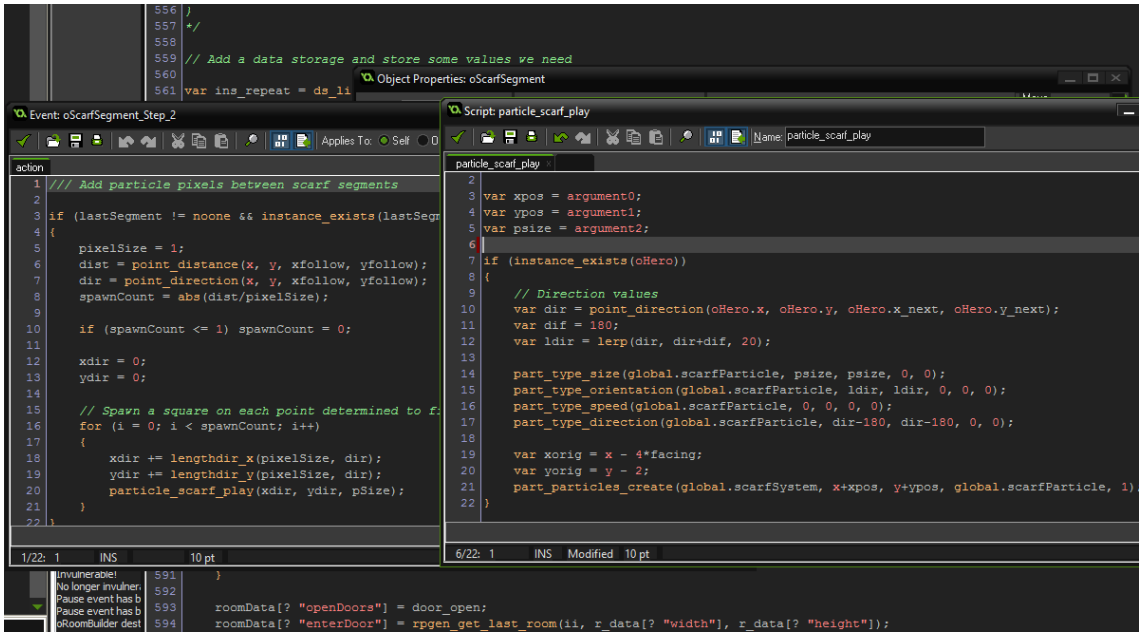


KUVA 13: Pelaaja ostamassa uutta hyökkäysjalokiveä

Kuten kuvasta 13 käy ilmi, ovat yksityiskohdat tällä hetkellä hiomattomia. Aiemmin mainitussa videossa suunniteltiin piirtäminen niin, että jos laatikko on liian suuri tekstille pystysuunnassa, pusketaan tekstiä ylöspäin niin, että ylin rivi häviää ikään kuin rullaavassa tekstitaulussa. Tämä on myös laitettu omaan projektiimme, mutta otettu käytöstä, sillä se on todettu tarpeettomaksi siitä syystä, että tekstit aiotaan pitää lyhyempänä ja täten helpommin luettavana.

3.6 PELIHAHMON DYNAAMINEN HUIVI

Yksi pelin erikoisempia efektejä on pelihahmon pitkä punaine huivi. Erikoita huivissa on se, että se seuraa pelaajaa dynaamisesti ja sen käyttäytyminen on pyritty saamaan mahdollisimman luonnolliseksi. Se liehuu hahmon juostessa suoraan tai tippuessa suoraan alaspäin. Huivi on rakennettu asettamalla kymmenen huivisegmentti, ”oScarfSegment” -objektia, niin, että ensimmäisenä luotu seuraa pelihahmoa, seuraava seuraa taas edellistäsegmenttiä ja niin edelleen. Objekteille on annettu tietty arvo pituusarvo, jonka ylittäessä sen on liikuttava kohti seuraamansa kohdetta. Seuraamisen nopeus on tehty ”lerppaamalla” nopeus, eli mitä kauempana segmentti on, sitä nopeampaa se liikkuu. Jokainen segmentti piirtää itsensä ja seuraamansa kohteen välille GameMakerin omalla partikkelijärjestelmällä punaisen vanan pikseli pikseliltä. Lerp-laskun avulla pikselien piirtoon saadaan sulava linja, jolla saadaan kuva yhtenäisestä, luonnollisesta huivista. Katso kuva 14.



KUVA 14: Pikselien paikkojen lasku ja lerp-laskun sisältävä piirtokohta.

3.7 HUD

HUD, eli heads-up display, on osa pelin graafista käyttöliittymää. Sen tehtävänä on välittää pelaajalle tietoa sen hetkisestä pelin tilanteesta, esimerkiksi pelihahmon elämäpisteistä, esineistä tai hyökkäyksistä. Riptalessa HUD:lla tarkastellaan pelaajan rahatilannetta, elämäpisteitä sekä hyökkäysjalokivien tilaa.

Olemme pyrkineet suunnittelmaan HUD:n ottaen huomioon pelin nopean toiminnan. Alunperin elämäpisteet oli tarkoitus merkata sydämminä keskelle ruudun ylälaitaa, missä jalokivet tällä hetkellä ovat. Huomasimme, että elämäpisteiden tilannetta on hankala seurata kesken taistelun, kun katse pitää irrottaa toiminnasta ja vilkaista ylös. Suunnittelimme sitten elämäpisteiden ilmaisu niin, että ne näkyvät pelihahmon yläpuolella. Ne eivät kuitenkaan näy koko ajan, vaan ainoastaan kolmessa tilanteessa: pelin alkaessa, hahmon ottaessa vauriota ja saadessa lisää elämää huoneista poimituista elämää palauttavista esineistä. Tällöin katsetta ei tarvitse tarpeettomasti irroittaa hahmosta ja taistelun sujuvuus ei siitä syystä kärsi.

Vahinkoa ottaessa ruutu myös vilkkuu muutaman kerran noin sekuntin kymmenyksen ajan niin, että kaikki valkoinen vaihtaa väriä punaiseksi ja takaisin. Hahmo myös lennähtää hieman taaksepäin osumapisteestä. Nämä on laadittu helpottamaan hahmon tilan ja toiminnan seuraamista.

Oikealla yläreunassa näkee myös pelaajan keräämä rahamäärä numerolukuna. Vaikka rahalukema on siirretty syrjemmäksi, sitä ei tarvitse olla vahtimassa yhtä tarkasti kuin elämäpisteitä, kauppoja kun ei tule vastaan niin tiheästi. (Masters, blog.digitaltutors.com)

4 POHDINTA

Siihen nähden, että peliä alettiin kehittämään vasta neljä kuukautta sitten, peli on edennyt valtavasti. Tästä huolimatta projekti on alkuperäisestä aikataulusta melko paljon myöhässä. Osittain tämä voi johtua siitä, että aikataulutus on tehty pieleen, mutta tähän vaikuttavat myös työryhmän pienuus ja kokemattomuus. Kyseessä on kuitenkin Super Godin ensimmäinen peli. Täysipäiväisenä kehittäjänä pelillä on lisäkseni ollut vain yksi henkilö, vastuuhenkilöni Super Godilla eli Juha Keränen, tuo iänikuinen jokapaikan höylä. Tulemme toki varsin hyvin toimeen keskenämme, mutta kuten olemme molemmat todenneet, on työryhmä vajavainen. Ryhmässä on ollut mukana useita henkilöitä neljän kuukauden aikana, joista osa on viihtynyt projektissa vain hetkeen ja sitten lähtenyt syystä tai toisesta.

Olen tykännyt työskennellä Super Godilla ja tämän projektin parissa. Ryhmän kanssa viihtyy ja Riptale on ollut haastava monelta osin, välillä on saanut grillata aivonystyröitä oikein urakalla ja välillä pähkäillä, miten GameMaker: Studion vajaatoimivuuksia saa kierrettyä. Jos aika, äly ja resurssit antavat myöten, kyllä tästä vielä hyvä peli saadaan. Olenhan sen itse lähes kokonaan ohjelmoinut, kjeh kjeh.

Tulen jatkossakin toimimaan pelinkehityksessä, se on varmaa. Se on joko Super Godin riveissä, jonkun toisen riveissä tai indienä, ihan miten vain. Ala tuntuu omalta nyt vajaan parin vuoden kokemuksella. Toki varmasti tulen tekemään muutakin softaa ja omistamaan aikaani muillekkin projekteille kuin pelkälle ohjelmoinnille, mutta mainittakoon, että allekirjoittaneen mielestä aivokopan käyttäminen tehokkaasti on hauskinta housut jalassa.

Mitä tulee GameMaker: Studioon, ehkä saatankin tehdä sillä vielä jotain jatkossakin. Toki, jotkut asiat sen suunnittelussa raivostuttavat, eikä sitä ole suunniteltu laajempiin projekteihin ei sitten ollenkaan. Pieniä pelejä sillä on varsin helppo ja nopea tehdä, ehkä se osoittautuu vielä hyväksi tavaksi tienata hieman extraa kaiken ohella. YoYo Games on vastaikää julkistanut GameMaker: Studio 2:n julkisen betatestin. Paljon on luvattu, mutta jää nähtäväksi, onko parannusta todellakin luvassa.

LÄHTEET

Norrgård, M. 2016. Tiled to Binary Converter
http://www.gamephase.net/tiled-to-binary-converter/

YoYo Games Ltd. GameMaker: Studio Version 1.4 Manual
http://docs.yoyogames.com/

Masters, M. Designing a HUD That Works for Your Game.
http://blog.digitaltutors.com/designing-a-hud-that-works-for-your-game/

Shaker, N. Togelius, J. Nelson, M. 2016. Procedural Content Generation in Games: A Textbook and an Overview of Current Research
http://pcgbook.com/

Chen, J, 2006. Flow in Games.
http://www.jenovachen.com/flowingames/Flow_in_games_final.pdf

Shiffman, D. 2016. The Nature of Code, Chapter 4. Particle Systems
http://natureofcode.com/book/chapter-4-particle-systems/

HeartBeast, 2014. Game Maker Tutorial: RPG Text Box Example
https://www.youtube.com/watch?v=b1hT-nzIds4

HeartBeast, 2014. [GameMaker Tutorial] Why Parent Objects?
https://www.youtube.com/watch?v=vteX3BfXnek