Markus Mäkinen

Pelin prototyypin kehitys Monogamella

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tietotekniikka

Insinöörityö

27.4.2015

Metropolia

Tekijä(t) Otsikko Sivumäärä Aika	Markus Mäkinen Pelin prototyypin kehitys Monogamella 35 sivua 5.5.2015
Tutkinto	Insinööri (AMK)
Koulutusohjelma	Tietotekniikka
Suuntautumisvaihtoehto	Ohjelmistotekniikka
Ohjaaja(t)	Lehtori Miikka Mäki-Uuro Lehtori Juha Kämäri

Insinöörityön tavoitteena oli kehittää pelin prototyyppi Monogamea käyttäen, samalla selvitettiin Monogamen hyvät ja huonot puolet. Tavoitteena oli myös oppia 3d-mallinnusta ja proseduraalisen sisällön tuottamista.

Monogamen yksi parhaista puolista on, että sen käyttäminen ei maksa mitään pelinkehityksessä Windows-alustoille. Monogameen on helppo tuoda sisältöä, kuten 3d-malleja ja kuvia, sekä piirtää ne ruudulle. Huonona puolena Monogamessa joutuu hyvin todennäköisesti ohjelmoimaan varjostimia, jos valmiit efektit eivät riitä toteuttamaan esimerkiksi haluttua valaistusta. Toisaalta tämä mahdollistaa myös juuri sellaisten varjostimien käytön kuin peli tarvitsee.

Peliin tarvittavien 3d-mallien teossa tutustuttiin syvällisemmin Blender 3d -mallinnussovelluksen käyttöön. 3d-mallien teko oli yksi eniten aikaa vievimmistä osa-alueista pelin kehityksessä. Pelihahmon mallinnuksen jälkeen mallille toteutettiin myös riggaus ja skinnaus.

Pelin tasot luotiin proseduraalisesti. Tason generointi proseduraalisesti oli melko helppo toteuttaa. Proseduraalisella generoinnilla on mahdollisuuksia vähentää pelinkehityksen vaatimaa aikaa huomattavasti, mutta sisällöstä on myös vaikeampi saada mielenkiintoista tai näyttävää.

Peliin saatiin toteutettua suurin osa määrittelyssä luetelluista vaatimuksista. Monogamen käytössä ei ilmennyt suuria ongelmia, ja apua ongelmatilanteisiin löytyi helposti. Yksi suurimmista ongelmista oli saada proseduraalisesti generoidun tason seinien graafinen esitys virheettömäksi.

Avainsanat	Monogame, pelinkehitys, proseduraalinen generointi

Metropolia

Author(s) Title Number of Pages Date	Markus Mäkinen Game prototype development with Monogame 35 pages 5 May 2015
Degree	Bachelor of Engineering
Degree Programme	Information technology
Specialisation option	Software engineering
Instructor(s)	Miikka Mäki-Uuro, Senior Lecturer Juha Kämäri, Senior Lecturer

The objective of this thesis was to develop a game prototype using Monogame and identifying the pros and cons of developing with Monogame. As an objective was also to learn 3d modeling and procedural content generation.

One of the best features of Monogame is that it is free to use for game development for Windows platforms. It is easy to bring content to Monogame, e.g. 3d-models and pictures, and to draw them on the screen. One of the drawbacks in Monogame is that it is very likely necessary to program shaders, for example if the premade effects are not sufficient enough for the required lighting. On the other hand this allows using just the kind of shaders that the game needs.

While doing the 3d models needed for the game, Blender 3d modeling software became increasingly familiar. 3d modeling was one of the most time consuming areas of developing the game. After modeling the game character, the model was also rigged and skinned.

Game levels were generated procedurally. Generating the levels procedurally was quite easily achieved. Procedural generation has the ability to possibly reduce the time needed for game development, but it is also harder to make the content interesting or visually pleasing.

Most of the requirements listed in the development plan were implemented in the game. While developing the game with Monogame, no serious problems were encountered, and help for problems was easily found. One of the biggest problems was to get the graphical display of the walls of the procedurally generated level error free.

Keywords	Monogame, game development, procedural generation

Metropolia

Sisällys

Lyhenteet

1 Johdanto 1

2 Monogame 2

2.1 3d-mallien piirtäminen 2
2.2 Pelisilmukka 8
2.3 Sisällön tuominen 9

3 Määrittely 10

3.1 Grafiikka 10
3.2 Taso 11
3.3 Tekoäly 12
3.4 Pelihahmot 13
3.5 Pelimekaniikka 14

4 Toteutus 14

4.1 Luolaston generointi 14
4.1.1 A*-algoritmi 15
4.1.2 Tasojen generointi 19
4.1.3 Huoneiden generointi 20
4.1.4 Käytävien generointi 21
4.1.5 Portaiden generointi 23
4.1.6 3d-mallien luonti 24
4.2 Grafiikka 25
4.3 Liikkuminen 27
4.4 Hyökkääminen 28
4.5 Vuorot 29
4.6 Näkökenttä 29

5 Yhteenveto 31

Lähteet 34

1 Johdanto

Insinöörityön aiheena on pelin prototyypin kehittäminen Monogamella. Monogame on avoimen lähdekoodin implementaatio Microsoftin XNA-ohjelmistokehyksestä. XNA:sta julkaistiin viimeinen versio vuonna 2011, ja sen kehitys on lopetettu. Monogamea kehitetään edelleen ja sillä voikin hyvin korvata XNA:n. Monogamella pelien kehittäminen ei ole myöskään rajoitettu vain Windows-alustoille, kuten XNA:lla oli. Jos on joskus ohjelmoinut XNA:lla, ei ole vaikea siirtyä käyttämään Monogamea, sillä ne ovat hyvin samankaltaiset. Työssä etsittiinkin ongelmatilanteissa yleensä ensimmäiseksi XNA:lle ohjeita, sillä XNA:lle löytyy helpommin ohjeita, ja ne toimivat lähes aina Monogamessakin.

Peli on 3d-grafiikalla toteutettu roguelike. Roguelike tarkoittaa pelien lajityyppiä, joka on saanut nimensä vuonna 1980 ilmestyneestä pelistä Rogue. Roguelike-peleille on tyypillistä, että pelaaja tutkii satunnaisesti luotua luolastoa ja vaikeustaso on korkea. Satunnaisesti luotu pelimaailma takaa sen, että yksikään pelikerta ei ole samanlainen. Pelien grafiikka on lähes aina toteutettu joko kirjoitusmerkein tai 2d-grafiikalla. [1.]

Työn päätarkoituksena oli oppia lisää 3d-grafiikan käytöstä ja teosta peliohjelmoinnissa ja proseduraalisesta sisällön luonnista. Pelissä luodaan proseduraalisesti vain luolaston tasot, mutta esimerkiksi Dwarf Fortress -pelissä luodaan mm. koko maailman historia, geografia ja maailman asukkaat persoonallisuuksineen proseduraalisesti. [2.]

Työssä käydään ensimmäiseksi läpi tärkeimpiä asioita Monogamella ohjelmoinnista, kuten mitä 3d-mallien piirtäminen ja sisällön tuominen peliin vaatii. Kolmannessa luvussa käydään läpi pelin määrittely, eli suunnitelma pelin ohjelmointia varten. Kaikki määrittelyssä luetellut vaatimukset eivät välttämättä päätyneet toteutukseen asti. Neljännessä luvussa kerrotaan miten ja millä toiminnoilla peli loppujen lopuksi toteutettiin. Neljännessä luvussa käydään läpi myös roguelike-pelien yhtä tärkeintä osa-aluetta, eli luolaston luontia.

2 Monogame

Monogamella voi kehittää pelejä täysin ilmaiseksi, kun käyttää kehitysympäristönä Visual Studion ilmaisversiota. Ohjelmointikielenä Monogamessa on C#. Pelin kehittämisessä käytettiin Visual Studio 2013 Express versiota. Androidille- ja Macille-pelejä kehittäessä pitää hankkia Xamarin Studio, joka ei ole ilmainen. Xamarin Studio vaatii toimiakseen myös vähintään Visual Studio Professional -version, joka ei myöskään ole ilmainen. Windows-alustoille kehittäminen voi kuitenkin olla ilmaista. [11.]

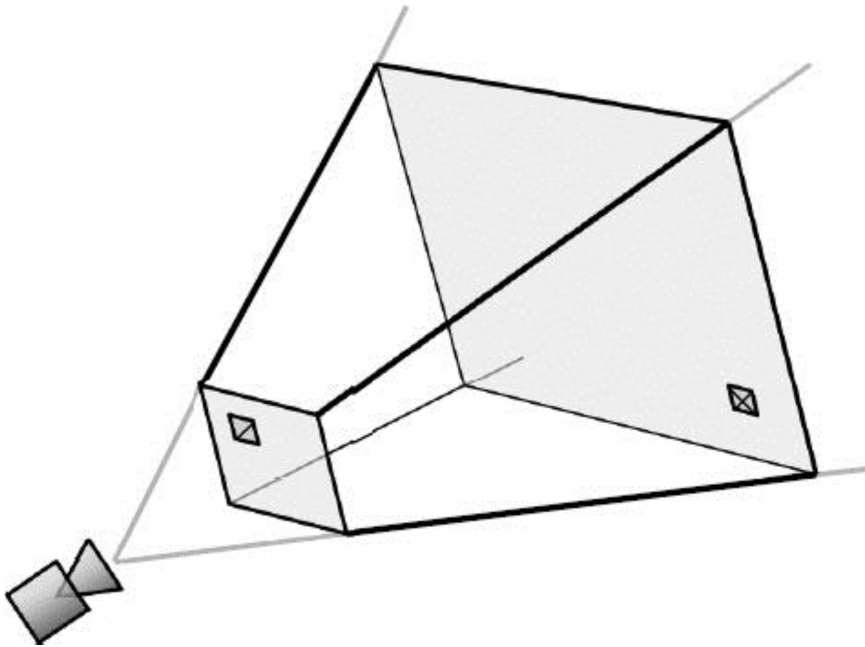
Pelissä käytettiin pääosin 3d-grafiikkaa. UI ja tekstit käyttävät 2d-grafiikkaa. Tässä osiossa kerrotaan tärkeimmät asiat 3d-grafiikan piirtämisestä Monogamessa. Osiossa kerrotaan myös Monogamen pelisilmukasta ja sisällön, kuten tekstuurien ja 3d-mallien tuomisesta Monogameen.

2.1 3d-mallien piirtäminen

Monogamessa 3d-mallien piirtäminen tapahtuu matriisien avulla. Matriiseja tarvitaan kolme, jotka ovat projektio-, näkymä- ja maailmamatriisi. Matriisien lisäksi tarvitaan efekti, joka määrittelee, miten 3d-mallit piirretään maailmaan.

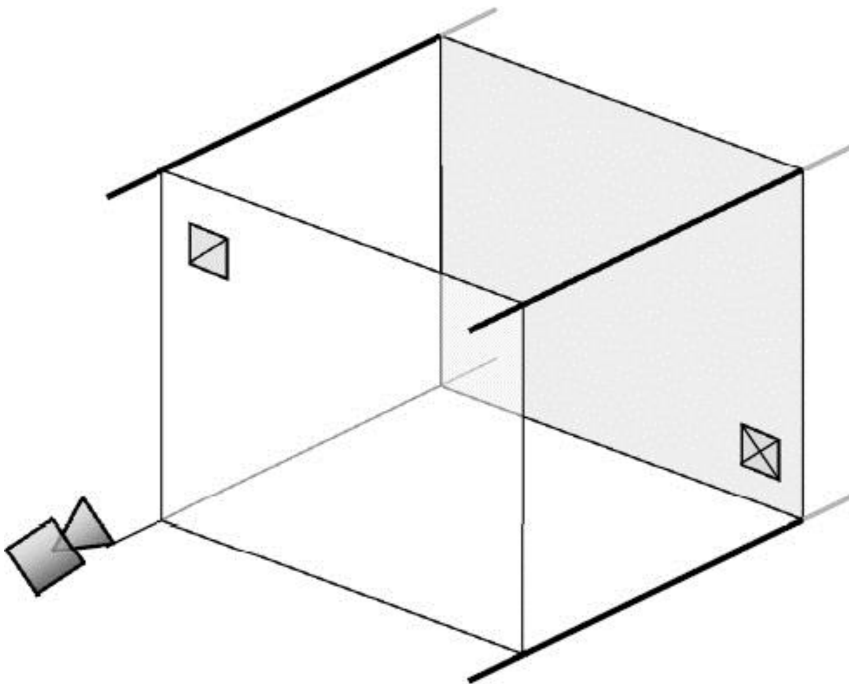
Matriisit

Projektiomatriisille määritetään perspektiivi projektiossa suorakulmion muotoinen katkaistu kartio, eli alue, jonka sisällä olevat 3d-mallit piirretään. Tämän alueen ulkopuolelle jäävät 3d-mallit eivät piirry. Ortografisessa projektiossa alue on suorakulmaisen särmiön muotoinen. Projektiomatriisin avulla muutetaan 3d-avaruudessa sijaitsevat mallit 2d-tasoon. Monogamesta löytyvät valmiit metodit perspektiivi- ja ortografisen projektion luomiseen. Perspektiivi-projektio luodaan Matrix.CreatePerspectiveFieldOfView -metodilla. Perspektiiviprojektiota havainnollistaa kuva 1.



Kuva 1. Perspektiiviprojektio [8. s. 212]

Ortografinen projektio luodaan Matrix.CreateOrtographic -metodilla ja projektiota havainnollistaa kuva 2.

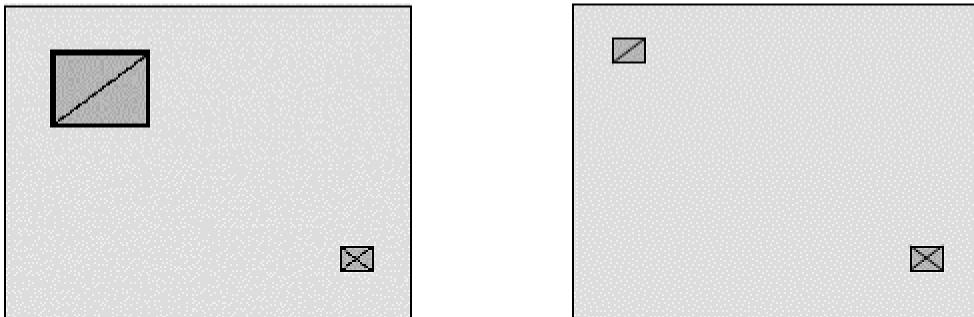


Kuva 2. Ortografinen projektio [8. s. 251]

Matrix.CreateOrtographic -metodin parametrit width ja height kertovat, kuinka monta yksikköä vaaka- ja pystysuunnassa näytetään. Esimerkkinä widthin arvolla 10 näytön

keskellä oleva kohta on nolla, vasen reuna -5 ja oikea reuna 5. Width-arvo kannattaa myös kertoa ikkunan kuvasuhteella, että projektion kuvasuhde on oikea. Kuvasuhde saadaan jakamalla ikkunan leveys ikkunan korkeudella.

Perspektiiviprojektio näyttää 3d-mallit pienempinä ja suurempina riippuen etäisyydestä kameraan. Ortografinen projektio näyttää 3d-mallit yhtä suurina riippumatta niiden etäisyydestä kameraan. Suurin osa peleistä käyttää perspektiiviprojektiota. Ortografinen projektio on kuitenkin hyödyllinen monissa strategia- ja roolipeleissä. Tässä pelissä käytettiin ortografista projektiota. Kuvassa 3 havainnollistetaan kahden samansuuruisen objektin, joista toinen on lähellä kameraa ja toinen kaukana kamerasta, näkymistä kamerassa perspektiivi- ja ortografisessa projektiossa. [8. s. 211 – 255.]



Kuva 3. Ensimmäisenä kuvassa on perspektiivi näkymä kamerasta (kuva 1) ja toisena ortografinen näkymä kamerasta (kuva 2). [8. s. 213, 251]

Näkymämatriisi on käytännössä katsoen kamera. Näkymämatriisin avulla selvitetään, mitkä 3d-mallit piirretään ja mihin kohtaan näkymässä. Näkymämatriisinkin tekoon Monogamesta löytyy valmis metodi Matrix.CreateLookAt. CreateLookAt tarvitsee seuraavat parametrit:

- cameraPosition: Vektori joka kertoo kameran sijainnin.
- cameraTarget: Vektori joka kertoo mihin kohtaan kamera katsoo.
- cameraUpVector: Vektori joka kertoo mihin suuntaan on ylös.

CameraUpVector:in arvoksi voi useimmissa tapauksissa antaa vektorin (0, 1, 0), eli y-koordinaatin arvo on 1. Arvoa muuttamalla voidaan saada aikaiseksi kameran pyörittäminen, mikä on tarpeen esimerkiksi lentosimulaattoreissa.

Maailmamatriisi on tarkoitettu 3d-mallien liikuttamiseen, kääntelyyn ja skaalaukseen. Lähes aina ennen kuin 3d-mallia piirretään pitää maailmamatriisi asettaa identiteettimatriisiksi. Identiteettimatriisilla asetetaan kaikkien transformaatioiden alkupisteeksi vektori (0, 0, 0), jolloin käännökset ja skaalaus on poistettu. Tämän jälkeen voidaan maailmamatriisiin asettaa halutut transformaatiot, kuten liikkuminen. Ensimmäisen transformaation jälkeen uusi transformaatio pitää kertoa maailmamatriisilla.

Monogamessa 3d-mallin transformaatioiden tekeminen on helppoa valmiiden metodien avulla. 3d-mallin liikuttamiseen voidaan käyttää metodia Matrix.CreateTranslation. Parametriksi annetaan Vector3, jolla määrätään, mihin kohtaan maailmassa malli halutaan siirtää. 3d-mallin kääntämiseen eri akseleiden suhteen löytyvät esimerkiksi seuraavat metodit, jotka ovat Matrix.CreateRotationX, Matrix.CreateRotationY ja Matrix.CreateRotationZ. Nämä metodit vaativat parametrikseen käännöskulman radiaaneina. Skaalaukseen löytyy metodi Matrix.CreateScale, jolle voi antaa parametrina liukuluvun, jonka verran mallia skaalataan tasaisesti joka akselilla. Metodeille löytyy myös erilaisia kuormituksia, jotka vaativat eri parametrit.

Transformaatioiden järjestyksellä on suuri merkitys. Lopputulos ei ole sama, jos mallia ensin siirretään ja sitten käännetään, kuin jos sitä ensin käännetään ja sitten siirretään. Tätä voisi ajatella siten, että päädytään eri kohtaan, jos kävellään suoraan eteenpäin ja sitten käännytään paikallaan, kuin jos ensin käännytään ja sitten kävellään suoraan eteenpäin. Skaalaus vaikuttaa myös siten, että jos ensin mallia skaalataan kaksinkertaiseksi, niin jokainen askel on myös kaksinkertaisesti enemmän. Skaalaus on usein järkevintä asettaa maailmamatriisiin viimeisenä, ettei se vaikuta muihin transformaatioihin. Transformaatiot siis asetetaan yksi kerrallaan maailmamatriisiin, ja jokainen uusi transformaatio on suhteessa siihen, mitä maailmamatriisissa on jo. Tämä on syy siihen, miksi maailmamatriisiin laitetaan aluksi identiteettimatriisi, että kaikki vanhat olemassa olevat transformaatiot poistuvat uusien tieltä. Kun kaikki halutut transformaatiot on asetettu maailmamatriisiin, voidaan malli piirtää. Seuraava piirrettävä malli asettaa taas aluksi maailmamatriisiksi identiteettimatriisin, että edellisen mallin transformaatiot eivät sekoita nyt vuorossa olevan mallin piirtoa.

Projektio-, näkymä- ja maailmamatriisi pitää aina asettaa efektiin, että niissä olevat tiedot tulevat voimaan. Projektiomatriisi tarvitsee asettaa efektiin ainakin kerran, mutta jos tähän tarvitsee tehdä muutoksia, on projektiomatriisi asetettava uudelleen efektiin. Näkymämatriisi pitää asettaa uudelleen efektiin, kun kamera liikkuu tai kääntyy, eli käy-

tännössä kerran pelisilmukan päivityksen aikana. Maailmamatriisia tarvitsee päivittää useimmiten, vähintäänkin kerran jokaisen eri 3d-mallin piirron yhteydessä. [8. s. 162 - 177.]

Efektit

Efektit ovat Monogamessa HLSL-kielellä toteutettuja pieniä ohjelmia. Efekti-tiedostot sisältävät varjostimia (engl. shader). Efektit ovat välttämättömiä 3d-grafiikan piirtämiseen Monogamessa. Monogamessa on monta valmista efektiä käytettäväksi, mutta niitä voi myös itse luoda. Esimerkiksi pistevaloa ei voi toteuttaa Monogamen valmiilla efekteillä, vaan efekti pistevaloa varten pitää itse luoda.

Efekteissä on ainakin kaksi erilaista varjostinta 3d-mallien piirtämistä varten. Pikselivarjostin muuttaa yksittäisiä pikseleitä, ja verteksivarjostimen tehtävänä on muuttaa 3d-mallin geometriaa. Verteksivarjostin käyttää aiemmin mainittuja matriiseja mallien transformaatioiden laskemiseen, eli projektio-, näkymä- ja maailmamatriiseja. Pikselivarjostin voi käyttää sitten verteksivarjostimen tulosta esimerkiksi mallin värjäämiseen punaiseksi. 2d-grafiikan muuttamisessa tarvitaan vain pikselivarjostinta.

Taulukossa 1 on lueteltuna kaikki Monogamen valmiit efektit ja mitä niillä on mahdollisuus tehdä. Taulukossa aina tarkoittaa, että tämä ominaisuus on aina käytössä ja sitä ei tarvitse aktivoida erikseen. Taulukossa valinnainen tarkoittaa, että jos ominaisuutta haluaa käyttää, se on ensin aktivoitava. Ominaisuus ei ole kuitenkaan pakollinen, ja efekti toimii ilman ominaisuuden aktivoimista. Osassa efektejä ei ole mahdollisuutta käyttää joitain ominaisuuksia ollenkaan. Efekteille pitää myös muistaa antaa oikeat muuttujat, esimerkiksi jos efekti käyttää teksturointia, se tarvitsee Texture2D:n toimiakseen.

Taulukko 1. Monogamesta löytyvät valmiit efektit ja niiden ominaisuudet. [8. s. 297 - 298]

Ominaisuus	**Basic**	**AlphaTest**	**EnvironmentMap**	**DualTexture**	**Skinned**
Projektio-, näkymä- ja maailmamatriisi	Aina	Aina	Aina	Aina	Aina
Diffuusi väri	Aina	Aina	Aina	Aina	Aina
Läpinäkyvyys	Valinnainen	Valinnainen	Valinnainen	Valinnainen	Valinnainen
Verteksin värjäys	Valinnainen	Valinnainen	-	Valinnainen	-
Teksturointi	Valinnainen	Aina	Aina	Aina	Aina
Taustavalaistus	Valinnainen	-	Aina	-	Aina
Suuntavalaistus	Valinnainen	-	Aina	-	Aina
Spekulaari valaistus	Valinnainen	-	-	-	Aina
Pikselintarkka valaistus	Valinnainen	-	-	-	Valinnainen
Itsesäteilyvalaistus	Valinnainen	-	Aina	-	Aina

Basic-efekti on valmiista efekteistä joustavin, ja se riittää useimpien 3d-mallien piirtämiseen. Mikäli mallit tarvitsevat monimutkaista animointia, pitää käyttää Skinned-efektiä tai luoda vastaavanlainen efekti itse. AlphaTest-efektiä voidaan käyttää Basic-efektiä paremmin mallien läpinäkyvyyden toteuttamiseen. EnvironmentMap-efektiä käytetään heijastusten tekoon. DualTexture-efektillä on mahdollista käyttää kahta tekstuuria samassa mallissa. AlphaTest- ja DualTexture-efektien suurena puutteena on valaistuksen puuttuminen. Basic-efektillä voi aktivoida perusvalaistuksen yhdellä metodikutsulla, joka on effect.EnableDefaultLighting. Valaistus koskettaa kuitenkin vain mallia, joka käyttää Basic-efektiä. Jokaista mallia ei myöskään tarvitse valaista samalla tavalla tai ollenkaan, vaan efektin ominaisuuksia voi muuttaa eri mallien piirroissa.

Pelissä käytettiin ainoastaan Basic-efektiä, mutta on mahdollista käyttää monia efektejä samaan aikaan. Esimerkiksi jos peliin tarvitsee animoituja 3d-malleja, voi niihin käyttää Skinned-efektiä, mutta muihin malleihin voi käyttää vaikka Basic-efektiä. Tällöin luodaan kaksi eri efektioliota, Basic-efekti ja Skinned-efekti oliot. Piirtometodissa pitää vain muistaa käyttää oikeata efektiä. [8. s. 296 - 308.]

2.2 Pelisilmukka

Monogamessa on valmis luokka pelisilmukkaa varten, joka sisältää kaiken tarvittavan pelin pyörittämiseen. Luokan nimi on Game, ja uutta projektia luotaessa Monogame luo oletusarvoisesti Game1-nimisen luokan, joka on periytynyt Game-luokasta. Game1:n nimi on hyvä vaihtaa vaikka pelin nimeksi.

Ensimmäinen metodi luokassa on Initialize. Tätä kutsutaan melkein heti konstruktorin jälkeen, ennen kuin pelisilmukka käynnistyy. Tässä metodissa alustetaan ennen peliä tarvittavat asiat kerran. Game-luokan tarvitsemat komponentit alustetaan tässä myös automaattisesti, kunhan base.Initialize -metodikutsua ei poisteta. Grafiikkasisältöä ei ladata tässä.

Seuraava metodi on LoadContent. Tämäkin metodi suoritetaan ennen kuin pelisilmukka käynnistyy. Tässä ladataan tarvittavat grafiikkaan liittyvät asiat, kuten tekstuurit, 3d-mallit ja fontit. Metodissa on hyvä myös luoda 2d-grafiikan piirtämistä varten SpriteBatch-olio tai 3d-grafiikan piirtämistä varten efekti. Kaikkea sisältöä ei välttämättä tarvitse tuoda tässä metodissa.

Alkuvalmistelujen jälkeen pelisilmukka käynnistyy. Pelisilmukassa on 2 metodia, jotka ovat Update ja Draw, eli päivitys ja piirto. Päivityksen ja piirron kutsumiskertoja sekunnissa voi vaihtaa, mutta oletusarvoisesti niitä kutsutaan 60 kertaa sekunnissa. Monogamen mobiilipeliprojekteissa kutsumistahti on 30 kertaa sekunnissa. Nopeutta voi vaihtaa muuttamalla TargetElapsedTime-ominaisuuden arvoa. Mikäli peli vaatii liikaa tehoja ja alkaa hidastua, Monogame vähentää piirtokutsuja, mutta pyrkii hoitamaan jokaisen päivityksen. Tämän takia on tärkeää, että vain piirrot suoritetaan Draw -metodissa ja esimerkiksi muuttujien päivitykset suoritetaan Update metodissa.

Jokaiselle peliobjektille kannattaa luokkia tehdessä luoda Update- ja Draw-metodit. Pelisilmukassa jokaisen peliobjektin näitä metodeja kutsutaan Game-luokan vastaavissa metodeissa. Tämä varmistaa sen, että vaikka piirtokutsu jäisi väliin, niin peliobjekti päivitetään kuitenkin. [8. s. 26 - 30.]

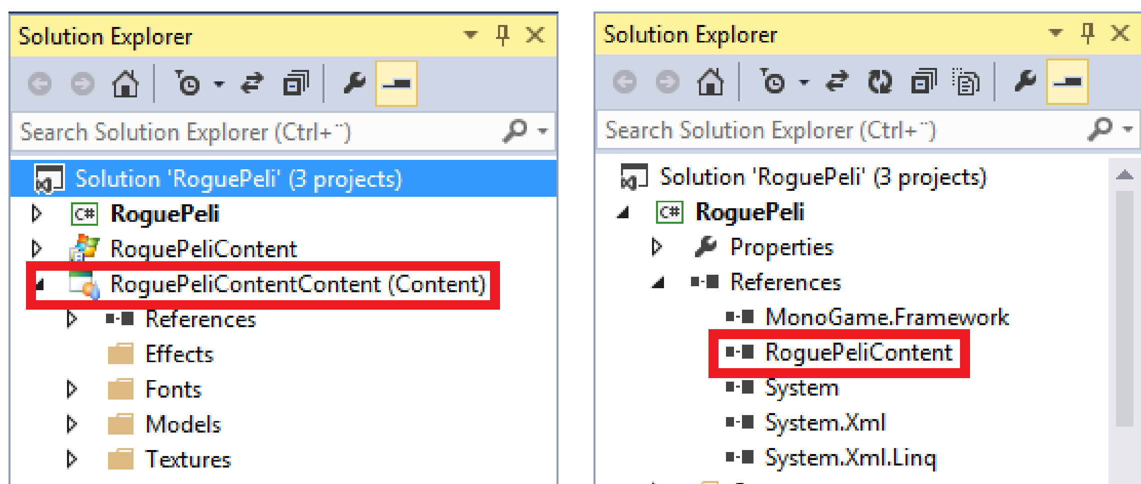
2.3 Sisällön tuominen

Monogamessa sisältö tuodaan peliin erillisen Content-projektin kautta tai Monogame Pipeline -sovelluksella. Tulevaisuudessa Monogame Pipeline Tool -sovellus on pääasiallinen tapa tuoda sisältö Monogameen, mutta pelissä sisältö tuotiin peliin Content-projektin kautta. Sisällöllä tarkoitetaan mm. tekstuureita, 3d-malleja, ääniä ja fontteja. Projektissa sisältö muutetaan .xnb -tiedostoiksi. Sisältö muutetaan siis Monogamelle ja esimerkiksi näytönohjaimelle sopivaan muotoon. Taulukossa 2 esitetään, mitä sisältöä Monogame tukee suoraan, mutta sisältökanavaan on mahdollista tehdä laajennus halutuille tiedostomuodoille. Sisältöä ei ole pakko tuoda Content-projektin sisältökanavan kautta, mutta se helpottaa sisällön lataamista huomattavasti. [10.]

Taulukko 2. Monogamen tukemat tiedostomuodot [9.]

Sisällön tyyppi	Tiedostomuodot
3d-mallit	.x, .fbx
Tekstuurit	.bmp, .dds, .dib, .jpg, .pfm, .png, .ppm, .tga
Äänet	.xap, .wma, .mp3, .wav
Fontit	.spritefont
Efektit	.fx

Tarvittava sisältö pitää ensin lisätä Content-projektiin (kuva 4). Content-projekti pitää myös muistaa linkittää varsinaiseen peliprojektiin lisäämällä peliprojektissa referenssi Content-projektiin. Kun haluttu sisältö on lisätty Content-projektiin, voidaan niitä käyttää pelissä.



Kuva 4. Vasemmalla projekti johon sisältötiedostot lisätään ja oikealla peliprojektiin linkitetty Content-projekti.

Sisältö ladataan peliin yleensä Game-luokan LoadContent-metodissa ennen pelisilmukan käynnistymistä. Sisältöä voi kuitenkin tarpeen mukaan ladata ja poistaa myös myöhemmin. Sisällön lataamisesta huolehtii luokka ContentManager, joka on Game-luokan komponentti. ContentManageriin saa yhteyden käyttämällä Game-luokan ominaisuutta Content. Käyttämällä kutsua Content.Load voidaan koodissa asettaa muuttujiin sisältö. Yleisimpiä sisällön käyttöön tarkoitettuja luokkia ovat seuraavat:

- Texture2D: kuva/tekstuuri
- Model: 3d-malli
- SpriteFont: fontti
- SoundEffect: ääniefekti

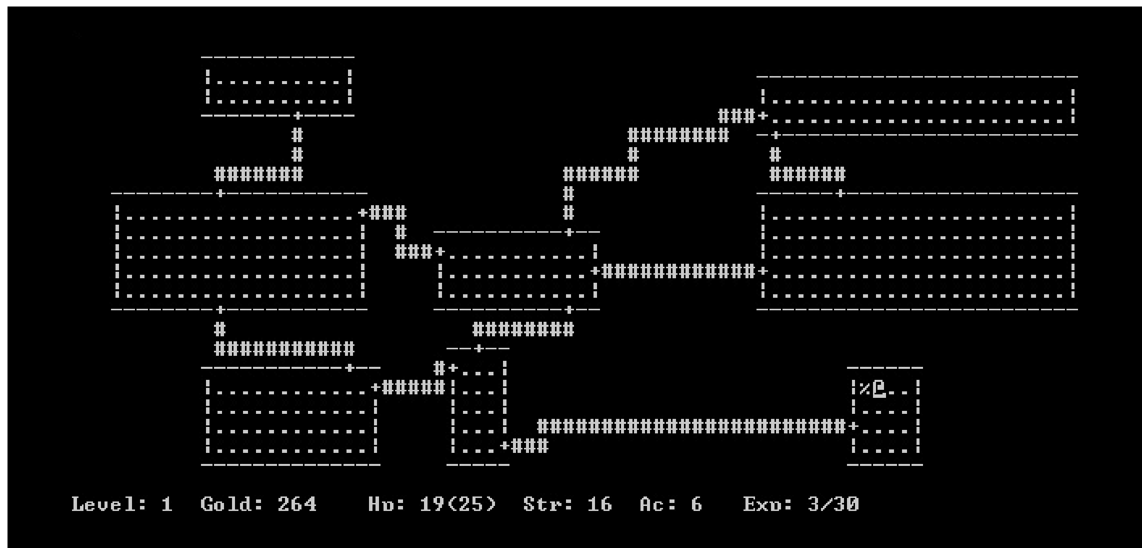
Kun ContentManager on ladannut muuttujiin sisällön, voidaan niitä käyttää pelissä. [9.]

3 Määrittely

Ennen pelin ohjelmointia oli tarvetta luoda suunnitelma pelissä tarvittavista asioista, kuten millaista grafiikkaa peli käyttää, millaisia pelihahmoja tarvitaan ja miten tasot luodaan. Kaikki määrittelyssä luetellut ominaisuudet eivät välttämättä pääse pelin toteutukseen ja saattavat muuttua alkuperäisestä suunnitelmasta.

3.1 Grafiikka

Tavanomaisesta roguelike-pelistä poiketen tämä peli toteutetaan 3d-grafiikalla. Lähes kaikki perinteisistä roguelike-peleistä on toteutettu joko kirjoitusmerkein tai 2d-grafiikalla (kuva 5). Syy 3d-grafiikan valintaan oli yksinkertaisesti halu oppia enemmän 3d-grafiikan käytöstä peliohjelmoinnissa. Pelin ohjelmoinnissa ei tarvitse juuri ottaa huomioon kolmatta ulottuvuutta, paitsi 3d-mallien sijoittamisessa. 3d-grafiikka vaatii hieman enemmän työtä kuin 2d-grafiikka, mutta siitä on myös hyötyä. Esimerkiksi syvyydestä ei tarvitse huolehtia matriisien ansiosta, kuten 2d-grafiikassa pitäisi. 2d-grafiikassa voi mahdollisesti myös joutua piirtämään isot määrät kuvia pelkän hahmon kääntelyyn, kun 3d-grafiikassa riittää kun luo yhden mallin.



Kuva 5. Kuvakaappaus pelistä Rogue. Pelaajaa edustaa kirjainmerkki @. [3.]

Pelin tasot piirretään ortografisesti, tarkemmin sanottuna isometrisesti. Tämä tarkoittaa pelissä käytännössä sitä, että kaukana ja lähellä kameraa olevat 3d-mallit näkyvät yhtä suurina. Myös kuvakulmaa on käännetty 45 astetta, ja se on ylhäältä.

Peliin tarvitaan 3d-malleja, jotka mallinnetaan Blenderillä. Blender on täysin ilmainen 3d-mallinnusohjelma. Vähimmäisvaatimuksena 3d-malleiksi tarvitaan ainakin pelaaja, yksi vihollinen ja tiilikarttaan tarvittavat seinät ja lattia. Pelaajalle tarvitaan myös kolme erilaista asetta ja vähintään yksi haarniska. Mallien animaatioita peliin ei tule ajan puutteen vuoksi.

3.2 Taso

Tärkeimpänä asiana pelin toimivuuden kannalta on, onko jokainen osa luolan tasoa saavutettavissa. Aivan kaikki roguelike-pelit eivät syystä tai toisesta pidä tätä vaatimusta pakollisena, mutta tässä pelissä tämä vaatimus on täytettävä. Ongelmia voisi syntyä ainakin, jos portaat alas ovat huoneessa, joka ei ole kytkeytynyt luolan osaan, jossa pelaaja on. Vähemmän haitallisina sivuvaikutuksina olisi, että pelaaja ei saisi kaikkia tavaroita kerättyä tasosta tai tapettua kaikkia vihollisia. Pelissä tasot generoidaan pelin alettua, joten tasoja ei generoida uudelleen, jos pelaaja menee portaat ylös ja alas, kuten jotkin roguelike-pelit tekevät. Tasojen uudelleen generoinnilla tasoa vaihtamalla tai mahdollistamalla pelaajalle seinien kaivauksen voisi estää yhteyksien puutteesta johtuvan umpikujan.

Tasossa pitää olla monta huonetta, jotka ovat erikokoisia ja ovat kytkeytyneet muuhun luolastoon käytävällä tai toisella huoneella. Huoneet ovat aluksi suorakulmion muotoisia, mutta jos ne ovat hieman päällekkäin, voidaan huoneet yhdistää yhdeksi isoksi huoneeksi. Samalla myös huoneen muoto muuttuu. Tämä tuo hieman vaihtelua ulkonäköön ja tasossa liikkumiseen. Huoneiden määrässä pitää olla vaihtelua eri tasojen välillä, ettei kaikissa tasoissa ole aina sama määrä huoneita. Tämäkin lisää tason ulkoasuun vaihtelua.

Käytäviä tarvitaan huoneiden yhdistämiseen muuhun tasoon. Käytävät eivät saa mennä huoneen nurkkien päältä tai huoneen seinän suuntaisesti. Käytävän pitää tehdä yhden ruudun kokoinen oviaukko kohtaan, josta se saapuu huoneeseen tai poistuu huoneesta. Jos käytävä menisi pelkästään huoneen nurkan päältä, jättäisi se ruman kolon huoneeseen. Jos käytävä menisi pitkin huoneen seinää, se poistaisi ison osan huoneen seinistä. Käytävien kaivamiseen käytetään A*-algoritmia.

Tasogeneraattorin pitää osata laittaa oikea 3d-malli jokaiseen ruutuun, eli katsoa viereisten ruutujen perusteella, mikä 3d-malli, millä rotaatiolla sopii juuri kyseiseen kohtaan. Kaikkiin ruutuihin ei tule 3d-mallia, vaan tasoissa on myös paljon tyhjiä ruutuja. 3d-mallien tekstuuria pitää myös voida vaihtaa satunnaisesti, etteivät kaikki seinät näytä samanlaisilta. Eri luolastoihin tai tasoihin pitää voida käyttää eri tekstuuria 3d-malleille.

Luolastossa pitää olla monta tasoa. Tasoja pitää voida vaihtaa siten, että tason tiedot tallentuvat aina. Tasoa vaihdettaessa tallennetaan ensin nykyinen taso ja ladataan taso, johon pelaaja on menossa. Jokainen esine ja vihollinen pitää myös tallentaa. Jokaisella tasolla pitää olla portaat alas ja ylös. Portaita käyttämällä pelaaja voi vaihtaa tasoa.

3.3 Tekoäly

Tekoälyn pitää osata liikuttaa vihollisia lähemmäksi pelaajaa, mutta vasta kun pelaaja on näköetäisyydellä. Tekoälyn pitää myös osata hyökätä pelaajan kimppuun, kun vihollinen on pelaajan vieressä. Jos pelaaja on kaukana vihollisesta, vihollisen pitää pysyä paikallaan tai liikkua satunnaisesti johonkin suuntaan. Tekoälyn päätöksenteko toteutetaan päätöspuutekniikalla, eli kun tietyt kriteerit täyttyvät, tekoäly toimii toimintasolmun

ohjeiden mukaisesti. Tekoälyn pitää myös liikkua omalla vuorollaan sääntöjen mukaisesti, eli toisten hahmojen päälle tai seinien päälle ei voi liikkua.

Tekoäly käyttää reitinhakuun A*-algoritmia. Reitinhakua käytetään vain, kun pelaaja on näköetäisyydellä. Muulloin vihollinen voi liikkua satunnaisesti johonkin suuntaan. Vihollisten pitää suorittaa reitinhaku uudelleen jokaisella vuorollaan, koska pelaaja on voinut liikkua tai eteen on voinut tulla este, kuten toinen vihollinen. Jos pelaaja siirtyy näköetäisyyden ulkopuolelle, eikä näy viholliselle moneen vuoroon, pitää tekoälyn lopettaa reitinhaku ja aloittaa satunnainen liikkuminen. Mikäli reitinhaku epäonnistuu jostain syystä, eikä hyökkäys ole mahdollinen, tekoälyn pitää osata lopettaa vuoronsa tekemättä mitään.

Käytävillä tai huoneissa voi välillä syntyä ruuhkaa. Tässä tapauksessa tekoälyn pitää osata järkevästi päättää, lähteäkö kiertoreittiä pelaajan kimppuun vai odottaako pelaajan lähellä olevien vihollisten kuolemista.

3.4 Pelihahmot

Pelihahmoja voi olla kahdenlaisia, eli pelaaja ja viholliset. Pelaajia pelissä on aina vain yksi ja vihollisia voi olla vaihteleva määrä. Jokaisella pelihahmolla tulee olla 3d-malli, jolla pelihahmo näytetään pelimaailmassa.

Pelaajalla ja vihollisilla on jokaisella omat ominaisuutensa, kuten hyökkäyksen aiheuttama minimi- ja maksimivahinko ja elämäpisteet. Näiden avulla voidaan suorittaa taistelu. Pelaaja ja viholliset voimistuvat pelin edetessä saamalla tasoja. Pelaaja saa tasoja tappamalla vihollisia, ja viholliset saavat tasoja riippuen kerroksesta.

Pelaajalla on myös mahdollisuus kerätä aseita, haarniskoita ja parannuspulloja. Aseilla parannetaan pelaajan hyökkäyskykyä, haarniskoilla vähennetään pelaajaan tulevaa vahinkoa ja parannuspulloilla voi saada nopeasti takaisin elämäpisteitä. Aseilla ja haarniskoilla on myös omat 3d-mallinsa, jotka näkyvät pelaajan päällä, kun ne puetaan päälle.

3.5 Pelimekaniikka

Peli on vuoropohjainen, eli jokainen pelaaja ja viholliset tekevät omalla vuorollaan tarvittavat asiat ja vuoron päätyttyä odottavat seuraavaa vuoroaan. Vuoro päättyy, kun vuorossa oleva hahmo esimerkiksi hyökkää tai liikkuu. Kaikki tapahtumat eivät vie saman verran aikaa, ja niihin vaikuttavat myös vuorossa olevan hahmon ominaisuudet. Osa hahmoista voi liikkua tavallista nopeammin tai ase vaikuttaa hyökkäyksen nopeuteen. Vuorot eivät siis tule aina samassa järjestyksessä vaan tehdystä toiminnosta riippuen vuoro voi alkaa aikaisemmin tai myöhemmin. Lähtökohtana on, että normaalilla nopeudella liikkuminen kuluttaa yhden aikayksikön, kuten myös hyökkääminen vaikka miekalla. Raskailla aseilla hyökkääminen taas kuluttaa enemmän aikaa ja kevyillä vähemmän.

Pelaajalla hyökkäys tapahtuu liikkumalla vihollista päin. Pelaajan tavaroista ja ominaisuuksista riippuen hyökkäys joko osuu tai ei. Vihollinen voi myös väistää onnistuneen hyökkäyksen. Hyökkäyksen onnistuttua pelaaja aiheuttaa vahinkoa viholliselle, eli vihollisen elämäpisteet vähenevät. Jos elämäpisteet putoavat nollaan tai sen alle, vihollinen poistetaan pelistä ja vihollisen tilalle asetetaan mahdollisesti jokin esine.

Pelaaja voi liikkua tasojen välillä käyttämällä portaita. Pelaaja aloittaa ylimmältä tasolta ja laskeutuu portaita pitkin alemmas. Mitä alempana pelaaja on, sitä vahvempia vihollisten pitää olla. Peli päättyy häviöön, jos pelaajan elämäpisteet putoavat nollaan tai alemmas. Peli päättyy myös, jos pelaaja pääsee alimmalle tasolle, jolloin pelaaja on voittanut pelin.

4 Toteutus

4.1 Luolaston generointi

Luolasto generoitiin peliin proseduraalisesti, eli tasot generoitiin pelkästään koodin avulla. Tämä säästää paljon aikaa sisällön luonnissa, koska tasoja ei tarvitse itse suunnitella tai toteuttaa. Jokaisesta tasosta tulee erilainen ja tasogeneraattorin parametreja muuttamalla vaihtelua voi lisätä entisestään eri tasoille. Tasoista on kuitenkin myös vaikeampi saada mielenkiintoisia tai järkevän näköisiä. Osa roguelike-peleistä

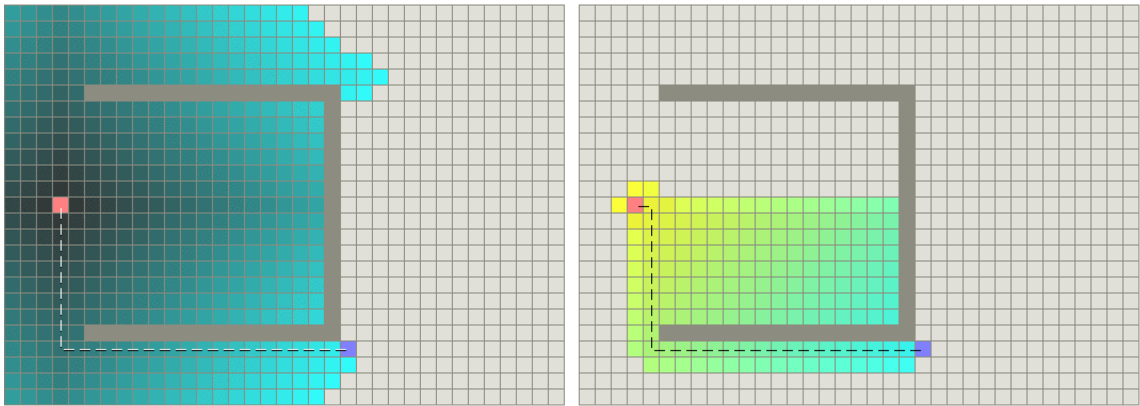
tekeekin suuren osan sisällöstään proseduraalisesti, mutta lisäävät muutamia karttoja, jotka on käsin tehty.

Algoritmeja tasojen generointiin löytyy monia erilaisia, jotka luovat hyvinkin erilaisia tasoja. Osa algoritmeista sopii hyvin esimerkiksi luolamaisen tason tekoon, kun osa taas tekee suorakulmion muotoisia huoneita ja yhdistää huoneet käytävillä. Pelissä kannattaakin käyttää montaa eri algoritmia tasojen generointiin, jotta tasoihin saa vaihtelua.

Sivustolta ”http://pcg.wikidot.com/pcg-algorithm:dungeon-generation” löytyy tietoa erilaisista tasonluontialgoritmeista. Tässä pelissä ei käytetty mitään valmista algoritmia, mutta niiden ideoita kylläkin. Ensimmäisessä tasogeneraattorin versiossa ongelmaksi muodostui käytävien kaivaminen. Käytävät menivät usein huoneiden seinien päältä, ja oviaukot muodostuivat välillä huoneen nurkkiin. Tämä ongelma ratkesi käyttämällä käytävien kaivamiseen A*-algoritmia, joka käydään seuraavaksi tarkemmin läpi.

4.1.1 A*-algoritmi

A*-algoritmia tarvittiin tasojen luonnissa käytävien kaivamiseen. Algoritmi pyrkii heuristiikan, eli arvauksen avulla löytämään nopeimman reitin aloituskohdasta loppukohtaan. Mikäli heuristiikkaa ei käytetä tai sen arvoksi annetaan nolla, kyseessä on Dijkstran-algoritmi. A*-algoritmi ei välttämättä aina löydä parasta reittiä, mutta on tehokas ja löytää reitin aina, jos se on mahdollista. Heuristiikkaa vaihtamalla algoritmia voi säädellä siten, että se löytää useammin lyhyimmän reitin tai suoriutuu nopeammin. Tässä pitää valita tehokkuuden ja reitin lyhyyden välillä. Yleistetysti pienillä heuristiikan arvoilla, eli aliarvioivalla heuristiikalla, löydetään lyhyempi reitti, mutta reitin haku kestää kauemmin. Suurilla heuristiikan arvoilla haku voi olla hyvin nopea, mutta reitti ei ole välttämättä lyhin. Kuvassa 6 nähdään Dijkstran-algoritmin ja A*-algoritmin käyttäytyminen samassa tilanteessa. Väritetyt ruudut ovat solmuja, jotka algoritmi käy läpi ennen maalia. Tapoja toteuttaa A*-algoritmi on monia. Seuraavaksi esitetään, miten se on pelissä toteutettu. [4.]



Kuva 6. Ensimmäisenä kuvassa Dijkstran-algoritmi ja toisena A*-algoritmi. [13.]

A*-algoritmi tarvitsee toimiakseen graafin eli verkon. Verkko muodostuu joukosta solmuja. Verkko on pelissä 2-ulotteinen taulukko solmuolioita. Solmut sisältävät seuraavat tiedot:

- x- ja y-koordinaatit, eli missä kohdassa verkkoa solmu on
- hinta: Kuinka paljon maksaa liikkua tässä solmussa
- g: Kokonaishinta alusta tähän asti, eli kuljettu matka
- h: Arvio matkasta maaliin tässä solmussa, eli heuristiikka
- f: Solmu jolla on alin f arvo, valitaan seuraavaksi vierailtavaksi solmuksi Lasketaan f = g + h
- naapurit: Lista solmuista, jotka ovat tämän solmun naapureita
- vanhempi: Naapuri solmu, jonka avulla lopullinen reitti selvitetään.

Verkko voidaan muodostaa, kun tasogeneraattori on sijoittanut huoneet. Pohjapiirustuksesta muodostetaan tiilien mukaan verkko. Pohjapiirustus käydään alusta loppuun läpi ja jokaisen alkion kohdalla luodaan uusi solmu olio. Solmulle annetaan luomisvaiheessa hinta, joka voi olla tyhjän- tai lattiatiilen kohdalla 10 ja seinätiilen kohdalla 30. Tämän takia algoritmi pyrkii välttämään kulkua seinien kohdalta. Jos seinän läpi kaivaminen maksaisi saman verran kuin muiden tiilien kaivaminen, algoritmi voisi helposti tehdä isoja aukkoja huoneisiin. Toinen tarvittava tieto luomisvaiheessa on x- ja y-koordinaatit, jotka vastaavat for-silmukan alkion indeksejä. Esimerkiksi alkion [4, 5] x-koordinaatti on neljä ja y-koordinaatti on viisi.

Kun verkossa on kaikki solmut, pitää niille asettaa naapurit. Naapureita voi olla jokaisella solmulla enintään neljä; verkon reunoilla niitä on vähemmän. Naapurit asetetaan pääilmansuuntien mukaisesti. Jokainen solmu käydään silmukassa lävitse ja samalla lisätään solmun naapurit listaan jokainen mahdollinen naapuri. Tämän vaiheen valmistuttua verkko on valmis käytettäväksi.

Reitin hakuun tarvitaan verkon lisäksi kaksi listaa: avointen ja suljettujen solmujen lista. Aluksi avointen listaan lisätään solmu, joka on reitinhaun alkukohta. Alkukohta on jonkin huoneen keskusta tai lähellä sitä, riippuen satunnaisluvuista. Alku- ja loppukohta talletetaan myöhempää vertailua varten solmuolioina. Nykyiseksi solmuksi asetetaan alkusolmu. Nykyinen solmu kertoo, missä kohtaa verkkoa algoritmi on. Nykyiselle solmulle lasketaan h:n arvo Manhattanin tavalla.

Manhattanin tapa lasketaan summaamalla matka pysty- ja vaakasuunnassa ja kertomalla liikkumisen hinnalla, joka on tässä kymmenen. Liikkumisen hintaan eivät vaikuta tässä esteet tai hidasteet. Liikkumisen hinnan tulisi olla aina sama, paitsi jos olisi mahdollista liikkua diagonaalisesti. Diagonaalisessa liikkumisessa hinnan pitää olla hieman suurempi, esimerkiksi neljätoista. Tässä pelissä käytävien kaivamisessa ei kuitenkaan haluta diagonaalista liikkumista. Seuraava kaava näyttää tavan laskea Manhattanin tapa, eli solmun heuristiikan arvon:

$$h = (|(x - endx)| + |(y - endy)|) * 10 \tag{1}$$

Avoimessa listassa on nyt yksi solmu, joka on aloitussolmu. Tämän jälkeen käydään avointen lista läpi ja etsitään pienin f:n arvo, joka on alussa tietenkin ainoalla solmulla listassa. Valitusta solmusta tulee uusi nykyinen solmu. Solmu siirretään avointen listasta suljettujen listalle. Tässä vaiheessa on hyvä tarkistaa, onko nykyinen solmu sama kuin loppusolmu. Reitinhaun voi päättää aina kun nykyinen solmu on loppusolmu, eli reitti alusta loppuun on silloin löytynyt.

Nykyisen solmun kaikki naapurit lisätään nyt avointen listaan, elleivät ne ole jo suljettujen listassa. Ennen lisäämistä avointen listaan naapureille lasketaan h-, g- ja f-arvot, sekä asetetaan nykyinen solmu vanhemmaksi. Jos naapuri oli jo avointen listassa, katsotaan, onko nykyinen reitti parempi naapurisolmuun vertaamalla g:n arvoja. Jos nykyinen reitti on parempi, vaihdetaan naapurin vanhemmaksi nykyinen solmu ja lasketaan sille uudestaan g:n ja f:n arvot.

Kun nykyisen solmun kaikki naapurit on käyty läpi, valitaan taas uusi nykyinen solmu. Avointen listasta katsotaan jälleen solmu, jolla on pienin f:n arvo. Tätä silmukkaa jatketaan niin kauan, kunnes nykyinen solmu on sama kuin loppusolmu tai avointen lista on tyhjä. Reittiä ei löytynyt, jos avointen lista on tyhjä eikä nykyinen solmu ollut loppusolmu. Käytävien kaivamisessa reitti kuitenkin löytyy aina.

Tässä A*-algoritmin toteutuksessa reitinhaun valmistuttua palautetaan tasogeneraattorille lista vektoreista, joiden kautta käytävä kaivetaan. Vektorit saadaan selville solmujen vanhempien avulla. Loppusolmusta aloittaen lisätään palautettavaan listaan käsiteltävän solmun x- ja y-koordinaateista muodostettu vektori. Tämän jälkeen vaihdetaan käsiteltäväksi solmuksi solmun vanhempi. Tätä jatketaan, kunnes käsiteltävällä solmulla ei ole enää vanhempaa, eli reitti on muodostettu lopusta alkuun vektoreina. Käytävien kaivamisessa ei ole väliä, aloitetaanko lopusta vai alusta. Lopputulos on kuitenkin sama. Pelihahmojen liikkumisessa lista pitäisi kuitenkin kääntää toisinpäin tai aloittaa reitinluku lopusta.

Yhteenveto pelissä käytetystä A*-algoritmista:

1) Lisää aloitussolmu avointen listaan.

2) Toista seuraavaa, kunnes nykyinen solmu on loppusolmu tai avointen lista on tyhjä:

   - Etsi avoimesta listasta solmu, jolla on pienin f:n arvo, josta tulee nykyinen solmu.
   - Vaihda nykyinen solmu avoimesta listasta suljettujen listaan.
   - Tee seuraavaa nykyisen solmun jokaiselle naapurille:

a) Jos naapuri on suljettujen listassa, älä tee mitään. Muuten jatka.

b) Jos naapuri ei ole avoimessa listassa, niin lisää se siihen. Aseta naapurin vanhemmaksi nykyinen solmu. Laske f, g ja h naapurille.

c) Jos naapuri on jo avoimessa listassa, katso onko nykyinen reitti parempi siihen. Jos reitti oli parempi, laske uudelleen naapurin g ja f, sekä aseta nykyinen solmu vanhemmaksi.

Pelissä käytetty A*-algoritmi ei ole missään tapauksessa ainut tai tehokkain tapa toteuttaa algoritmi. Esimerkiksi avointen ja suljettujen listan tyypin valinnalla voi olla suurikin merkitys algoritmin tehokkuuteen. [5.]

4.1.2 Tasojen generointi

Tasogeneraattori toteutettiin siten, että ensin tasoon generoidaan huoneet, tämän jälkeen käytävät ja lopuksi sijoitetaan portaat. Kaikki tästä saatu tieto tallennetaan 2-ulotteiseen kokonaislukutaulukkoon, jota kutsutaan pohjapiirustukseksi. Tässä vaiheessa taulukon alkiossa voi olla jokin seuraavista luetelluista tyypeistä: EMPTY, WALL tai FLOOR. Tällä tiedolla suoritetaan viimeinen vaihe, jossa luodaan 3d-mallit, eli graafinen esitys tasosta. Tähän tarvitaan toinen 2-ulotteinen taulukko Tile-olioita, jota kutsutaan tilemapiksi. Jokaisella luolaston tasolla on pohjapiirustus ja tilemap, jotka tasogeneraattori sille luo.

Tile-olio pitää sisällään mm. 3d-mallin, sijainnin ja rotaation. Rotaatiota tarvitaan siihen, että yhtä 3d-mallia voidaan käyttää useammin. Ei olisi järkevää mallintaa esimerkiksi pohjois- ja eteläseinää erikseen, vaan on parempi käyttää yhtä 3d-mallia, ja siirroilla ja rotaatiolla asettaa se oikeaan kohtaan tiilessä. Sijainti kertoo, mihin kohtaan 3d-maailmassa 3d-malli piirretään.

Pelin karttaan tarvittiin 7 erilaista 3d-mallia, kuten kulmaseinä, tuplaseinä ja lattia. Aluksi seinät oli tehty Monogamessa koodin avulla, mutta tarve erilaisille seinille kasvoi sen verran, että oli järkevämpää luoda 3d-mallit Blenderissä. Monimutkaisempien seinien tekstuurikoordinaattien asettaminen Blenderissä oli myös paljon helpompaa kuin kooditasolla olisi ollut. Huonona puolena tässä oli se, että kun alun perin tason graafinen esitys oli yhdessä isossa verteksi-puskurissa, niin nyt jokainen malli joudutaan piirtämään erikseen. Hyviä puolia oli taas paljon enemmän, kuten että nyt jokaisen 3d-mallin piirtämisen pystyy helposti estämään ja sallimaan. Tästä oli paljon hyötyä LoS-algoritmin toteutuksessa.

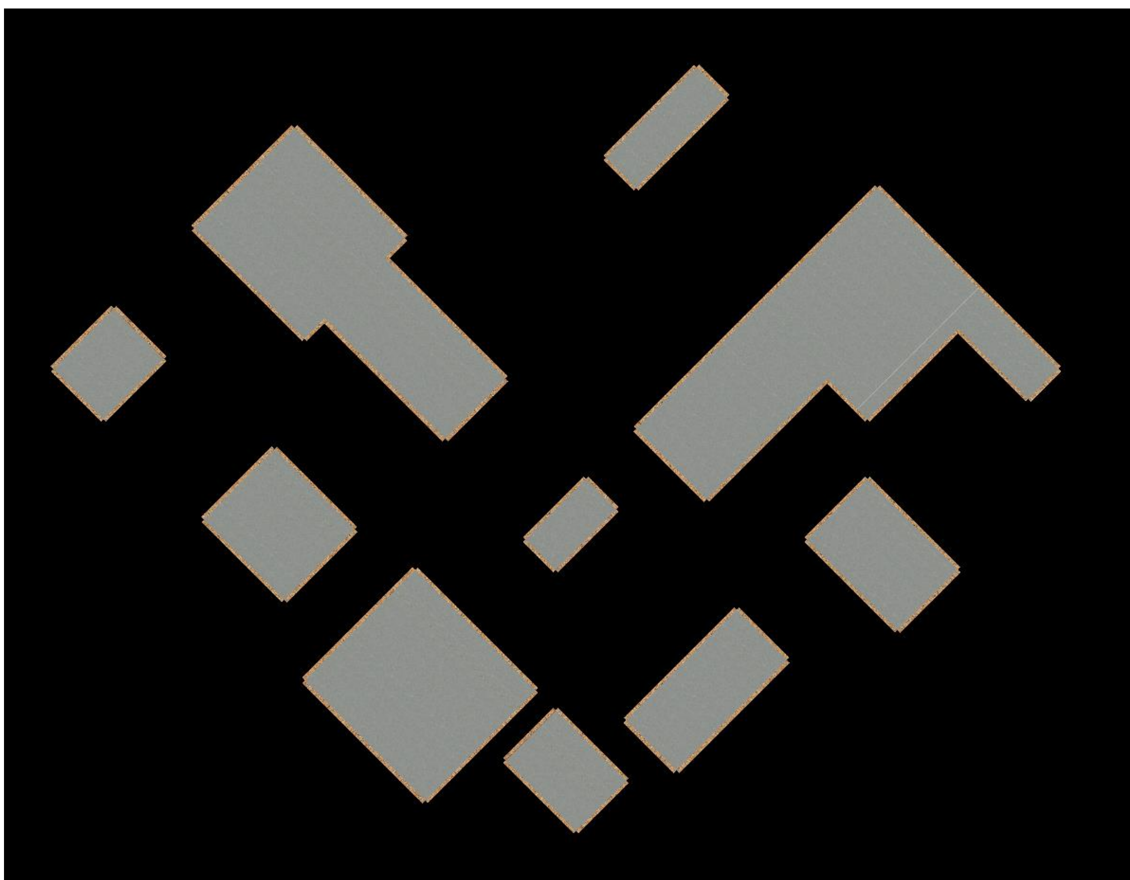
Jokainen taso on jaettu soluihin. Soluja voi olla kartan koosta riippuen vaihteleva määrä. Solut ovat suorakulmioita, jotka muodostetaan esimerkiksi jakamalla aluksi koko kartta neljään osaan ja nämä neljä osaa vielä neljään osaan. Näin saadaan 16 solua, joihin jokaiseen on mahdollista muodostaa huone. Jakamiskohdat ovat hieman satunnaisia, eli joistain soluista tulee todennäköisesti pienempiä ja joistain isompia. Solut eroavat toisistaan hieman kooltaan ja tietenkin sijainniltaan. Yksikään solu ei ole toisen päällä.

4.1.3 Huoneiden generointi

Ensimmäinen vaihe huoneiden generoinnissa on solujen teko, joihin huoneet lopulta voidaan sijoittaa. Halutulla todennäköisyydellä jokaiseen soluun voidaan muodostaa huone. Huone on vaihtelevan kokoinen suorakulmio. Huoneen kokoon ja sijaintiin vaikuttaa se, missä solussa se sijaitsee, sekä parametrit huoneen maksimi- ja minimikoko.

Huoneen saatua sijaintinsa ja kokonsa pitää huoneelle luoda seinät. Tämä tapahtuu yksinkertaisesti asettamalla jokaiseen suorakulmion reunakohtaan pohjapiirustuksessa luetellun tyypin WALL-arvo. Keskelle taas laitetaan lattiaa. Seinät luodaan jokaiselle huoneelle tässä vaiheessa. Tässä vaiheessa poistetaan myös vierekkäisten huoneiden välistä seinät ja asetetaan lattia tilalle, että saadaan isompia ja erimuotoisia huoneita.

Lopuksi tehdään vielä varmistuskierros koko taulukolle alusta loppuun, että yksikään lattia tiili ei ole tyhjän tiilen vieressä. Pelissä ei ole missään tapauksessa sallittua, että lattian vieressä olisi tyhjyyttä. Muutamassa tapauksessa huoneiden yhdistämisessä voi huoneen nurkkaan jäädä kolo ilman tätä varmistuskierrosta. Kuvassa 7 näkyy huoneiden generoimisen jälkeinen tilanne pohjapiirustuksessa.



Kuva 7. Kartta ylhäältäpäin kuvattuna, kun huoneet on generoitu.

Tässä vaiheessa tasolla ei ole oikeasti vielä graafista esitystä, mutta algoritmia muokattiin kuvaa varten siten, että käytävien ja portaiden teko jätettiin pois.

4.1.4 Käytävien generointi

Tässä vaiheessa taulukosta löytyvät huoneet ja tyhjät kohdat. Aluksi kaikki huoneet lisätään listaan notConnected, sekä luodaan toinen lista connected, joka on aluksi tyhjä. Tämä vaihe on valmis, kun notConnected-lista on saatu tyhjäksi, eli kaikki huoneet ovat yhdistyneet ainakin yhden huoneen kanssa.

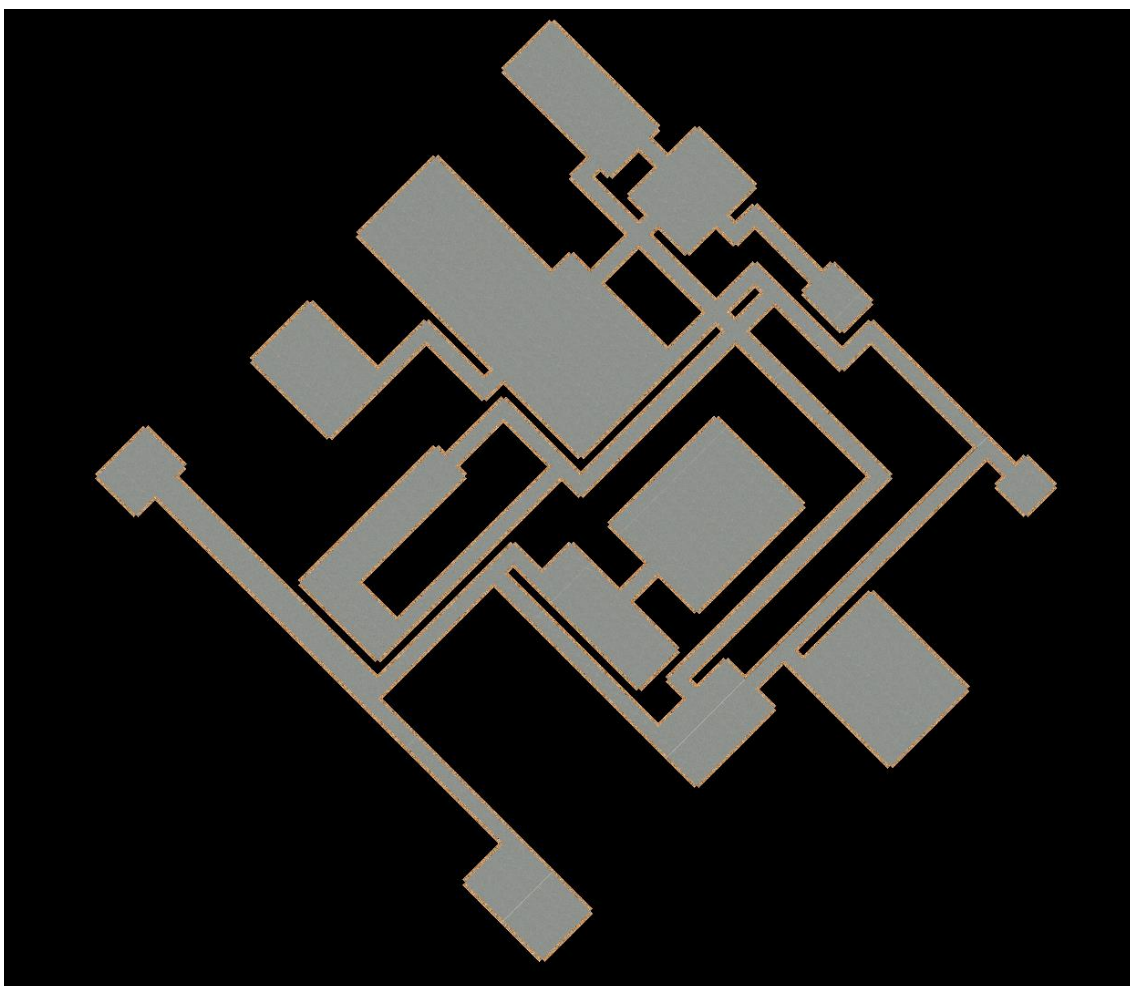
Aluksi arvotaan satunnaisluvulla kaksi huonetta notConnected-listasta, jotka yhdistetään. Nämä kaksi huonetta voivat sijaita hyvin lähellä tai kaukana toisistaan, sillä ei ole väliä. On parempi, että käytäviä voi olla pitkiä ja lyhyitä. Huoneet eivät tietenkään saa olla sama huone, joka myös varmistetaan tässä. Yhdistäminen tapahtuu A*-algoritmilla, joka etsii halvimman reitin kahden huoneen välillä. Seinien läpi kulkeminen on kolme kertaa kalliimpaa kuin tyhjän tai lattian kautta kulkeminen. Hintoja muuttamalla A*-

algoritmin käytöstä voi muuttaa paljon, esimerkiksi laittamalla lattian kautta kulkemisen halvimmaksi, jolloin se suosisi valmiita käytäviä enemmän.

A*-algoritmi palauttaa tasogeneraattorille listan vektoreista, joiden kautta reitti kahden huoneen välillä kulkee. Vektori sisältää kaksi koordinaattia, jotka kertovat, mihin kohtaan pohjapiirustuksessa muutokset tehdään. Nämä vektorit käydään läpi ja muutetaan jokainen tyhjäksi tai seinä lattiaksi. Lattian kohdalla ei tarvitse tehdä mitään. Jos reitti kulkee huoneen läpi, katsotaan, onko se jo kytkeytynyt. Jos huone ei ollut kytkeytynyt, poistetaan se notConnected-listasta ja lisätään connected-listaan. Näin yksi käytävä voi yhdistää monta huonetta, ja tehtävien käytävien määrä vähenee.

Kun ensimmäiset kaksi huonetta on yhdistetty, valitaan seuraavat kaksi huonetta siten, että yksi huone on notConnected-listalta ja toinen connected-listalta. Näin varmistetaan, että huoneet yhdistyvät muuhun tasoon. Nämä huoneet yhdistetään ja jatketaan niin kauan, kunnes notConnected-lista on tyhjä.

Käytävien kaivuun jälkeen käydään taas koko pohjapiirustus alkio kerrallaan lävitse ja katsotaan, onko lattian vierellä tyhjiä kohtia. Tässä vaiheessa tyhjiä kohtia on runsaasti, sillä käytävät eivät tee seiniä kuin vasta tässä vaiheessa. Jokainen tyhjä kohta lattian vierellä vaihdetaan seinäksi. Tämän kierroksen jälkeen kartta on melkein valmis, eli lattiat ja seinät ovat asetettuina. Kuvassa 8 nähdään kartta, kun huoneet ja käytävät on generoitu.



Kuva 8. Kartta ylhäältäpäin kuvattuna kun huoneet ja käytävät on generoitu.

Kuten kuvan 7 kohdalla tässäkään vaiheessa kartalla ei ole todellisuudessa vielä graafista esitystä.

4.1.5 Portaiden generointi

Tässä vaiheessa pitää asettaa kahdet portaat joka tasolle, mutta ei viimeiselle tasolle. Tämä on hyvin yksinkertaisesti toteutettu: luodaan kaksi satunnaislukua, eli x- ja y-koordinaatti. Tämän jälkeen katsotaan pohjapiirustuksesta, onko kyseisessä koordinaatissa lattia. Jos ei ole, niin arvotaan uudet koordinaatit ja katsotaan uudestaan, kunnes lattiakohta löytyy. Lattian löydyttyä porras voidaan asettaa siihen paikkaan. Sama tehdään toiselle portaalle, paitsi jos kyseessä on alin taso. Alimmassa tasossa ei saa olla portaita alas.

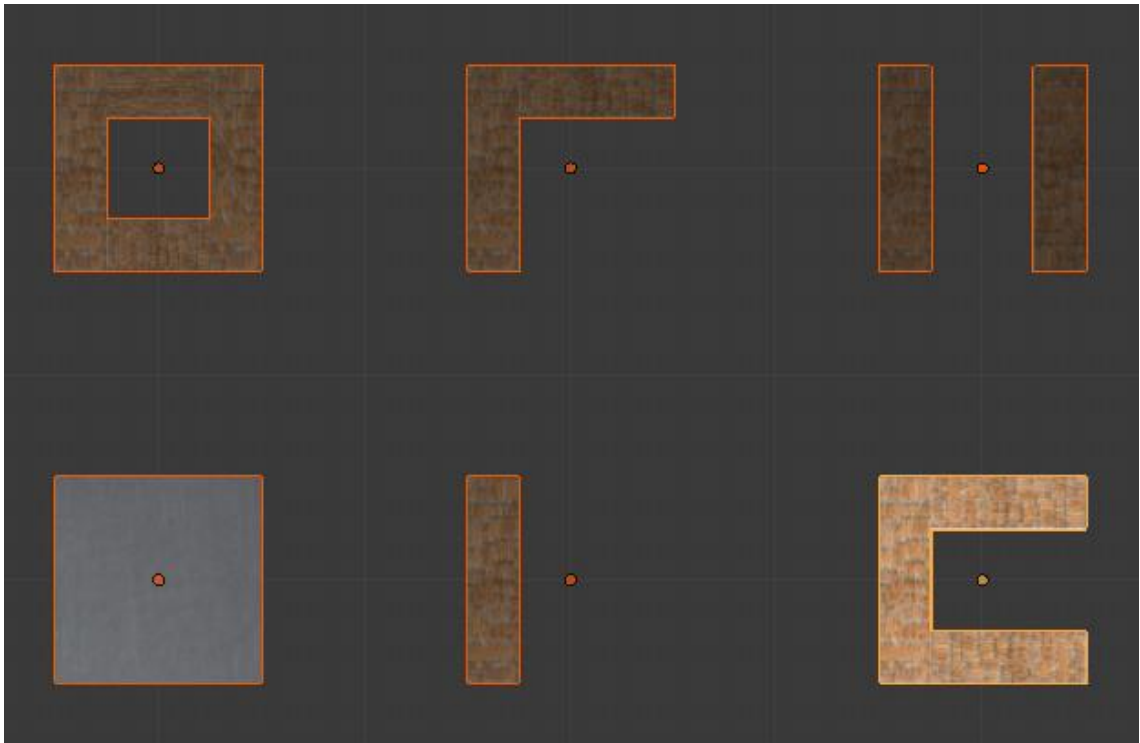
Taso on nyt valmis muutettavaksi 3d-malleiksi. Pohjapiirustus sisältää nyt kaikki huoneet, portaat ja käytävät seinineen.

4.1.6 3d-mallien luonti

Tason graafinen esitys kävi läpi monta muutosta. Ensimmäiseksi seinät olivat kuution muotoisia ja muodostettu koodin avulla vertekseistä. Kaikki seinät ja lattiat olivat yhdessä isossa verteksipuskurissa, eli koko kartta oli yksi malli. Tämä oli yksinkertainen ja tehokas ratkaisu piirron kannalta, mutta ei näyttänyt kovin hyvältä, eikä ollut tarpeeksi helposti muokattavissa kesken pelin. Lopulta tiilikarttaan tarvittavat 3d-mallit tehtiin Blenderillä.

Tilemapin jokaisella alkiolla on mahdollista olla yksi 3d-malli. Lattian 3d-malliksi riitti yksi neliön muotoinen taso. Seinien tapauksessa taas tarvittaisiin 15 erilaista mallia, jos ne haluaisi suoraan kartalle. Järkevämpää oli kuitenkin luoda 5 mallia, joita kääntämällä sai jokaisen halutun seinän.

Tasogeneraattori luo Tile-oliot pohjapiirustuksen valmistuttua. Pohjapiirustuksen avulla nähdään, mitä viereisissä koordinaateissa sijaitsee. Jokaisen seinän kohdalla tarvitsee katsoa naapureiden perusteella, minkälainen seinä kyseiseen kohtaan kuuluu. Esimerkiksi pilariseinän tunnistaa yksinkertaisesti siitä, että sen ympärillä on pelkkää lattiaa. Pilariseinä on seinä, jolla on neljä sivua eikä sitä tarvitse kääntää. Toisena esimerkkinä yksittäisen seinän tunnistaa siitä, että kolmella naapurilla ei ole lattiaa ja yhdellä on. Riippuen naapurista, jolla lattia oli, mallia käännetään siten että se sopii kohtaan. Yksittäinen seinä voi olla neljässä eri asennossa, eli neljään eri pääilmansuuntaan. Muut kolme seinää on tupla-, tripla ja kulmaseinä. Kuvassa 9 näkyvät kaikki tiilikarttaan tarvitut 3d-mallit ylhäältäpäin kuvattuna. Malleja käännetään pelissä kuvassa näkyvien pisteiden ympäri, jotta ne sopivat useampaan kohtaan tiilikartassa.



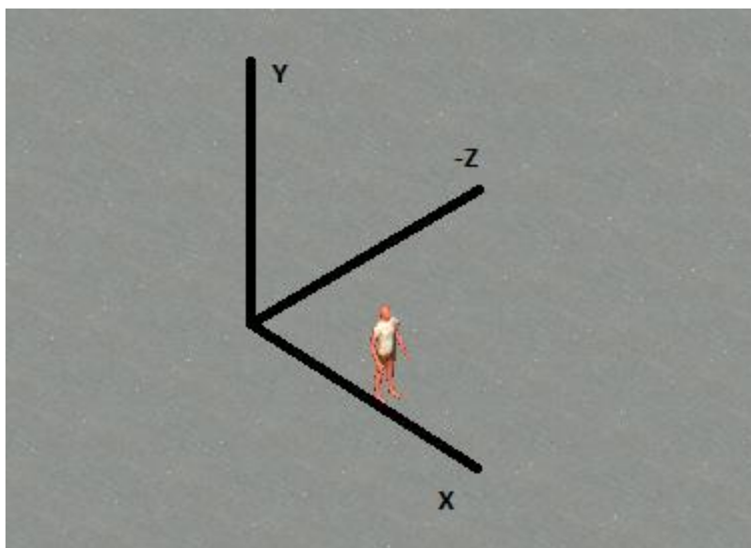
Kuva 9. Tiilikartan kaikki 3d-mallit keskipisteineen, jotka ovat pilariseinä, kulmaseinä, tuplaseinä, lattia, seinä ja triplaseinä.

Tasogeneraattorin päätettyä se tiili, joka sopii kuhunkin kohtaan, se siirtää ne tilemap-listaan. Tilemap-lista sisältää siis Tile-oliot, jotka ovat graafinen esitys tasolle. Tätä listaa läpikäymällä jokainen taso voi piirtää itsensä.

Tämän vaiheen päädyttyä tasogeneraattori voi viimein antaa tasolle pohjapiirustuksen ja tilemapin. Tämän enempää tasogeneraattorin ei tarvitse tehdä. Taso on kaikin puolin valmis käytettäväksi. Tasogeneraattori ei luo vihollisia tai esineitä.

4.2 Grafiikka

Pelissä käytettiin isometristä 3d-grafiikkaa. Tämä toteutettiin asettamalla projektiomatriiksi ortografinen projektio ja asettamalla kamera ylös, taakse ja 45 asteen kulmaan käännettynä katselupisteeseen suhteutettuna. Kameran liikkuessa katselupistettä liikutetaan samassa suhteessa. Koordinaatisto ja pelinäkymä ovat kuvan 10 mukaiset.



Kuva 10. 3d-koordinaatisto pelin isometrisessä näkymässä.

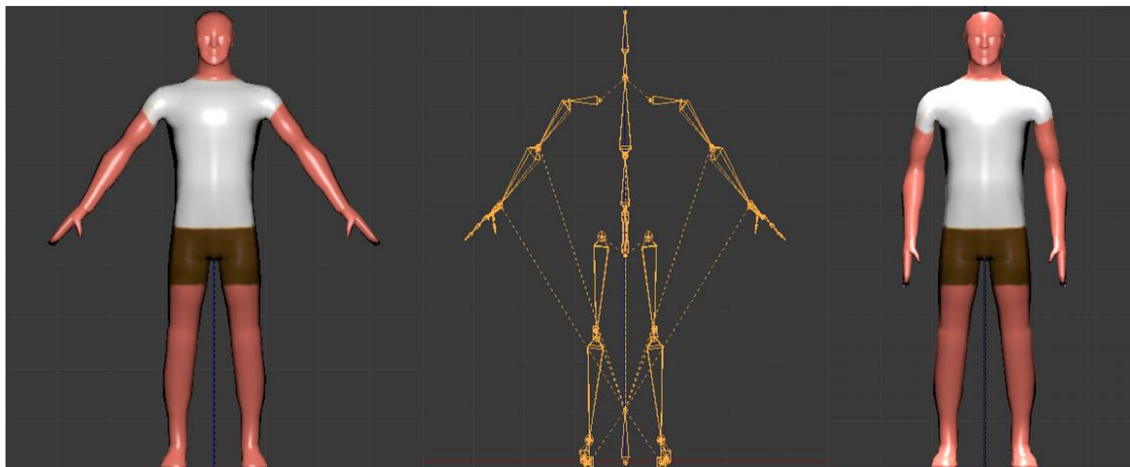
Isometristä näkymää käyttäessä pitää muistaa, että jos haluaa 3d-mallin liikkuvan oikealle, sitä ei voi siirtää x-akselilla pelkästään. Mallia pitää siirtää myös z-akselilla, muuten malli liikkuisi oikealle alas. Y-akselilla ei ole mahdollista liikkua pelissä, sillä sitä käytetään vain mallien sijoittamisessa maailmaan.

Pelin valaistus toteutettiin käyttämällä Basic-efektin perusvalaistusta. Kaikki mallit käyttävät Basic-efektiä. Kaikilla malleilla on tekstuurit, ja Tiili-oliot käyttävät yhteistä tekstuuriatlasta, eli kuvatiedostoa, jossa on monta kuvaa. Tekstuurit joita ei ole itse tehty, on hankittu sivustolta ”http://www.cgtextures.com”, joka sisältää paljon ilmaisia tekstuureja.

3d-mallit tehtiin Blender sovelluksella. Pelaajan 3d-malliin tehtiin lisäksi riggaus ja skinnaus, eli mallia voi animoida luiden avulla. 3d-mallien luominen ja riggaus on iso aihealue, eikä sitä käydä tässä tarkasti lävitse. Pelihahmot eivät ole animoituja, mutta riggauksen avulla mallien asentoja voi helposti vaihtaa. Asennon vaihtamisella ja tekstuurin vaihdolla voitiin yhdestä mallista luoda useampia vihollisia, eli alun perin pelaajan hahmoksi tarkoitettu malli kävi myös ihmisen kaltaisten vihollisten tekoon.

Pelaajan 3d-malli tehtiin laatikkomallinnustekniikalla. Mallin teko aloitettiin sylinteristä jalan kohdalta. Sylinteriä muokattiin vastaamaan ihmisvartalon piirustusta edestä ja sivulta jalkapohjista kaulaan asti. Vartalosta mallinnettiin vain toinen puoli ja toinen puoli saatiin käyttämällä Blenderin peilikuvamääritettä. Tällä tavalla malli on varmasti identtinen molemmilta puolilta. Mallin pää tehtiin erikseen ja lisättiin vartaloon myö-

hemmin. Kuvassa 11 näytetään valmis 3d-malli, luuranko ja 3d-malli luurangon avulla eri asentoon laitettuna.



Kuva 11. Ensimmäisenä kuvassa on teksturoitu valmis pelihahmo, keskellä luuranko animointia varten ja viimeisenä pelihahmo luurangon avulla muutettuna eri asennossa.

Riggaus toteutettiin käyttämällä IK-ketjua, joka tarkoittaa, että liikuttamalla ketjun viimeistä luuta aiemmat luut seuraavat perässä. IK-ketju helpottaa animointia, sillä esimerkiksi FK-ketjua käytettäessä joutuisi jokaisen luun asettamaan erikseen haluttuun paikkaan ja asentoon. Riggauksen jälkeen malli piti vielä skinnata, eli asettaa verteksit seuraamaan luita. Blenderissä tämän voi toteuttaa ainakin käyttämällä työkalua Weight Paint tai automaattisesti, kun luuranko ja malli yhdistetään. [12.]

4.3 Liikkuminen

Pelaajan liikkuminen toteutettiin lukemalla näppäinkomentoja. Pelaaja voi liikkua kahdeksaan eri suuntaan, eli pysty-, vaaka- ja vinottaissuunnassa. Vinottain liikkuminen kuluttaa yhtä paljon aikaa kuin muutkin liikkumissuunnat. Pelaajan yrittäessä liikkua käännetään ensimmäiseksi 3d-malli liikkeen suuntaiseksi. Kun malli käännetään aina liikkumista yrittäessä, saa pelaaja visuaalisen varmistuksen siitä, että liikettä yritettiin. Tämän jälkeen tarkistetaan, onko liike mahdollinen. Pelaaja voi liikkua vain lattian päällä, eikä kohdassa saa olla vihollista. Jos pelaaja liikkuu esimerkiksi seinää päin, 3d-malli käännetään, mutta vuoro ei pääty.

Viholliset ja pelaaja käyttävät liikkumiseen A*-algoritmia. Algoritmi on hieman erilainen kuin käytävien kaivamisessa käytetty. Algoritmia piti muokata siten, että pääilmansuun-

tien lisäksi voidaan liikkua diagonaalisesti. Verkon solmuilla voi olla nyt kahdeksan naapurisolmua. Tämän lisäksi solmut voivat olla varattuja, eli niissä voi olla jokin pelihahmoista. Varattujen solmujen läpi ei saa kulkea, ja tämä tarkistus piti lisätä myös algoritmiin. Käytävien kaivamisessa algoritmi sai mennä seinien läpi, mutta liikkumisessa tämä ei saa olla mahdollista. Tämäkin joudutaan tarkistamaan joka solmun kohdalla. Verkosta olisi voinut jättää pois kohdat, joissa on seiniä, mutta nämä lisättiin silti verkkoon, jos myöhemmin on tarvetta tehdä esimerkiksi vihollinen, joka voi liikkua seinien läpi.

Liikkumisen onnistuttua päivitetään A*-verkkoa. Pelaajan vanhassa sijainnissa oleva solmu asetetaan vapaaksi liikkeelle ja uudessa sijainnissa oleva solmu asetetaan varatuksi. Liikkumisen lopuksi pelaajan vuoro lopetetaan.

Vihollisille annettiin kaksi eri tapaa liikkua kartalla. Kun viholliset eivät näe pelaajaa, tekoäly yrittää liikuttaa vihollista satunnaiseen suuntaan. Tätä liikkumista kokeillaan kerran, ja jos liike ei ollut sallittu, lopetetaan vuoro. Viholliset päivittävät myös sijaintinsa verkkoon onnistuneen liikkumisen jälkeen.

Kun vihollinen havaitsee pelaajan, lopetetaan satunnaiseen suuntaan liikkuminen ja liikutaan lähemmäs pelaajaa. Reitti pelaajan luokse etsitään A*-algoritmilla. Reitti haetaan uudelleen joka vuorolla, koska pelaaja ja muut viholliset muuttavat jatkuvasti verkkoa.

4.4 Hyökkääminen

Hyökkäyksen mahdollisuus toteutettiin käyttämällä hyväksi A*-algoritmin löytämän reitin pituutta. Jos reitin pituus on yksi, se tarkoittaa, että hyökkäys voi olla mahdollinen. Vihollinen tai pelaaja katsoo, onko tiilikartassa reitin lopussa olevassa alkiossa pelihahmo, jota vastaan voi hyökätä. Viholliset eivät voi hyökätä toisten vihollisten kimppuun. Ainoastaan pelaaja ja vihollinen voivat hyökätä keskenään.

Hyökkäyksen lopputulos selvitettiin yksinkertaisella metodilla. Ensiksi katsotaan, osuuko hyökkäys, jonka perusonnistumismahdollisuus on 80 prosenttia. Jos hyökkäys osuu, voi pelihahmo vielä yrittää väistää hyökkäyksen ominaisuuksista riippuvalla todennäköisyydellä. Väistön epäonnistuttua voidaan aiheuttaa vahinkoa osuneeseen pelihah-

moon. Pelaajan tekemän vahingon määrä riippuu aseesta ja voimasta. Vihollisilla ei ole aseita, ja vahingon määrä riippuu pelkästään ominaisuuksista. Aiheutettu vahingon määrä vähennetään pelihahmon elämäpisteistä. Mikäli elämäpisteet putoavat nollaan tai sen alle, pelihahmo kuolee ja poistetaan pelistä. Pelaajan kuollessa peli loppuu.

4.5 Vuorot

Pelin vuorojen toteutuksessa päädyttiin käyttämään "tiimalasia" selvittämään, kenen vuoro on seuraavaksi. Pelin alettua kaikilla pelihahmoilla on tiimalasissa 100 aikayksikköä. Jokainen pelihahmojen suorittama toiminto lisää tietyn verran aikayksiköitä. Normaalilla nopeudella liikkuminen lisää 100 aikayksikköä. Aseet vaikuttavat pelaajan hyökkäysnopeuteen. Raskaat aseet lisäävät enemmän aikayksiköitä kuin kevyemmät. Pelihahmojen ominaisuudet vaikuttavat myös nopeuteen.

Vuorojen vaihtumisesta ja järjestyksestä huolehtii staattinen luokka TurnResolver. TurnResolver etsii pelihahmon, jolla on vähiten aikayksiköitä tiimalasissaan. Pelihahmon vuoron päätyttyä vähennetään jokaisen pelihahmon tiimalasista saman verran aikayksiköitä kuin vuorossa olleella pelihahmolla oli aikayksiköitä tiimalasissa vuoron alussa. Vuoronsa päättäneelle pelihahmolle lisätään aikayksiköitä tiimalasiin tehdyn toiminnon verran. Tämän jälkeen etsitään taas pelihahmo, jolla on vähiten aikayksiköitä tiimalasissaan. Tasapelin kohdalla annetaan vuoro listalta ensimmäisenä löytyneelle pelihahmolle.

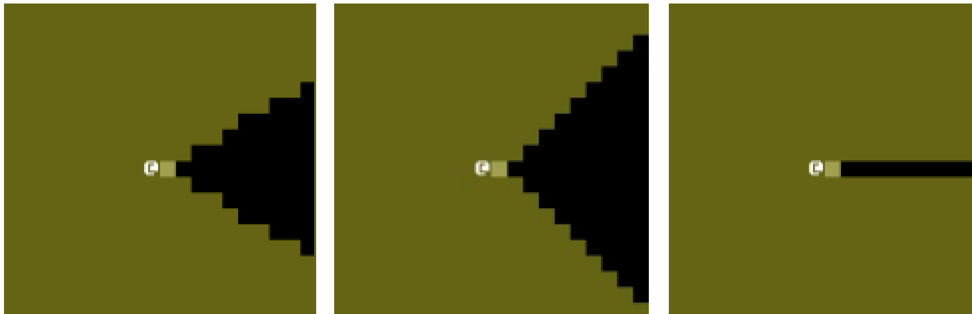
Vihollisen tullessa vuoroon TurnResolver käskyttää tätä tekemään päätöksen vuorostaan. Vihollisen tekoäly ottaa tässä vaiheessa hallinnan ja päättää, mitä tekee vuorollaan. Pelaajan vuoro päättyy, kun pelaaja liikkuu, hyökkää tai odottaa.

4.6 Näkökenttä

Pelaajalle piti toteuttaa näkökenttä, jonka avulla karttaa ja vihollisia piirretään sen mukaan, mitä pelaaja näkee. Erilaisista LoS-algoritmeista oli vaikea päättää mikä valitaan. Moni algoritmeista oli vaikeasti toteutettavissa tai eivät ominaisuuksiltaan sopineet peliin. LoS-algoritmin valinnassa pitää valita, haluaako symmetrisen vai epäsymmetrisen näkökentän. Symmetrinen näkökenttä tarkoittaa, että jos vihollinen voi nähdä pelaajan, niin pelaajakin voi nähdä vihollisen. Epäsymmetrisessä näkökentässä voi käydä niin,

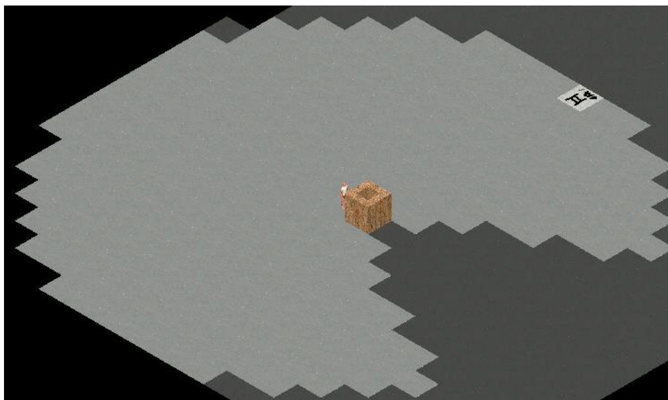
että vihollinen näkee pelaajan, mutta pelaaja ei näe vihollista. Tässä valinta kohdistui symmetriseen näkökenttään.

Toiseksi LoS-algoritmin valinnassa piti miettiä, miten haluaa kartan ruutujen näkyvyyden. Lähes kaikki suositut LoS-algoritmit eroavat toisistaan tässä. Kuvassa 12 voidaan nähdä, miten muutama suosittu algoritmi käyttäytyy, kun pelaajan edessä on pilari. @-merkki edustaa kuvassa pelaajaa, musta väri aluetta, jota ei voi nähdä ja pelaajan oikealla puolella oleva neliö pilaria. Loput alueesta on aluetta, joka näkyy pelaajalle. [6.]



Kuva 12. Basic, shadow ja diamond/digital LoS-algoritmien toiminta pilarin takaa. [3.]

Pelissä käytössä oleva LoS-algoritmi muistuttaa eniten Basicin toimintaa. Kuvassa 13 on näkyvissä pelin käyttämän LoS-algoritmin toiminta pilarin takaa. Kuvasta voi huomata, että toiminta eroaa hieman Basicin toiminnasta ja suuresti muiden kuvassa 12 näkyvien algoritmien toiminnasta. Tumman harmaa on aluetta, johon pelaaja ei tällä hetkellä näe, mutta on kerran aikaisemmin nähnyt. Musta alue on aluetta, jota ei ole vielä ikinä tutkittu. Vaalean harmaa alue on alue, jonka pelaaja pystyy nyt näkemään.



Kuva 13. Pelissä käytössä olevan LoS-algoritmin toiminta pilarin takaa.

LoS-algoritmin valintaan vaikutti lopulta vain sattuma. Ensimmäinen algoritmi jota pelissä testattiin, olikin juuri sopiva. Algoritmi löytyi roguebasin sivustolta, josta löytyy paljon muitakin LoS-algoritmeja. Tälle algoritmille ei löytynyt varsinaista nimeä, mutta se löytyi sivulta "http://www.roguebasin.com/index.php?title=Eligloscode". Algoritmin perusideana on ampua säteitä pelihahmon ympärille. Näiden säteiden perusteella päätetään, mitkä osat kartasta ovat näkyvillä.

Pelihahmon muuttujasta visionRange saadaan säteiden pituus. Pelaajalla visionRange on oletusarvoisesti 10. Ennen säteiden ampumista pitää jokaisen potentiaalisesti näkökentän piirissä olevan tiilen isVisible-muuttujan arvo muuttaa epätodeksi, kuitenkin varmistaen, ettei liikkumisen seurauksena jollekin tiilelle jää isVisible-arvo todeksi.

Kun kaikki tiilet ovat isVisible arvoltaan epätosia, voidaan ampua säteet. Säteitä ammutaan 720 kappaletta. Pienemmillä määrillä säteitä algoritmi ei toiminut tarpeeksi tarkasti. Esimerkiksi 360 säteellä suorassa käytävässä tuli keskelle käytävää kohta seinässä, jota pelaaja ei voinut nähdä. Tästä kauempana oleva seinä kuitenkin näkyi. Vasta kohottamalla säteiden määrän 720:een kaikki virheet hävisivät.

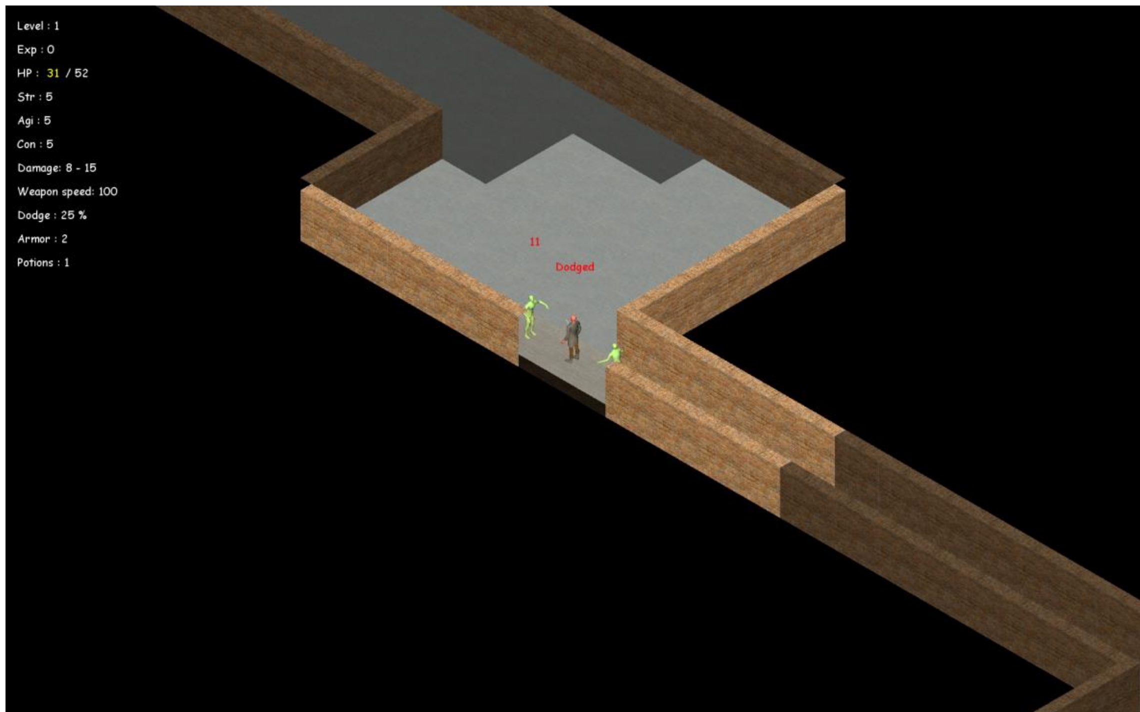
Säde tutkii jokaista Tile-oliota matkansa varrella ja asettaa tämän isVisible-arvon todeksi. Kun vastaan tulee seinä, asetetaan sen isVisible-arvoksi tosi, mutta säteen eteneminen pysäytetään. Tämä estää sen, että pelaaja voisi nähdä seinän toiselle puolelle. Tämän jälkeen ammutaan uusi säde eri kulmasta ja toistetaan samat operaatiot. Tätä jatketaan kunnes kaikki 720 sädettä on ammuttu.

Tile-olioilla on myös isDiscovered-muuttuja, joka kertoo, onko tiiltä nähty kertaakaan aikaisemmin. Säde asettaa jokaiseen osuneeseen tiileen tämän arvoksi toden. Tiili käyttää isDiscovered- ja isVisible-arvoja piirtämisessä. Jos tiiltä ei ole ikinä nähty, eli isDiscovered on epätosi, ei 3d-mallia piirretä ollenkaan. Jos tiili on joskus nähty, mutta ei nähdä juuri nyt, malli piirretään harmaalla värillä. Jos tiili nähdään, niin se piirretään normaalisti. [7.]

5 Yhteenveto

Pelin prototyyppiin saatiin toteutettua suurin osa määritelmässä luetelluista vaatimuksista. Kaikkea märittelyssä lueteltua ei saatu täysin toteutettua, tai ne muuttuivat to-

teutuksessa tarpeen vaatiessa. Määrittelykin muuttui hieman alkuperäisestä, koska jo kehityksen alkuvaiheessa huomattiin, että jotkin ideat eivät olleetkaan käytännössä sopivia peliin tai keksittiin parempi tapa tehdä ne. Kuvassa 14 nähdään kuvakaappaus pelin prototyypistä kehityksen valmistuttua.



Kuva 14. Pelaaja ja kaksi zombieta taistelemassa.

Monogamen käytössä ei ilmaantunut suurempia vaikeuksia, ja moneen ongelmatilanteeseen löytyi neuvoja eri verkkosivuilta. Ongelmatilanteessa voi hakea myös XNA:lle ohjeita, sillä on suuri todennäköisyys, että ohjeet toimivat Monogamessakin. Monogamella on hyvin helppo ja nopea saada jotain näkymään ruudulle, esimerkiksi muutamalla rivillä koodia saa jo kuvan piirrettyä ruudulle. Sisällön tuonti peleihin on helppoa, jos ei tarvitse erikoisempia tiedostomuotoja. HLSL-kielen opettelu voi olla hyvin todennäköisesti edessä, jos Monogamella haluaa tehdä 3d-pelejä. Monogamen valmiilla efekteillä onnistuvat 3d-piirron perusasiat, mutta esimerkiksi pistevalon joutuu jo toteuttamaan itse tehdyllä efektillä. Varjostimien ohjelmointi mahdollistaa kuitenkin sen, että kaiken tarvittavan saa piirrettyä haluamallaan tavalla. Monogamen tarkoitus ei olekaan antaa kaikkea valmiina, vaan helpottaa ja nopeuttaa asioiden tekoa.

Proseduraalinen sisällön generointi, eli tässä pelissä tasojen generointi oli yllättävän helppo toteuttaa. Toteutettu tasogeneraattori oli melko yksinkertainen, mutta toimiva ja

sen käyttäytymistä voi säädellä muuttamalla parametreja. Pienillä muutoksilla tasogeneraattorin koodiin ja parametreihin voi saada isoja muutoksia aikaan tason lopputulokseen. Huoneiden yhdistämisessä todettiin, että helpoin tapa tehdä käytävät huoneiden välille on reitinhaku-algoritmilla.

3d-mallien tekeminen oli odotetusti paljon aikaa vievä osa-alue pelin kehityksessä. Varsinainen peli tarvitsisi paljon enemmän 3d-malleja kuin prototyyppiin tehtiin, sekä animoinnin toteuttamisen Blenderissä ja Monogamessa. Mallien animointi elävöittäisi peliä huomattavasti, mutta työssä ei ollut riittävästi aikaa tämän toteuttamiseen. Yksi 3d-malli tehtiin kuitenkin animointi valmiiksi, josta oli hyötyä vihollisten teossa, esimerkiksi zombien kohdalla vaihtamalla mallin asentoa. Asentoa muuttamalla saatiin myös pelaajan hahmolle eri aseille sopivat asennot. Yksi 3d-malli voikin riittää monen pelihahmon esitykseen peleissä muuttamalla asentoa, varusteita ja tekstuureita.

Suurimpia ongelmia aiheutti tasojen seinien esittäminen graafisesti. Erityisesti kulmaseinät olivat ongelmallisia, ja moni ratkaisu toi uusia ongelmia. Ongelmat ilmenivät LoS-algoritmin toteutuksen jälkeen, kun pelaaja tutkii luolaa ja paljastaa uusia seiniä ne nähdessään. Seinää ei voi esimerkiksi kulmakohdassa venyttää isommaksi, että se peittäisi kulmaan jäävän raon, sillä se paljastaisi kulmakohdan pelaajalle liian aikaisin. Ratkaisuna voisi olla väliaikainen malli, joka vaihdetaan oikeaksi, kun kulmakohdan molemmat seinät on nähty. Eripituiset seinät vaativat myös teksturointiin korjauksia, ettei tekstuuri näytä venytetyltä. Myös tiilien kohdat, joissa on monta seinää esitettynä yhdellä 3d-mallilla, aiheuttivat ongelmia. Esimerkiksi jos kahden käytävän välillä on yksi seinä, niin se esitetään yhdellä 3d-mallilla, jossa on kaksi vastakkaista seinää. Tässä piti tarkistaa, minkä seinistä pelaaja voi nähdä sijainnistaan. Seinän 3d-malli piti jakaa eri mesheihin Blenderissä ja lisätä koodiin ehdot, milloin mikäkin seinän osa piirretään, jätetään harmaaksi tai ei piirretä ollenkaan riippuen, onko seinää ikinä nähty tai nähdäänkö se juuri nyt. Ongelmat pelin kehityksessä olivat kuitenkin melko pieniä, ja pienellä suunnittelulla ja vaivannäöllä korjattu tai korjattavissa.

Lähteet

1 Rogue. 2013. Verkkodokumentti. <http://www.roguebasin.com/index.php?title=Rogue>. Luettu 25.4.2015.

2 Dwarf Fortress. 2015. Verkkodokumentti. <http://www.bay12games.com/dwarves/>. Luettu 25.4.2015.

3 The "dungeon" as it looked on an ASCII Terminal. 2008. Verkkodokumentti. <http://en.wikipedia.org/wiki/Rogue_%28video_game%29#/media/File:Rogue_Unix_Screenshot_CAR.PNG>. Luettu 25.4.2015.

4 Patel, Amit. Heuristics. 2015. Verkkodokumentti. <http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html>. Luettu 28.3.2015.

5 Lester, Patrick. A* Pathfinding for Beginners. 2005. Verkkodokumentti. <http://www.policyalmanac.org/games/aStarTutorial.htm>. Luettu 28.3.2015.

6 Jice(nimimerkki). Comparative study of field of view algorithms for 2D grid based worlds. 2009. Verkkodokumentti. <http://www.roguebasin.com/index.php?title=Comparative_study_of_field_of_view_algorithms_for_2D_grid_based_worlds>. Luettu 7.4.2015.

7 Elig(nimimerkki). Eligloscode. 2009. Verkkodokumentti. <http://www.roguebasin.com/index.php?title=Eligloscode>. Luettu 8.4.2015.

8 Daves, Adam. 2010. Windows Phone 7 Game Development. New York: Apress.

9 Whitaker. Managing content. 2014. Verkkodokumentti. <http://rbwhitaker.wikidot.com/monogame-managing-content>. Luettu 3.3.2015.

10 Monogame. Using The Pipeline Tool. 2015. Verkkodokumentti. <http://www.monogame.net/documentation/?page=Using_The_Pipeline_Tool>. Luettu 14.4.2015.

11 Xamarin. Installing Xamarin.iOS on Windows. 2015. Verkkodokumentti. <http://developer.xamarin.com/guides/ios/getting_started/installation/windows/>. Luettu 1.3.2015.

12 Blender. Inverse Kinematics. 2014. Verkkodokumentti. <http://wiki.blender.org/index.php/Doc:2.6/Manual/Rigging/Posing/Inverse_Kinematics>. Luettu 15.4.2015.

13 Patel, Amit. Introduction to A*. 2015. Verkkodokumentti. <http://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html>. Luettu 17.4.2015.