PELIYMPÄRISTÖN SUUNNITTELUEDITORI

HAMK
HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

Ammattikorkeakoulututkinnon opinnäytetyö

Riihimäen kampus, Tieto- ja viestintätekniikan koulutusohjelma

Syksy, 2017

Ville Happo

HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

Tieto- ja viestintätekniikan ko.
Riihimäki

Tekijä Ville Happo **Vuosi** 2017

Työn nimi Peliympäristön suunnittelueditori

Työn ohjaaja/t Petri Kuittinen

TIIVISTELMÄ

Opinnäytetyössäni lähestytään pelien kenttäsuunnittelua modulaarisesta näkökulmasta, kehittämällä peliympäristön suunnittelueditoria. Tätä varten käydään läpi modulaarisuuden toimintaa pelien ulkopuolella ja peleissä. Pelien osalta käydään läpi modulaarisuuden sovelluksia vanhoissa ja uusissa peleissä. Moderneissa peleissä modulaarisuus toteutetaan 3D-malleilla, joten niiden osalta käydään läpi 3D-mallin suunnitteluvaiheessa olennaisia asioita, kuten kääntöpisteen sijainti ja koon suunnittelu sopivaksi käytettävän pelimoottorin ruudukkoon.

Toiminnallisessa osuudessa kehitin suunnitteluedi torin Unreal Engine 4 -pelimoottorilla, jolla voidaan sijoittaa modulaarisia 3D-malleja peliympäristöön ja käsitellä niitä. Tähän osuuteen sisältyy modulaaristen 3D-mallien suunnittelu, editorin työkalujen kehittäminen ja niiden toimintaperiaatteet. Sovelluksen kehittämisen ohessa käydään lyhyesti läpi editorille välttämättömän käyttöliittymän toteutus.

Projektin lopputuloksena oli toimiva suunnitteluedi tori, joka sisälsi välttämättömät työkalut kentän suunnitteluun ja modulaaristen 3D-mallien käsittelyyn. Kenttien sisältö on mahdollista tallentaa ja ladata tiedostosta.

Avainsanat Modulaarisuus, pelisuunnittelu, suunnitteluedi tori, Unreal Engine

Sivut 35 sivua

HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

Degree Programme in Information and Communication Technology
Riihimäki

Author Ville Happo **Year** 2017

Subject Game Environment Design Editor

Supervisors Petri Kuittinen

ABSTRACT

This thesis approaches game level design, from the perspective of modularity, by designing a game environment editor. This includes applications of modularity in real life, and for existing and new games. Modularity in modern games can be accomplished with 3D-models which must be designed with some important features in mind. These features include pivot and designing the module sizes to fit each other, and the grid of a game engine that is being used.

This thesis included developing a level design editor with Unreal Engine 4. The editor was designed for positioning and manipulating modular 3D-models at a game level. This part of the project included designing the 3D-models, developing the tools for the editor and explaining how they functioned. The functionality and development of a minimal user interface are explained briefly in the thesis.

The outcome of the project was a functional application, which contained the necessary tools for the level design editor and manipulation of modular 3D-models at the game level. It is possible to save and load the contents of a level from a file.

Keywords Design Editor, Game development, Modularity, Unreal Engine

Pages 35 pages

SISÄLLYS

1 JOHDANTO

Opinnäytetyössäni tutustutaan modulaarisuuteen, mitä se tarkoittaa ja sen soveltamiseen peliympäristön suunnittelussa. Tätä varten käydään läpi modulaarisuuden sovelluksia pelien ulkopuolella ja peleissä. Pelien osalta käydään läpi huomioon otettavia asioista 3D-mallien suunnittelussa, jotta ne voivat toimia modulaarisesti.

Projektiosuus keskittyy suunnitteluedeitorin kehittämiseen. Tähän sisältyy sovelluksen ominaisuuksien suunnittelu ja toteutus, joilla voidaan lisätä, poistaa, kääntää ja liikuttaa itse tehtyjä modulaarisia 3D-malleja peliympäristössä ja tallentaa kentän sisältö tiedostoon. Editorin toiminnan lisäksi käydään läpi osia käyttöliittymän toteutuksesta, joka on välttämätön osa editorin toimintaa. Editorin käyttöliittymä ei ole projektin keskeinen aihe, joten sen toiminnasta kerrotaan vain lyhyesti. Editorin toimintojen toteuttamiseen käytettiin Unreal Engine Visual Scriptingiä, joka on Unreal Enginen visuaalinen skriptausjärjestelmä. Sovelluksen on tarkoitus olla helppo ja nopea käyttää, jotta kokematon käyttäjä voi rakentaa peliympäristöjä, ilman teknistä tietotaitoa suunnitteluohjelmista tai pelimoottorin toiminnasta.

Suunnitteluedeitoriin sisältyville toiminnoille ei ollut minkäänlaista pohjatyötä, ennen projektin aloittamista ja niiden toiminta on suunniteltu projektin tekemisen ohella. Eri ominaisuuksien toimintaperiaatteet muuttuivat useaan kertaan projektin aikana, jotta ne sopivat yhteen ja tukivat vasta valmistuneita toimintoja. Opinnäytetyöni tehtiin itsenäisenä projektina ilman toimeksiantoa. Sen aihe liittyy ammattiini pelisuunnittelijana ja projektiin käytetty aika ja vaiva edistävät osaamistani nykyisessä ammatissani.

2 MODULAARISUUS

Modulaarisessa suunnittelussa pyritään uudelleenkäytettävyyteen, yhteensopivuuteen, resurssien kulutuksen vähentämiseen ja helppokäyttöisyyteen. Modulaarisesti toteutettuja kokonaisuuksia voi löytää useilta eri aihealueilta, kuten tietotekniikasta, arkkitehtuurista, ajonneuvoteollisuudesta, ohjelmoinnista, prosessiteollisuudesta ja hissiteollisuudesta. (Modular Design n.d.)

2.1 **Pelien ulkopuolella**

Tietotekniikassa modulaarisuus tarkoittaa sitä, että tietokoneiden komponentit ovat helposti vaihdettavia ja keskenään yhteensopivia

standardisoitujen liitäntätapojen ansiosta. Tämä antaa käyttäjille mahdollisuuden parantaa tietokoneen toimintaa ilman, että käyttäjän täytyy ostaa kokonaan uusi tietokone. Tyypillisiin moduuleihin kuuluu virtalähteet, prosessorit, emolevyt, näytönohjaimet, kovalevyt ja optiset asemat. Näiden osatyyppien vaihtaminen keskenään pitäisi olla mahdollista, kunhan osat tukevat samaa liitäntä standardia. (Modular Design n.d.)

Modulaarinen talonrakentaminen on prosessi, jossa talo rakennetaan työmaan ulkopuolella hallituissa olosuhteissa, käyttäen samoja materiaaleja ja suunnitellaan taloja, noudattaen samoja sääntöjä ja standardeja, kuin perinteisessä rakentamisessa, käyttäen vain puolet sen vaatimasta ajasta. Talot valmistetaan moduuleissa, jotka työmaalla yhdistämisen jälkeen vastaavat suunnittelultaan ja ominaisuuksiltaan useimpia hienostuneita, paikan päällä rakennettuja rakennuksia. Rakenteellisesti modulaariset rakennukset ovat kestävämpiä, kuin perinteisesti rakennetut talot, koska jokainen moduuli on suunniteltu kestämään kuljetusta ja nostamista yksinään. Kun moduulit on kiinnitetty toisiinsa, niistä tulee integroitu seinä-, lattia- tai katto-asetelma. (Modular Building Institute n.d.)

2.2 **Peleissä**

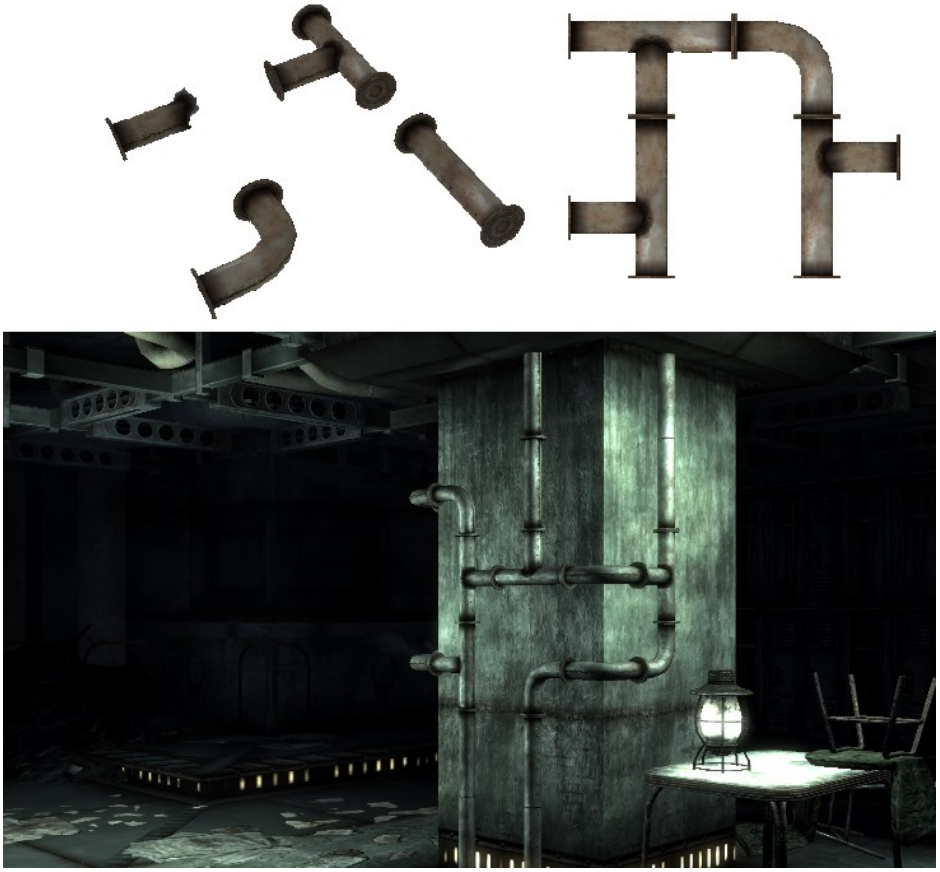
Modulaarisuus on prosessi, jossa uudelleen käytetään samoja 3D-malleja. Tämä tarkoittaa, että modulaarisuuden toimimiseen tarvitaan vähintään yksi 3D-malli ja se antaa mahdollisuuden rakentaa peliympäristöjä. Lopputulos ei välttämättä ole visuaalisesti miellyttävä, mutta modulaarisen ympäristön 3D-mallien valmistamiseen kuluu vähemmän aikaa, kuin sellaisen ympäristön, jossa käytetään paljon uniikkeja 3D-malleja. Vaikka modulaarisuutta ei vietäisi äärimmäisyyksiin, se on silti usein nopeampaa, kuin ympäristön rakentaminen uniikeista malleista, joilla on samantasoinen laatu ja yksityiskohdat. (Jones 2011, 7.)

Jos jokin malli täytyy vaihtaa, se tapahtuu perustuen modulaarisuuden toimintaperiaatteeseen. Kun artisti on päivittänyt modulaarisen 3D-mallin, sillä korvataan jo pelissä oleva 3D-malli, jolloin sen kaikki instanssit päivittyvät automaattisesti, säästäen aikaa ja vaivaa.

Modulaarisuuden sivutuotteena tulee mahdollisuus käyttää 3D-malleja paikoissa, joissa niitä ei ole tarkoitettu käytettävän. Modulaaristen osasarjojen tyyli on yleensä sama, joten mallien käyttäminen eri tarkoitukseen ei välttämättä pistä pelaajan silmään. Tällä keinolla voidaan myös lisätä vaihtelua kentän ulkonäköön ja rikkoa kentän geometrian toistuvuutta. (Jones 2011, 11.)

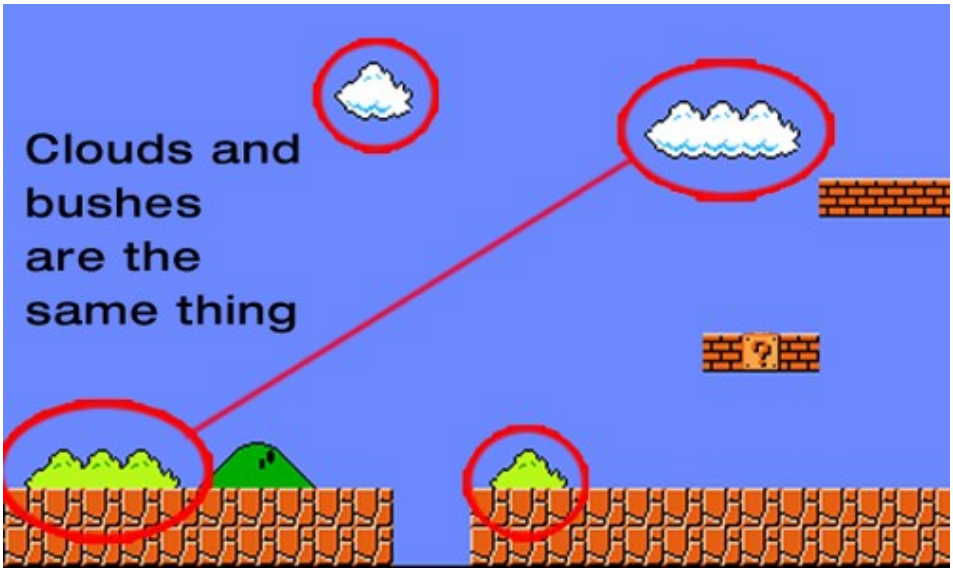
Modulaariset järjestelmät voivat koostua vain muutamasta osasta, kuten Fallout 3:ssa käytetty putki-sarja, joka koostuu neljästä putkesta. Kuvassa 1 näkyvän putkisarjan tärkein ominaisuus on se, että siitä voidaan saada

aikaan paljon enemmän kokonaisuuksia, kuin sen osien summa on. Artisti ei vain tehnyt neljää putkea, vaan järjestelmän, jonka avulla voidaan rakentaa loputtomasti putkiasetelmia. (Burgess 2013.)



Kuva 1. Fallout 3 putki-sarja (Burgess 2013).

Modulaarisuus peleissä on ollut olemassa pitkän aikaa. Jo Nintendo Entertainment Systemin ajoilta 2D-peleissä, kuten kuvan 2 Mario ja Segan Sonic, käyttivät modulaarista suunnittelua. Marion ensimmäisessä tasossa kaikkia peliobjekteja on käytetty useita kertoja, eri tarkoituksissa. Kaikista käytetyin objekti on lattiaruutu, jota käytetään koko kentän lattiana ja ainoa paikka lattiassa, jossa sitä ei käytetä, on lattiassa olevat aukot.



Kuva 2. Marion modulaarisuus (Warped Factor 2016).

Sonic, joka julkaistiin myöhemmin kuin Mario, käytti samantyylistä modulaarista suunnittelua, mutta monimutkaisemmissa kentissä. Niissä uudelleen käytettiin objekteja ja tekstuureja, joita oli välillä skaalattu täyttämään tilaa paremmin. Suurin ero Sonicin ja Marion välillä oli se, että Sonicin kentät koostuivat päällekkäin asetetuista tasoista, jotka sisälsivät joitain monimutkaisia muotoja ja reittejä, joita pitkin voitiin kulkea eri korkuisten tasojen välillä ja ne kytkivät modulaarisia elementtejä toisiinsa. (Jones 2011, 20.)

2.3 **Pelimoottorien ruudukot**

Kaikissa pelimoottoreissa on jonkinlainen ruudukkojärjestelmä. Ruudukon tarkoitus on helpottaa 3D-mallien asettamista peliympäristöön. Jotkin pelit, kuten *Far Cry* tai *Max Payne,* käyttävät metreihin perustuvaa ruudukkoa. Metreihin perustuva ruudukko on varsin yksinkertainen. Ruudukon koko voidaan asettaa esimerkiksi arvoihin: 1, 5, 10, 20 tai 50 metriä. Pelaajan korkeus on yleensä noin 1,8 metriä, joten kappaleiden vertaaminen niiden todellisiin vastakappaleisiin on helppoa. Toinen ruudukkojärjestelmä perustuu yksiköihin. Tätä käytetään esimerkiksi Id Softwaren *Doom*-sarjassa ja Epic Gamesin *Unreal*-sarjassa. Tässä ruudukossa ihmisen kokoinen pelihahmo on noin 96 yksikköä korkea, joka tarkoittaa, että 1 metri on noin 53 yksikköä. Tässä ruudukkotyypissä ruutujen koko perustuu kahden kerrannaisiin, kuten tekstuurien resoluutio. Kuvassa 3 on esimerkki kahdella kerrannaisen ruudukkojärjestelmän mitoista. (Gamasutra 2005.)

2^x = Grid size

2^0 = 1 Unit

2^1 = 2 Units

2^2 = 4 Units

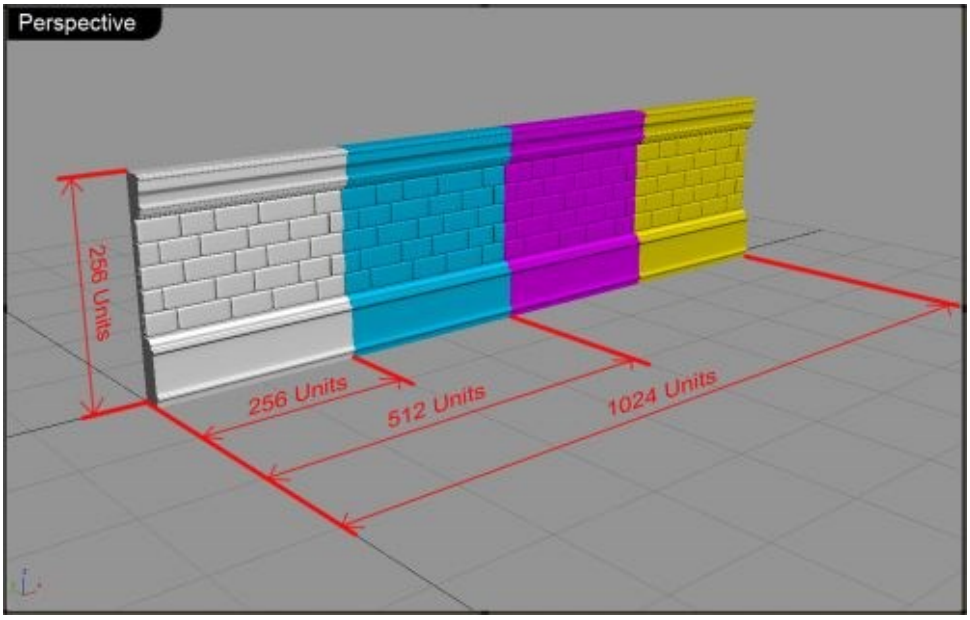
2^3 = 8 Units

2^8 = 256 Units

2^{10} = 1024 Units

Kuva 3. Yksikköjärjestelmä (Gamasutra 2005).

Jos käytössä oleva pelimoottori tukee kahden kerrannaisiin perustuvaa ruudukkoa, modulaariset mallit tulee mitoittaa ruudukkoon sopivaksi. Jos mitat eivät ole kahden kerrannaisia, 3D-malli ei tule sopimaan ruudukkoon. Kuvassa 4 on modulaarinen seinä, jonka leveys ja korkeus ovat 256 yksikköä, joten sen mitat ovat kuvan 2 mukaan kahdella kerrannaisia. 3D-mallit, jotka sopivat mitoiltaan täydellisesti ruudukkoon, tulevat liittymään toisiinsa saumattomasti. (Gamasutra 2005.)



Kuva 4. Kahdella kerrannainen (Gamasutra 2005).

Kun suunnitellaan 3D-malleja, joita tullaan käyttämään yhdessä, täytyy varmistaa, että niiden mitat vastaavat toisiaan. Jos esimerkiksi on jo mallinnettu seinä, jonka leveys on 256 yksikköä, sen kanssa käytettävien lattia- ja katto 3D-mallien tulee myös olla 256 yksikköä leveitä, jotta niitä voidaan asettaa vierekkäin saumattomasti. (Gamasutra 2005.)

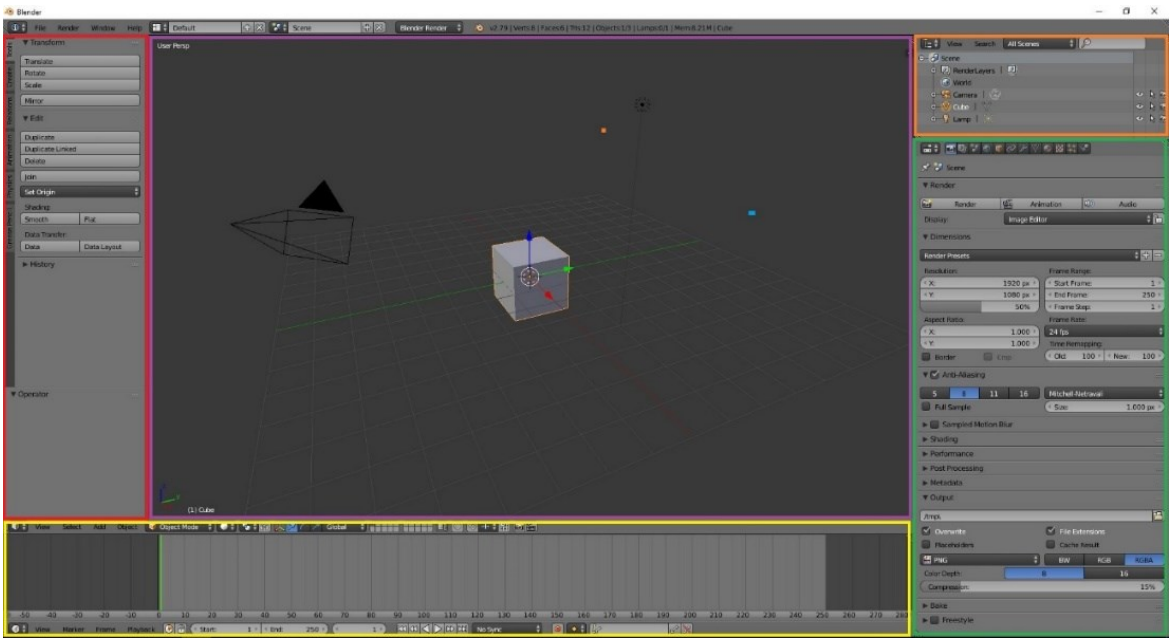
3 KÄYTETTÄVÄT OHJELMAT

3.1 Blender

Blender-säätiö on hollantilainen julkisen sektorin yritys, joka perustettu tukemaan ja helpottamaan Blender.org projekteja. Blender on ilmainen ja vapaan lähdekoodin 3D-luomisympäristö. Se tukee 3D-tuotantoa kokonaisuudessaan mallintamisessa, rigauksessa, animoinnissa, simuloinnissa, renderöinnissä, kompositoinnissa, liikkeen jäljentämisessä, videon editoinnissa ja pelien tekemisessä. Edistyneet käyttäjät hyödyntävät Blenderin ohjelmointirajapintaa python-skriptien kirjoittamisessa, muokatakseen Blenderiä ja kirjoittaakseen siihen lisäosia. Yleensä näitä sisällytetään Blenderin tulevissa versioissa. Blender sopii yksityishenkilöille ja pienille yrityksille, joilla on tarve yhtenäiselle 3D-tuotannolle. Blender on ristiin yhteensopiva eri alustojen kanssa ja tukee yhtä hyvin Linux, Windows ja Macintosh tietokoneita. (Blender n.d.)

Valitsin Blenderin projektin 3D-suunnittelu ohjelmaksi, koska se on ilmainen ja minulla on lähes kolmen vuoden kokemus sen käyttämisestä. Olen tehnyt sillä 3D-mallit ja animaatioita kolmeen aikaisempaa peliprojektiin ja käyttänyt sitä opintojeni aikana tehtävien suorittamiseen.

Kuvassa 5 näkyy Blenderin käyttöliittymä. Sen ikkunoita voi liikuttaa ja niiden kokoa voi muuttaa hiirellä raahaamalla. Blenderin värejä, fontin kokoa ja muita ulkonäköön liittyviä asioita voi muuttaa täysin vapaasti asetuksista. Blender perustuu vahvasti pikanäppäimiin ja sen käyttäminen valikkojen nappeja painamalla on hidasta.



Kuva 5. Blender käyttöliittymä.

Alla on listattuna kuvan 5 värillisten laatikoiden sisältö.

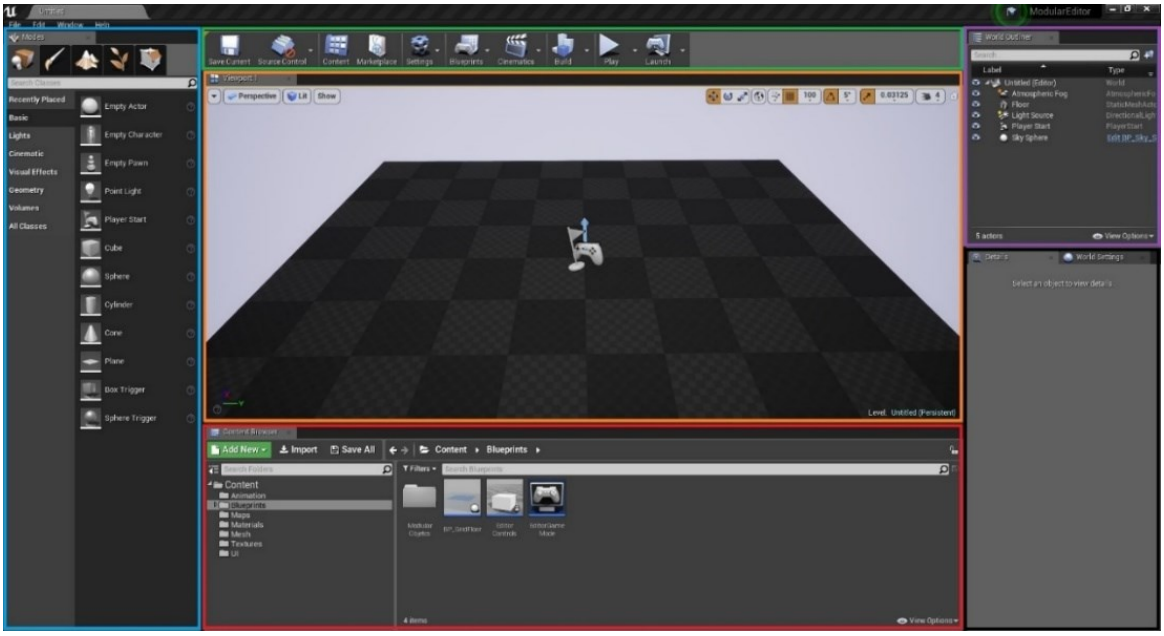
- Punainen – Työkalut, joilla voidaan manipuloida valittua objektia, animoida sitä ja lisätä uusia objekteja.
- Violetti – Editorinäkymä, jossa käsitellään 3D-malleja. Editorin keskeisin ikkuna
- Keltainen – Animaatio aikajana, jonka avulla lisätään avain ruutuja animaatioihin
- Oranssi – Listaus kyseisen projektin objekteista ja niiden perintäsuhteista toisiinsa nähden.
- Vihreä – Ominaisuudet, joka sisältää renderöinnin, materiaalien, tekstuurien, rigien ja 3D-ympäristön asetukset.

3.2 **Unreal Engine 4**

Unreal Engine 4 on paketti kehitystyökaluja, joka on luotu kenelle tahansa, joka työskentelee reaaliaikaisen visualisoinnin parissa. Unreal Engine 4 antaa käyttäjälle kaiken tarvittavan projektin aloittamiseen, julkaisemiseen ja yrityksen kasvuun. Se on tarkoitettu yritysten sovellusten, elokuvamaisten kokemusten ja PC-, konsoli-, mobiili-, VR- ja AR-pelien kehittämiseen. (Epic Games n.d.a.)

Valitsin Unreal Engine 4 -pelimoottorin projektia varten, koska olen käyttänyt sitä kolmessa aikaisemmassa projektissa ja minulla on siitä noin kahden vuoden kokemus. Olen myös kokeillut Unity-pelimoottoria, mutta en harkinnut sen käyttämistä, koska valtaosa opinnäytetyön ajasta kuluisi pelimoottorin oppimiseen, eikä itse ohjelman kehittämiseen. Käytin projektissa Unreal Engine 4:n versiota 4.18.

Unreal Enginen editorin käyttöliittymä koostuu liikuteltavista ikkunoista, joiden sijaintia ja kokoa voidaan muokata vapaasti. Kahden monitorin käyttäminen onnistuu, kun jokin ikkuna raahataan toiseen monitoriin. Oletusnäkymässä on vain ehdottomasti tarpeelliset ikkunat, mutta lisää löytyy valikoista. Kuvassa 6 on näkymä, johon Unreal Engine 4 aukeaa, kun uusi projekti avataan. Kuvaan 6 on lisätty värejä, jotka rajaavat eri ikkunoita.



Kuva 6. Unreal Enginen editorin käyttöliittymä.

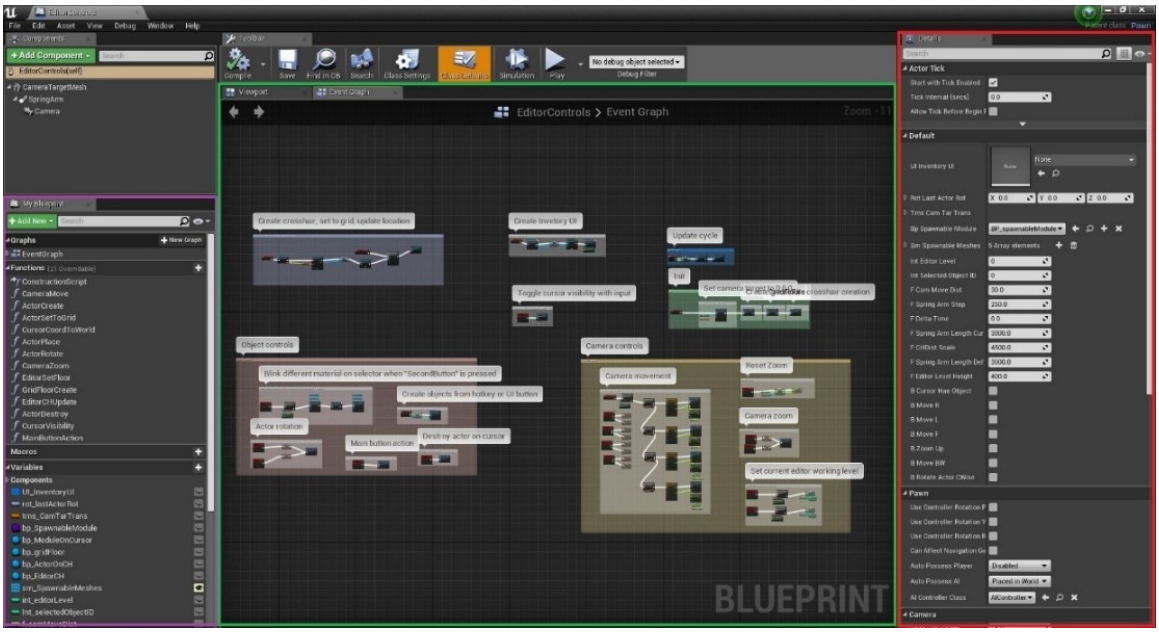
Alla on listattuna kuvan 6 värillisten laatikoiden sisältö.

- Sininen – Modes, jossa vaihdetaan työskentelymoodia ja josta tuodaan peruselementtejä, kuten valoja, perusmuotoja ja visuaalisia efektejä kenttään.
- Vihreä – Työkalu-ikkuna, josta esimerkiksi tallennetaan kenttä, käynnistetään peli ja rakennetaan ohjelma lähdekoodista.
- Oranssi – Pelinäkymä, jossa nähdään pelissä oleva editorikameran kautta ja voidaan esikatsella peliä eri kuvakulmista ja eri suodattimien läpi.
- Punainen – Hakemisto, josta löytyy kaikki projektin materiaalit, kuten 3D-mallit, luokat, Blueprintit, kentät ja tekstuurit.
- Violetti – Lista pelin kaikista objekteista ja niiden perintäsuhteista.
- Musta – Yksityiskohdat, jossa näkyy valitun objektit tiedot ja säädettävät arvot.

Blueprints Visual Scripting on Unreal Enginen skriptaus-järjestelmä, joka perustuu solmukohtien, eli noodien käyttämiseen pelielementtien skriptaamisessa. Kuten useita muitakin ohjelmointikieliä, sitä käytetään olio-ohjelmointiin perustuvien luokkien tai objektien määrittämiseen pelimoottorissa. Blueprints Visual Scriptingillä määritettyihin peliobjekteihin ja luokkiin viitataan yleensä vain nimellä *Blueprint*, Unreal Engine 4:n sisällä. Tämä järjestelmä on joustava ja tehokas, koska se antaa suunnittelijoille mahdollisuuden käyttää teoriassa mitä tahansa konsepteja ja työkaluja, jotka ovat yleisesti vain ohjelmoijien käytettävissä. Sen lisäksi erityisesti Blueprinteille tarkoitettu merkintätapa on saatavilla Unreal Engine 4:n C++ sovelluksessa, joka antaa ohjelmoijille

mahdollisuuden kehittää perustason järjestelmiä, joiden päälle suunnittelijat voivat laajentaa uusia toiminnallisuuksia. (Epic Games n.d.b.)

Blueprint Visual Scripting käyttöliittymä näyttää lähes samalta kuin editorin, mutta sen sisältö ja käyttötarkoitus ovat eri. Kuva 7 Keskellä oleva kaavioruutu on olennaisin. Sen sisällä yhdistetään noodeja ja muuttujia toisiinsa skriptin luomiseksi.



Kuva 7. Blueprint-ikkunan käyttöliittymä.

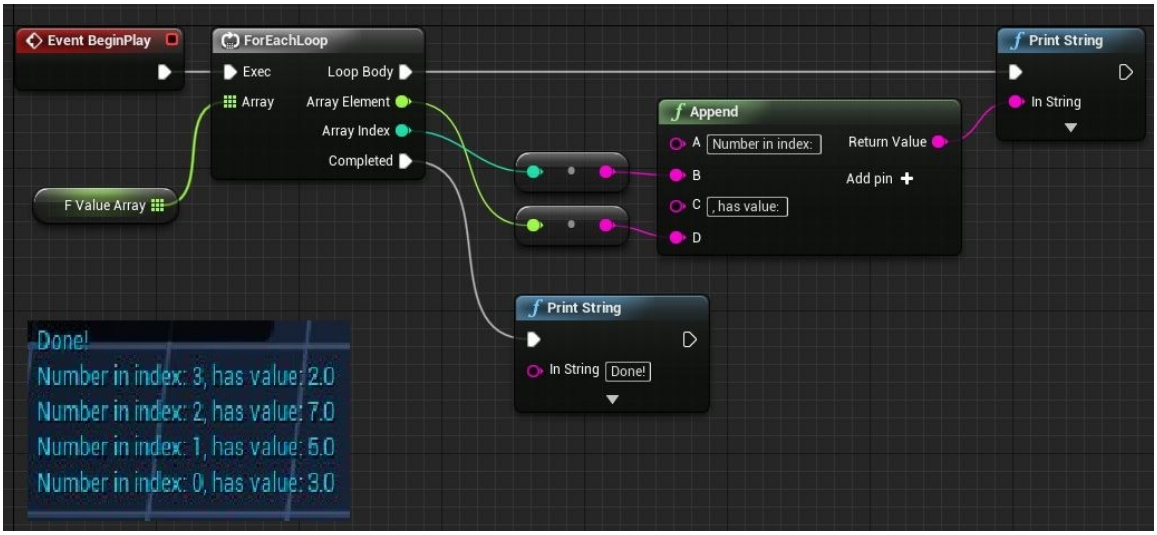
Alla on listattuna kuvan 10 värillisten laatikoiden sisältö.

- Vihreä – Kaavioruutu, jossa koodin rakennetaan solmukohdista, eli noodeista.
- Violetti – Kyseisen Blueprintin muuttujat, tapahtumat, funktiot ja makrot.
- Punainen – Yksityiskohdat, jossa voidaan säätää funktioiden ja muuttujien arvoja ja tarkastella arvoja ajon aikana.

Uusia noodeja haetaan avaamalla haku-ikkuna hiiren oikealla painikkeella. Kun halutut noodit ovat kaaviossa, niiden tulo- ja lähtöportit yhdistetään muihin noodeihin kaapeleilla. Blueprint ajetaan siten, että jokin tapahtuma laukeaa, jonka jälkeen siihen kytketyt noodit ajetaan kytkentäjärjestyksessä. Tällaisia tapahtumia on esimerkiksi pelin alkaminen, tai jonkin näppäimen painaminen.

Kuvassa 8 on esimerkki Blueprint, joka tulostaa sarjan merkkijonoja. Pelin alkaessa *EventBeginPlay*-tapahtuma käynnistää *ForEachLoop*-silmukan, joka käy *f_ValueArrayn* kaikki arvot läpi. *ForEachLoop*-silmukka syöttää arrayn jokaisen elementin arvon ja indeksin *Append*-noodille, jossa arvot

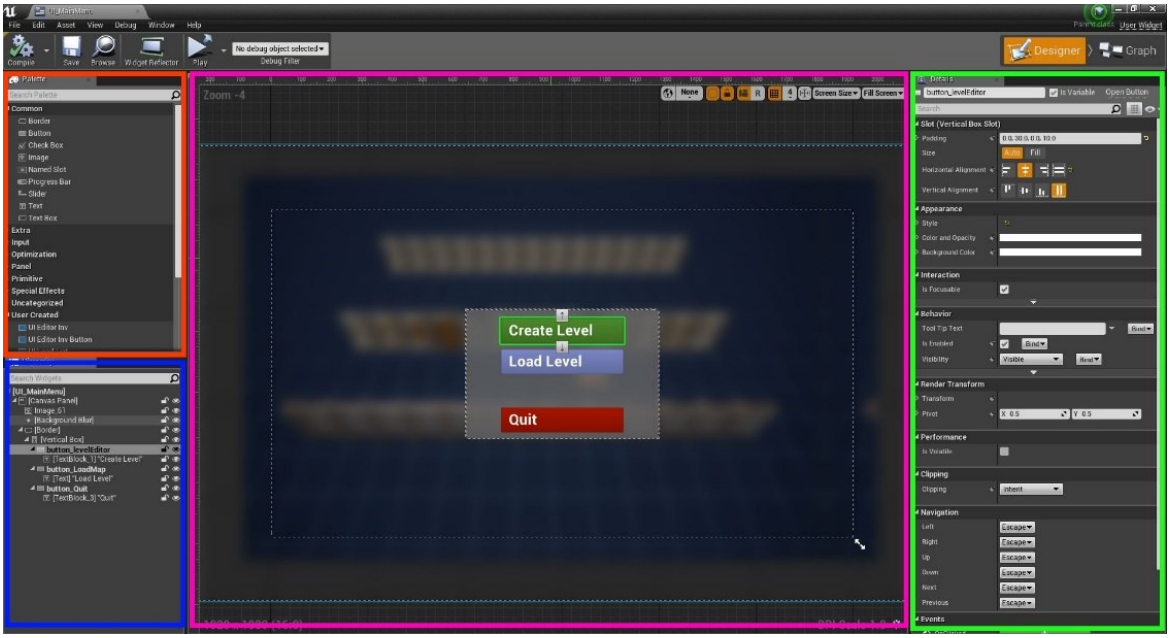
ja esitäytetyt merkkijonot yhdistetään merkkijonoksi. Tämän jälkeen kuvan 8 yläoikeassa kulmassa oleva *PrintString*-noodi tulostaa merkkijonot. Kun *ForEachLoop*-silmukka on käynyt läpi koko *f_ValueArrayn*, se suorittaa alemman *PrintString*-noodin, joka tulostaa merkkijonon *”Done!”.*



Kuva 8. Esimerkki Blueprint.

Unreal Motion Graphics UI Designer (UMG), on visuaalisen käyttöliittymän suunnittelutyökalu. Sillä voidaan suunnitella käyttöliittymän elementtejä, kuten pelin sisäisiä heijastusnäyttöjä (HUD), valikoita ja muuta käyttöliittymään kuuluvaa grafiikkaa, jonka pelintekijä haluaa näyttää pelaajalle. Pohjimmiltaan UMG:t ovat sarja Widgettejä, eli esivalmistettuja funktioita, kuten nappeja, valintaruutuja, liukureita ja edistymispalkkeja, joista suunnittelija voi kasata käyttöliittymän. Näitä Widgettejä editoidaan erityisessä Widget-Blueprintissä, joka sisältää kaksi välilehteä. Designer-välilehdessä suunnitellaan käyttöliittymän ulkonäkö, layout ja asetetaan perusfunktiot kuvaruutuun. Kaavio-välilehti tarjoaa Widgettien taustalla olevan toiminnallisuuden ja sen avulla Widgeteille voidaan lisätä skriptejä. (Epic Games n.d.c.)

Kuvassa 9 on UMG:n designer välilehden yleisnäkymä. Kuvan keskellä on editorin päävalikko suunnitteluvaiheessa. Se sisältää kolme nappia, taustakuvan, sumennuksen ja harmaan taustan napeille.



Kuva 9. UMG, designer välilehden yleisnäkymä.

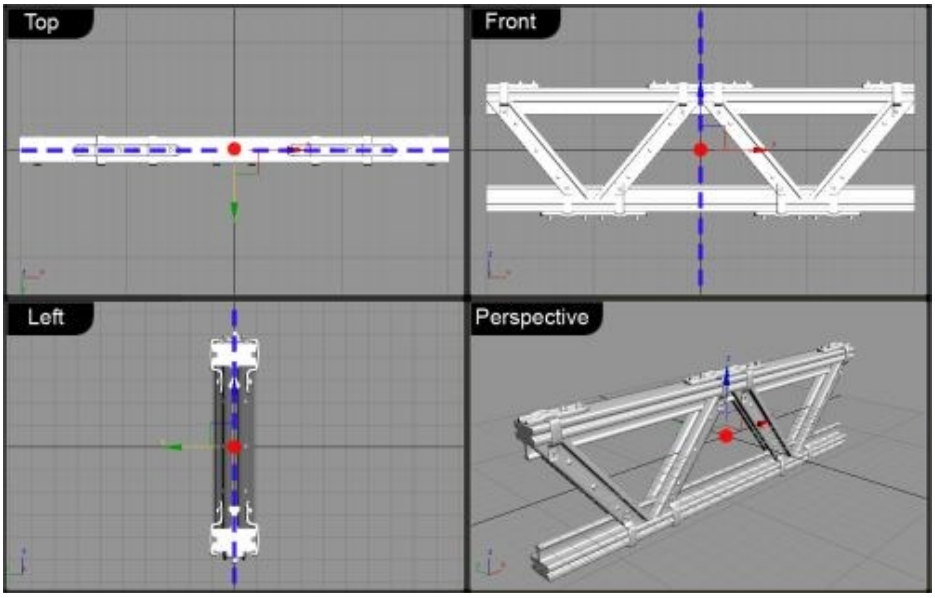
Alla on listattuna kuvan 9 värillisten laatikoiden sisältö.

- Oranssi – Widget paletti, josta voidaan raahata uusia elementtejä visuaaliseen näkymään.
- Sininen – Käytössä olevien Widgettien hierarkia.
- Pinkki – Designer-välilehden suunnittelutila
- Vihreä – Ominaisuudet-ikkuna, jossa voidaan muuttaa valitun Widgetin arvoja.

4 MODULAARISET 3D-MALLIT

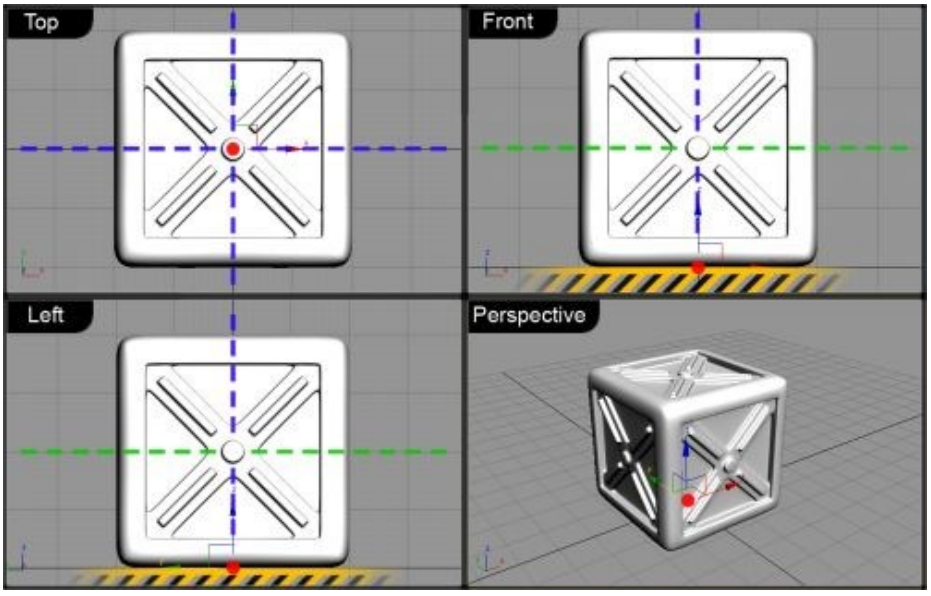
4.1 3D-mallin kääntöpiste

Kääntöpiste on 3D-mallille määritelty piste, jonka ympäri se kääntyy pelimoottorissa. Sen sijainti modulaarisella mallilla on tärkeä ja sen sijainnin päättämiseen kannattaa käyttää aikaa ja sen toimintaa testata, koska hyvin sijoitettu kääntöpiste tekee mallien asettamisesta helppoa, mutta huonosti asetettu kääntöpiste voi tehdä modulaarisesta mallista käyttökelvottoman. Jos mallilla on symmetrisiä muotoja yhteen tai useampaan suuntaan, kääntöpisteen on hyvä olla symmetrian risteyksen keskellä. Jos malli on symmetrinen x-, y- ja z-akselilla, kuten kuvan 10 tukipilari, symmetrioiden risteys on myös mallin keskipiste. (Gamasutra 2005.)



Kuva 10. Symmetrisellä teräksisellä tukipilarilla on hyvä kääntöpiste (Gamasutra 2005).

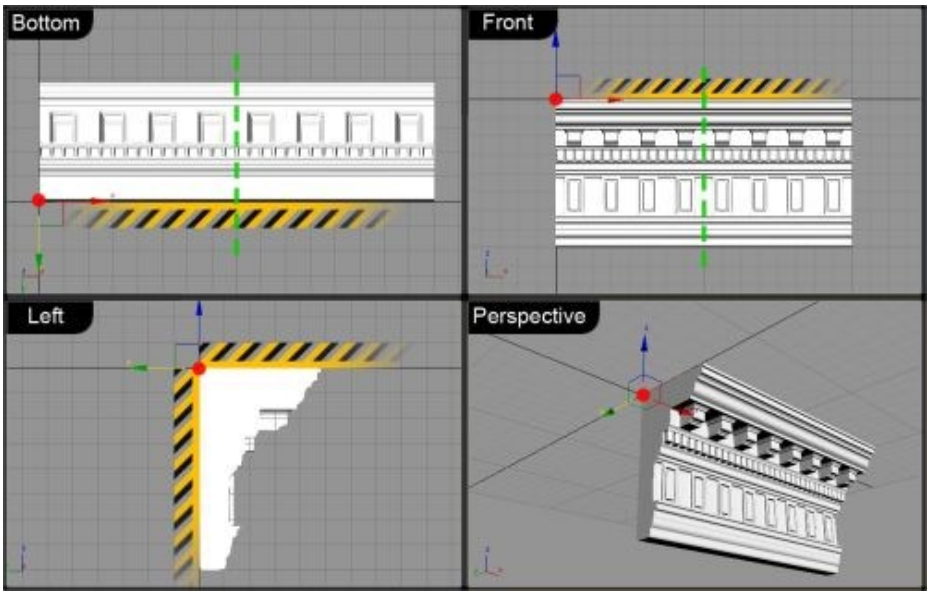
Mallin käyttö ja asettelu pelissä tulee ottaa huomioon kääntöpistettä asetettaessa. Jos kyseessä on 3D-malli, joka tulee aina olemaan maassa tai lattialla kuten Kuvan 11 laatikko, kannattaa kääntöpiste asettaa pintaan, joka koskee maahan, jotta malli on aina tasan lattian kanssa asettamisen jälkeen. (Gamasutra 2005.)



Kuva 11. Symmetrinen laatikko, joka asetetaan maata vasten (Gamasutra 2005).

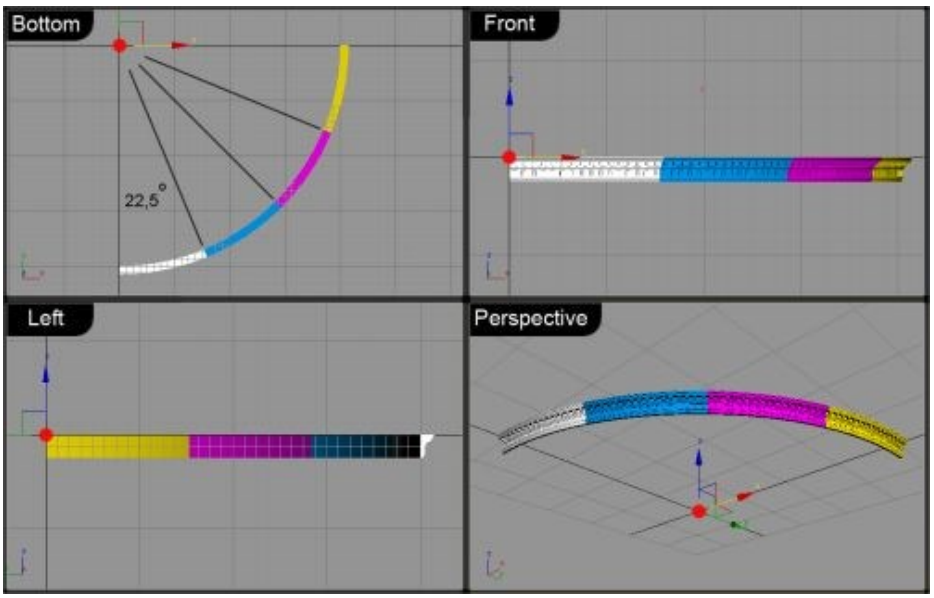
Jos kyseessä on malli, joita asetetaan pelissä useita peräkkäin ja ne toistavat samaa muotoa jatkuvasti, kuten kuvassa 12, kannattaa kääntöpiste asettaa kulmaan, josta malli kiinnittyy seuraavaan malliin.

Tästä on se hyöty, että malleja voidaan skaalata siten, että yksi pääty pysyy paikallaan ja malli venyy kiinnittyneestä reunasta poispäin. (Gamasutra 2005.)



Kuva 12. Malli, jota asetetaan useita peräkkäin ja luodaan saumattomasti jatkuva muoto. (Gamasutra 2005).

Jotkin modulaariset kappaleet ovat suunniteltu osaksi kehää. Niitä täytyy kääntää ennen yhdistämistä, jotta saadaan aikaseksi kaareva muoto. Kuvan 13 tilanteessa kääntöpisteen tulee olla yhdistettyjen mallien ympyrän keskellä. Jossain tapauksissa tämä tarkoittaa, että kääntöpiste tulee olemaan kaukana mallista, joka on yleensä huono asia, mutta tässä tapauksessa se toimii paljon paremmin, kuin kääntöpisteen ollessa keskellä mallia. (Gamasutra 2005.)

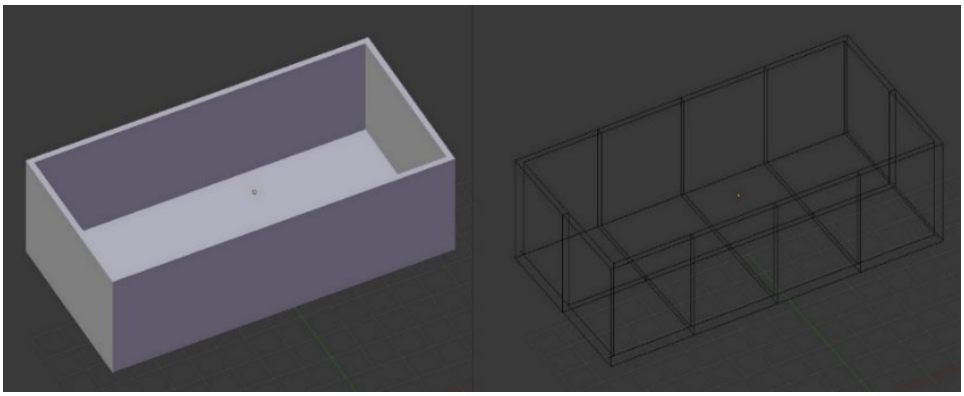


Kuva 13. Kaareva ja toistuva malli, jota voidaan pyörittää 22,5 asteen askelissa ilman rakoja. (Gamasutra 2005).

4.2 **Testimoduulit**

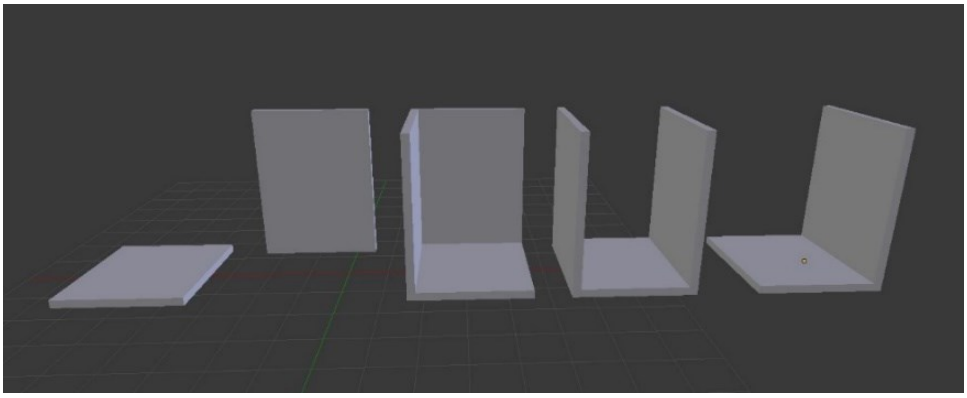
Modulaarisuus perustuu mallien asetteluun peräkkäin ja vierekkäin, saman muodon jatkuvaan toistumiseen tai useiden mallien saumattomaan yhteensopivuuteen. 3D-mallit tulee suunnitella siten, että ne sopivat toisiinsa geometrian kannalta ja noudattavat samoja mittasuhteita.

Moduulisarjaa suunnitellessa täytyy kiinnittää erityisesti huomiota kappeleiden mittoihin ja yhteensopivuuteen. Tämän lisäksi tulee testata osien yhteensopivuus monissa eri asennoissa ja asetelmissa. Tämän projektin rakentamissarjan tapauksessa, kaikkien vierekkäin asetettavien seinämoduulien tulee olla saman korkuisia ja mikä tahansa seinä pitää pystyä liittämään muihin seiniin ongelmitta, asennosta riippumatta. Tämä tarkoittaa, että seinien tulee olla yhtä paksuja ja moduulien tulee noudattaa samaa pituus- ja leveysskaalaa. Kuvassa 14 on ensimmäisen testisarjan moduuleista kasattu huone, jonka seinät sopivat saumattomasti yhteen. Moduulien korkeus on 400cm, leveys 300cm ja syvyys 300cm. Seinien paksuus on 20cm.



Kuva 14. Moduulien saumaton yhteensopivuus.

Kuvassa 15 näkyvä, projektin ensimmäinen testisarja sisälsi viisi osaa: seinän, lattian, lattian yhdellä seinällä, lattian kahdella vastakkaisella seinällä ja lattian kulmaseinällä. Sarja toimi hyvänä lähtökohtana kunnollisen sarjan kehittämiseen, koska sen avulla löytyi toistuvia ongelmia, jotka voidaan välttää seuraavissa versioissa.

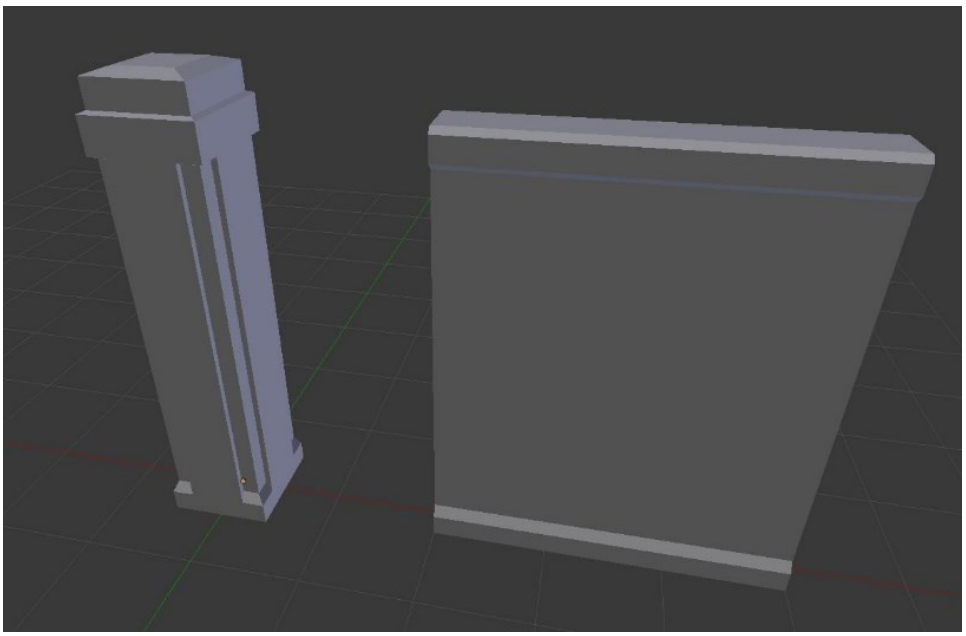


Kuva 15. Ensimmäinen testisarja moduulit.

Lattian sisällyttäminen seinämoduuleihin oli turhaa, koska sarjaan kuului lattiamoduuli, joka voidaan asettaa niiden tilalle. Lattian poistaminen moduuleista tarkoittaa myös, että seinien tekstuureille jää enemmän käytettäviä pikseleitä UV-kartoituksen aikana, jonka ansiosta moduulien tekstuureilla on korkeampi resoluutio pelissä. Lisäksi lattialle tehtävät ulkonäkömuutokset tulisi tehdä kaikille lattian sisältävillä moduuleille.

4.3 **Valmiit moduulit**

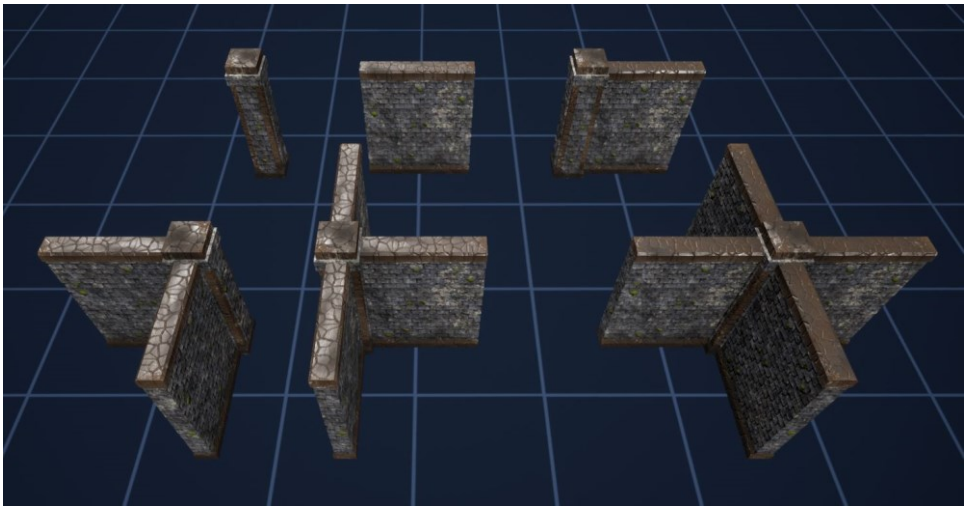
Päädyin yksinkertaistamaan moduulien toimintaperiaatetta reilusti, verrattuna testisarjaan. Vähensin moduulien määrän vain kahteen osaan, eli pylvääseen ja seinään. Kuvassa 16 on seinän ja pylvään 3D-mallit Blenderissä.



Kuva 16. Modulaarinen pylväs ja seinä.

Pylväs ja seinä koostuvat yhteensä 135:stä polygonista, joka on vähän verrattuna korkealaatuisiin pelihahmoihin, joiden polygonimäärät voivat vaihdella kymmenistä tuhansista satoihin tuhansiin polygoneihin. Pylvään ja seinän idea on se, että niistä voidaan kasata erilaisia yhdistelmiä Unreal Enginen sisällä. Kuvassa 17 on näistä 3D-malleista kasattuja yhdistelmiä editorissa. Moduuleille tehtiin omat Blueprintit, jonka sisällä 3D-mallit asetettiin haluttuihin paikkoihin toisiinsa nähden. Editorissa voidaan sen jälkeen käyttää näitä Blueprinttejä ja sen sisältämät 3D-mallit noudattavat Blueprintin sijaintia, säilyttäen omat sijaintinsa toisiinsa nähden.

Moduulien kääntöpiste on pylvään keskellä, sen pohjassa. Näin moduulit tulevat aina pohjasta kiinni lattiatasoon ja niitä voidaan kääntää siten, että pylväs pysyy aina keskellä. Kuvan 17 ylärivin keskellä olevasta moduulista puuttuu pylväs, joten sen kääntöpiste on seinän päädyssä, lattiatasossa. Se voidaan kiinnittää muiden moduulien pylvääseen tai käyttää sellaisenaan, pitkän seinän rakentamisessa.



Kuva 17. Kasatut muurimoduulit.

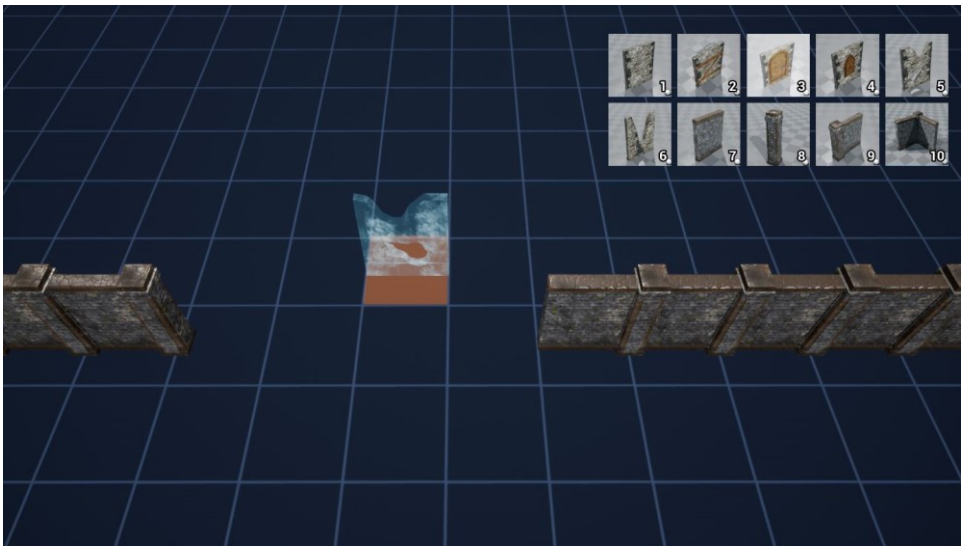
Lattiaa ei enää tehdä moduuleilla, vaan se jätetään kokonaan pois editorin ominaisuuksista opinnäytetyön osana. Tarkoituksenani on lisätä editoriin maalaustyökalu, jolla voidaan maalata lattiaan eri materiaaleja, kuten nurmikkoa ja kiviä. Tämän toiminnon valmistuminen vie niin kauan, ettei se mahdu opinnäytetyön laajuuteen.

5 SUUNNITTELUEDITORI

Opinnäytetyössäni mainitaan näppäimistön ja hiiren painikkeita eri toimintojen käyttämiseksi, koska editori on suunniteltu ja testattu ensisijaisesti Windowsilla. Toimintojen suorittamiseksi ei tarkkailla tiettyjä

näppäimiä vaan *InputAction*-tapahtumia, joihin voidaan kytkeä lähes mitä tahansa näppäimiä tietokoneen näppäimistöstä ja pelikonsolien ohjaimista. Unreal Engine on yhteensopiva lähes kaikkien suosittujen konsolien ja käyttöjärjestelmien kanssa, joten editorin kääntäminen niille tulevaisuudessa on mahdollista.

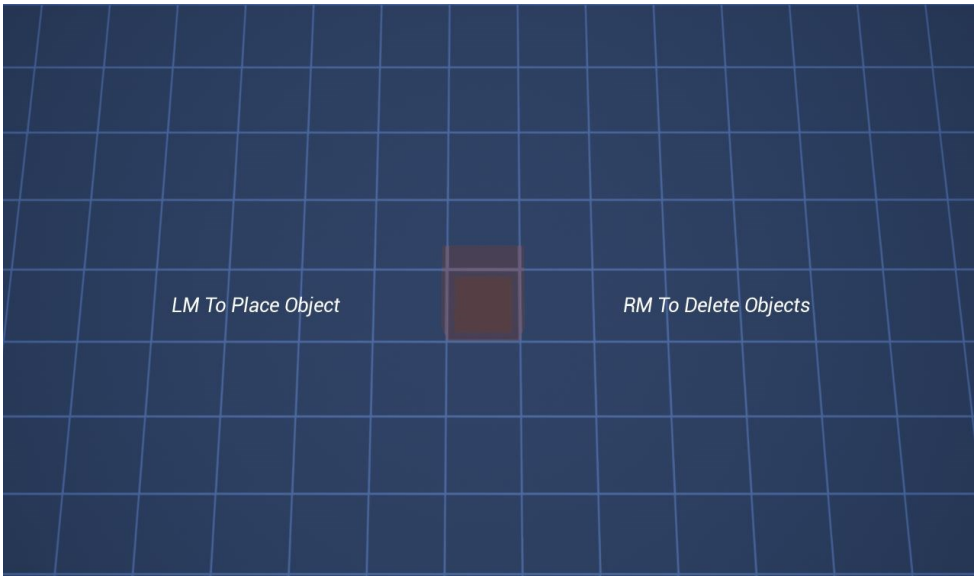
Kuvassa 18 on yleisnäkymä editorista. Kuvassa näkyy kenttään asetettuja moduuleja, valitsin ja moduulivalikko. Lopulliseen editoriin sisältyy mahdollisuus valita 3D-malli valikosta pikkukuvan perusteella, jolloin se luodaan pelikenttään. Kyseinen malli on sen jälkeen kiinnitetty hiiren ohjaamaan valitsimeen. Malli voidaan jättää kenttään ja sitä voidaan sen liikuttaa tai se voidaan poistaa jälkikäteen. Kentän sisältö voidaan tallentaa tiedostoon käyttäjän haluamalla nimellä ja aiemmin tallennetun kentän suunnittelua voidaan jatkaa, lataamalla tiedosto ja tallentamalla vanhan kentän päälle tai uutena tiedostona.



Kuva 18. Editorin yleisnäkymä.

5.1 **Ruudukko**

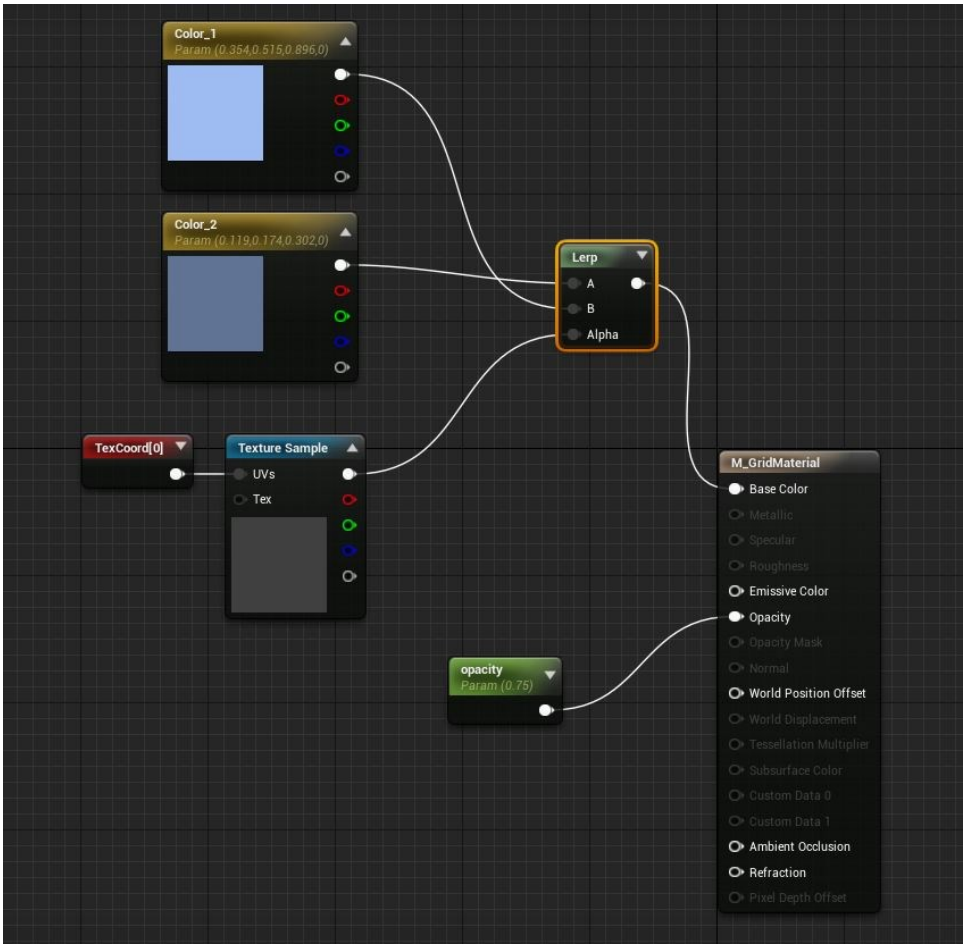
Ruudukko toimii editorissa hahmottamista avustavana elementtinä. Sillä ei ole koodillisia toiminnallisuuksia, mutta se on olennainen osa moduulien asettelussa peliympäristöön. Suunnitteluedeitorissa käytettiin metreihin perustuvaa ruudukkoa, jossa yksi ruutu on 3x3 metriä. Metreihin perustuvaa ruudukkoa käytetään, koska Blenderin oletusmittasuhteet toimivat metrien perusteella ja 3D-mallien suunnittelu on mielestäni luontevampaa metreissä. Kuvassa 19 on editorin ruudukko.



Kuva 19. Editorin ruudukko.

Ruudukolle on tehty Blueprint nimeltään *BP_GridFloor*. Se sisältää tason, joka on kooltaan 3000 x 3000 metriä. Se on projektin aikana tekemieni testieni perusteella riittävän kokoinen ja tason kokoa voidaan muuttaa, jos sille tulee tarvetta.

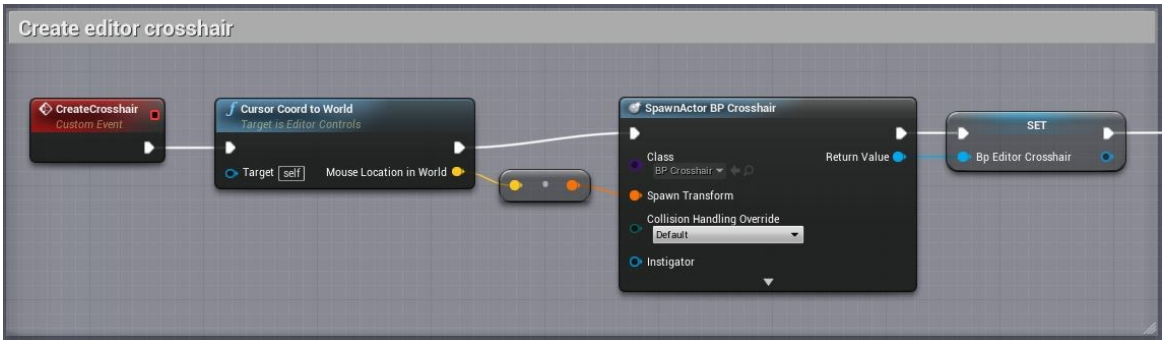
Tasolle on asetettu läpikuultava materiaali, jossa kaksi sinisen sävyä ja maski yhdistetään yhdeksi tekstuuriksi. Tämä tekstuuri vastaa kooltaan yhtä *BP_GridFloorin* ruutua. Kuvassa 20 on materiaalin kaavio, Unreal Engine 4:n materiaalieditorissa. Kaksi väriä ja mustavalkoinen maski syötetään lineaariselle interpolointi-, eli *Lerp*-noodille. Sen ulostulosta saadaan sama kuva, kuin mikä syötettiin sen alpha-kanavaan, mutta sen musta ja valkoinen väri on korvattu A- ja B- sisääntuloon kytketyillä väreillä. Tässä tapauksessa kahdella eri sinisen sävyllä, joista toinen on ruudukon pohjaväri ja toinen on ruudukon rajaviiva.



Kuva 20. Ruudukon materiaali.

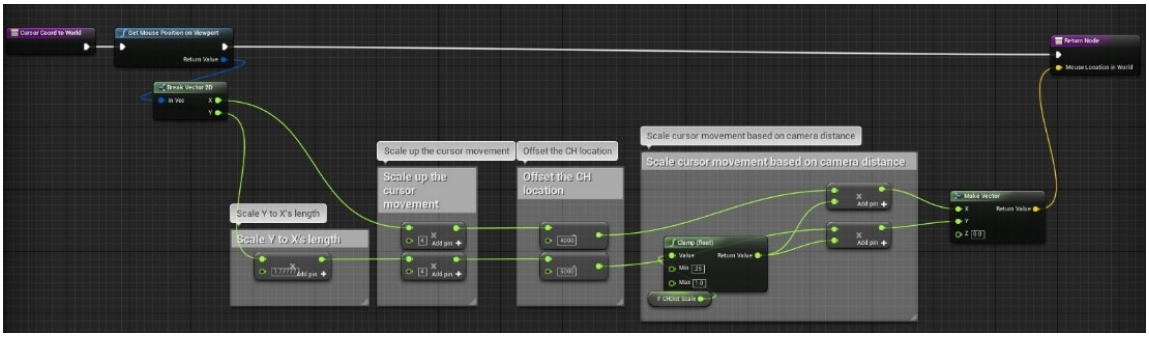
5.2 **Valitsin**

Valitsin on olennaisin työkalu, jolla käyttäjä työskentelee editorissa. Sen avulla käyttäjä valitsee, liikuttaa ja poistaa moduuleja kentästä. Valitsin, on *BP_Crosshair_Blueprintin* instanssi, joka sisältää visualisointiin tarkoitetun suorakulmaisen särmiön. Tämän särmiön ja kentässä olevien objektien päällekkäisyyksien perusteella tehdään poisto- ja siirto-operaatiot modulaarisille 3D-malleille. Uudet moduulit asetetaan luomishetkellä valitsimen sijaintiin ja niillä on läpikuultava animoitu materiaali, jonka avulla valitsimeen kiinnitetty moduuli erottuu kenttään jo asetetuista moduuleista. Valitsimen luominen tapahtuu EditorControls-Blueprintissä, joka hallitsee lähes kaikkia editorin perustoimintoja. Valitsin luodaan kuvan 21 skriptillä, editori-näkymän käynnistyessä.



Kuva 21. Valitsiminen luomis-skripti.

Kuvassa 21 näkyvä *CreateCrosshair*-tapahtuma käynnistää kuvassa 22 olevan *CursorCoordToWord*-funktion, joka kääntää kursorin sijainnin monitorin 2D-tilasta, pelin 3D-tilaan. Tämä tapahtuu ottamalla hiiren sijainti-vektori *GetMousePositionOnViewport*-noodilla. Se palauttaa 2D-vektorin, josta otetaan sen X- ja Y-komponentit. Y:n arvoa vahvistetaan 1,777…-kertaiseksi, jotta 2D-vektori ei noudata enää monitorin alkuperäistä 16:9 kuvasuhdetta, vaan 1:1 kuvasuhdetta. Tämä tehtiin, koska editorin kuvakulma ei ole kohtisuora kenttään nähden ja 16:9 kuvasuhdetta noudattavan valitsimen liike toimisi siten, että valitsin liikkuu voimakkaammin X-akselilla, kuin Y-akselilla. Tämä osoittautui ongelmalliseksi testausten aikana. Seuraavaksi vektorin X- ja Y-komponentteja vahvistetaan nelinkertaiseksi, jotta hiiren liike vastaa valitsimen liikettä editorikentän reunasta toiseen. Ilman tätä vahvistusta editorin valitsin liikkuisi vai vähän, keskellä editorinäkymää. Sijaintivektorin X- ja Y-komponentteja kerrotaan vielä *f_CHDist_Scale*-floatilla, joka vahvistaa valitsimen liikenopeutta kameran ollessa kaukana työskentelypinnasta ja hidastaa sitä kameran ollessa lähellä. Lopuksi 2D-vektori kasataan uudelleen ja sitä käytetään valitsimen sijainnin määrittämiseen.



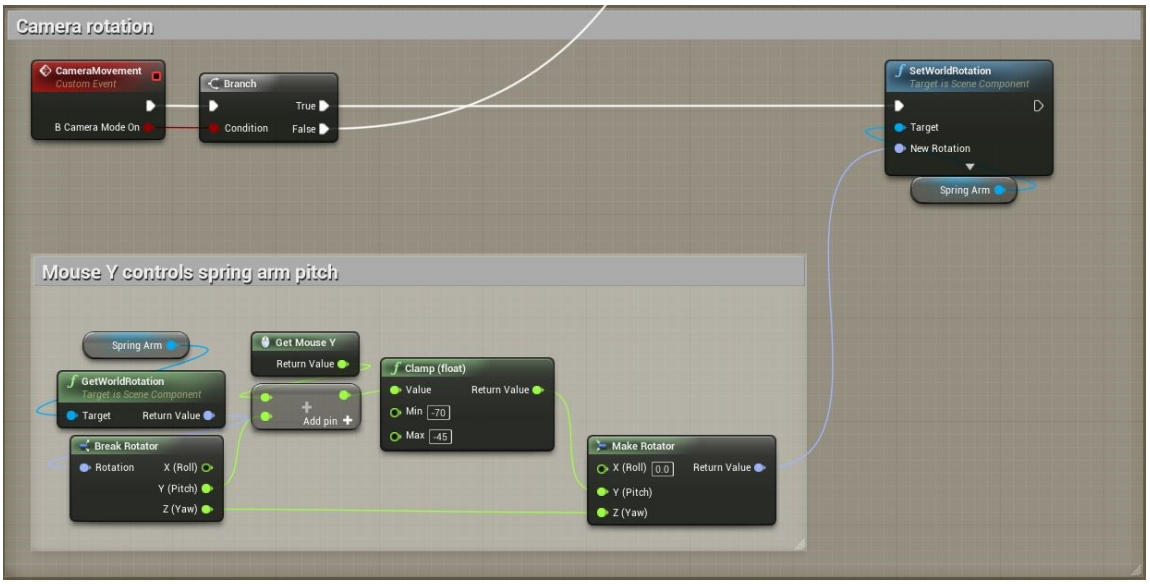
Kuva 22. Hiiren koordinaattien kääntäminen peliympäristöön.

5.3 **Kamera**

Kameralle toteutettuja ominaisuuksia ovat kuvakulman muuttaminen, kameran liikuttaminen WASD-näppäimillä, zoomaaminen ja zoomauksen palauttaminen oletusetäisyydelle.

A- ja D-näppäimet säätävät *f_MoveRight*-floatin arvoa välillä -1 ja 1. W- ja S-näppäimet säätävät *f_MoveForward*-floatin arvoa välillä -1 ja 1. Jos edes toinen näistä muuttujista on jotain muuta kuin nolla, tiedetään että pelaajan haluavan liikuttaa kameraa. Ctrl-painikkeen painaminen muuttaa *b_CameraModeOn*-booleanin arvoa. Sen perusteella ohjelma tietää, haluaako pelaaja kääntää kameraa, vai säätää sen kuvakulmaa. *EditorControls-Blueprint* tarkistaa jokaisen kuvan aikana yrittääkö pelaaja liikuttaa tai kääntää kameraa.

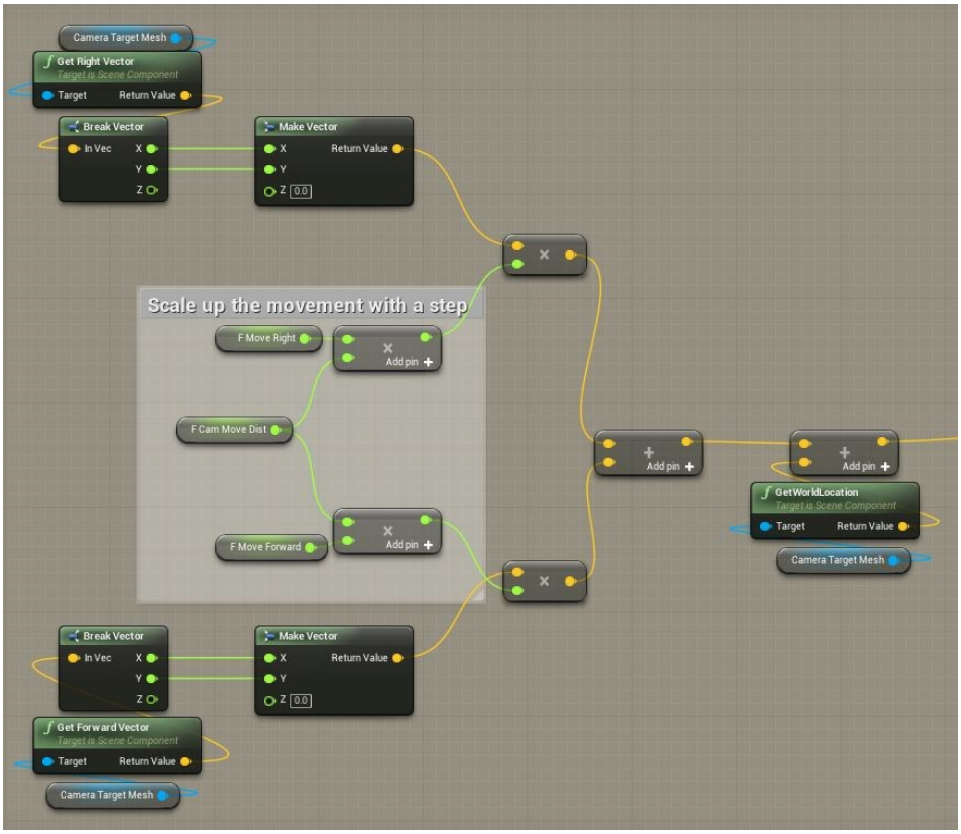
Kuvassa 23 on kameran kuvakulman hallintaskripti. *b_CameraModeOn*-booleanin ollessa totta, kameran varren nykyiseen rotaatioon Y-akselilla lisätään hiiren liike Y-akselilla. Rotaatiolle on asetettu liikkuma-alue *Clamp(float)*-noodilla. *Clamp(float)*-noodi rajoittaa sen läpi kulkevan arvon jollekin välille. Käyttäjä voi syöttää sille minimi ja maksimi arvon ja ulostulo-arvo on aina niiden välissä, vaikka tuloarvo olisikin rajojen ulkopuolella. Tällä estetään kameran liikkuminen ruudukon alle ja yli pystysuoran asennon, jolloin kamera kääntyisi ylösalaisin.



Kuva 23. Kameran kuvakulman säätäminen.

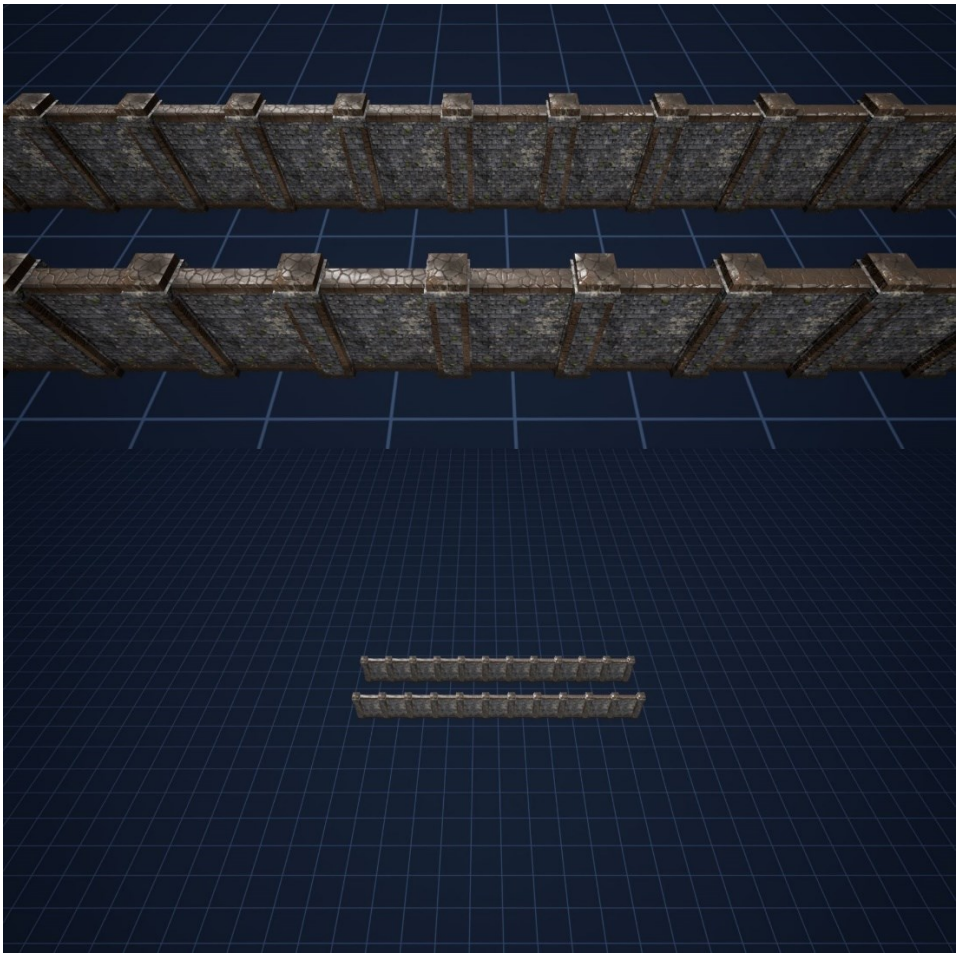
Kuvassa 24 on kameran liikevektorin laskemisskripti. *b_CameraModeOn*-booleanin ollessa epätosi, kameran kohdeobjektin eteen ja oikealle suuntaavia liikevektoreita vahvistetaan *f_MoveForward*- ja *f_MoveRight*-floateilla. Kahta edellä mainittua lukua vahvistetaan myös *f_CamMoveDist*-floatilla. Se on testaamalla saatu luku, jolla vahvistetaan kameran liikenopeutta 30 kertaiseksi, alkuperäiseen arvoon nähden.

Tämän jälkeen liikevektorit yhdistetään ja saadaan lopullinen liike X- ja Y-akselilla. Tämä vektori syötetään kameralle uutena sijaintina ja sama liikevektori syötetään myös valitsimelle, jotta se liikkuu kameran mukana.



Kuva 24. Kameran liikevektorin laskeminen.

Pelaajan pyörittäessä hiiren rullaa ylös, *b_ZoomUp-boolean* on tosi ja rullatessa alas se on epätosi. Kummassakin tapauksessa suoritetaan *CameraZoom-funktio*. Riippuen *b_ZoomUp*-booleanin arvosta, kameran varren pituuteen joko lisätään, tai siitä vähennetään testaamalla toimivaksi todettu 250-yksikköä, per rullan askel. Kuvassa 25 näkyy maksimi ja minimizoomausten ero. Kun kameran varren pituus on asetettu, säädetään *f_CHDistScale*-floatia, joka vahvistaa tai heikentää valitsimien liikenopeutta kameran etäisyyden perusteella. Kameran etäisyyden palauttaminen oletusarvoon tapahtuu pelaajan painaessa hiiren rullaa. Kameran varren pituudeksi asetetaan testaamalla toimivaksi todettu 3000 yksikön pituus.



Kuva 25. Kameran zoomauksen minimi ja maksimietäisyys.

5.4 **Moduulivalikko**

Editorin moduulivalikko on suunniteltu toimimaan siten, että se laajenee automaattisesti, kun *EditorControls*-Blueprintin sisältämään *struct_BasicModules*-array laajenee. Se on rakennetaulukko, joka sisältää kaikkien käytössä olevien moduulien tyypin ja pikkukuvan. Tein tämän toiminnon, koska en tiennyt kuinka monta moduulia projektissa tulee loppujen lopulta olemaan ja tiesin, että niiden lisääminen manuaalisesti on raskasta, varsinkin jos joudun tekemään useasti. Editori luo moduuleille valikon ja lisää jokaiselle valikon napille kuvan siitä moduulista, jota se edustaa. Näin pelaaja voi päättä kuvan perusteella, mitä nappia hän painaa.

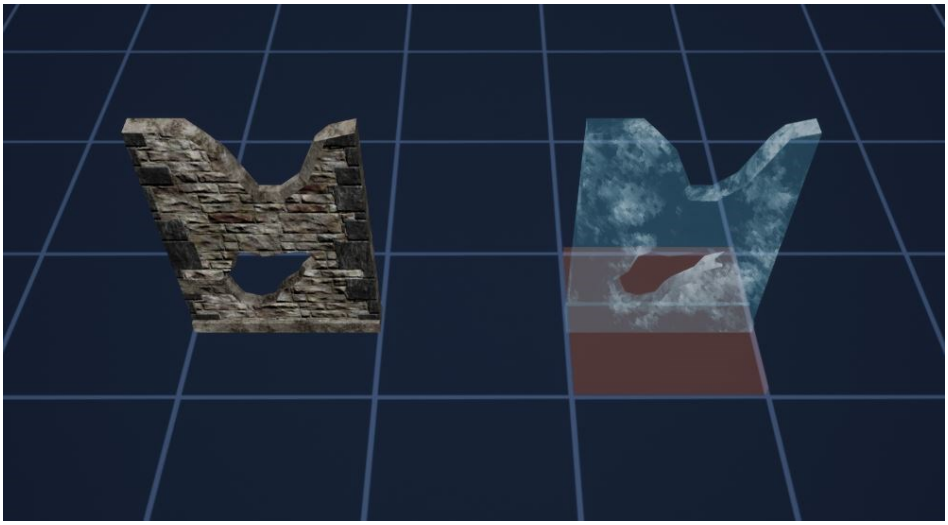
Kuvassa 26 on editorin moduulivalikko, joka sisältää kaikki moduuli-Blueprintit. Hiiren kursori on näkyvissä valikon ollessa auki. Valikko aukeaa ja sulkeutuu painamalla sarkainta. Moduuli valitaan klikkaamalla kuvaketta tai painamalla kuvakkeen alareunassa näkyvää pikanäppäintä. Kun moduuli on valittu, sen luodaan valitsimen sijaintiin ja se seuraa valitsimen liikkeitä.

Kuva 26. Editorin Moduulivalikko.

Valikko luodaan editorin käynnistyessä ja valikkoon lisätään yhtä monta nappia, kuin *struct_basicModules*-arrayssa on elementtejä, jolloin napin järjestysluku vastaa *struct_basicModulesin* moduulia. Napin taustakuvaksi asetetaan nappia vastaavan *struct_BasicModules-arrayssa olevan* rakenteen sisältämä kuva.

Valikon napit sisältävät skriptin, joka odottaa pikanäppäimen painamista tai napin klikkaamista. Kun nappia käytetään, se suorittaa *ActorCreate-funktion* ja syöttää tälle funktiolle oman järjestyslukunsa, jota käytetään nappia vastaavan moduulin luomiseen *struct_BasicModules*-arraysta.

ActorCreate-funktio sisältää skriptin, jossa napin järjestysluvulla haetaan sitä vastaava moduuli *struct_BasicModules*-arraysta, asetetaan moduulille sijainniksi editorin valitsimen sijainti ja asetetaan moduulin kaikille 3D-malleille animoitu ja läpinäkyvä materiaali, joka näkyy kuvassa 27.

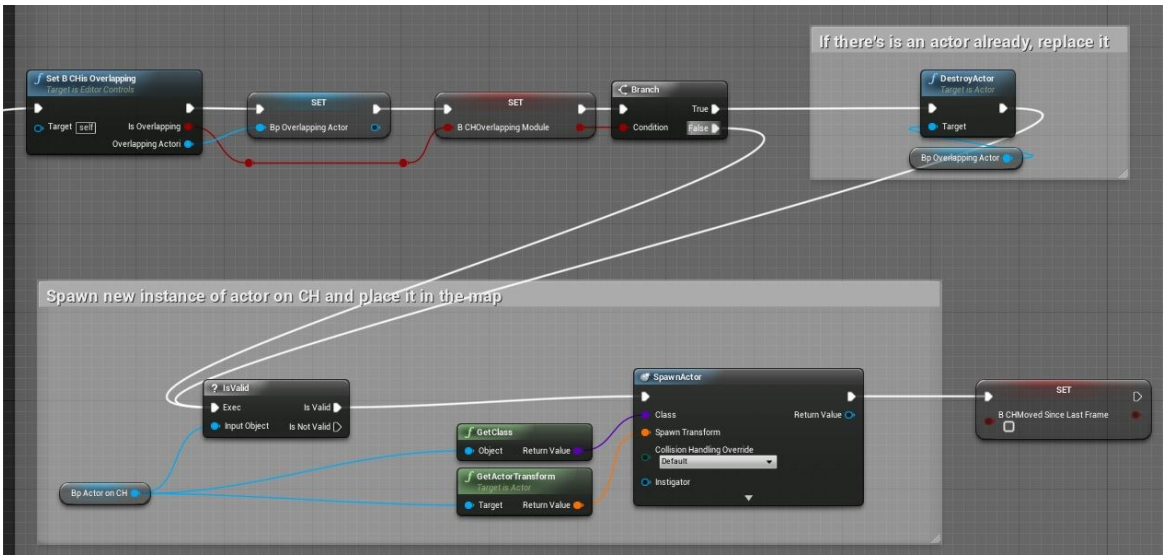


Kuva 27. Kenttään asetettu ja valitsimeen kiinnitetty moduuli vierekkäin.

5.5 Moduulien käsittely

Käyttäjän kyky lisätä, poistaa ja käsitellä kentässä olevia moduuleita on editorin keskeisin toiminto. Tähän liittyviä toimintoja on moduuleiden lisääminen yksitellen klikkaamalla hiiren vasenta nappia tai pitämällä sitä pohjassa ja raahaamalla hiirtä, jolloin moduuleja lisätään joka kerta, kun valitsin vaihtaa ruudukon ruutua. Poistamiselle on samat toiminnot, eli moduuleita voidaan poistaa yksitellen tai hiirtä raahaamalla nappi pohjassa.

Kuvassa 28 on moduulien asettamisen skripti, joka suoritetaan, jos pelaaja painaa hiiren vasenta painiketta. Se tarkistaa ensin, onko valitsimen kanssa päällekkäin jokin muu moduuli, kuin siihen sillä hetkellä kytketty moduuli. Jos on, niin kyseinen moduuli poistetaan ja korvataan valitsimeen kytketyn moduulin instanssilla. Jos valitsimen kanssa ei ole mitään päällekkäin, tapahtuu sama prosessi ilman vanhan objektin poistamista.



Kuva 28. Moduulin asettaminen kenttään.

Poistaminen tapahtuu hiiren oikealla painikkeella ja sen toiminnallisuus on osittain sama kuin objektin lisäämisessä. Ensin se tarkistaa onko valitsin päällekkäin jonkin moduulin päällä ja jos on, niin se poistetaan.

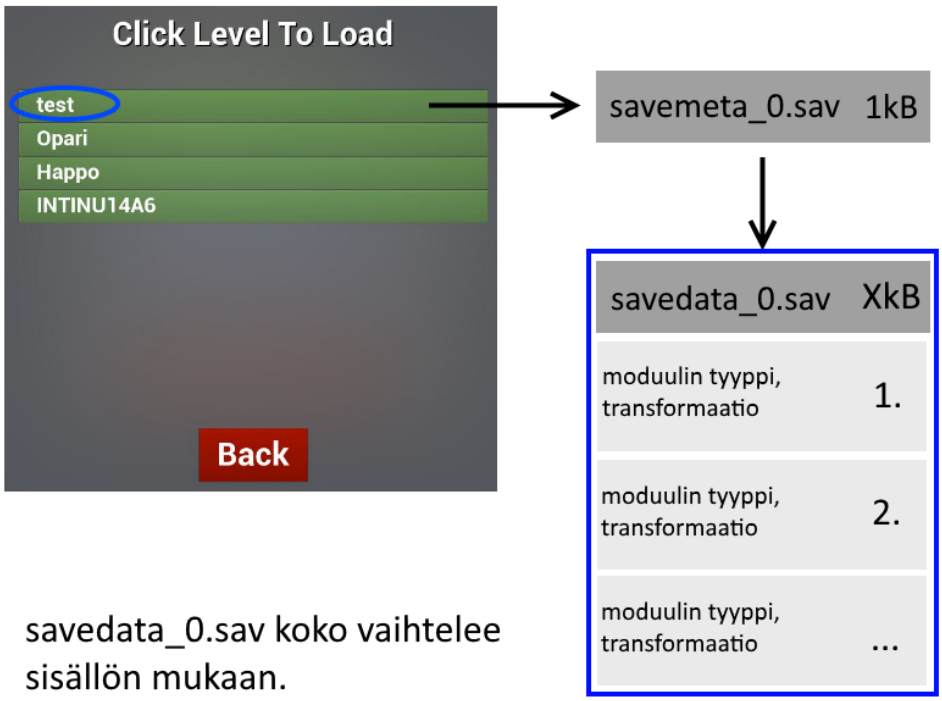
Moduulin kääntäminen vaakatasossa tapahtuu Q- ja E-napeilla. Ne ohjaavat *b_RotateActorCWise*-booleanin tilaa. Q on epätosi ja E on tosi. Kumpaakin nappia painettaessa suoritetaan *ActorRotate*-funktio, joka kääntää valitsimeen kytkettyä moduulia 90-astetta myötäpäivään tai vastapäivään riippuen *b_RotateActorCWise*-booleanin tilasta.

5.6 Kentän sisällön tallentaminen

Tietojen tallentamiseen käytetään *SaveGame*-luokkaa, josta voidaan periä uusia tallennustyyppejä ja niille voidaan määrittää tallennettavia

muuttujia. Kun tallennuskomento suoritetaan, Unreal Engine luo ohjelman kansioon sav-päätteisen tiedoston määrittelyllä sisällöllä. Tiedostot säilyvät, vaikka ohjelma suljetaan ja niistä voidaan lukea muuttujien arvoja ajon aikana.

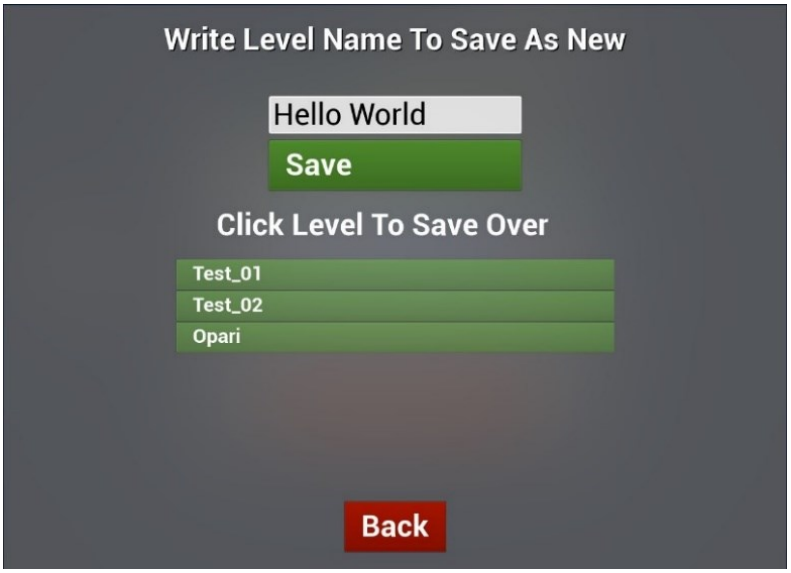
SuunnitteluEditorissa käytetään tallennuksille *sav*-tiedosto paria, jotka ovat *savedata* ja *savemeta*. *Savedata* sisältää arrayn kentän moduulien tyypeistä ja niiden transformaatioista. Transformaatio koostuu kolmesta vektorista, jotka ovat objektin sijainti, rotaatio ja skaalaus. *Savemeta* sisältää tallennetun kentän nimen, tallennuspaikan numeron ja savedatan tallennusnimen. Tiedostoparia käytetään, koska *savedatan* koko kasvaa kentässä olevien moduulien määrän perusteella ja sen käyttäminen valikoissa voi olla hidasta, jos savedata-tiedostoja on useita ja ne ovat kooltaan suuria. *Savemeta*-tiedostot sisältävät aina lähes saman verran tietoa ja niiden koko on noin yksi kilotavu. *Savemeta* on periaatteessa *Savedatan* edustaja, jota käytetään valikoissa. Kuva 29 kuvaa *savemeta*- ja *savedata*-tiedostojen yhteistoimintaa valikossa. *Savemetasta* haetaan valikoille olennaisin tieto, eli kentän nimi, jonka perusteella käyttäjä tietää mitä haluaa ladata.



Kuva 29. Tallennustiedon hakeminen *savedatasta*, *savemetan* avulla.

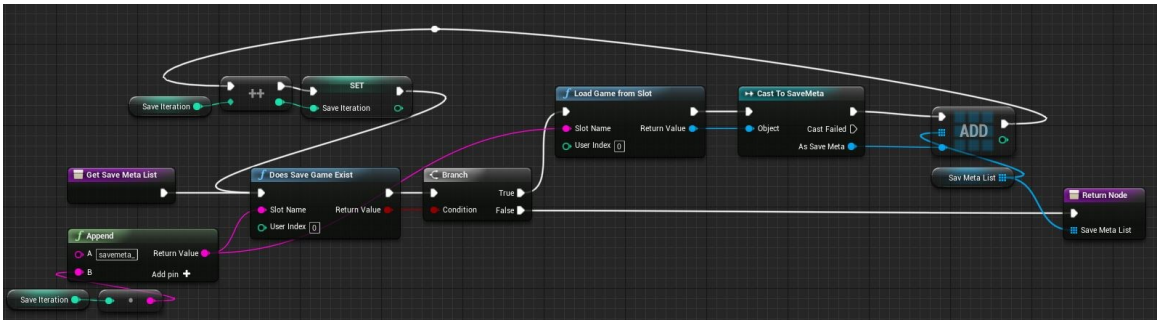
Kuvassa 30 näkyvä tallennusikkuna sisältää tekstiruudun, listan olemassa olevista tallenteista ja *Save*-napin. Tekstiruutuun voidaan syöttää kentän nimi, jos se halutaan tallentaa uutena tiedostona tai jotakin olemassa olevaa tallennetta voidaan klikata, jos nykyinen kenttä halutaan tallentaa sen tilalle. Olemassa olevan tallenteen klikkaaminen asettaa sen kentän

nimen tekstiruutuun, jotta käyttäjälle ei jää epäselväksi millä nimellä tiedosto tullaan tallentamaan.



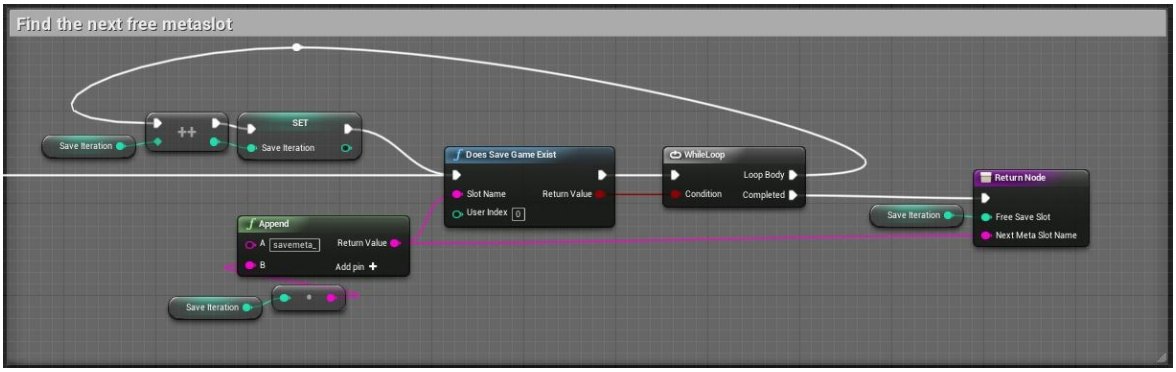
Kuva 30. Tallennusvalikko.

Save-napin painaminen käynnistää tallentamisen, jonka alussa suoritetaan kuvan 31 *GetSaveMetaList*-funktio. Se iteroi läpi kaikki olemassa olevat *savemeta*-tiedostot ja vertaa niiden sisältämiä kentän nimiä, tekstiruudussa olevaan syötteeseen. Jos kentän nimi ei ole käytössä, uuden tiedoston luomisprosessi käynnistetään.



Kuva 31. *GetSaveMetaList*-funktio.

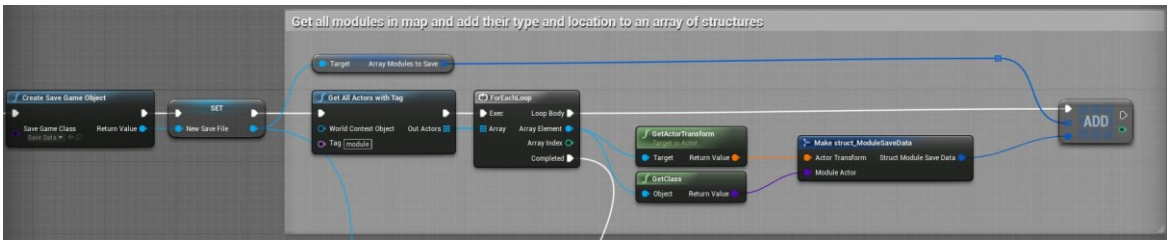
Tallenteelle haetaan vapaa tallennuspaikka kuvan 32 *NextFreeSaveSlot*-funktiolla. Se tarkistaa nimen perusteella onko metatiedosto olemassa. *DoesSaveGameExist*-noodille syötetään merkkijono, joka sisältää tekstin: ”*savemeta_”* + numero. Jos esimerkiksi *savemeta_1* on jo olemassa, numeroa kasvatetaan yhdellä ja tarkistetaan uudelleen. Tämä toistetaan, kunnes vapaa paikka löytyy. Vapaan paikan numero ja tämän löytämiseksi käytetty merkkijono palautetaan.



Kuva 32. *NextFreeSaveSlot*-funktio.

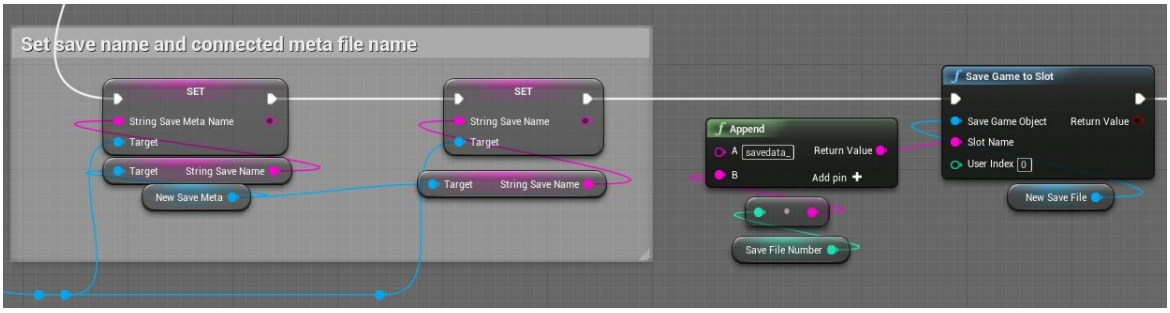
Tallennuspaikka ja metatiedoston tuleva nimi on määritetty, joten ohjelma aloittaa tiedostojen sisällön määrittämisen. *CreateSaveGameObject*-noodilla luodaan uusi instanssi *savemeta*-luokasta. Tälle syötetään käyttäjän päättämä nimi, tallennuspaikan numero ja datatiedoston nimi, joka on ”*savedata_”* + tallenuspaikan numero. *savemeta* tallennetaan tiedostoksi *SaveGameToSlot*-noodilla.

Kuvassa 33 *savedata*-luokasta tehdään uusi instanssi ja kentän kaikki moduulit haetaan *GetAllActorsWithTag*-noodilla. Se palauttaa arrayn moduuleista, joilla on tagi ”module”. Ne käydään läpi *ForEachLoop*-noodilla. Jokaisen moduulin luokka ja transformaatio tallennettaan *scruct_ModuleSaveData*-rakenteeseen. Nämä rakenteet lisätään sen jälkeen *savedatan* sisältämään arrayyn.



Kuva 33. Savedatan sisältämien moduulien määrittäminen.

Kuvassa 34 näkyy, kun *ForEachLoop*-silmukka on lopettanut moduulien lisäämisen, jonka jälkeen *savedata*-tiedostoon tallennetaan sille kuuluvan *savemeta*-tiedoston nimi, pelaajan antama kentän nimi ja sille annetaan nimeksi ”*savedata_”* + tallennuspaikan numero. Tiedosto tallennetaan *SaveGameToSlot*-noodilla.



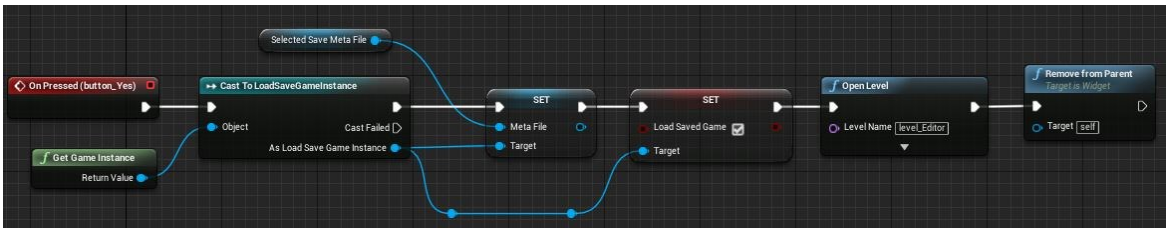
Kuva 34. *Savemetan* nimen ja tallenteen nimen syöttäminen *savedatalle*.

Vanhan tallenteen korvaaminen noudattaa samaa logiikkaa, mutta tallenteen paikkaa ei tarvitse etsiä, vaan se haetaan vanhan tallenteen *savemeta*-tiedostosta, jonka jälkeen olemassa olevan *savedatan* sisältö poistetaan ja sen tilalle lisätään uuden moduulit ja niiden sijainnit.

5.7 **Kentän sisällön lataaminen**

Moduulien lataamiseen käytetään *savedatan* ja *savemetan* lisäksi *GameInstance*-luokkaa. Se on ohjelmanlaajuisesti saatavilla oleva *UObjekti*, johon voidaan tallentaa mitä tahansa dataa, joka halutaan kuljettaa kenttien välillä. Yleisesti ottaen data pitää tallentaa määritys- tai binääri-tiedostoon, jotta sitä voidaan siirtää kenttien välillä latauksen yli, mutta *GameInstance*-luokka säilyttää datan niin kauan, kuin ohjelma on käynnissä. (Epic Games n.d.d)

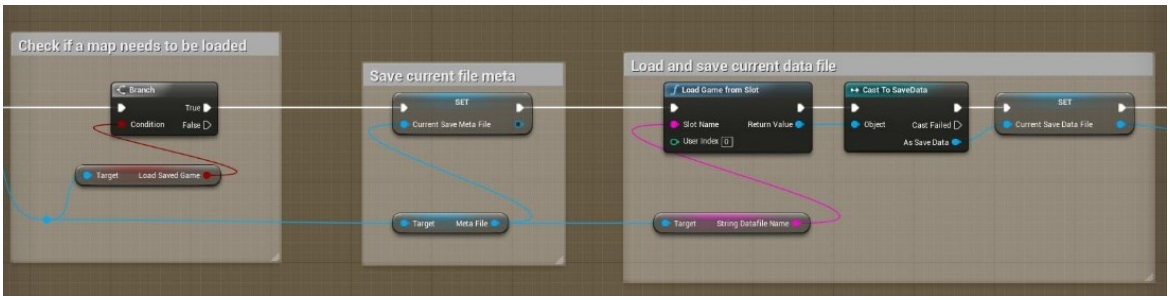
Latausvalikkoon päästään ohjelman päävalikon tai taukovalikon kautta, painamalla *Load Level*-nappia. Tämä avaa latausvalikon, joka sisältää listan nappeja, joiden tekstinä on *savemeta*-tiedostoista haetut kenttien nimet. Latausprosessi alkaa, kun jotain kenttää vastaavaa nappia painetaan ja vahvistetaan valinta. Napille liitetyn savemeta-tiedosto viite syötetään kuvan 35 mukaisesti *GameInstance*-luokan instanssille nimeltä *LoadSaveGameInstance*. Sen sisältämä *LoadSavedGame*-booleani asetetaan todeksi. Seuraavaksi ladataan tyhjä editorikenttä *OpenLevel*-noodilla.



Kuva 35. Metatiedoston viitteen syöttäminen *LoadSaveGameInstancelle*.

Kun editorikenttä aukeaa, editorin toiminnot sisältävä *EditorControls*-Blueprint suorittaa käynnistystoimintonsa ja niiden jälkeen se tarkistaa

kuvan 36 mukaisesti onko *LoadSavedGame*-booleani tosi. Jos on, niin *LoadSaveGameInstancesta* haetaan *savemeta*-tiedosto ja se tallennetaan muuttujaan *CurrentSaveMetaFile*. *Savemetan* avulle haetaan siihen liittyvä *savedata*-tiedosto, joka puolestaan tallennetaan muuttujaan *CurrentSaveDataFile*.



Kuva 36. *EditorControls-Blueprint* sisältämä *savemeta*-tiedoston lataus.

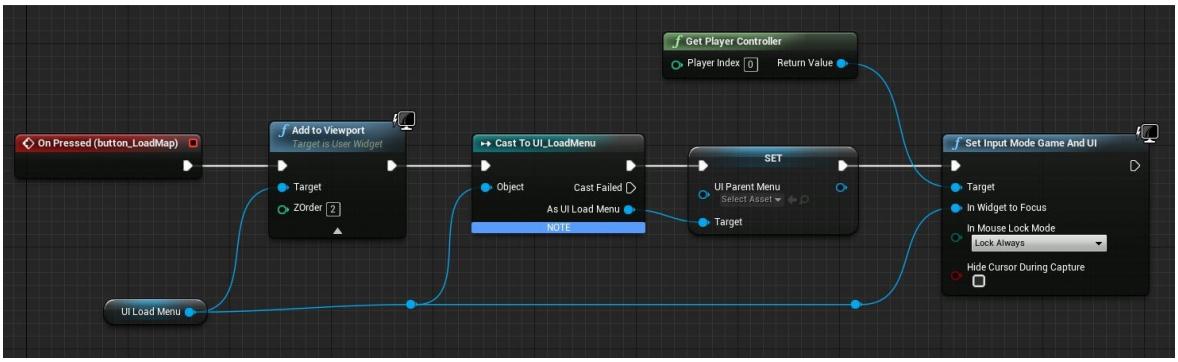
Savedatasta haetaan array, joka sisältää *struct_ModuleSaveData*-rakenteet ja ne käydään läpi *ForEachLoop*-silmukalla. Jokaisen rakenteen kohdalla suoritetaan *SpawnActor*-funktion, jolle syötetään rakenteen sisältämä moduulin luokka ja sen transformaatio. Kun *ForEachLoop* on valmis, kenttään on lisätty kaikki tallenteen moduulit niiden omilla paikoillaan. Lopuksi *LoadSavedGame-booleani* asetetaan epätodeksi, jotta tallenteen lataaminen tulee määrittää uudelleen ja sama kenttä ei lataudu, jos käyttäjä aloittaa työskentelyn tyhjässä kentässä.

5.8 **Editorin valikot**

Projektin suunnitteluEditori sisältää kaksi pääasiallista valikkoa, joista kummastakin päästään sisempiin valikoihin. Ohjelman auetessa, ensimmäinen ruutuun tuleva asia on kuvassa 37 näkyvä päävalikko. Se sisältää kolme nappia. *Create Level*-nappi avaa tyhjän kentän, *Load Level*-nappi avaa alivalikon, josta voidaan ladata aiempi tallennus ja *Quit*-napin painaminen suorittaa konsolikomennon, joka sulkee koko ohjelman. Päävalikko on teemaltaan ja toiminnaltaan samanlainen, kuin kaikki muut ohjelman valikot. Eteenpäin vievät tai tallentavat napit ovat vihreitä, uuden valikon avaavat napit ovat sinisiä ja peruutus tai poistumisnapit ovat punaisia. Kaikkien valikoiden taustalla on vahva sumennus, jonka tarkoituksena on tuoda käyttäjän huomio terävänä näkyvään valikkoon, eikä esimerkiksi taustalla olevaan editorinäkymään.

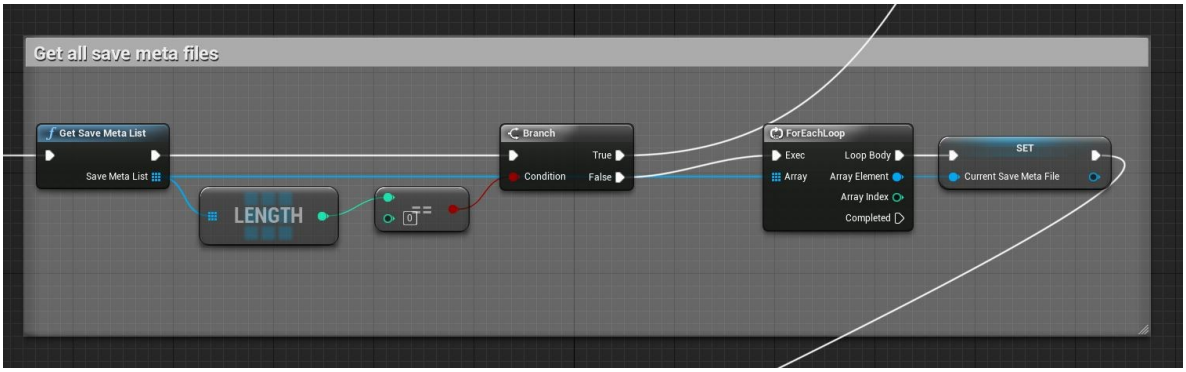
Kuva 37. Editorin päävalikko.

Kuvassa 38 on päävalikon *Load Level*-napin painamisella suoritettava skripti. Ensin ajetaan *AddToViewport*-noodi, jonka referenssiksi on asetettu *UI_Load_Menu*-Widgetin referenssi. Se lisää näytölle uuden kentänlatausvalikon. Skriptin lopussa asetetaan käyttäjän syötteen uudeksi kohteeksi, hetki sitten luotu *UI_Load_Menu*-Widget. Tämä tehdään, koska Widgetit eivät oletusarvoisesti kuuntele käyttäjän syötettä. Se tarkoittaa, että tiettyihin näppäimiin sidotut toiminnot eivät toimi, ilman syötetilan muuttamista Widgetille.



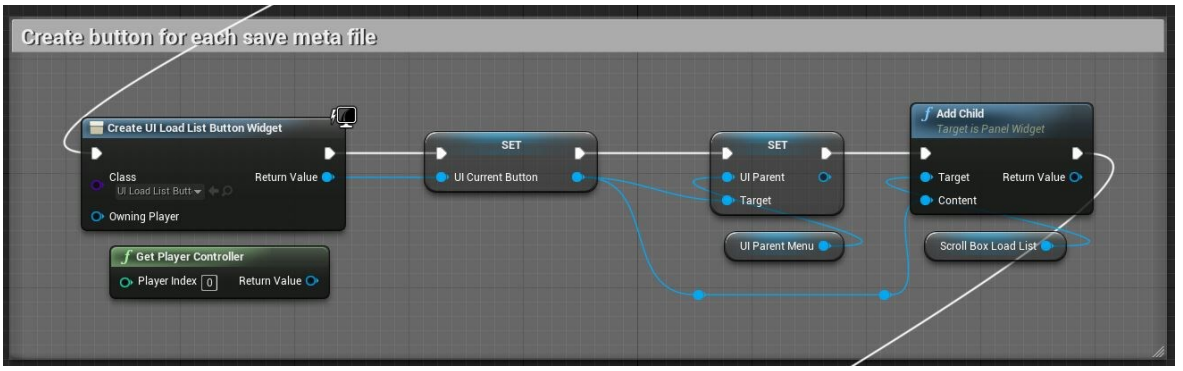
Kuva 38. Päävalikon *Load Level*-napin taustalla oleva kaavio.

Kuvassa 39 haetaan *savemeta*-tiedostot *GetSaveMetaList*-funktiolla. Se palauttaa arrayn kaikista *savemeta*-tiedostoista, jos niiden lukumäärä on suurempi kuin 0. Tämän jälkeen arrayn *savemeta*-tiedostoille aletaan tekemään toimenpiteitä.



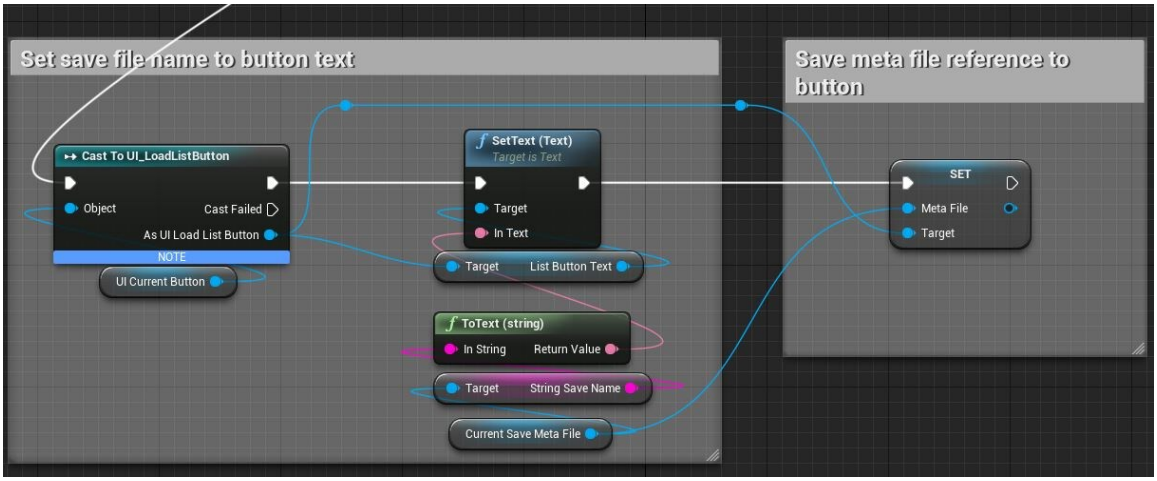
Kuva 39. *LoadList*-menuun tulevien metatiedostojen hakeminen.

Kuvassa 40 jatketaan edellisen kuvan tilanteesta. Jokaiselle löytyneelle metatiedostolle luodaan *UI_LoadListButton*-Widget, jotka lisätään *ScrollBoxLoadList*-säiliön sisälle. Se on laatikko, jonka sisälle napit asetetaan päällekkäin ja sen sisältöä voidaan rullata hiirellä.



Kuva 40. Tallennenappien luominen ja lisääminen tallennuslistaan.

Kuvassa 41 jatketaan edelleen samaa tilannetta. Lopuksi napeille asetetaan *savemeta*-tiedostosta haettu kentän nimi *string_SaveName* ja niiden sisältämälle *MetaFile*-tyypin referenssiksi asetetaan se *savemeta*-tiedosto, jonka tiedoilla nappi luotiin.



Kuva 41. Tallennenappien tekstin asettaminen.

6 POHDINTA

Olen tyytyväinen siihen, että kaikki editorille olennaiset toiminnot tulivat ajoissa valmiiksi. Niiden toteuttaminen yksittäisinä toimintoina oli helppoa, mutta mitä enemmän niitä oli valmiina, sitä enemmän tuli yhteensopivuusongelmia ja vanhoja toimintoja piti muokata. Olennaisin syy tähän oli se, että opinnäytetyön aihe muuttui juuri ennen sen aloittamista, enkä halunnut käyttää kuin muutaman päivän editorin sisällön määrittämiseen ja ominaisuuksien suunnitteluun. Tämä johtui siitä, että oletin projektin vaativan niin paljon työtä, että se oli saatava käyntiin mahdollisimman nopeasti. Kyseinen päätös saattoi aiheuttaa enemmän harmia kuin hyötyä, koska käytin lähes 20 tuntia valmiin koodiin korjaamiseen. Tästä huolimatta olen tyytyväinen päätökseeni muuttaa projektin aihetta, koska nykyinen aihe menee syvemmälle Unreal Engine-kehittämiseen, kuin alkuperäisesti suunniteltu aihe, jossa olisin rakentanut modulaarisen kentän käsin. Tämä projekti oli myös ohjelmointinäkökulmasta haastavampi, kuin alkuperäisesti ajateltu projekti, johon ei olisi sisältynyt editoria.

Unreal Engine 4 oli loistava valinta projektin alustaksi, koska se on vakaa ja sisältää paljon hyödyllisiä työkaluja. Ainoa projektia hidastava asia oli se, että Blueprintit vaikuttavat olevan suunniteltu käytettäväksi jo editorin sisällä olevien tiedostojen käsittelyyn ja niiden käyttäminen ulkoisten tiedostojen käsittelyssä on rajoittunutta, verrattuna perinteiseen ohjelmointiin. Tämä ongelma olisi voitu välttää käyttämällä C++-ohjelmointia, mutta halusin testata mitä Blueprinteillä voi saada aikaiseksi. Blueprinteillä saa tehtyä helposti yksinkertaisia toimintoja, mutta tulen käyttämään tästä lähtien perinteistä ohjelmointia visuaalisen sijaan. Visuaalinen ohjelmointi on hyvä tapa aloittaa ohjelmointiin tutustuminen, koska sen hahmottaminen on helppoa, mutta se on hidasta noodien etsimisen ja kytkemisen takia. Blueprintit eivät myöskään käytä samoja

kaikkia samoja toimintoja, kuin perinteiset ohjelmointikielet, koska ne eivät aina toimi hyvin visuaalisessa käyttöliittymässä. Esimerkiksi if-lause on korvattu branch-noodilla, joka ei sisällä sellaisenaan else if- ja else-lauseita vaan ne tulee rakentaa useasta branch-noodista. For-silmukkaa tarvittaessa täytyy etukäteen valita käyttäkö ForEachLoop-silmukkaa vai ForEachLoopWithBreak-silmukkaa, koska pelkkä ForEachLoop ei sisällä Break-toimintoa. Blueprinteillä saa paljon aikaiseksi, mutta niihin totuttautuminen voi viedä aikaa, jos käyttäjä on tottunut perinteiseen ohjelmointiin.

Tärkein asia jonka opin tästä projektista oli se, että mitä isompi projekti on kyseessä, sitä enemmän kannattaa käyttää aikaa suunnitelmaan, ettei tule tehtyä turhaa työtä. Turhan työn tekemisen lisäksi, työ pitää korjata, eli jokainen tunti turhaa työtä saattaa vaatia toisen tunnin korjaamiseen. Kaikkia ongelmia ei voi mitenkään estää, mutta käytän tästä lähtien aikani ennemmin suunnitteluun, kuin korjaamiseen.

Parasta opinnäytetyössäni oli se, että tein sen ilman toimeksiantoa haluamastani aiheesta ja minulla oli täysin vapaat kädet sen suunnittelussa. Opinnäytetyön aikana valmistunut editori toimii hyvänä pohjana peliprojektille, jota olen suunnittelut. En välttämättä käytä tätä nimenomaista projektia, mutta nyt ymmärrän samantyylisen editorin toimintaa ja vaatimuksia paremmin.

LÄHTEET

Blender (n.d.) About. Viitattu 16.11.2017
https://www.blender.org/about/

Burgess, J. (2013). Skyrim's Modular Approach to Level Design. Blogijulkaisu 19.4.2013. Viitattu 28.10.2017
http://blog.joelburgess.com/2013/04/skyrims-modular-level-design-gdc-2013.html

Epic Games (n.d.a). Unreal Engine 4 Features. Viitattu 6.11.2017
https://www.unrealengine.com/en-US/features

Epic Games (n.d.b). Blueprints Visual Scripting. Viitattu 29.10.2017
https://docs.unrealengine.com/latest/INT/Engine/Blueprints/

Epic Game (n.d.c). UMG UI Designer User Guide. Viitattu 20.12.2017
https://docs.unrealengine.com/latest/INT/Engine/UMG/UserGuide/

Epic Games (n.d.d). Game Instance, Custom Game Instance For Inter-Level Persistent Data Storage. Viitattu 14.12.2017
https://wiki.unrealengine.com/Game_Instance,_Custom_Game_Instance_For_Inter-Level_Persistent_Data_Storage

Gamasutra (2005). Creating Modular Game Art For Fast Level Design. Viitattu 28.10.2017
https://www.gamasutra.com/view/feature/130885/creating_modular_game_art_for_fast_.php

Jones, S. (2011). Investigation into modular design within computer games. Viitattu 29.10.2017
http://www.scottjonescg.co.uk/FYPResearch/Investigation_into_modular_design_within_computer_games_v1.0.pdf

Modular Building Institute (n.d.) Why Build Modular. Viitattu 29.10.2017
http://www.modular.org/htmlpage.aspx?name=why_modular

Modular Design. (n.d.) Modular Design. Viitattu 28.10.2017
https://en.wikipedia.org/wiki/Modular_design

Warped Factor (2016). 10 Things You Might Not Know About SUPER MARIO BROS. Viitattu 29.10.2017
http://www.warpedfactor.com/2015/01/10-things-you-might-not-know-about_21.html