Henna Intke

Lautapasianssipelin suunnittelu ja toteutus

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tieto- ja viestintätekniikka

Insinöörityö

12.9.2024

Tiivistelmä

Tekijä: Henna Intke
Otsikko: Lautapasianssipelin suunnittelu ja toteutus
Sivumäärä: 58 sivua + 1 liite
Aika: 12.9.2024

Tutkinto: Insinööri (AMK)
Tutkinto-ohjelma: Tieto- ja viestintätekniikka

Ohjaaja: ICT ja tuotantotalouden osaamisaluejohtaja Janne Salonen

---

Tässä insinöörityössä toteutettiin olemassa olevasta lautapasianssipelistä digitaalinen versio. Tavoitteena oli tehdä toimiva ja visuaalisesti hyvännäköinen peli. Lisäksi tavoitteena oli laatia pelikehityksen vaiheista selkeä ja helposti ymmärrettävä raportti niin, että aloitteleva pelisuunnittelija voi hyödyntää raporttia tietolähteenä ensimmäisissä pienissä peliprojekteissaan.

Työ aloitettiin ottamalla selvää lautapasianssista ja sen säännöistä. Tämän jälkeen etsittiin tietoa pelikehityksestä ja sen eri vaiheista. Näistä vaiheista muodostettiin selkeä kokonaisuus, kuinka pelikehitysprosessi kulkee suunnittelusta toteutukseen. Jokaiseen vaiheeseen ja niissä huomioitaviin asioihin perehdyttiin huolellisesti.

Kun pelikehityksen vaiheet oli käyty läpi teoriatasolla, aloitettiin lautapasianssipelin toteuttaminen näiden vaiheiden mukaisesti. Kyseessä oli jo olemassa oleva peli, joten suunnitteluvaihe oli huomattavasti lyhyempi kuin silloin, kun toteutetaan täysin uusi peli. Nyt suunnitteluvaiheessa riitti, kun pohdittiin teemaa, pelin ulkoasua ja käyttöliittymää. Suunnitteluvaiheen jälkeen pelissä tarvittava 3D-grafiikka tehtiin Blender-mallinnusohjelmaa käyttäen. Valmis grafiikka vietiin Unity-pelimoottoriin, jota käytettiin pelin lopullisessa toteutuksessa. Pelissä tarvittava ohjelmakoodi kirjoitettiin Visual Studiossa C#-kielellä.

Lopputuloksena saatiin toimiva, Ankkojen Rantapäivä -niminen, lautapasianssipeli. Lisäksi saatiin selkeä ja havainnollinen raportti, joka esittelee pelikehityksen vaiheet sekä teoriassa että käytännössä. Niin pelin tekninen kuin visuaalinenkin toteutus onnistui juuri suunnitelmien mukaisesti. Peli on täysin toimiva, joten sitä voidaan pelata ja sen parissa työskentelyä voidaan jatkaa, sillä peliprojekteille tyypilliseen tapaan, myös tässä työssä jatkokehitysmahdollisuuksia on runsaasti.

Avainsanat: pelikehitys, Blender, Unity, lautapasianssi

Tämän opinnäytetyön alkuperä on tarkastettu Turnitin Originality Check -ohjelmalla.

Abstract

Author:	Henna Intke
Title:	Design and implementation of the peg solitaire game
Number of Pages:	58 pages + 1 appendix
Date:	12 September 2024
Degree:	Bachelor of Engineering
Degree Programme:	Information and Communication Technology
Supervisor:	Janne Salonen, Director of school, School of ICT

In this thesis, a digital version of peg solitaire game was made. The aim was to make a game that works properly and looks great. In addition, the aim was to write a clear and easy-to-understand report about the stages of game development, so that novice game designers can use the report as a source of information in their first small game projects.

The work started by finding out the rules of peg solitaire. After this, literature about game development was searched. The stages of game development were carefully studied and a clear understanding of how game development progresses from design to production was formed.

After the theory part, the digital version of peg solitaire was started to implement according to these stages of game development. Since peg solitaire is an existing game, the design stage was shorter than in the case of making a completely new game. Now it was only necessary to design theme, visual appearance and user interface. After this, 3D graphics were made with 3D creation suite called Blender. The graphics were transferred to game engine called Unity, which was used for the final implementation of the game. The program code needed for the game was written in C# in Visual Studio.

The end result was the peg solitaire game called Ankkojen Rantapäivä, in English Ducks' Beach Day. In addition, the clear and easy-to-understand report was written. This report presents the stages of game development both in theory and in practice. The technical and visual implementation of the game went exactly as planned, so the game can be played and developed further. It is typical for game projects that there are many possibilities for further development, and this project is no exception.

Keywords: Game development, Blender, Unity, Peg Solitaire

Sisällys

1 Johdanto 1

2 Lautapasianssi 2

3 Pelikehitys 5

3.1 Pelin suunnittelu 5
3.2 Grafiikan suunnittelu ja toteutus 10
3.3 Ohjelmointi 11

4 Lautapasianssipelin toteutus: suunnittelu 13

5 Lautapasianssipelin toteutus: grafiikka 15

5.1 Blenderin käyttöliittymä 15
5.2 3D-mallinnus Blenderillä 17
5.2.1 Pelilaudan mallinnus 18
5.2.2 Pelilaudan somisteiden mallinnus 25
5.2.3 Pelinappulan mallinnus 32

6 Lautapasianssipelin toteutus: ohjelmointi 33

6.1 Unityn käyttöliittymä 33
6.2 Ohjelmakoodin kirjoittaminen ja pelin kokoaminen Unityssä 34
6.2.1 Alkuvalmistelut 34
6.2.2 Koordinaatisto 36
6.2.3 Valitsimella liikkuminen 39
6.2.4 Pelinaikainen toiminta ja pelinappuloiden hallinta 40
6.2.5 Animointi 45
6.2.6 Viimeistely 50

7 Yhteenveto 54

Lähteet 56

Liitteet

Liite 1: Kuvakaappauksia Ankkojen Rantapäivä -pelistä

1 Johdanto

Pelikehitys on monivaiheinen projekti, joka alkaa ideoinnista ja päättyy parhaassa tapauksessa valmiin pelin julkaisuun. Kaikki pelit eivät etene suunnittelupöydältä toteutusvaiheeseen tai sen loppuun asti, sillä toisinaan pelisuunnittelija joutuu toteamaan, ettei idea olekaan toimiva, peli ei saavuta riittävää kiinnostusta tai kulut nousevat liian suuriksi. Pienimmillään pelikehitys on yhden henkilön harrasteprojekti, jolla ei välttämättä edes tavoitella taloudellista hyötyä. Suurimmillaan pelikehitys on vakaata yritystoimintaa, jossa peleillä tavoitellaan tuottoa ja pelikehitykseen osallistuu suuri tiimi eri alojen osaajia, kuten graafikoita, ohjelmoijia, äänisuunnittelijoita ja testaajia.

Tässä opinnäytetyössä perehdytään pelikehitykseen toteuttamalla lautapasianssipelistä digitaalinen versio. Aluksi esitellään lyhyesti lautapasianssi ja sen säännöt, jonka jälkeen pelikehitysprosessi käydään läpi teoriatasolla. Tämän jälkeen lautapasianssista tehdään digitaalinen versio teoriaosassa esitettyjen vaiheiden mukaisesti. Työssä ei siis kehitetä uutta peli-ideaa, vaan toteutetaan jo olemassa olevasta pelistä digitaalinen versio. Lautapasianssi valikoitui peliksi sen vuoksi, että se on sopivan yksinkertainen, jolloin pelin ja sen sääntöjen kuvailu ei vie liikaa huomiota varsinaiselta pelikehitysprosessilta. Toisekseen lautapasianssista ei ole tehty tähän mennessä kovinkaan paljon digitaalisia versioita, vaikka se tunnetaan maailmanlaajuisesti.

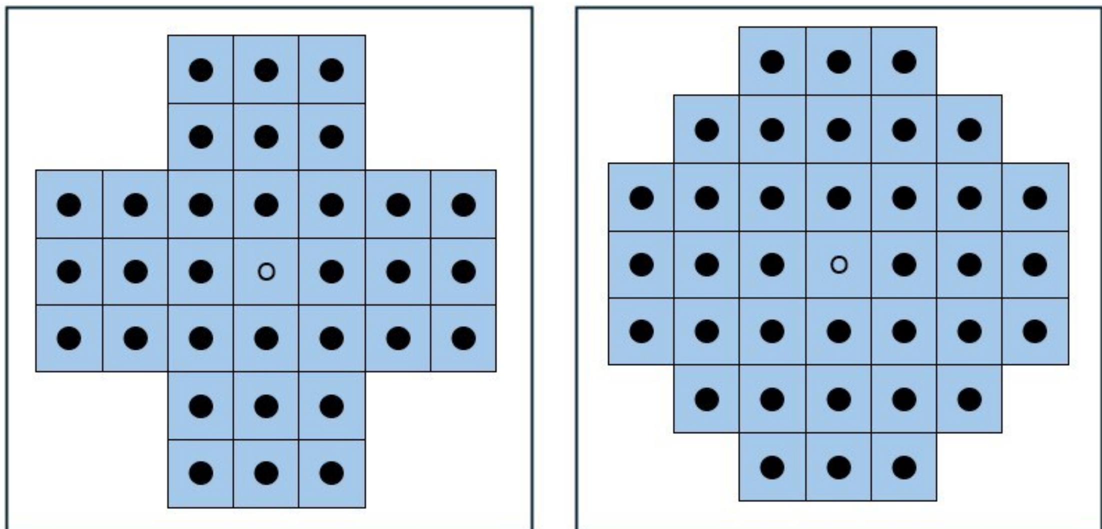
Tämän insinöörityön tärkeimpänä tavoitteena on tehdä lautapasianssista toimiva ja visuaalisesti hyvännäköinen digitaalinen versio. Lisäksi tavoitteena on laatia raportti, jossa pelikehityksen eri vaiheet kuvataan selkeästi ja ymmärrettävästi. Tarkoitus on, että aloitteleva pelisuunnittelija tai pelikehityksestä kiinnostunut henkilö pystyy hyödyntämään raporttia ja käyttämään sitä tietolähteenä ensimmäisissä pienissä peliprojekteissaan.

2 Lautapasianssi

Lautapasianssi (englanniksi peg solitaire) on kansainvälisesti tunnettu yhden hengen lautapeli. Suomessa siitä käytetään myös nimeä erakkopeli. Tässä luvussa esitellään lyhyesti lautapasianssin säännöt sekä pelilaudat, joilla sitä pelataan.

Pelilaudat

Lautapasianssia pelataan tavallisesti joko englantilaisella tai ranskalaisella pelilaudalla. Englantilaisessa pelilaudassa on 33 koloa, kun taas ranskalaisessa niitä on 37. Pelin alkuvalmistelu tehdään kummallakin pelilaudalla samalla tavalla: jokaiseen pelilaudan koloon asetetaan pelinappula paitsi keskimmäiseen. [1, s. 171.] Kuvassa 1 esitetään pelilautojen muodot sekä pelinappuloiden paikat alkutilanteessa.



Kuva 1. Vasemmalla englantilainen ja oikealla ranskalainen pelilauta [1, s. 171]. Musta piste kuvastaa pelin alkuvalmistelussa pelilaudalle asetettua pelinappulaa.

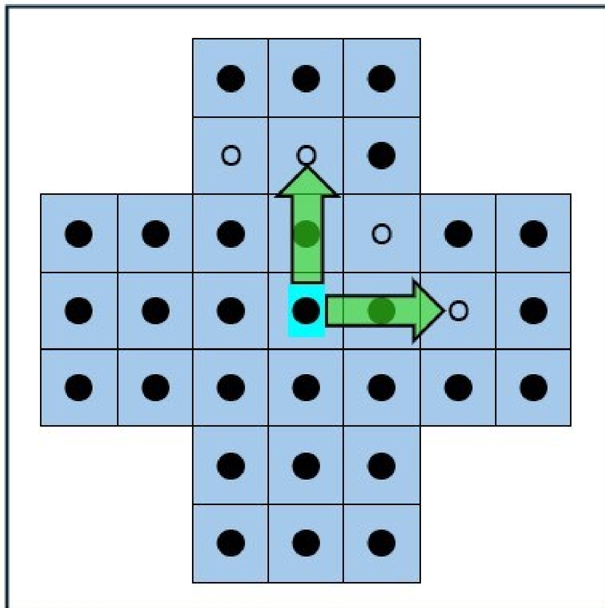
Myös pelin säännöt ovat samanlaiset molemmilla pelilaudoilla. Pelilautojen ainoat eroavaisuudet ovat siis muoto ja koko, jotka tietenkin vaikuttavat siihen,

kuinka peli pelataan läpi. Etenkin lautapasianssin digitaalisissa versioissa voi nähdä käytettävän englantilaisen ja ranskalaisen pelilaudan lisäksi monia muitakin pelilautoja.

Säännöt

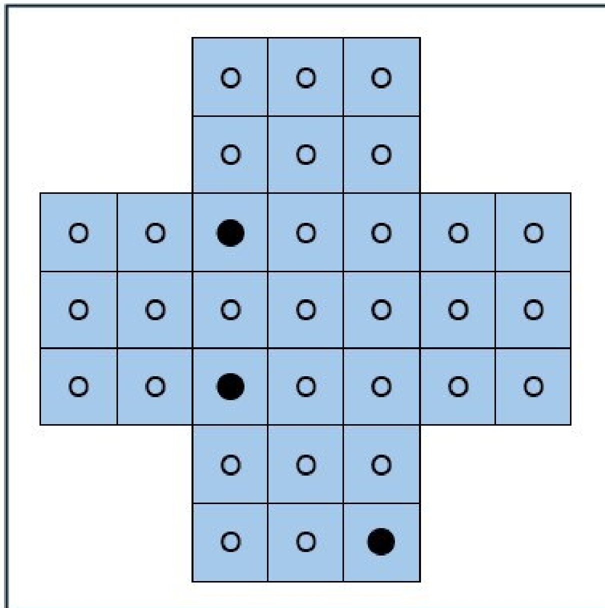
Lautapasianssin säännöt ja tavoite ovat hyvin yksinkertaiset. Tavoitteena on poistaa pelinappulat pelilaudalta siten, että lopuksi jäljellä on vain yksi pelinappula, joka on laudan keskimmäisessä kolossa [1, s. 171].

Pelinappula poistetaan pelilaudalta siten, että valitulla nappulalla siirrytään viereisen nappulan yli sen takana olevaan vapaaseen koloon (kuva 2). Suunta voi olla ylös, alas, vasemmalle tai oikealle, mutta ei viistoon. Pelinappula, joka ylitetään, poistetaan pelilaudalta. Tällainen siirto on pelin ainoa sallittu siirto. [1, s. 171.] Esimerkiksi kahden pelinappulan tai tyhjän kolon ylittäminen ei ole sallittua.



Kuva 2. Kuvitteellinen pelitilanne. Nuolet osoittavat valitun pelinappulan mahdolliset siirrot.

Peli päättyy, kun mahdollisia siirtoja ei enää ole (kuva 3) tai kun pelin tavoite on saavutettu, eli pelilaudalla on vain yksi pelinappula jäljellä [1, s. 171].



Kuva 3. Kuvitteellinen pelitilanne. Peli on päättynyt, koska mahdollisia siirtoja ei enää ole.

Useiden pelinappuloiden siirtäminen saman aikaisesti ei ole sallittua, vaan nappuloita siirretään yksi kerrallaan. Mitä tahansa nappulaa saa siirtää, kunhan siirto tapahtuu sääntöjen mukaisesti. Yhdessä kolossa voi olla vain yksi pelinappula kerrallaan, eli sellaiseen koloon, jossa on jo pelinappula, ei saa siirtää enää toista nappulaa.

3 Pelikehitys

Kemppaisen [2, s. 82] mukaan pelisuunnittelija pystyy tuottamaan pelejä itsenäisesti, mikäli suunnittelija hallitsee kolme eri osa-aluetta: suunnittelun, grafiikan ja ohjelmoinnin. Tässä luvussa käydään läpi pelikehitysprosessi näiden osa-alueiden mukaisesti, eli ensimmäisenä perehdytään suunnitteluvaiheeseen, sen jälkeen grafiikan toteuttamiseen ja lopuksi pelin ohjelmointiin.

3.1 Pelin suunnittelu

Pelin kohderyhmä, sisältö ja mekaniikka muodostavat toisiinsa vahvan yhteyden siten, että kun yhtä näistä kolmesta osasta muutetaan merkittävästi, vaikuttaa muutos myös kahteen muuhun osa-alueeseen [2, s. 102]. Kohderyhmää, sisältöä ja mekaniikkaa voidaankin pitää pelisuunnittelun kulmakivinä ja siksi tässä insinöörityössä tarkastellaan suunnitteluvaihetta juuri näiden kolmen osa-alueen kautta.

Se, mistä osasta suunnittelu aloitetaan, riippuu täysin projektista. Jos on esimerkiksi tietty teema, joka peliin halutaan, voidaan lähteä siitä liikkeelle ja sen jälkeen pohditaan mekaniikkaa ja kenelle peli tehdään. Joskus suunnittelua ohjaa kohderyhmä. Tehtävänä voi olla lapsille sopivan pelin toteuttaminen, jolloin sisältöä ja mekaniikkaa on mietittävä siitä näkökulmasta, mikä on sopivaa lapsipelaajille. Jos keksitään mekaniikka, jonka ympärille peli halutaan toteuttaa, on sen jälkeen suunniteltava pelille sopiva sisältö ja kohderyhmä. Kemppainen [2, s. 55] toteaakin hyvin kirjassaan, ettei pelisuunnittelussa ole yhtä ainoaa totuutta, jonka mukaan tulee aina toimia, vaan hyvään lopputulokseen voidaan päätyä montaa eri tietä pitkin.

Kohderyhmä

Kuten ei ole yhtä ainoaa oikeaa tapaa suunnitella pelejä, ei myöskään ole olemassa yhtä ainoaa pelaajatyyppiä. Toiset pelaavat harrastuksena, toiset pelaavat ammatikseen. Jotkut viettävät päivittäin tunteja pelimaailmassa, kun taas

joillekin pelaaminen on satunnaista huvia silloin tällöin. Yksi nauttii toiminnantäyteisistä moninpeleistä, joissa strategia on suuressa roolissa ja toinen puolestaan viihtyy pelien parissa, joissa pelimaailmaa saa tutkia itsenäisesti kaikessa rauhassa ilman paineita. Hyvän pelisuunnittelijan ominaisuuksiin kuuluukin kyky ymmärtää erilaisia pelaajia ja heidän tapojaan toimia erilaisissa tilanteissa [2, s. 75–76].

Peliä ei voi suunnitella siten, että kohderyhmänä on kaikki tai valtaosa maailman pelaajista, vaan pelaajia on luokiteltava ja kategorisoitava. Pelaajien jakaminen akselille, jonka toisessa päässä ovat kevyesti ja satunnaisesti pelaavat kasuaalipelaajat ja toisessa intohimolla peliin suhtautuvat core-pelaajat, on harhaanjohtavaa, sillä mekaniikaltaan syvällisiä ja strategialtaan monimutkaisia pelejäkin voidaan pelata lyhyesti vain silloin tällöin. Toisaalta kevyttä kasuaalipeliä on mahdollista pelata tuntikausia yhteen menoon. Eli on aivan mahdollista, että sama peli tarjoaa kasuaali-core-akselin molemmille ääripäille sopivan pelikokemuksen. International Hobon BrainHex-malli on yksi esimerkki siitä, kuinka pelaajia voidaan luokitella. [2, s. 100–101, 106–107.]

BrainHex-mallin mukaan pelaajatyyppejä on seitsemän. Etsijä (seeker) on kiinnostunut pelimaailmasta ja nauttii sen ihmeistä. Selviytyjä (survivor) pitää kauhuun liittyvästä voimakkaasta kokemuksesta. Mallia laatiessa ei kuitenkaan ole ollut selvää, tuleeko pelosta nauttimista arvioida kauhun kokemuksen intensiivisyyden vai kauhun jälkeen koetun helpotuksen kautta. Hurjapää (daredevil) on riskinottaja, joka nauttii esimerkiksi takaa-ajon tuomasta jännityksestä, kun taas nero (mastermind) tykkää ratkaista pulmia ja suunnitella strategioita. Valloittaja (conqueror) ei ole tyytyväinen helppoihin voittoihin, vaan haluaa taistella vastoinkäymisiä vastaan. Valloittaja nauttii voittoon asti kamppailemisesta sekä mahdottoman vaikeiden vastustajien ja muiden pelaajien voittamisesta. [3.]

Sosialisoija (socialiser) on luottavainen pelaaja, joka tykkää jutella muiden ihmisten kanssa ja auttaa heitä. Sosialisoija kuitenkin suuttuu pelaajille, jotka käyttävät luottamusta väärin. Valloittaja voidaan nähdä haasteorientoituneena, kun taas saavuttaja (achiever) on selkeämmin tavoiteorientoitunut ja

motivoitunut pitkän aikavälin saavutuksista. Ero näiden kahden pelaajatyypin välillä voi olla hiuksenhieno, mutta silti tärkeä. Saavuttajan pelaaminen perustuu haasteiden suorittamiseen, kun taas valloittajan pelaaminen perustuu haasteiden voittamiseen. [3.] Saavuttajan ja valloittajan eroa voidaankin kuvailla siten, että saavuttaja nauttii haasteiden varsinaisesta suorittamisesta ja valloittaja haasteiden suorittamisen tuomasta voitosta.

Pelaajatyyppejä kuvaavia malleja tarkastellessa on muistettava kriittisyys, sillä malleissa joudutaan yksinkertaistamaan asioita ja on myös mahdollista, että mallia kehittäessä on jäänyt jokin oleellinen näkökulma huomioimatta. Mallit ovat aina vain suuntaa antavia, eikä niillä voida koskaan määritellä ihmisiä täysin tarkasti. [2, s. 108–109.]

Jos pelin kohderyhmää etsitään pelaajatyyppien kautta, ei pidä jumiutua vain yhteen tyyppiin, sillä on hyvä, jos erityyppisille pelaajille pystytään tarjoamaan erilaisia tapoja nauttia samasta pelistä [2, s. 55]. Mikäli suunnitellaan seikkailupeliä etsijöille, voidaan esimerkiksi pohtia, mitä elementtejä peliin täytyy lisätä, jotta myös saavuttajat tai valloittajat saadaan kiinnostumaan. Toisinaan pelin kohderyhmä määräytyy luontevasti pelin teeman kautta. Esimerkiksi nuorille suunnatun hevosaiheisen pelin kohderyhmä voi olla 10–15-vuotiaat hevosista kiinnostuneet. Kohderyhmän ikä on tärkeää määritellä, sillä se vaikuttaa merkittävästi pelin sisältöön. Lapsille tarkoitettujen pelien sisältöä valvotaan tarkasti, eikä kaikkien ansaintamallien käyttökään ole mahdollista lasten peleissä [2, s. 101].

Mekaniikka

Pelimekaniikka on aihe, josta on esitetty paljon erilaisia näkemyksiä ja määritelmiä. Tämä työ käsittelee pelimekaniikkaa yhtenä pelielementtinä, kuten Järvinen [4, s. 55] on sen määritellyt.

Pelimekaniikka on siis yksi pelielementeistä. Muita elementtejä ovat esimerkiksi ympäristö (pelilauta tai -maailma), komponentit (esimerkiksi pelinappulat tai -hahmot), säännöt, teema, käyttöliittymä, tieto ja pelaajat. Pelimekaniikka on yksi

mahdollinen keino, kuinka pelaaja voi olla vuorovaikutuksessa pelin muihin elementteihin. [4, s. 55, 73.]

Yksinkertaisimmillaan pelimekaniikkaa voidaan kuvailla niin, että se on sitä, mitä pelin pelaaminen on, sillä pelimekaniikka merkitsee pelaajan toimintaa ja suoritusta. Tämän vuoksi verbit kuvaavat parhaiten pelimekaniikkaa: arvata, liikkua, kerätä, tähdätä, ampua, potkaista, suorittaa, käydä kauppaa, tarjota, valita ja niin edelleen. [4, s. 74.]

Esimerkiksi kivi-sakset-paperi-pelissä mekaniikka on valitseminen, sillä pelaajan on tehtävä valinta useista eri vaihtoehdoista. Shakkipelin mekaniikka on liike pisteestä pisteeseen, kun pelaajat siirtävät vuorotelleen nappuloitaan pelilaudalla. Päämekaniikalla voi olla myös alimekaniikka. Esimerkiksi kilpa-ajopelissä päämekaniikka on ohjaaminen, kun taas sen alimekaniikkana toimii kiihdyttäminen ja hidastaminen. [4, s. 385, 387, 392.]

Yksinkertaistettuna voidaan siis sanoa, että pelimekaniikkaa suunnitellessa pelisuunnittelija pohtii, mitä pelaajan halutaan pelissä tekevän. Mekaniikka saattaa usein tulla peliin suoraan teemaan mukana, kuten vaikkapa ralli- ja urheilupeleissä, eikä mekaniikkaa välttämättä aina tarvitse erikseen edes suunnitella.

Sisältö

Koska mekaniikka nähdään suunnitteluvaiheen yhtenä osana kohderyhmän ja sisällön rinnalla, voidaan ajatella, että sisällön suunnittelu käsittää tällöin muiden pelielementtien, kuten sääntöjen, ympäristön, komponenttien, teeman ja käyttöliittymän suunnittelun.

Kemppainen [2, s. 71] ehdottaa, että sääntöjä laatiessa aihetta voidaan jakaa pienempiin osiin esimerkiksi seuraavalla tavalla:

- Mitä pelissä voi tehdä ja mitä ei?
- Kuinka peli alkaa ja milloin se loppuu?
- Voittaako joku pelin ja kuinka voittaja määräytyy?
- Onko peli yksin-, monin- vai joukkuepeli?
- Pelataanko peli vain yhden kerran vai koostuuko se eristä tai otteluista?
- Millainen peliympäristö on?
- Koostuuko peli vain muutamasta osasta vai onko siinä lukuisia eri osia, jotka kaikki voivat vaikuttaa toisiinsa?

Pelissä voi olla teema tai sitten peli voi olla teematon eli abstrakti. Perinteiset tammi, shakki ja go ovat esimerkkejä abstrakteista peleistä. [5, s. 197.] Myös shakin kaltaiseen abstraktiin peliin on mahdollista lisätä teema, jolloin teema näkyy pelaajalle pelilaudan ja -nappuloiden kautta. Esimerkiksi maatilateemaisessa shakissa pelilauta esittää maatilan pihaa, ja nappulat ovat erilaisia maatilalta tuttuja asioita, kuten traktoreita, eläimiä ja työvälineitä. Saaren [5, s. 197] mukaan tällaisessa tapauksessa maatilateema voi kuitenkin tuntua hyvin päälle liimatulta ja pelaaja kokee teemasta huolimatta pelin abstraktiksi sen vuoksi, ettei pelin teemalla ole mitään tekemistä pelimekaniikan kanssa.

Mikäli pelissä on tarina, on hahmojen ja pelimaailman lisäksi suunniteltava myös tarinan juoni. Lisäksi joudutaan pohtimaan, kuinka kaikki tämä saadaan peleille ominaiseen interaktiiviseen muotoon. On muistettava, että pelaajalla on aina jonkinlainen vapaus toimia pelissä haluamallaan tavalla, joten pelejä ei ole mahdollista käsikirjoittaa niin suoraviivaisesti kuin vaikkapa kirjoja ja elokuvia käsikirjoitetaan. Pelien tarinoiden käsikirjoittamisen ytimessä on se, kuinka pelaaja saadaan vietyä tarinan läpi, kuinka pelaajan huomio kiinnitettään ja mitä muuta kertomukseen tuodaan päätarinan lisäksi. [2, s. 72–74.]

Pelimaailman ja -hahmojen ohella myös pelin käyttöliittymä on olennainen osa pelikokemusta. Kemppaisen [2, s. 77] mukaan käyttöliittymällä on suuri merkitys siihen, miten pelaaja pystyy uppoutumaan pelin maailmaan. Tietoa pitää antaa

sopivasti, mutta ei liikaa. Lisäksi pelin käyttäminen on oltava loogista ja helposti opittavaa.

3.2 Grafiikan suunnittelu ja toteutus

Kun pelin kohderyhmä, sisältö ja mekaniikka on suunniteltu, voidaan siirtyä grafiikan suunnitteluun ja toteutukseen. Toki grafiikasta on voitu tehdä joitakin suunnitelmia jo pelin sisällön suunnittelun yhteydessä.

Designin ja arkkitehtuurin tuntemisesta on hyötyä peligraafikolle [2, s. 80]. Mikäli alat ovat tuntemattomia, peligraafikon on oltava valmis ottamaan niistä selvää. Esimerkiksi toimintapelissä, joka sijoittuu vanhaan Roomaan kaupungin kapeille kujille, rakennusten arkkitehtuuri ei voi olla mitä sattuu, vaan pelaajan on pelimaailmaa katsoessaan koettava olevansa Roomassa. Muuten pelikokemus jää vajaaksi ja pelaaja voi kokea tulleensa johdetuksi harhaan, jos pelimaailma ei vastaa sitä, mitä sen on luvattu olevan.

Niin 2D- kuin 3D-peleissäkin on tavallista, että pelimaailmaan luodut tilat ovat selkeästi avarampia kuin vastaavat tilat oikeassa maailmassa. Tilaa tarvitaan, jotta pelihahmolla liikkuminen ja sen kontrollointi olisi mahdollisimman sujuvaa. 3D-peleissä hahmoa seuraava kamerakin vaatii tilaa. [2, s. 80–81.] Pelimaailman toteutuksessa ei siis ole tärkeintä täydellisen realistinen mittakaava, vaan riittävä tila, jotta pelikokemus pysyy miellyttävänä.

Photoshop-kuvankäsittelyohjelma on erinomainen työkalu 2D-grafiikoiden toteuttamiseen. 3D-elementtien ja -animaatioiden tekemiseen voidaan käyttää esimerkiksi Blender-mallinnusohjelmaa. Grafiikan suunnittelussa ja toteutuksessa kannattaa rohkeasti kokeilla erilaisia värejä ja niiden yhdistelmiä. Esimerkiksi Adams [6, s. 11] noudattaa kirjassaan periaatetta, ettei vääriä väriyhdistelmiä ole olemassa, vaan erikoisemmallakin värien yhdistelmällä voidaan tuoda projektiin ihan uudenlaista dynamiikkaa.

Järjestys, että ensin tehdään grafiikka täysin valmiiksi ja vasta sen jälkeen ohjelmoidaan, ei ole mitenkään ehdoton. Toimintapelien tuotannolle onkin tyypillistä, että pelin maailmasta tehdään nopea laatikkoversio, jossa hahmojen toiminta hiotaan valmiiksi, ja vasta tämän jälkeen peligraafikko somistaa maailman. [2, s. 80.]

3.3 Ohjelmointi

Ohjelmointivaihe aloitetaan viemällä edellisessä vaiheessa toteutettu grafiikka pelimoottoriin, kuten Unityyn, jossa peli kootaan lopulliseen muotoonsa. Pelin varsinaisen toiminnallisuuden toteuttamiseen tarvitaan ohjelmointia. Unity tukee C#-ohjelmointikieltä, jota voidaan kirjoittaa esimerkiksi Visual Studiossa.

Huolellisesti laaditusta pelisuunnitelmasta on apua ohjelmointivaiheessa. Kun pelin halutut ominaisuudet ja niiden väliset suhteet ovat tiedossa, on eri ominaisuudet helppo asettaa tärkeysjärjestykseen. Tämän jälkeen voidaan lähteä toteuttamaan ominaisuuksia yksi kerrallaan tärkeysjärjestyksen mukaisesti. [2, s. 98.]

Suunnitteluvaiheessa kirjoitetut säännöt on vietävä ohjelmakoodiin, jotta peli tulee toimimaan halutulla tavalla. Tietenkään sääntöjä ei kirjoiteta koodiin samalla tavalla kuin ne on kirjoitettu suunnitteluvaiheessa, vaan pelin säännöt ilmenevät ohjelmakoodissa esimerkiksi erilaisissa toisto- ja ehtolauseissa. Pelin toiminta puretaan ohjelmakoodiin pienen pieniin osiin, ja siellä kerrotaan, miten pelin tulee reagoida pelaajan erilaisiin toimintoihin [2, s. 71, 79]. Mitä isompi ja mutkikkaampi peli on kyseessä, sitä monimutkaisemmaksi ohjelmakoodikin lopulta muodostuu. Aina, kun ohjelmakoodiin lisätään pienikin uusi osa, kannattaa sitä heti testata, sillä näin ohjelmointivirheiden löytäminen ja korjaaminen on helpompaa. Jos ohjelmoidaan kerralla lukuisia monimutkaisia toimintoja, ja vasta lopuksi testataan ja havaitaan virhe, voi olla melko työlästä jäljittää, mikä tai mitkä osat uudesta koodista virheen aiheuttavat.

Ohjelmoidessa kannattaa heti alusta alkaen kiinnittää huomiota siihen, että jokaisella kooditiedostolla on oma selkeä vastuualueensa. Näin koodin rakenne pysyy loogisena ja koodin käsittely, korjaaminen ja päivittäminen tulee olemaan myöhemmin helpompaa, kun ei tarvitse käydä sekalaisia kooditiedostoja läpi ja pohtia, missähän jokin tietty koodin pätkä mahtoikaan olla.

Ohjelmakoodin kommentoiminen voi auttaa hahmottamaan, mitä koodissa tapahtuu. Kommentti on sellainen osa koodia, jota ei suoriteta. Ohjelmoija voi siten merkitä kommenteilla koodin joukkoon itselleen tai muille saman koodin parissa työskenteleville erilaisia huomioita tai vaikka kuvauksen, mihin jotakin muuttujaa tai metodia käytetään. Kommentilla voidaan myös merkitä esimerkiksi lopettavan sulkeen perään, mitä kyseisellä kohdalla suljetaan. Esimerkkikoodissa 1 esitetään, kuinka C#-ohjelmointikielessä kirjoitetaan kommentteja.

```
//Tämä on yhden rivin kommentti.

/*Tässä kommentti on
kirjoitettu kahdelle riville.*/
```

Esimerkkikoodi 1. Koodin kommentoiminen C#-kielessä.

Kommentoinnin ohella myös luokkien, muuttujien ja metodien kuvaava ja johdonmukainen nimeäminen selkeyttää ohjelmakoodia.

4 Lautapasianssipelin toteutus: suunnittelu

Koska lautapasianssi on olemassa oleva peli, suunnitteluvaiheessa ei ollut tarvetta suunnitella peliä alusta alkaen, sillä pelillä oli jo mekaniikka ja säännöt. Sillä oli myös ympäristö ja komponentit eli pelilauta ja pelinappulat. Suunniteltavana oli siten kohderyhmä, käyttöliittymä ja mahdollinen teema, joka tietenkin näkyisi pelilaudan ja -nappuloiden toteutuksessa.

Suunnitteluvaihe aloitettiin kirjoittamalla ylös kaikki mieleen tulleet ideat teemasta, kohderyhmästä ja siitä, millainen olisi toimiva käyttöliittymä. Todettiin, että käyttöliittymään tarvitaan painikkeet, joilla pelaaja voi navigoida etusivun, ohjeiden ja pelin välillä. Lisäksi pelinäkymässä haluttiin näyttää pelaajalle tieto siitä, kuinka monta pelinappulaa pelilaudalla on jäljellä.

Kohderyhmäksi valittiin 30–40-vuotiaat aikuiset. Lautapasianssin voittaminen on haastavaa, joten ajateltiin, että tämän vuoksi pelin parissa viihtyvistä suurin osa on varmasti aikuisia. Suunniteltiin, että pelistä toteutettaisiin mahdollisimman klassinen versio, eikä siihen lisättäisi mitään ylimääräistä teemaa. Pelin värimaailma tulisi olemaan neutraali, jota elävöitettäisiin yhdellä kirkkaammalla tehostevärillä.

Saaren [5, s. 136] mukaan kevyelläkin teemalla varustettua peliä on helpompi myydä kuin täysin teematonta peliä, joten päädyttiin pohtimaan, pitäisikö pelillä sittenkin olla jokin teema. Todettiin myös, että insinöörityössä teemallinen peli olisi toteutuksen kannalta mielekkäämpi, sillä klassiseen versioon mallinnettavaa olisi vain todella vähän ja nekin kappaleet hyvin yksinkertaisia. Tämän vuoksi pelille päädyttiin kuitenkin suunnittelemaan teema.

Teeman suunnittelu aloitettiin pohtimalla, mikä sopisi lautapasianssin ympärille. Pelissä liikutaan yhdellä pelinappulalla toisen yli, ja ylitetty nappula poistetaan pelilaudalta. Tästä tuli mieleen, että pelilauta voisi olla vesialue ja pelinappulat vedessä olevia eläimiä. Pelinappula eli eläin, joka ylitetään, sukeltaisi veteen. Ensimmäisenä ajateltiin värikästä viidakkoympäristöä, jossa viidakon keskellä on lampi, joka on täynnä virtahepoja. Viidakosta tulee mieleen sademetsä,

jossa on liaaneja ja korkeita puita. Tämä olisi ollut pelin selkeyden kannalta haaste, sillä lukuisat korkeat elementit pelilaudan ympärillä olisivat häirinneet pelilaudan näkyvyyttä. Jos korkeita puita olisi ollut vain muutama pelilaudan takaosassa ja etuosassa matalampaa kasvillisuutta, kuten pensaita, vaikutelma viidakosta olisi muuttunut enemmän puistomaiseksi.

Seuraavaksi mietittiin, mitä muuta vesialue voisi kuvastaa kuin viidakkolampea. Mieleen tuli uimaranta, sillä sellainen ympäristö olisi helposti toteutettavissa elementeillä, jotka eivät estä näkyvyyttä pelilaudalle. Virtahevot eivät tuntuneet sopivalta hiekkarannalle, vaan rantaympäristöstä tuli enemmän mieleen ankat. Näin peliä lähdettiin toteuttamaan teemalla ankat rannalla, ja pelin nimeksi valittiin Ankkojen Rantapäivä.

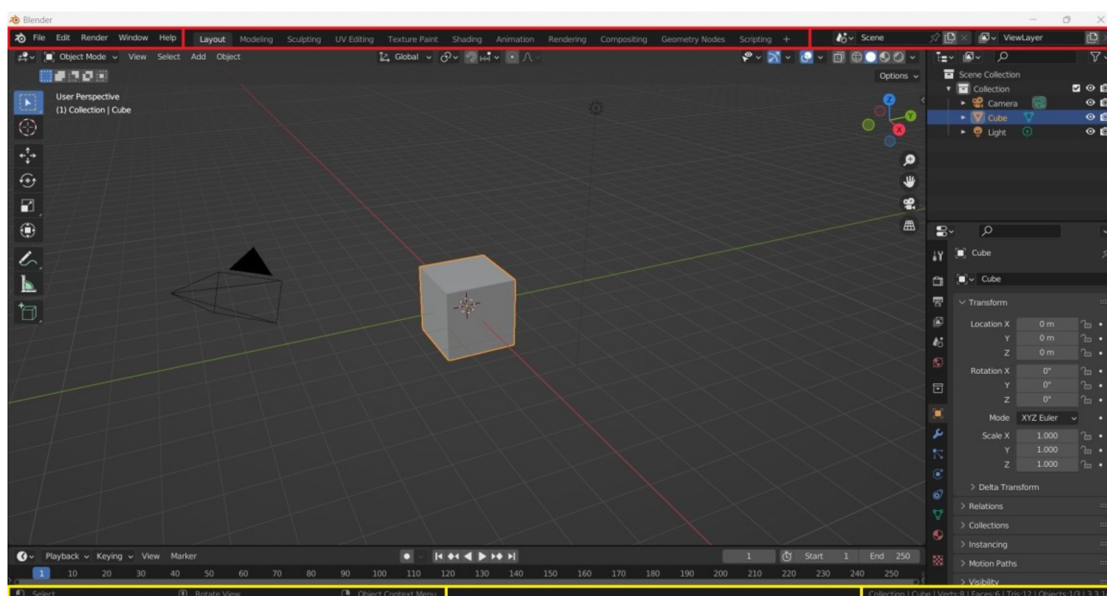
Vaikka peli saikin hauskan teeman, ei kohderyhmää muutettu, sillä hauskan ja värikkään pelin arveltiin edelleen puhuttelevan myös monia aikuisia. Toki tiedostettiin, että pieni osa kohderyhmästä saattaa mieltää pelin nyt liian lapselliseksi. Tämän vastapainona nähtiin kuitenkin se, että värikäs ja hauskannäköinen lautapasianssi voi olla ihan kaiken ikäisille peliä tuntemattomille jopa houkuttelevampi ja helpommin lähestyttävä vaihtoehto kuin perinteinen, neutraali versio. Näin teemallisen version avulla voidaan mahdollisesti tavoittaa paremmin niitäkin pelaajia, jotka eivät ole peliä ennen pelanneet. Nyt myös useampi lapsipelaaja saattaa viihtyä lautapasianssin äärellä kierroksen tai pari ankkojen liikkumista seuraten, vaikka pelin idea ei välttämättä muuten kiinnostaisikaan.

5 Lautapasianssipelin toteutus: grafiikka

Blender on ilmainen, avoimen lähdekoodin 3D-mallinnusohjelma [7]. Tässä luvussa esitellään Blenderin käyttöliittymä ja mallinnetaan pelissä tarvittavat 3D-kappaleet eli pelilauta ja pelinappulat.

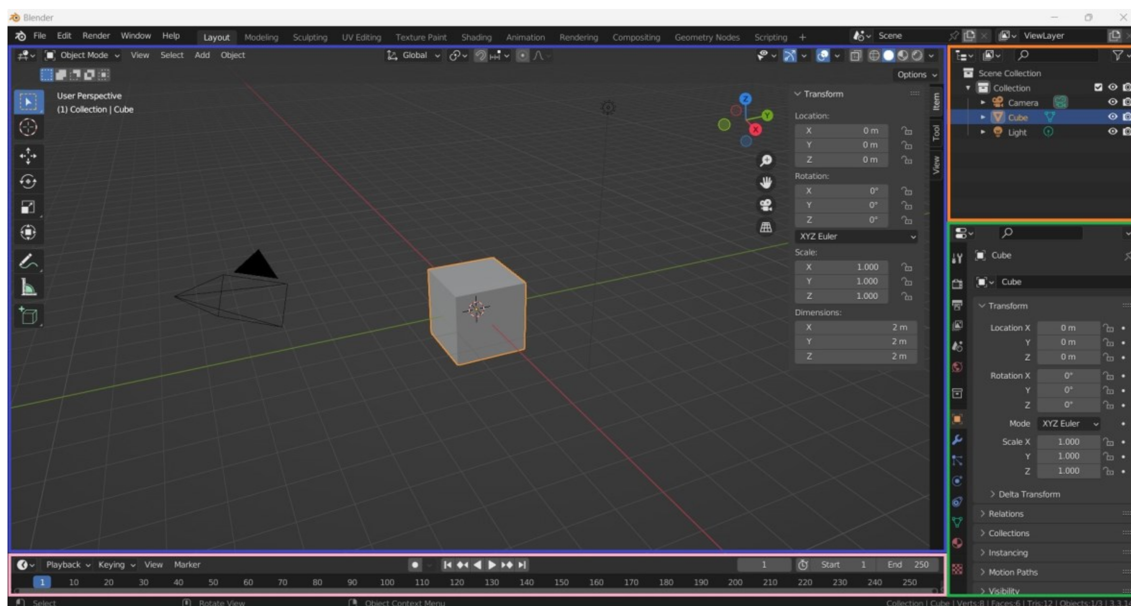
5.1 Blenderin käyttöliittymä

Blenderin käyttöliittymässä on kolme pääosaa: yläpalkki, statuspalkki ja alueet. Alueet ovat editorien näyttämistä varten. Sekä yläpalkki että statuspalkki koostuvat kolmesta osasta (kuva 4). Yläpalkki jakautuu siten, että vasemmalla ovat valikot, keskellä työtilat ja oikealla näkymät ja tasot. Statuspalkissa vasemmalla näytetään näppäintiedot, eli mitä toimintoja hiiren painikkeet tekevät, kun kursori on kyseisellä kohdalla. Jos jokin työkalu on valittu aktiiviseksi, näytetään myös työkaluun liittyvät pikanäppäimet. Keskellä tilaviesteissä näytetään meneillään olevan tehtävän, esimerkiksi renderöinnin, edistyminen. Lisäksi siinä voidaan näyttää tietoa tai varoituksia sisältäviä viestejä. Oikealla resurssitiedoissa kerrotaan tietoja näkymästä, arvio Blenderin käyttämästä RAM-muistista sekä käytössä olevan Blender-ohjelman versio. [8; 9; 10; 11.]



Kuva 4. Blenderin käyttöliittymä. Yläpalkki on korostettu punaisella ja statuspalkki keltaisella. [10; 11.]

Se, mitä editoreja näytetään, riippuu valitusta työtilasta. Esimerkiksi Layout-työtilassa on neljä editoria: 3D Viewport, Timeline, Outliner ja Properties (kuva 5).

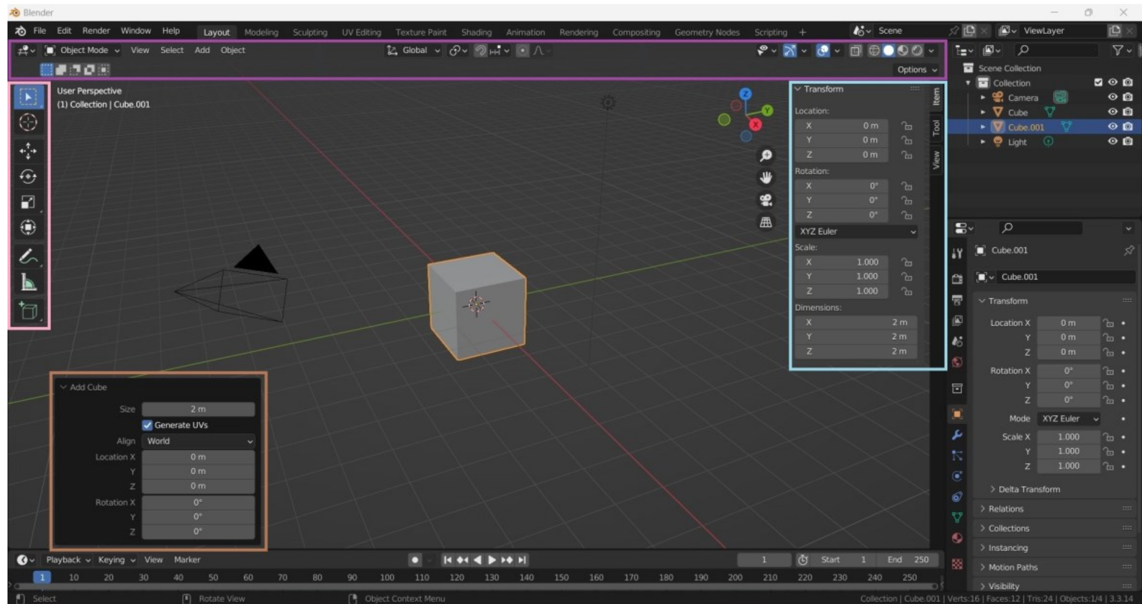


Kuva 5. Layout-työtilan editorit. 3D Viewport on korostettu sinisellä, Timeline vaaleanpunaisella, Outliner oranssilla ja Properties vihreällä.

3D Viewport -editoria käytetään muun muassa mallintamiseen ja animaatioiden tekemiseen. Silloin, kun tehdään animaatioita, tarvitaan myös Timeline-editoria. Outliner-editori listaa kaiken, mitä kyseisessä Blender-tiedostossa on. Properties-editorissa näytetään ja siellä voidaan muokata aktiivisen näkymän tai kappaleen tietoja. [12; 13; 14; 15.]

Käyttäjä voi missä tahansa työtilassa vaihtaa siellä näytettäviä editoreja. Etenkään aloittelevan Blenderin käyttäjän ei kannata siirtyä editorista toiseen vaihtelemalla niitä yhden työtilan sisällä, vaan on järkevämpää siirtyä editorista toiseen työtilaa vaihtamalla. Näin kannattaa toimia sen vuoksi, että jokaiseen työtilaan on määritelty näytettäväksi editorit, jotka ovat kyseisen tehtävän kannalta oleellisia [16]. Blenderissä työtilat on nimetty kuvaavasti, joten valitsemalla tehtävää vastaavan työtilan, käyttäjä saa varmasti käyttöönsä juuri ne editorit, joita tehtävässään tarvitsee.

Kaikki editorit Blenderissä jakautuvat osiin. Kuvassa 6 3D Viewport -editorissa on näkyvissä otsikko, työkalupalkki, sivupalkki ja viimeisimmän toiminnon säätöpaneeli. Näiden keskellä on editorin pääosa, joka näyttää 3D-maailman. [17.]

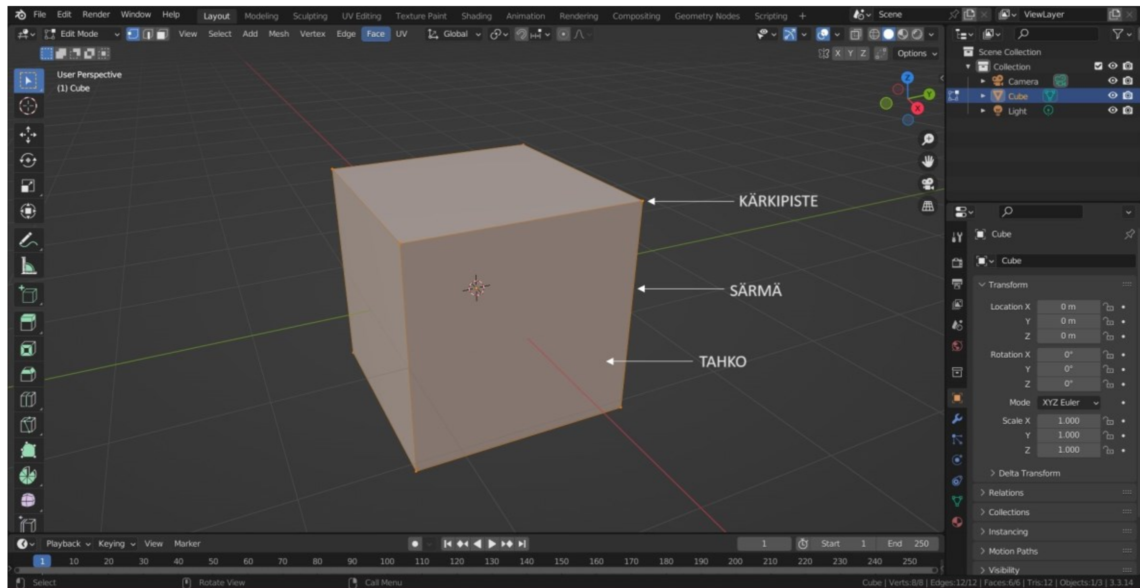


Kuva 6. 3D Viewport -editorin osat. Otsikko on korostettu violetilla, työkalupalkki vaaleanpunaisella, sivupalkki vaaleansinisellä ja viimeisimmän toiminnon säätöpaneeli ruskealla. Näiden keskellä näkyy editorin pääosa. [17.]

Editorin osat voivat pitää sisällään pienempiä jäsentäviä elementtejä, kuten välilehtiä ja paneeleja, joissa on painikkeita ja säätimiä [17].

5.2 3D-mallinnus Blenderillä

3D-malli voi koostua erityyppisistä kappaleista. Mesh eli verkko on yleinen kappaleen tyyppi. Mesh-tyyppinen kappale muodostuu kärkipisteistä, särmistä ja tahkoista (kuva 7). Kärkipiste on yksittäinen piste 3D-avaruudessa ja särmä on suora linja, joka yhdistää kaksi kärkipistettä toisiinsa. Tahko puolestaan on kolmen tai useamman särmän muodostama alue. [18; 19; 20.]



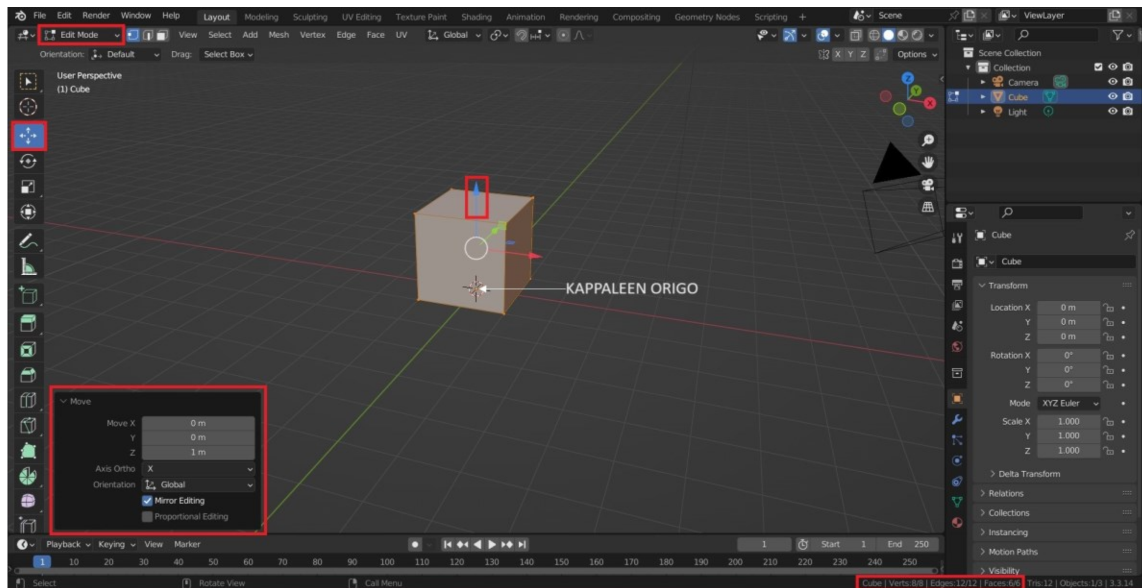
Kuva 7. Mesh-tyyppisen kappaleen rakenne [20].

Mitä tahansa mesh-tyyppistä kappaletta voidaan muokata kärkipisteitä, särmiä ja tahkoja siirtämällä sekä käyttämällä erilaisia työkaluja. Muokkaus tehdään aina muokkaustilassa.

5.2.1 Pelilaudan mallinnus

Pelilaudan mallinnus aloitettiin siirtämällä kaikkia kuution kärkipisteitä, särmiä ja tahkoja yhden metrin verran ylöspäin. Muokkaustilaan siirryttiin otsikkorivin tila-valikosta. Tämän jälkeen katsottiin statuspalkista, että kaikki kappaleen kärkipisteet, särmät ja tahkot ovat valittuina. Sitten työkalupalkista valittiin siirtotyökalu ja vedettiin kuutiota hieman ylöspäin sinisestä, z-akselin suuntaisesta nuolesta. Lopuksi avattiin alalaidasta viimeisimmän toiminnon säätöpaneeli ja asetettiin siirron määräksi yksi metri. Siirto tehtiin, jotta kuution origo saatiin sen keskeltä pohjaan. Origon sijainti on tärkeä, kun kappaletta siirretään, pyöritetään tai skaalataan [21].

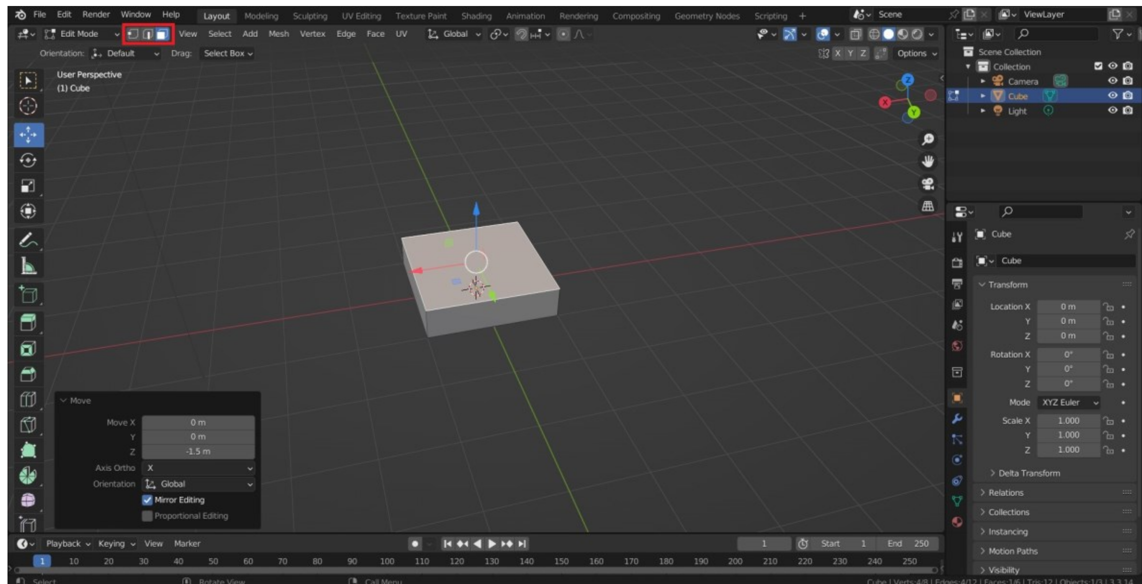
Tilavalikko, siirtämiseen liittyvät vaiheet sekä kuution origoa osoittava pieni oranssi ympyrä esitetään kuvassa 8.



Kuva 8. Kuution kärkipisteiden, särmien ja tahkojen siirtäminen, tilavalikko sekä kuution origo.

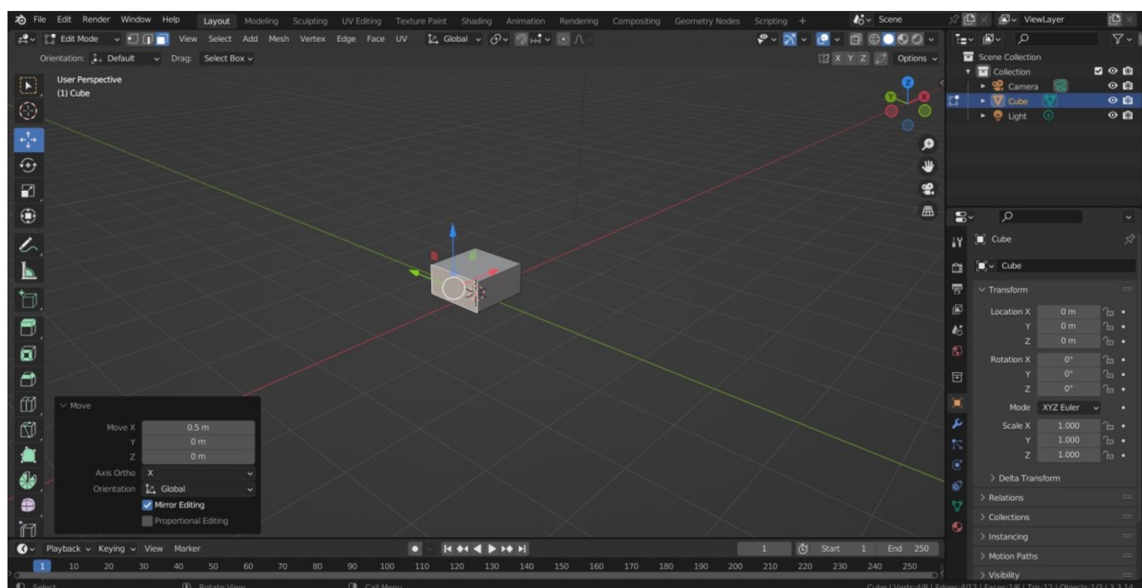
Kun origo oli saatu kuution pohjaan, voitiin aloittaa kuution koon muokkaaminen. Otsikkorivillä vaihdettiin valintatilaksi tahkovalinta ja tämän jälkeen klikattiin kappaleen päällystahkoa. Sitten katsottiin, että siirtotyökalu on aktiivinen ja lähdettiin viemään tahkoa z-akselin suunnassa alaspäin. Tarkka siirron määrä annettiin tälläkin kertaa viimeisimmän toiminnon säätöpaneelissa.

Madallettu kuutio ja painikkeet, joilla valintatilaa voidaan muuttaa, esitetään kuvassa 9.



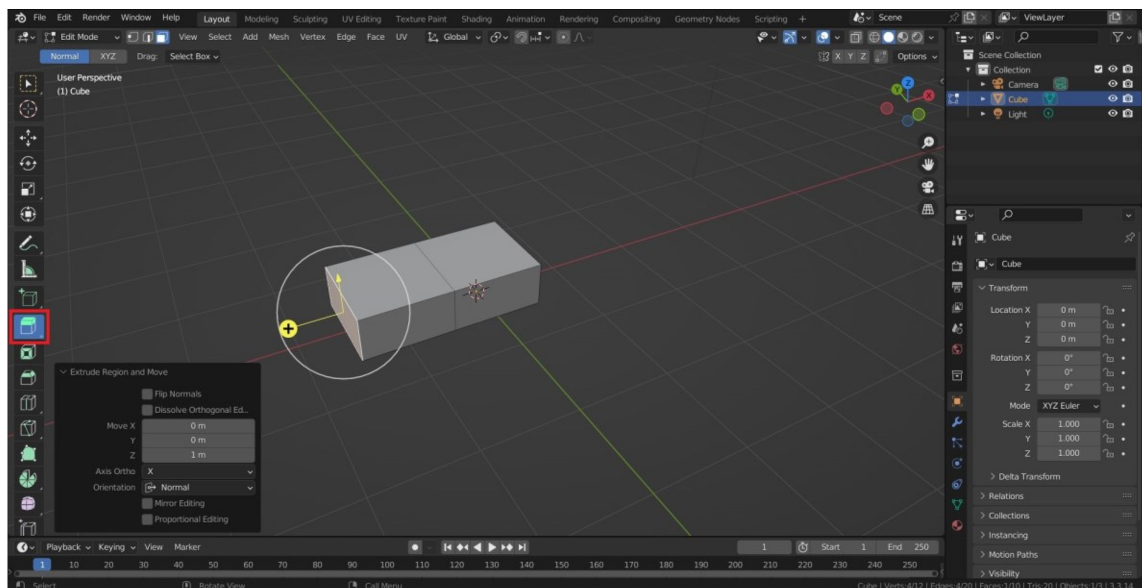
Kuva 9. Madallettu kuutio. Otsikkoriviltä on korostettu painikkeet, joilla voidaan muuttaa valintatilaa.

Jokaista kappaleen sivutahkoa siirrettiin kohti sen keskustaa samaan tapaan kuin päällystahkoa siirrettiin alaspäin (kuva 10). Lopputuloksena saatiin halutun kokoinen kappale, josta pelilautaa voitiin lähteä muodostamaan.



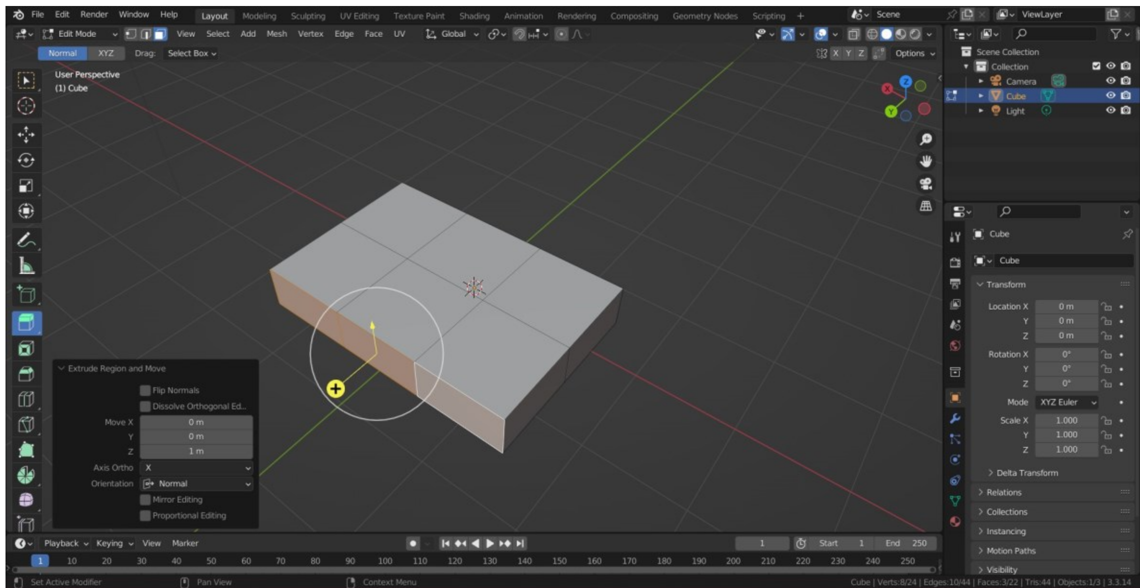
Kuva 10. Kuutio muokattuna sopivan kokoiseksi pelilaudan osaksi.

Lautapasianssin pelilaudan voidaan ajatella muodostuvan pienistä ruuduista, joten valmiina oli ensimmäinen ruutu. Seuraavaksi käyttöön valittiin Extrude Region -työkalu, jonka avulla loput pelilaudan ruudut voitiin tehdä. Työkalun valinnan jälkeen katsottiin otsikkoriviltä, että tahkovalinta on edelleen aktiivinen, klikattiin kappaleen sivutahkoa ja vedettiin sitä keltaisesta plusmerkistä ulospäin. Jälleen tarkka siirron määrä annettiin viimeisimmän toiminnon säätöpaneelissa (kuva 11).



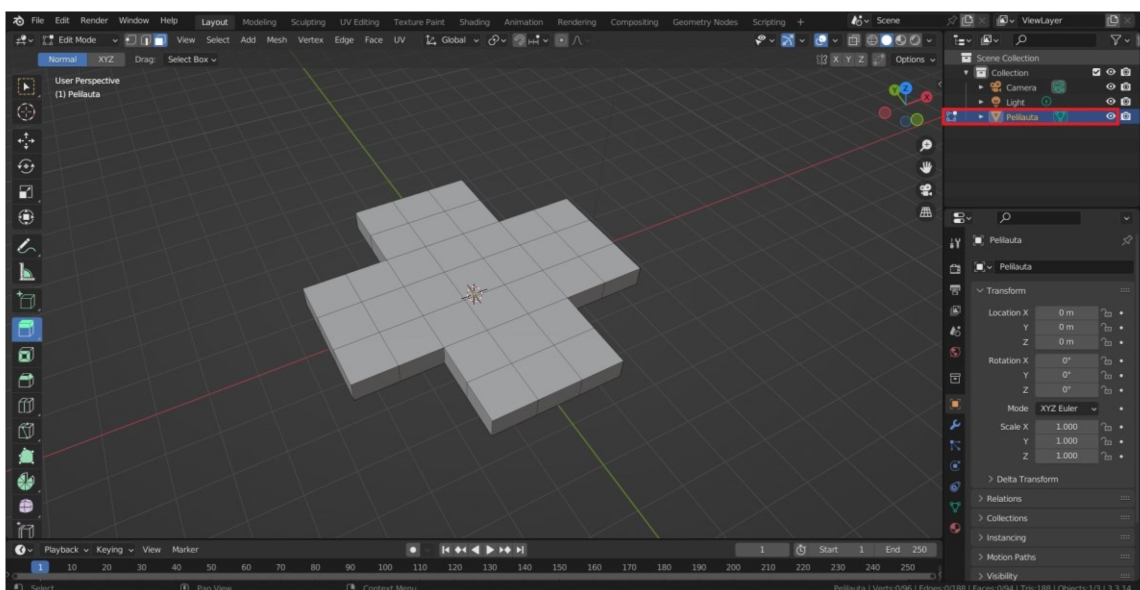
Kuva 11. Pelilaudan toinen ruutu muodostettu Extrude Region -työkalulla.

Ctrl-painiketta pohjassa pitämällä voitiin valita ja käsitellä useita tahkoja samanaikaisesti (kuva 12).



Kuva 12. Extrude Region -työkalu käytettynä useaan tahkoon samanaikaisesti.

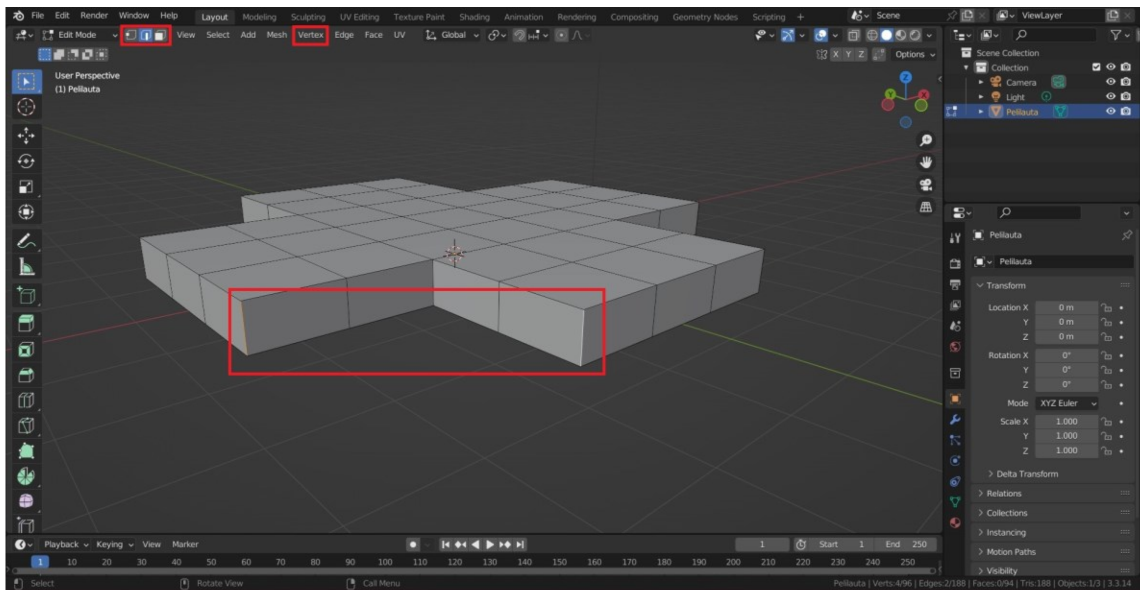
Pelilaudan rakentamista Extrude Region -työkalulla jatkettiin, kunnes ruudukko oli englantilaista pelilautaa vastaava. Kappaleen nimi oli alun perin Cube, mutta se muutettiin Pelilaudaksi Outliner-editorissa (kuva 13). Kappaleen uudelleen nimeäminen onnistuu Outliner-editorissa esimerkiksi kappaleen nimeä kaksoisklikkaamalla.



Kuva 13. Englantilaista pelilautaa vastaava 3D-kappale, joka on nimetty Pelilaudaksi.

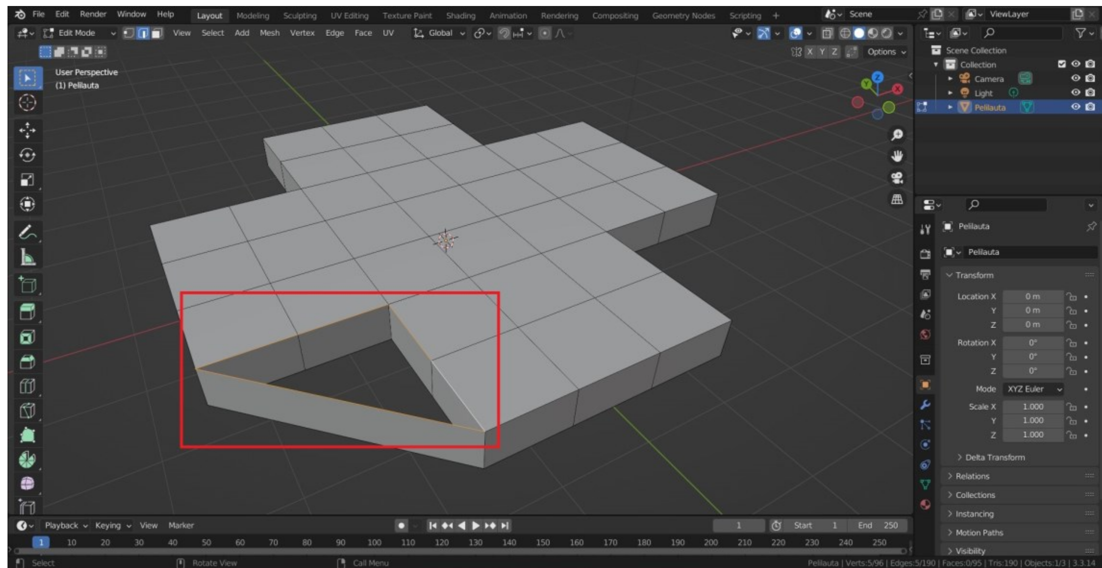
Kuten Outliner-editorista havaitaan, Extrude Region -työkalu ei muodosta uusia kappaleita, vaan se ainoastaan lisää kärkispisteitä, särmiä ja tahkoja muokattavaan kappaleeseen.

Pelilaudan kulmien täyttäminen aloitettiin siten, että otsikkorivillä vaihdettiin valintatilaksi särmävalinta, klikattiin Ctrl-painike pohjassa kulma-alueen molemmat uloimmat särmät aktiivisiksi ja lopuksi valittiin otsikkorivin valikosta Vertex -> New Edge/Face from Vertices (kuva 14).



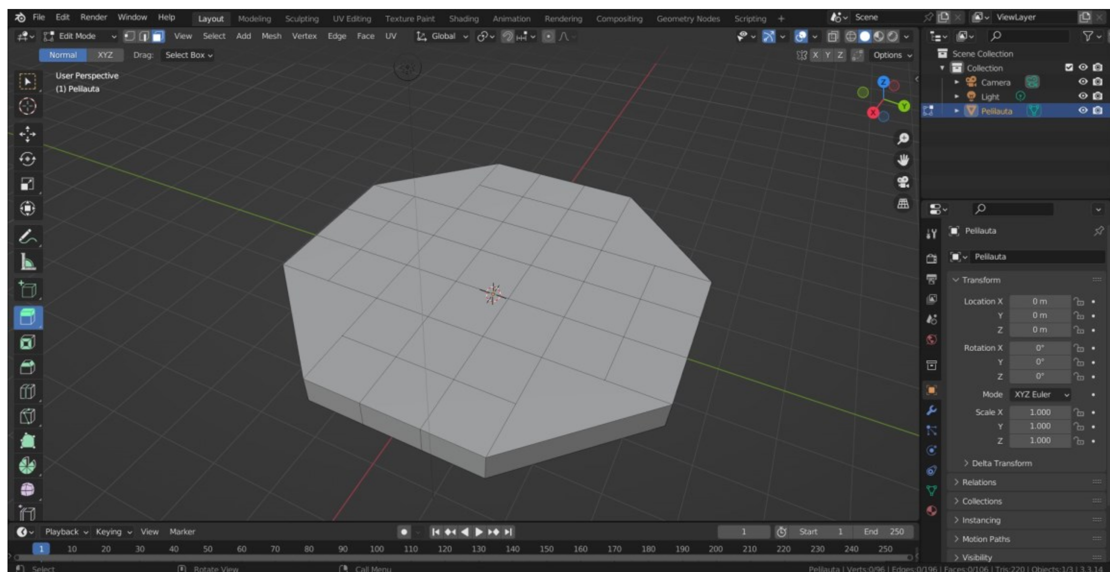
Kuva 14. Uuden tahkon muodostaminen olemassa olevien särmien välille.

Uusi tahko teki kulmaan kolmion muotoisen alueen. Seuraavaksi valittiin aktiivisiksi kolmion muotoista aluetta pelilaudan yläpinnassa rajaavat särmät ja muodostettiin niidenkin välille uusi tahko (kuva 15). Sama toistettiin pelilaudan alapinnassa.



Kuva 15. Kulmaan muodostunutta kolmion mallista aluetta pelilaudan yläpinnassa rajaavat särmät.

Jokainen pelilaudan kulma käsiteltiin edellä kuvatulla tavalla (kuva 16).

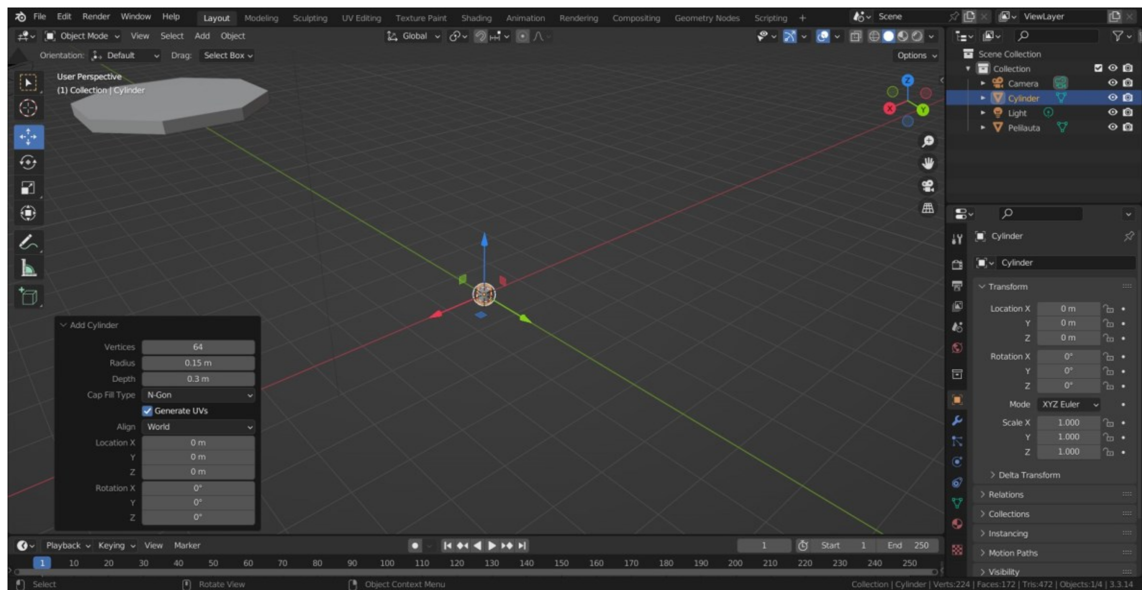


Kuva 16. Pelilaudan kulmiin muodostetut uudet tahkot.

Kun pelilaudan pohja oli valmis, voitiin siirtyä somisteiden mallintamiseen.

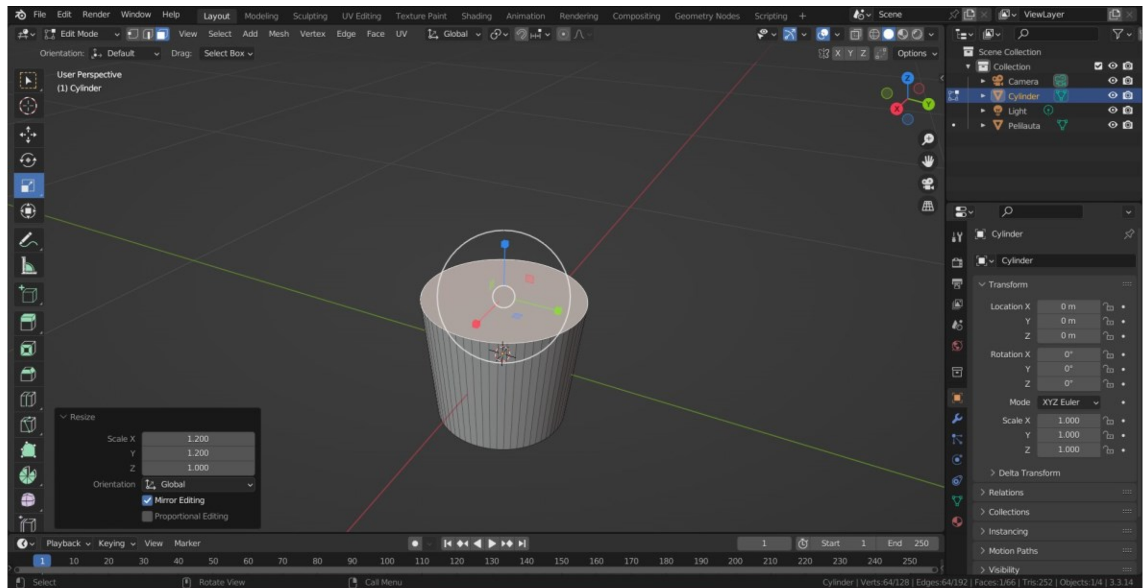
5.2.2 Pelilaudan somisteiden mallinnus

Otsikkorivillä valittiin tilaksi kappaletila ja siirrettiin pelilauta sivummalle siirtotyökalua käyttäen. Kun uusi kappale luodaan, Blender sijoittaa sen koordinaatteihin 0, 0, 0. Siirtämällä pelilauta sivuun, saatiin keskelle tilaa uuden kappaleen käsittelyyn. Uusi kappale, josta tehtäisiin palmun rungon osa, lisättiin otsikkorivin valikosta Add -> Mesh -> Cylinder. Kappaleelle määriteltiin halutut arvot viimeisimmän toiminnon säätöpaneelissa (kuva 17).



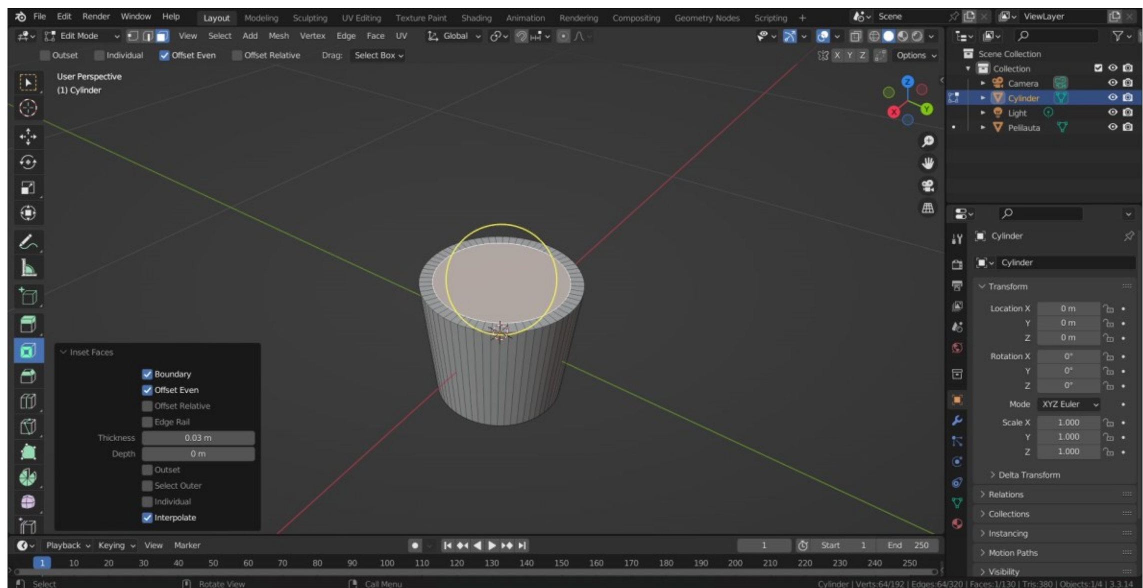
Kuva 17. Uusi kappale 3D-maailmassa.

Seuraavaksi siirryttiin muokkaustilaan, jotta kappaleen päällystahkoa voitiin skaalata. Päällystahko klikattiin aktiiviseksi ja työkalupalkista valittiin skaalaus. Ottamalla kiinni x- ja y-skaala-akseleiden välissä olevasta pienestä sinisestä ruudusta, kappaletta voitiin skaalata x- ja y-akseleiden suhteen z-akselin skaalan pysyessä muuttumattomana (kuva 18).



Kuva 18. Kappaleen päällystahkon skaalaaminen.

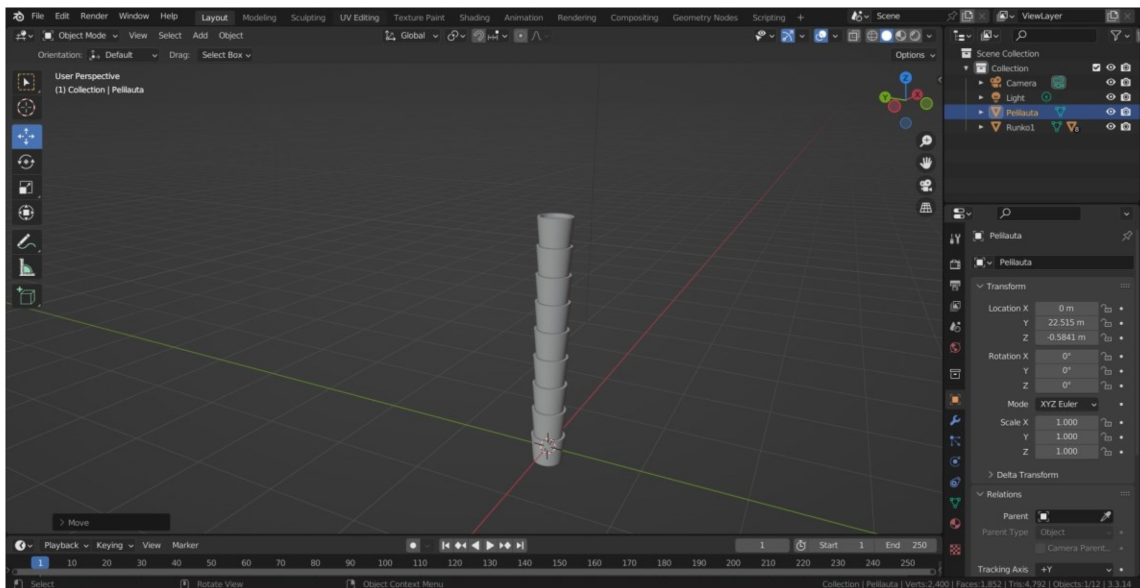
Päällystahkon ollessa edelleen aktiivinen, työkalupalkista otettiin käyttöön Inset Faces -työkalu. Päällystahkon sisään muodostettiin uusi tahko ottamalla kiinni keltaisesta ympyrän kehästä (kuva 19).



Kuva 19. Päällystahkon sisään muodostettu uusi tahko.

Tämän jälkeen käytettiin vielä Extrude Region -työkalua ja työnnettiin sisempää tahkoa aavistuksen verran alaspäin. Näin saatiin valmiiksi palmun rungon yksi osa. Tilaksi vaihdettiin kappaletila, kappale nimettiin uudelleen Outliner-editorissa ja sitä kopioitiin siellä Ctrl+C ja Ctrl+V näppäinkomennoilla, kunnes rungon osia oli riittävä määrä. Siirtotyökalua käyttäen rungon osat järjesteltiin päällekkäin ja muodostettiin kokonainen runko.

Lopuksi Outliner-editorissa otettiin Ctrl-painike pohjassa aktiivisiksi kaikki rungon osat siten, että viimeisenä klikattiin rungon alimmainen osa. Tämän jälkeen otsikkorivin valikosta valittiin Object -> Parent -> Object. Tämä toiminto teki viimeiseksi valitusta osasta eli rungon alimmasta osasta vanhemman ja muista valituista osista sen lapsia (kuva 20). Kun vanhempaa eli rungon alinta osaa siirtää, lapset seuraavat perässä. Kun tehdään 3D-malleja, jotka koostuvat useista kappaleista, on järkevää ryhmitellä kappaleita näin. Se helpottaa huomattavasti valmiin mallin käsittelyä.

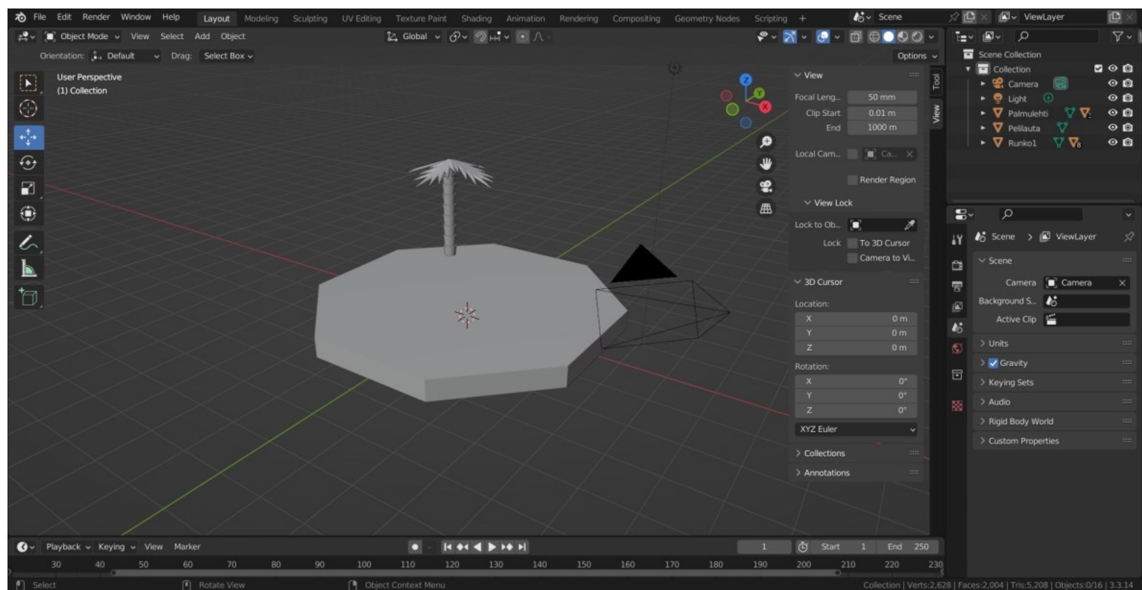


Kuva 20. Palmun rungon ylemmät osat asetettuna alimman osan lapsiksi.

Palmulle tehtiin lehdet siten, että kappaletilassa valittiin työkalupalkista Annotate-työkalu ja piirrettiin haluttu lehden muoto. Tämän jälkeen 3D-maailmaan lisättiin uusi kappale Add -> Mesh -> Plane. Kappale klikattiin aktiiviseksi ja

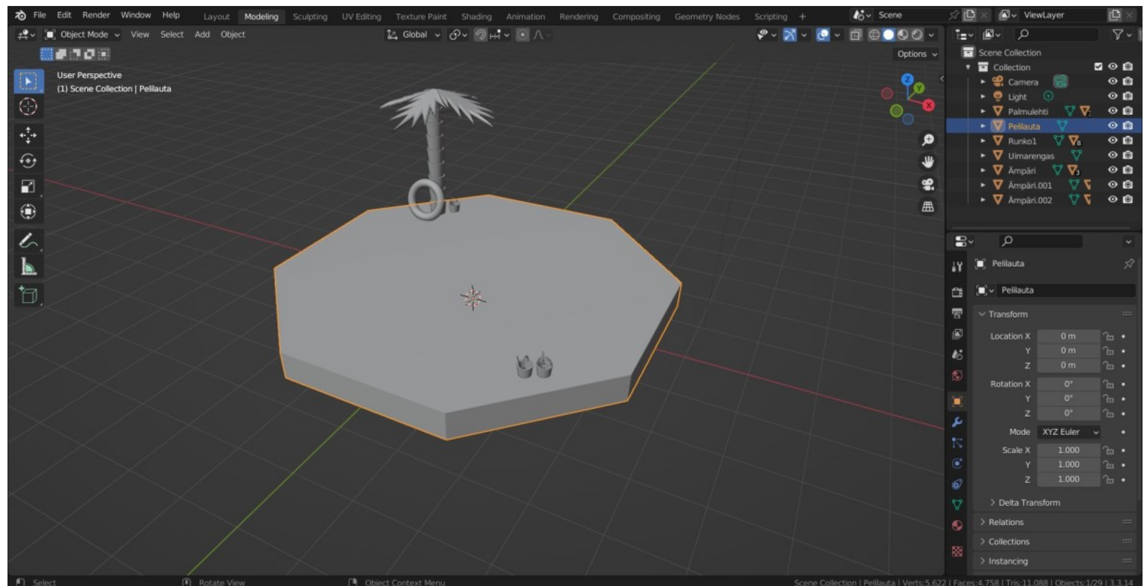
siirryttiin muokkaustilaan. Valintatilaksi otettiin kärkipistevalinta ja klikattiin yhtä kappaleen kärkipistettä. Tämän jälkeen otsikkorivin valikosta valittiin Mesh -> Duplicate ja sijoitettiin uusi kärkipiste piirroksen mukaisesti sopivaan kohtaan. Kärkipisteiden monistamista jatkettiin niin kauan, että kärkipisteillä oli saatu muodostettua palmun lehden muoto piirretyn mallikuvan mukaisesti. Kärkipisteiden välille luotiin särmät ja särmien väliin tahko. Alkuperäisen Plane-kappaleen tahko, särmät ja kärkipisteet poistettiin, jolloin jäljelle jäi ainoastaan juuri muodostettu lehden muotoinen 2D-kappale.

Muokkaustilan työkaluja käyttäen lehdestä työstettiin halutun mallinen 3D-kappale. Kun muoto oli hyvä, kappaletta monistettiin ja lehdet aseteltiin palmun rungon päälle. Lehtien asettelua viimeisteltiin muokkaamalla joidenkin lehtien muotoa särmiä ja kärkipisteitä siirtämällä. Myös palmun lehdille asetettiin vanhempi-lapsi-suhde. Valmis palmu asetettuna pelilaudan päälle esitetään kuvassa 21.



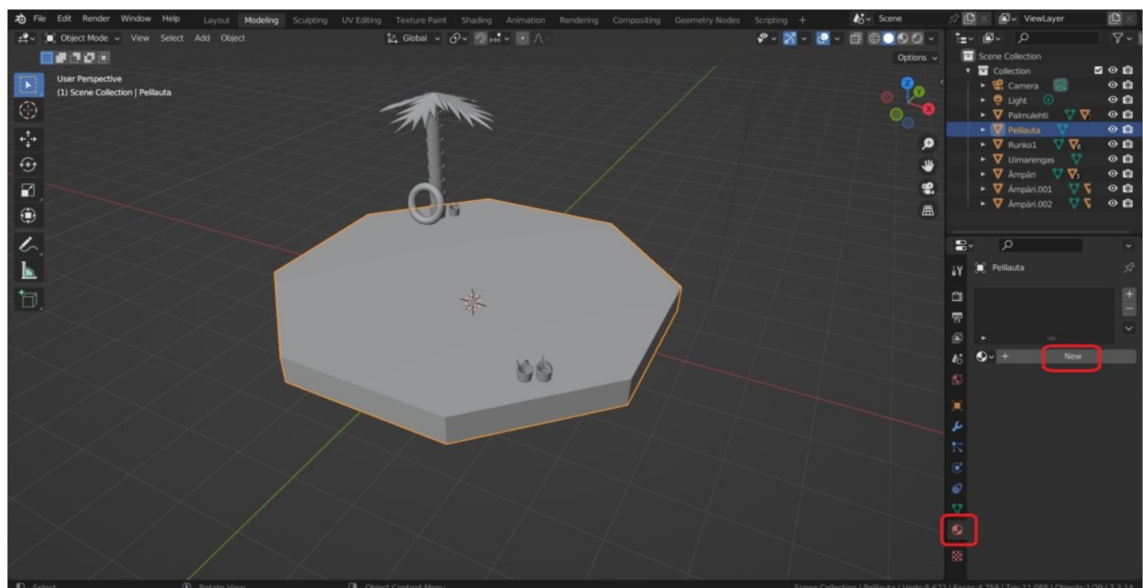
Kuva 21. Palmu asetettuna pelilaudan reunalle.

Pelilautaa somistettiin palmun lisäksi hiekkaämpäreillä ja -lapioilla sekä uimarenkaalla (kuva 22). Nämäkin toteutettiin hyödyntäen muokkaustilan työkaluja sekä poistamalla ja luomalla kärkipisteitä, särmiä ja tahkoja.



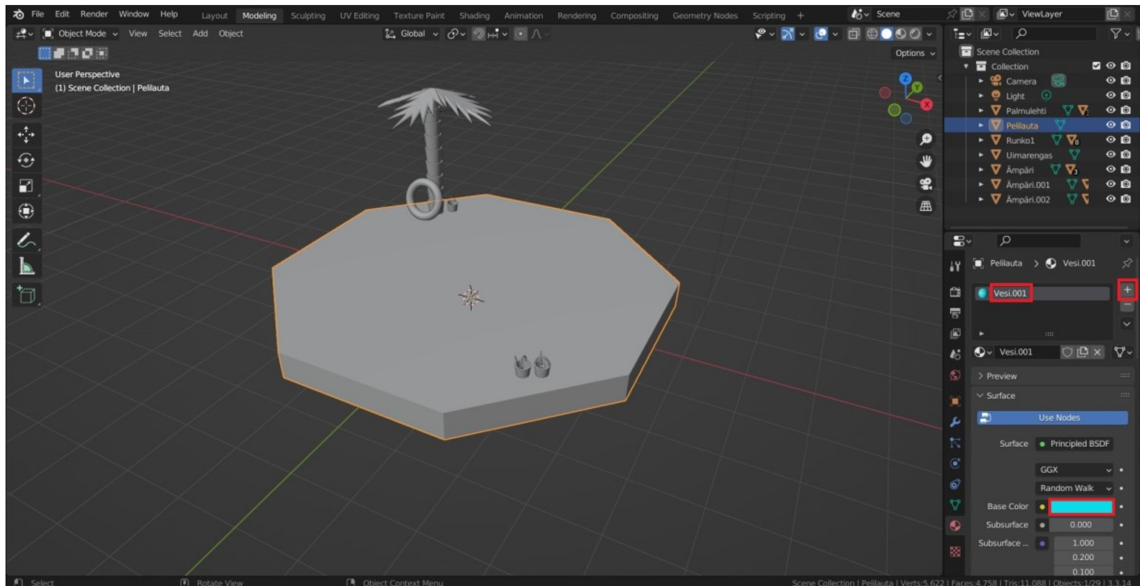
Kuva 22. Pelilauta somistettuna.

Seuraavaksi pelilautaan ja somisteisiin lisättiin värejä. Pelilauta valittiin aktiiviseksi Outliner-editorissa ja Properties-editorissa avattiin Material Properties -välilehti. Uusi materiaali lisättiin klikkaamalla New-painiketta (kuva 23).



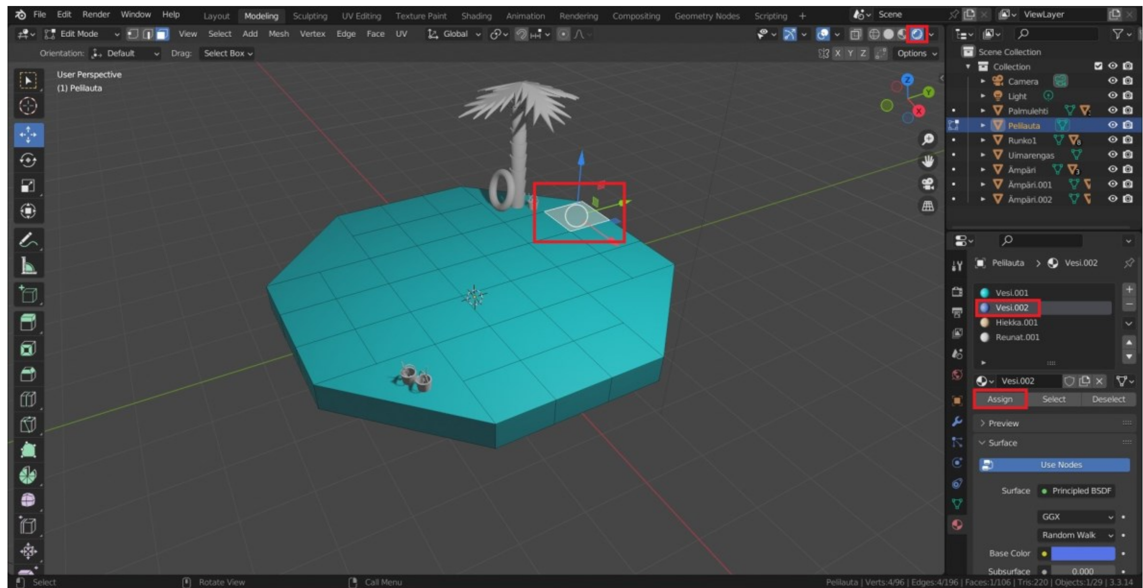
Kuva 23. Uuden materiaalin lisääminen.

Kuvassa 24 esitetään uusien materiaalien määrittely. Aluksi materiaali nimettiin uudelleen sen nimeä tuplaklikkaamalla. Väri määriteltiin kohdassa pohjaväri. Väripaletti aukesi klikkaamalla laatikkoa, joka näyttää käytössä olevan värin. Pelilautaan tarvittiin useampia värejä, joten oikeassa reunassa olevaa plus-merkkiä painamalla päästiin lisäämään ja määrittelemään taas seuraava materiaali.



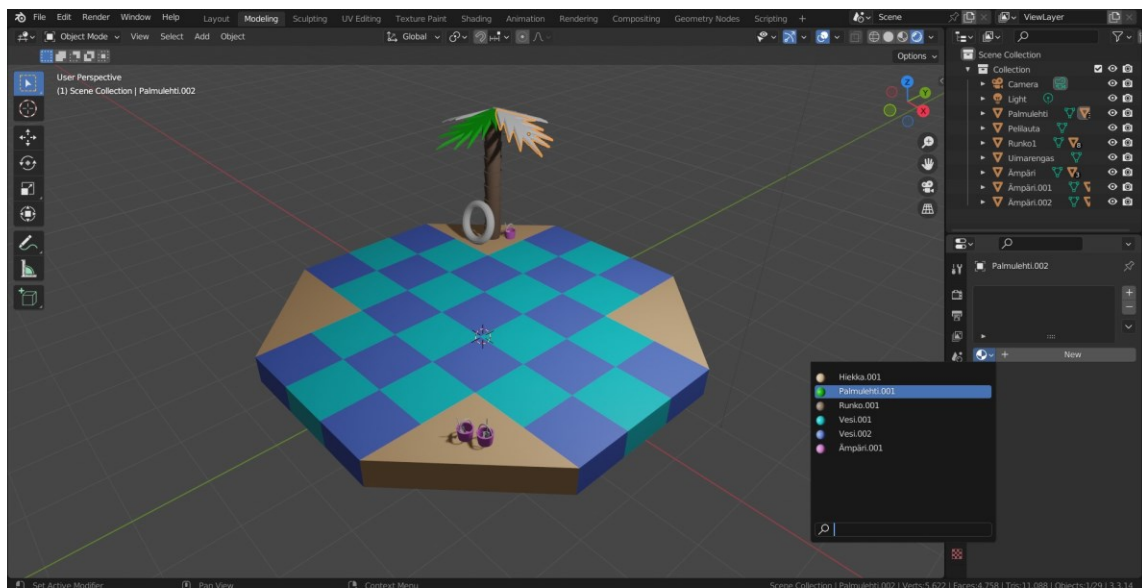
Kuva 24. Uusien materiaalien määrittely.

Kun muutama materiaali oli lisätty ja määritelty, valittiin otsikkorivillä Viewport Shading -kohdasta vaihtoehto rendered, sillä aiemmin valittuna ollut solid ei näytä kappaleille määriteltyjä materiaaleja. Koko pelilaudassa oli käytössä ensimmäisenä tehty materiaali, mutta tarvittaessa vaikka jokaiselle tahkolle voidaan asettaa oma materiaalinsa. Tahkolle asetetaan materiaali valitsemalla se aktiiviseksi, klikkaamalla listasta haluttu materiaali ja painamalla Assign-painiketta. Nämä esitetään kuvassa 25.



Kuva 25. Materiaalin asettaminen yhdelle tahkolle.

Koska palmun lehdissä käytettiin vain yhtä väriä, niille voitiin tehdä ja asettaa materiaali kappaletilassakin. Kun materiaali oli luotu ja määritelty ensimmäiselle lehdelle, voitiin seuraaville lehdille materiaali ottaa käyttöön valitsemalla se avautuvasta luettelosta (kuva 26).

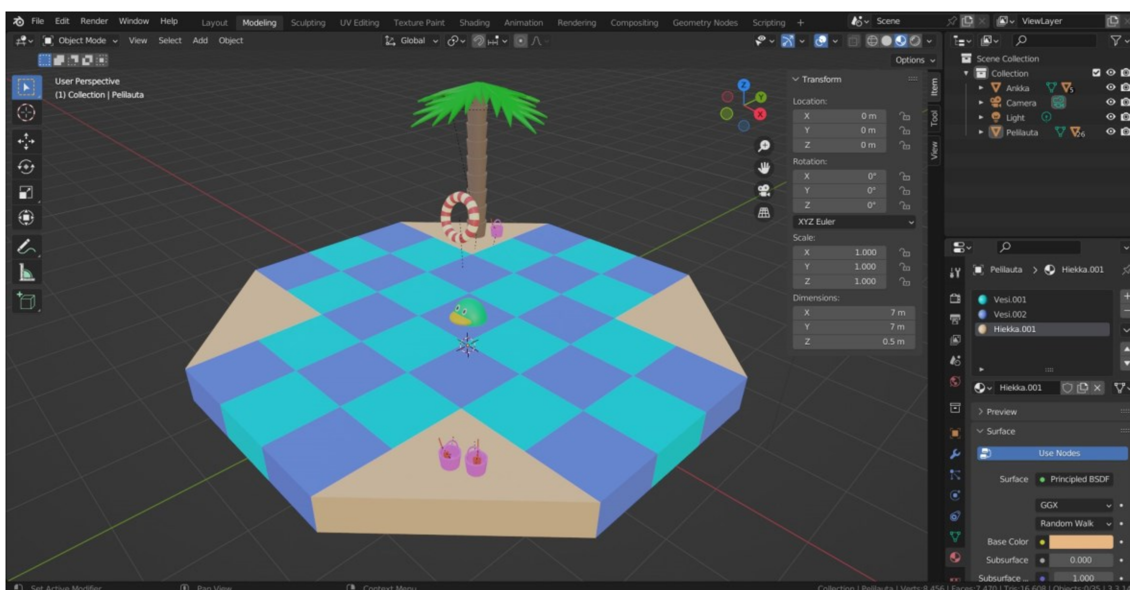


Kuva 26. Materiaalin valitseminen luettelosta.

Pelilauta ja sen somisteet väritettiin loppuun ja tämän jälkeen mallinnettiin vielä viimeinen puuttuva kappale eli pelinappula.

5.2.3 Pelinappulan mallinnus

Myös pelinappula toteutettiin muokkaustilan työkaluja käyttämällä sekä poistamalla ja luomalla kärkipisteitä, särmiä ja tahkoja. Lopuksi vielä määriteltiin muutama uusi materiaali, jotta pelinappulallekin saatiin väritys. Pelilauta ja ankan muotoinen pelinappula esitetään kuvassa 27.



Kuva 27. Pelilauta ja ankan muotoinen pelinappula.

Kaikkien pelinappuloiden haluttiin olevan samanvärisiä, koska somisteissa ja pelilaudassa oli jo käytössä useampi eri väri. Jos pelinappuloita olisi toteutettu esimerkiksi kahdessa eri värissä, kokonaisuudesta olisi tullut liian kirjava. Mikäli pelilauta olisi ollut yksivärinen tai vaikkapa mustavalkoinen, olisi kokonaisuuteen tällöin voitu tuoda lisäväriä monenvärisillä pelinappuloilla.

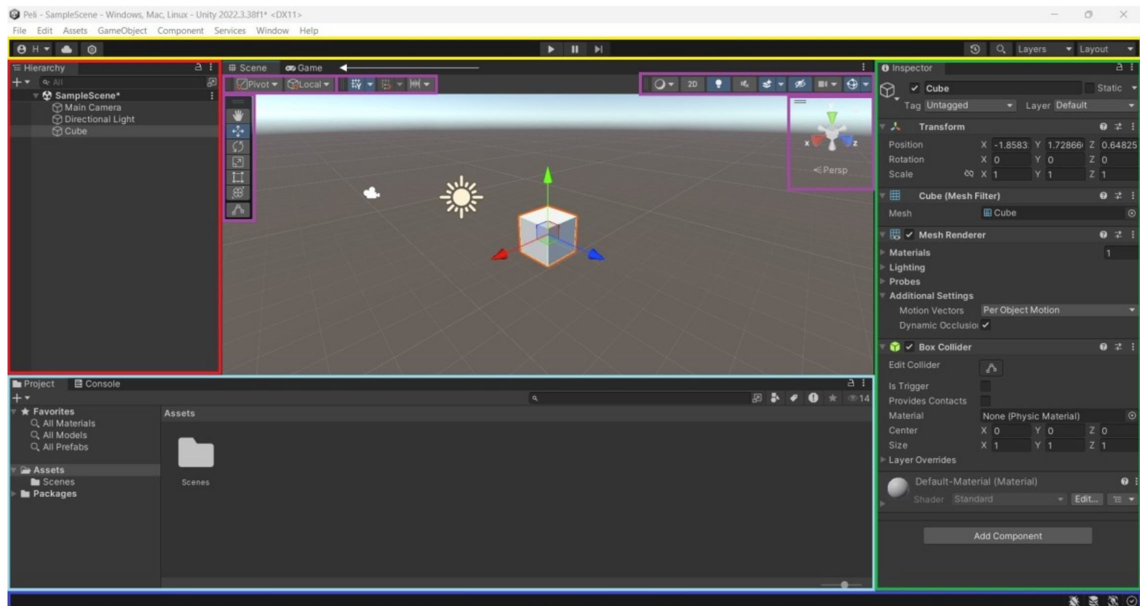
Kaikki peliin tarvittavat erilaiset osat oli mallinnettu, joten seuraavaksi voitiin viedä mallit Unityyn ja aloittaa pelin kokoaminen ja ohjelmointi.

6 Lautapasianssipelin toteutus: ohjelmointi

6.1 Unityn käyttöliittymä

Myös Unityssä käyttöliittymä koostuu useasta eri osasta (kuva 28). Ylhäällä, valikkorivin alapuolella, on työkalupalkki. Käyttöliittymän vasemmassa reunassa on Hierarchy-ikkuna, jossa on listattuna kaikki kohtauksessa olevat peliobjektit. Oikeassa reunassa puolestaan on Inspector-ikkuna. Siellä voidaan katsoa ja muokata aktiiviseksi valitun peliobjektin kaikkia ominaisuuksia. Erityyppisillä peliobjekteilla on erilaiset ominaisuudet, joten Inspector-ikkunan sisältö muuttuu joka kerta, kun valitsee eri peliobjektin. Alalaidan iso ruutu on Project-ikkuna. Siellä näytetään kyseisen Unity-projektin kansiorakenne ja kaikki projektiin liittyvät tiedostot. Käyttöliittymän alimpana osana on statuspalkki, joka näyttää esimerkiksi ilmoituksia Unityn eri prosesseista. [22; 23.]

Käyttöliittymän keskellä on kohtausnäkymä, joka on tarkoitettu meneillään olevan projektin rakentamiseen, tarkasteluun ja muokkaamiseen. Kohtausnäkymän yläpuolella ja päällä on paneeleja ja työkalupalkkeja. Ne sisältävät perustyökalut kohtausnäkymän ja peliobjektien muokkaamiseen. Kohtausnäkymän lisäksi projektia voidaan katsoa myös pelinäkymässä. Pelinäkymässä projekti näytetään renderöitynä ja siellä projektia voidaan myös testata. [22; 24; 25; 26.] Näytettävä näkymä valitaan käyttämällä esimerkiksi välilehtiä, jotka ovat näkymän yläpuolella.



Kuva 28. Unityn käyttöliittymä. Työkalupalkki on korostettu keltaisella, Hierarchy-ikkuna punaisella, Inspector-ikkuna vihreällä, Project-ikkuna vaaleansinisellä ja statuspalkki tummansinisellä. Keskellä on kohtausnäkymä, johon kuuluvat paneelit ja työkalupalkit on korostettu violetilla. [22; 27.] Valkoinen nuoli osoittaa välilehdet, joilla näytettävä näkymä voidaan valita.

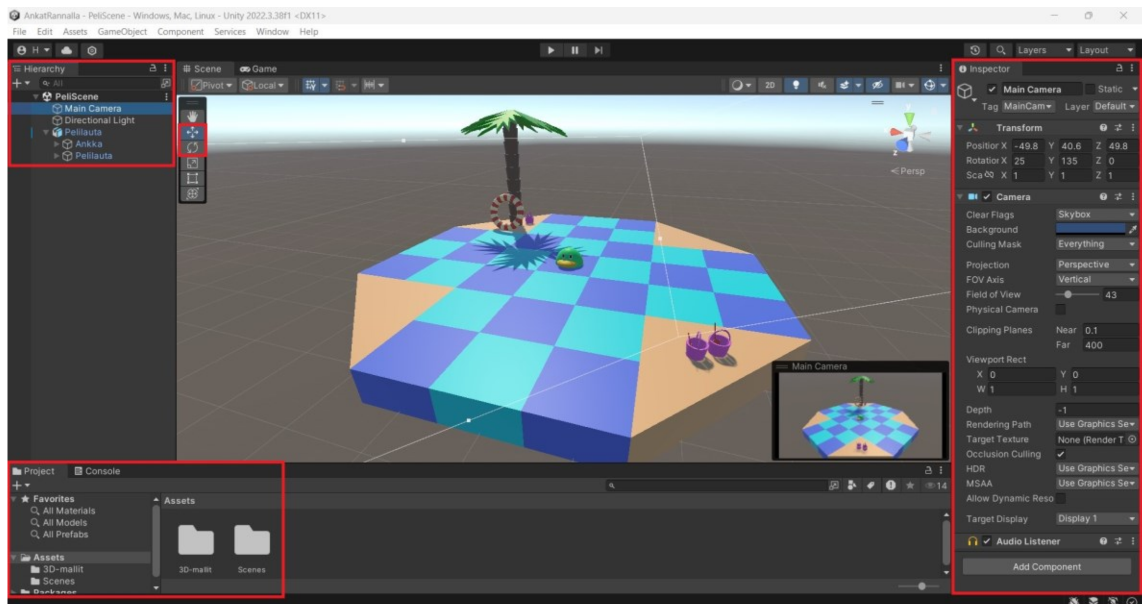
Tässä esiteltiin Unityn käyttöliittymän oletusasettelu. Käyttäjä voi kuitenkin halutessaan järjestellä ikkunoiden paikkoja hyvinkin vapaasti ja mukauttaa käyttöliittymän juuri omiin työskentelytapoihin sopivaksi. [22; 28.]

6.2 Ohjelmakoodin kirjoittaminen ja pelin kokoaminen Unityssä

6.2.1 Alkuvalmistelut

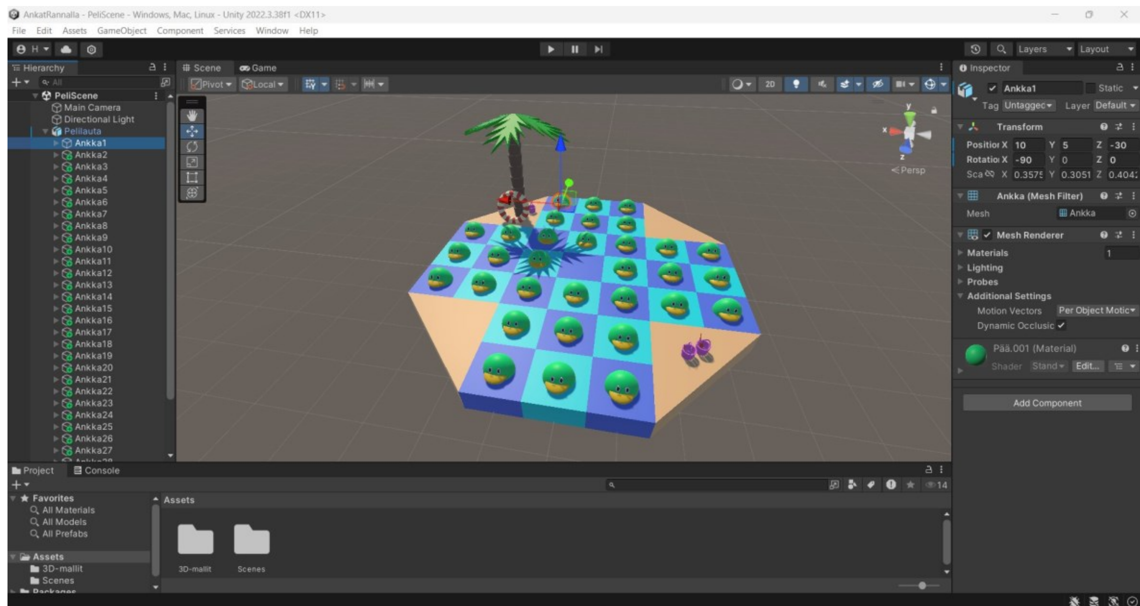
3D-mallit tuotiin Unity-projektin käyttöön tallentamalla Blender-tiedosto kyseisen Unity-projektin Assets-kansioon. Mallien tuominen Unityyn tällä tavoin on erittäin kätevää, sillä jos malleja tarvitsee muokata, avataan vain Assets-kansiossa oleva tiedosto, tehdään tarvittavat muutokset ja tallennetaan. Tehdyt muutokset päivittyvät välittömästi Unityyn. Assets-kansion haluttiin pysyvän selkeänä ja loogisena, joten sen sisään tehtiin oma kansio mallitiedostoja varten.

Blender-tiedoston sisältämät mallit saatiin käyttöön Unityssa raahaamalla mallitiedosto Project-ikkunasta kohtausnäkymään. Blender-tiedoston mukana tulleet kamera ja valo poistettiin valitsemalla ne aktiivisiksi Ctrl-painike pohjassa Hierarchy-ikkunassa ja painamalla sitten Delete-näppäintä. Tämän jälkeen Unity-projektin kamera valittiin aktiiviseksi Hierarchy-ikkunassa ja aseteltiin sopivalle paikalle käyttäen Inspector-ikkunaa ja kohtausnäkymän työkaluja. Kameran asettelun yhteydessä myös pelilaudan sekä ankan sijainti ja kierto tarkistettiin ja säädettiin sopivaksi Inspector-ikkunassa. Lopuksi kohtaus nimettiin uudelleen siirtymällä Project-ikkunan kansiorakenteessa Assets -> Scenes ja klikkaamalla hiiren oikealla painikkeella kansiossa olevaa Scene-tiedostoa. Edellä kuvatuissa toiminnoissa käytetyt ikkunat ja työkalut esitetään kuvassa 29.



Kuva 29. 3D-mallin tuonti ja asettelu.

Pelinappuloita tarvittiin 32, joten ankkaa kopioitiin Hierarchy-ikkunassa Ctrl+C ja Ctrl+V näppäinkomennoilla. Jokainen pelinappula nimettiin uudelleen ja asetettiin oikealle paikalleen Inspector-ikkunassa (kuva 30).

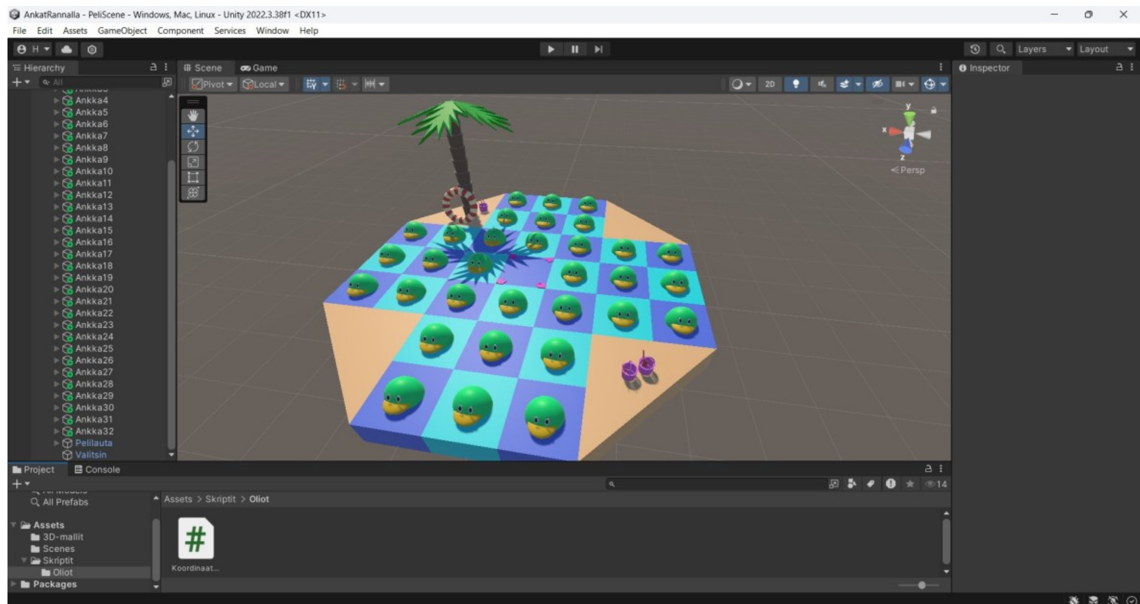


Kuva 30. Pelinappulat aseteltuna pelilaudalle omille paikoilleen.

Kaikki tarvittavat kappaleet oli mallinnettu Blenderissä ja pelilaudan somisteetkin oli siellä aseteltu paikoilleen, joten tämän enempää ei alkuvalmisteluja tarvittu, vaan seuraavaksi voitiin siirtyä jo ohjelmakoodin kirjoittamiseen.

6.2.2 Koordinaatisto

Kooditiedostoillekin tehtiin oma kansio Assets-kansion sisään. Unityssa uusi kansio luodaan Project-ikkunassa siirtymällä kansiorakenteessa haluttuun kohteeseen, klikkaamalla siellä hiiren oikeaa painiketta ja valitsemalla Create -> Folder. Selvyyden vuoksi kooditiedostojen kansion sisään tehtiin vielä erillinen kansio oliokooditiedostoja varten. Uusi kooditiedosto lisättiin samaan tapaan kuin uusi kansiokin eli klikattiin halutun kohdekansion sisällä hiiren oikeaa painiketta ja valittiin Create -> C# Script (kuva 31).



Kuva 31. Unity-projektin kansiorakenne ja uusi kooditiedosto.

Lautapasianssissa pelinappuloita siirretään pelilaudalla tiettyjen sääntöjen mukaisesti, joten pelin oikeanlaisen toiminnan kannalta ohjelmakoodissa oli välttämätöntä määritellä pelialueen rajat ja saada selville, missä koordinaateissa pelinappulat ja valitsin milloinkin ovat. Ohjelmointi aloitettiin poistamalla uudessa kooditiedostossa valmiina olevat Start- ja Update-funktiot. Lisäksi poistettiin MonoBehaviour-luokan perintä ja kirjoitettiin Koordinaatisto-luokalle muodostin, kuten esimerkkikoodissa 2 esitetään.

```
using UnityEngine;

namespace Koordinaatit
{
	public class Koordinaatisto
	{
		public Koordinaatisto()
		{
		}
	}
}
```

Esimerkkikoodi 2. Koordinaatisto-luokka ja muodostin.

Pelialue määriteltiin tallentamalla pelilaudan ruutujen koordinaatit kaksiulotteiseen taulukkoon. Tieto siitä, ollaanko pelialueella vai sen ulkopuolella, saatiin

metodilla, joka vertaa parametreina saamiaan koordinaatteja taulukon koordinaatteihin. Mikäli parametreina saadut koordinaatit löytyvät taulukosta eli ovat pelialueella, metodi palauttaa arvon true ja muussa tapauksessa arvon false (esimerkkikoodi 3).

```
private int[,] ruutujenKoordinaatitXZ = {
     {10, -30}, {0, -30}, {-10, -30},
     {10, -20}, {0, -20}, {-10, -20},
     {30, -10}, {20, -10}, {10, -10}, {0, -10}, {-10, -10},
          {-20, -10}, {-30, -10},
     {30, 0}, {20, 0}, {10, 0}, {0, 0}, {-10, 0}, {-20, 0},
          {-30, 0},
     {30, 10}, {20, 10}, {10, 10}, {0, 10}, {-10, 10},
          {-20, 10}, {-30, 10},
     {10, 20}, {0, 20}, {-10, 20},
     {10, 30}, {0, 30}, {-10, 30}
};

public bool OnkoKoordinaatitPelialueella(int x, int z)
{
     for (int i = 0; i < ruutujenKoordinaatitXZ.Length/2; i++)
     {
          if (x == ruutujenKoordinaatitXZ[i, 0] && z == ruutujenKoordi-
          naatitXZ[i, 1])
          {
               return true;
          }
     }
     return false;
}
```

Esimerkkikoodi 3. Kaksiulotteisen taulukon ja sen arvot läpikäyvän metodin määrittely.

Lopuksi luokalle kirjoitettiin metodit, joilla saadaan selville minkä tahansa peliobjektin sijainti. Esimerkkikoodissa 4 esitetään metodi, joka palauttaa parametrina saamansa peliobjektin x-koordinaatin.

```
public int SijaintiX(GameObject peliobjekti)
{
     return (int)peliobjekti.GetComponent<Transform>().position.x;
}
```

Esimerkkikoodi 4. Metodi, joka palauttaa minkä tahansa peliobjektin x-koordinaatin.

Vastaavat metodit kirjoitettiin myös peliobjektin y- ja z-koordinaattien selvittämiseen. Mitään monimutkaista toimintaa Koordinaatisto-luokkaan ei siis tarvittu,

vaan tieto pelialueesta ja mahdollisuus selvittää peliobjektin sijainti olivat täysin riittävä pohja sille, että pelin varsinainen toiminnallisuus voitiin toteuttaa.

6.2.3 Valitsimella liikkuminen

Ohjelmointia aloittaessa huomattiin, että grafiikan toteutusvaiheessa oli unohtunut mallintaa valitsin, jolla pelaaja valitsee siirrettävän pelinappulan ja paikan, mihin kyseinen pelinappula siirretään. Koska 3D-mallit oli tuotu Unityyn tallentamalla Blender-tiedosto Assets-kansioon, valitsimen lisääminen oli hyvin helppoa. Avattiin vain Assets-kansiossa oleva mallitiedosto, tehtiin valitsin ja tallennettiin. Valitsin ilmestyi välittömästi Unityyn. Valitsimen lisäksi mallinnettiin myös korostuselementti, jolla osoitetaan pelissä valittuna oleva ankka.

Valitsimen liikkumisen hallinnalle tehtiin oma kooditiedosto samalla tavalla kuin pelin koordinaatistollekin. Kooditiedostosta poistettiin siellä valmiina olevat funktiot sekä MonoBehaviour-luokan perintä ja määriteltiin luokalle muodostin. Jotta valitsin pysyisi pelialueella, eikä sillä voisi lähteä pelilaudan ulkopuolelle, tarvittiin Koordinaatisto-luokasta tietoa pelialueesta ja oli myös voitava tarkastaa, missä koordinaateissa valitsin on. Tätä varten ValitsimenHallinta-luokkaan luotiin Koordinaatisto-olio, kuten esimerkkikoodissa 5 esitetään.

```
using UnityEngine;
using Koordinaatit;

namespace ValitsimenHallinnointi
{
     public class ValitsimenHallinta
     {
          private Koordinaatisto xyz = new Koordinaatisto();

          public ValitsimenHallinta()
          {
          }
     }
}
```

Esimerkkikoodi 5. ValitsimenHallinta-luokka, muodostin ja Koordinaatisto-olio.

Valitsimella voidaan liikkua ylös, alas, vasemmalle ja oikealle, joten luokkaan kirjoitettiin neljä metodia niin, että kukin pitää huolta yhdestä

liikkumissuunnasta. Metodi tarkastaa Koordinaatisto-luokkaa käyttäen, pysyykö valitsin metodin vastuusuuntaan siirtyessään edelleen pelialueella. Mikäli pysyy, valitsin siirtyy kyseiseen suuntaan (esimerkkikoodi 6).

```
public void ValitsinVasemmalle(GameObject valitsin)
{
     if (xyz.OnkoKoordinaatitPelialueella(xyz.SijaintiX(valitsin) + 10,
     xyz.SijaintiZ(valitsin)))
     {
          valitsin.transform.position = new Vector3(xyz.SijaintiX
          (valitsin) + 10, xyz.SijaintiY(valitsin), xyz.SijaintiZ
          (valitsin));
     }
}
```

Esimerkkikoodi 6. Metodi, joka siirtää valitsinta vasemmalle pelialueella.

Myös muille liikkumissuunnille kirjoitettiin metodit samalla tavalla.

6.2.4 Pelinaikainen toiminta ja pelinappuloiden hallinta

Pelin ohjaaminen haluttiin toteuttaa niin, että valitsinta liikutetaan nuolinäppäimillä, ankka valitaan välilyönnillä ja paikka, jonne ankka siirretään, vahvistetaan Enter-painikkeella. Tarvittiin siis kooditiedosto, joka tarkkailee, mitä painiketta pelaaja painaa ja määrittelee, mitä painikkeen painamisesta seuraa. Siksi luotiin uusi kooditiedosto Ankat. Koska Ankat-luokassa haluttiin tarkkailla pelinaikaisia tapahtumia, eli painaako pelaaja jotain painiketta, kooditiedostoon jätettiin siellä valmiina olevat Start- ja Update-funktiot sekä MonoBehaviour-luokan perintä. Start-funktio suoritetaan yhden kerran, kun kohtaus käynnistyy. Update-funktio puolestaan on toiminnassa niin kauan kuin kohtauskin on käynnissä.

Esimerkkikoodissa 7 esitetään, kuinka Update-funktiossa voidaan tarkastaa, onko pelaaja painanut vasemman nuolinäppäimen alas. Jos kyseinen painike on painettu alas, kutsutaan ValitsimenHallinta-luokasta luodun olion kautta metodia, joka siirtää valitsinta pelialueella vasemmalle.

```
void Update()
{
     if (Input.GetKeyDown(KeyCode.LeftArrow))
     {
          hallintaValitsin.ValitsinVasemmalle(valitsin);
     }
}
```

Esimerkkikoodi 7. If-lause, joka tarkastaa, onko vasen nuolinäppäin painettu alas.

Myös muille nuolinäppäimille kirjoitettiin omat if-lauseet Update-funktion sisään. Tämän jälkeen pohdittiin, kuinka pelinappulan valitseminen ja siirtäminen toteutetaan, jotta ainoastaan sääntöjen mukaiset siirrot ovat mahdollisia. Ongelmaa pilkottiin pienemmäksi kirjoittamalla ylös kaikki seikat, jotka vaikuttavat siihen, onko siirto sääntöjen mukainen:

- Onko valitussa ruudussa siirrettävä pelinappula?
- Onko siirto yhden ruudun yli vasemmalle, oikealle, ylös tai alas?
- Onko kohderuutu vapaa?
- Onko ylitettävässä ruudussa pelinappula?

Huomattiin, että kysymyksistä saa helposti kirjoitettua neljä metodia, joilla valvotaan, onko siirto sääntöjen mukainen. Kukin metodi vastaa yhteen kysymykseen asettamalla muuttujansa arvoksi joko true tai false. Mikäli jokainen metodi asettaa muuttujalleen arvon true, on siirto sääntöjen mukainen ja se voidaan tehdä. Metodeja varten luotiin uusi kooditiedosto AnkkojenHallinta. Sinne kirjoitettiin luokan muodostin ja poistettiin Start- ja Update-funktiot sekä MonoBehaviour-luokan perintä.

Ennen kuin metodeja voitiin kirjoittaa, oli määriteltävä muutama muuttuja. Tarvittiin kokonaislukumuuttujat, joihin tallennetaan mistä koordinaateista, minkä koordinaattien yli ja mihin koordinaatteihin siirto tehdään. Tarvittiin myös muuttujat, joihin tallennetaan siirrettävä pelinappula, ylitettävä pelinappula ja valitun pelinappulan korostuselementti. Lisäksi oli määriteltävä kaksi listaa. Ensimmäinen lista sisältää pysyvästi kaikki pelinappulat. Toisen listan sisältö puolestaan päivittyy koko pelin ajan, sillä se sisältää ainoastaan ne pelinappulat, joita ei ole

poistettu pelilaudalta. Lopuksi oli määriteltävä neljä totuusmuuttujaa, yksi kutakin metodia varten.

Ensimmäinen metodi, OnkoRuudussaSiirrettavaAnkka, tarkastaa nimensä mukaisesti, onko valitussa ruudussa ankka, jota siirtää. Jos on, muuttujiin otetaan talteen sekä lähtöruudun koordinaatit että tieto siitä, mikä pelinappula kyseisessä ruudussa on. Valittu ruutu korostetaan ja metodi asettaa oman totuusmuuttujansa arvoksi true.

Tämän jälkeen vuorossa on SiirrytaankoYhdenRuudunYli-metodi, joka tutkii, tehdäänkö siirto yhden ruudun yli vasemmalle, oikealle, ylös tai alas. Mikäli tehdään, ylitettävän ruudun koordinaatit tallennetaan muuttujiin. Metodi myös asettaa totuusmuuttujansa arvoksi true.

Kolmas metodi on OnkoKohderuutuVapaa. Se tarkastaa, onko kohderuutu tyhjä. Jos on, kohderuudun koordinaatit tallennetaan muuttujiin ja metodi asettaa totuusmuuttujansa arvoksi true.

Viimeisenä kirjoitettiin metodi OnkoYlitettavassaRuudussaAnkka. Se tutkii, onko ylitettävä ruutu tyhjä vai onko siinä ankka. Mikäli ruudussa on ankka, tallennetaan muuttujaan tieto siitä, mikä pelinappula kyseisessä ruudussa on ja metodi myös asettaa oman totuusmuuttujansa arvoksi true.

Edellä kuvatut metodit vain tutkivat, onko siirto sääntöjen mukainen. Tarvittiin siis vielä metodit, jotka huolehtivat pelinappuloiden siirtämisestä, mikäli siirto voidaan tehdä. Esimerkkikoodissa 8 esitetään pelinappuloiden siirtämiseen tarkoitettujen metodien rungot. Varsinainen toiminnallisuus metodien sisään toteutettiin hieman myöhemmin.

```
public void AnkkaSiirtyy()
{
     if (ruudussaSiirrettavaAnkka == true && siirtoYhdenRuudunYli ==
     true && kohderuutuVapaa == true && ylitettavassaRuudussaAnkka ==
     true)
     {
          //Tähän toiminnallisuus
     }
}

public void AnkkaSukeltaa()
{
     if (ruudussaSiirrettavaAnkka == true && siirtoYhdenRuudunYli ==
     true && kohderuutuVapaa == true && ylitettavassaRuudussaAnkka ==
     true)
     {
          //Tähän toiminnallisuus
     }
}
```

Esimerkkikoodi 8. Pelinappuloiden siirtämisestä vastaavat metodit.

Metodien kirjoittamisen jälkeen siirryttiin takaisin Ankat-luokkaan ja luotiin sinne olio AnkkojenHallinta-luokasta. Start-funktiossa tuotiin ohjelmakoodin käyttöön tarvittavat peliobjektit eli pelinappulat, valitsin ja valitun pelinappulan korostus-elementti (esimerkkikoodi 9).

```
private string nimi = "";
public GameObject valitsin = null;

void Start()
{
     for (int i = 1; i < 33; i++)
     {
          nimi = "Ankka"+i;
          hallintaAnkat.ankkalista.Add(GameObject.Find(nimi));
     }

     this.valitsin = GameObject.Find("Valitsin");
     hallintaAnkat.korostus = GameObject.Find("Korostus");
     hallintaAnkat.siirrettavaAnkka = GameObject.Find("AloitusAnkka");
     hallintaAnkat.ylitettavaAnkka = GameObject.Find("AloitusAnkka");
}
```

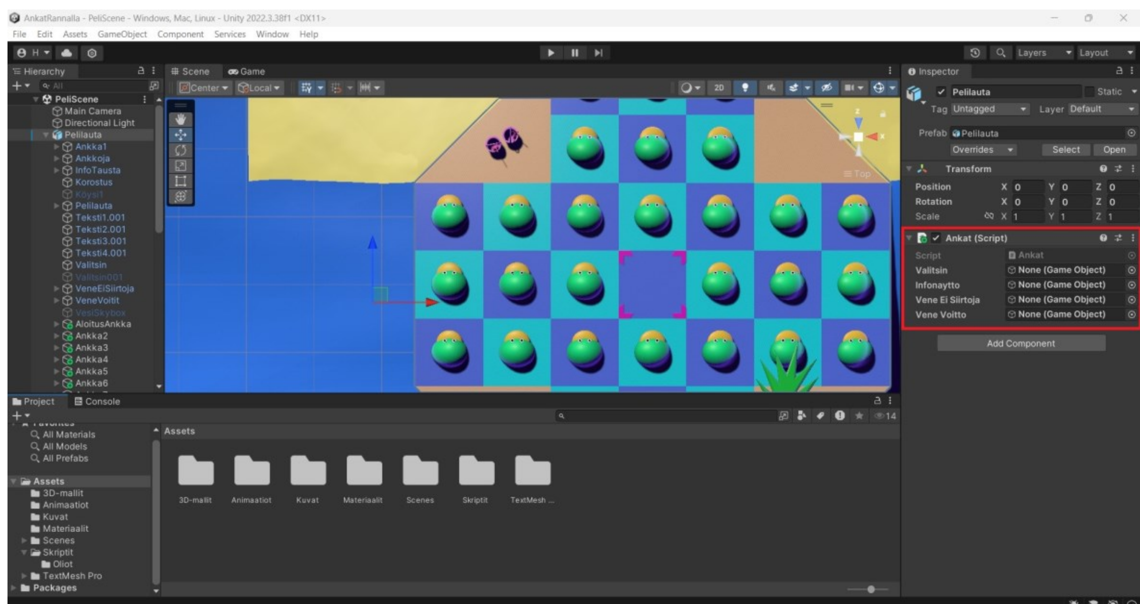
Esimerkkikoodi 9. Peliobjektien hakeminen ohjelmakoodin käyttöön.

Ankat-luokan Update-funktioon lisättiin äsken kirjoitettujen metodien kutsut if-lauseiden sisään, kuten esimerkkikoodissa 10 esitetään.

```
if (Input.GetKeyDown(KeyCode.Space))
{
     hallintaAnkat.OnkoRuudussaSiirrettavaAnkka(valitsin);
}

if (Input.GetKeyDown(KeyCode.Return))
{
     hallintaAnkat.SiirrytaankoYhdenRuudunYli(valitsin);
     hallintaAnkat.OnkoKohderuutuVapaa(valitsin);
     hallintaAnkat.OnkoYlitettavassaRuudussaAnkka();
     hallintaAnkat.AnkkaSiirtyy();
     hallintaAnkat.AnkkaSukeltaa();
}
```

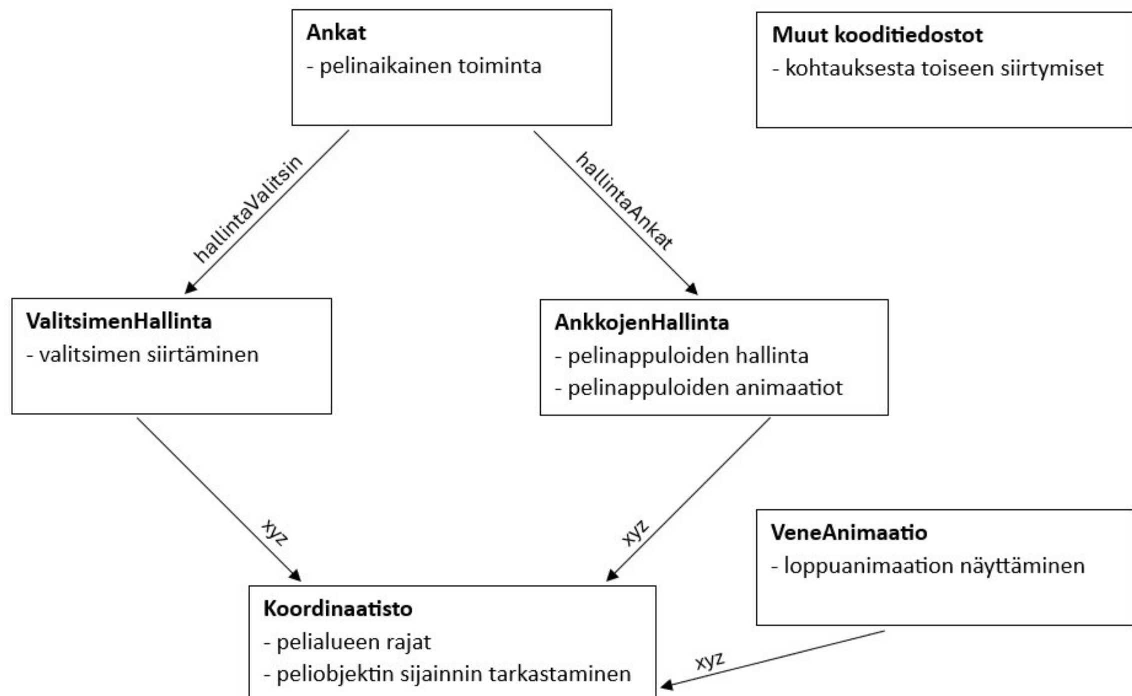
Esimerkkikoodi 10. Siirron oikeellisuuden tarkastaminen ja pelinappuloiden siirtäminen pelin aikana.

Ankat-luokassa hallittiin pelinaikaisia tapahtumia Start- ja Update-funktioissa, joten luokan kooditiedosto oli sisällytettävä kohtaukseen Unityssa. Muuten tiedostoa ei suoritettaisi. Kooditiedosto sisällytetään kohtaukseen liittämällä se kohtauksessa mukana olevan peliobjektin komponentiksi. Koska Ankat-luokassa ei käsitelty yhtäkään peliobjektia suoralla this-viittauksella, tiedosto voitiin liittää minkä tahansa kohtauksessa mukana olevan peliobjektin komponentiksi. Komponentiksi liittäminen kävi helposti raahaamalla kooditiedosto Project-ikkunasta Hierarchy-ikkunaan halutun peliobjektin päälle. Kuvassa 32 esitetään peliobjektin komponentiksi liitetty kooditiedosto.



Kuva 32. Kooditiedosto peliobjektin komponenttina.

Ohjelmakoodia varten luotiin useita eri kooditiedostoja, koska jokaiselle kooditiedostolle haluttiin rajata oma selkeä vastuualueensa (kuva 33). Kaiken pohjana toimii Koordinaatisto-luokka. ValitsimenHallinta- ja AnkkojenHallinta-luokat luovat siitä oliot ja kutsuvat Koordinaatiston metodeja. Vastaavasti Ankat-luokka luo ValitsimenHallinta- ja AnkkojenHallinta-luokista oliot ja kutsuu näiden luokkien metodeja.



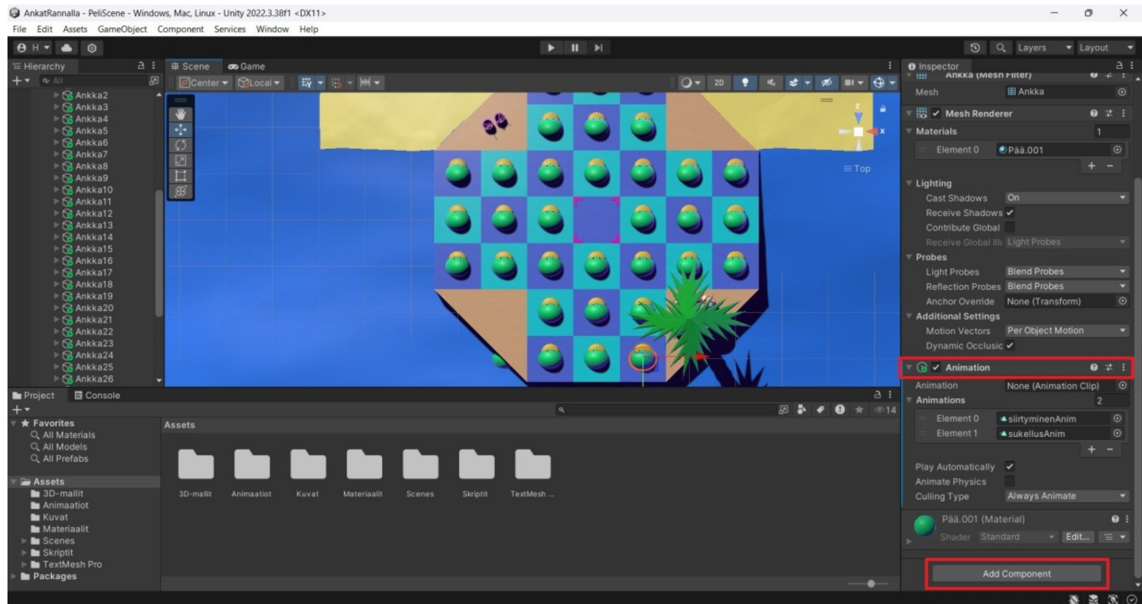
Kuva 33. Pelin taustalla toimivat kooditiedostot ja niiden väliset suhteet.

Pelin varsinainen toiminnallisuus oli nyt toteutettu, joten seuraavaksi voitiin siirtyä animointiin.

6.2.5 Animointi

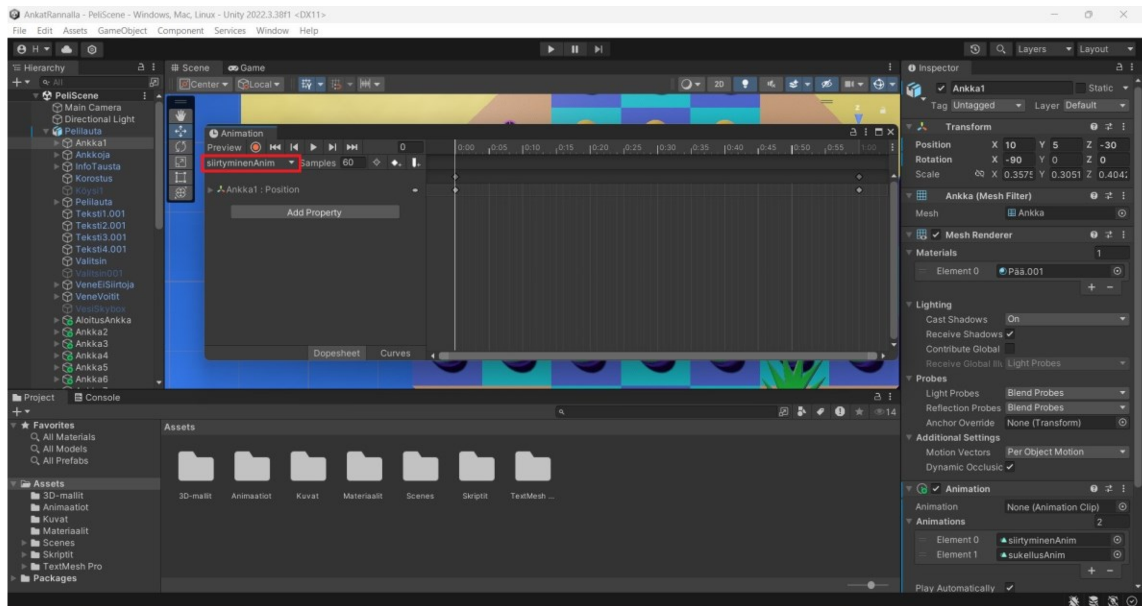
Ankkojen liikkumiseen tarvittiin kaksi erilaista animaatiota. Ensimmäisessä animaatiossa ankka siirtyy ruudusta toiseen ja toisessa animaatiossa ankka sukeltaa. Kuten kaikille muillekin peliin liittyville tiedostoille, myös animaatioille tehtiin oma kansio Assets-kansioon.

Jotta peliobjekti voidaan animoida, sillä on oltava Animation-komponentti. Siksi kaikki pelinappulat valittiin Shift- ja Ctrl-painikkeiden avulla aktiivisiksi Hierarchy-ikkunassa, klikattiin Inspector-ikkunassa Add Component -painiketta ja etsittiin listasta Animation (kuva 34).



Kuva 34. Add Component -painike, jolla peliobjektille lisättiin Animation-komponentti.

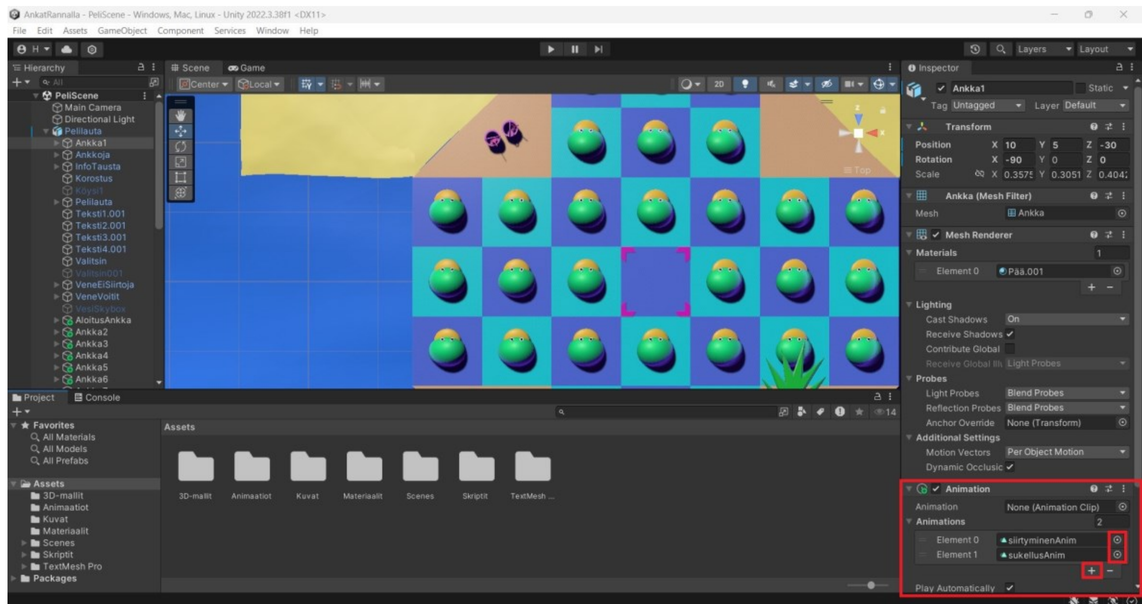
Seuraavaksi avattiin Animation-ikkuna valitsemalla valikkorivillä Window -> Animation -> Animation. Animaatioklippi luotiin valitsemalla ensimmäinen pelinappula aktiiviseksi Hierarchy-ikkunassa ja painamalla Animation-ikkunaan ilmestynyttä Create-painiketta. Animaatioklipille annettiin nimi ja se tallennettiin kansioon, joka niitä varten oli tehty. Toinen animaatioklippi lisättiin klikkaamalla alaspäin osoittavaa nuolta ja valitsemalla Create New Clip (kuva 35).



Kuva 35. Toisen animaatioklipin lisääminen Animation-ikkunassa.

Kun toinenkin animaatioklippi oli nimetty ja tallennettu, voitiin Animation-ikkuna sulkea. Valitulle pelinappulalle lisätyt kaksi animaatioklippiä näkyivät Inspector-ikkunassa pelinappulan Animation-komponentissa. Muille pelinappuloille animaatioklipit lisättiin valitsemalla kyseiset pelinappulat aktiivisiksi Hierarchy-ikkunassa ja klikkaamalla Inspector-ikkunassa Animation-komponentissa Animations-otsikon alla olevaa pluspainiketta kaksi kertaa. Lisättyihin kenttiin voitiin valita animaatioklipit klikkaamalla kentän oikeassa reunassa olevaa pampulaa.

Pelinappulalle lisätyt animaatioklipit ja niiden lisäämiseen käytetyt painikkeet esitetään kuvassa 36.



Kuva 36. Animaatioklippien lisääminen peliobjektille Inspector-ikkunassa.

Animaatioklippien sisältö lisättiin ohjelmakoodissa AnkkojenHallinta-luokassa AnkkaSiirtyy- ja AnkkaSukeltaa-metodeilla. Näiden metodien rungot kirjoitettiin kooditiedostoon jo valmiiksi edellisen luvun lopussa.

Esimerkkikoodissa 11 esitetään valmis AnkkaSukeltaa-metodi ja sen käyttämät muuttujat. Jos siirto voidaan tehdä, haetaan sukeltavaan ankkaan liitetty animaatioklippi. Koska animaatiota käytetään Animation-komponentin kautta Animator-komponentin sijan, on AnimationClip-luokan legacy-muuttujalle asetettava arvo true. Seuraavaksi varmistetaan, että animaatioklippi on tyhjä ja asetetaan klippiin haluttu siirtymä sekä sen kesto. Lopuksi animaatio käynnistetään.

```
private AnimationClip sukellus = null;
private AnimationCurve sukellusCurveX = null;
private AnimationCurve sukellusCurveY = null;
private AnimationCurve sukellusCurveZ = null;

public void AnkkaSukeltaa()
{
     if (ruudussaSiirrettavaAnkka == true && siirtoYhdenRuudunYli ==
     true && kohderuutuVapaa == true && ylitettavassaRuudussaAnkka ==
     true)
     {
          this.sukellus = ylitettavaAnkka.GetComponent<Animation>().
          GetClip("sukellusAnim");

          sukellus.legacy = true;
          sukellus.ClearCurves();

          sukellusCurveX = AnimationCurve.Linear(0, xyz.SijaintiX(yli-
          tettavaAnkka), 1, xyz.SijaintiX(ylitettavaAnkka));

          sukellusCurveY = AnimationCurve.Linear(0, xyz.SijaintiY(yli-
          tettavaAnkka), 1, 0);

          sukellusCurveZ = AnimationCurve.Linear(0, xyz.SijaintiZ(yli-
          tettavaAnkka), 1, xyz.SijaintiZ(ylitettavaAnkka));

          sukellus.SetCurve("", typeof(Transform), "localPosition.x",
          sukellusCurveX);

          sukellus.SetCurve("", typeof(Transform), "localPosition.y",
          sukellusCurveY);

          sukellus.SetCurve("", typeof(Transform), "localPosition.z",
          sukellusCurveZ);

          ylitettavaAnkka.GetComponent<Animation>().Play("sukel-
          lusAnim");
     }
}
```

Esimerkkikoodi 11. Metodi, joka kirjoittaa ja näyttää ankan sukellusanimaation.

Samaan tapaan kirjoitettiin ankan ruudusta toiseen siirtävä animaatio AnkkaSiirtyy-metodin sisälle.

Peliin haluttiin tuoda lisää visuaalisuutta ja korostaa mielikuvaa sukeltavista ja uivista ankoista, joten sukeltavan ankan ympärille lisättiin vesirenkaat ja uivan ankan ympärille kuplia. Kuten muutkin pelin elementit, myös vesirenkaat ja kuplat mallinnettiin Blenderissä ja tuotiin Unityyn tallentamalla Blender-tiedosto Assets-kansiossa olevaan 3D-mallit-kansioon. Vesirenkaiden ja kuplien animointi toteutettiin täysin samoja vaiheita noudattaen kuin ankkojenkin animointi, eli

lisättiin Unityssa peliobjekteille Animation-komponentit, luotiin animaatioklipit ja liitettiin ne peliobjekteihin. Lopuksi AnkkojenHallinta-luokassa määriteltiin animaatiot kirjoittavat metodit.

6.2.6 Viimeistely

Näin oli toteutettu toimiva lautapasianssipeli. Pelaaja voi valita siirrettävän pelinappulan ja paikan, jonne pelinappulan haluaa siirtyvän. Jos siirto on sääntöjen mukainen, pelinappulat siirtyvät sen mukaisesti. Vaikka itse peli olikin nyt toimiva, puuttui siitä kuitenkin vielä muutamia oleellisia asioita.

AnkkojenHallinta-luokkaan kirjoitettiin metodit AnimaatioKaynnissa ja OnkoSiirtoja. AnimaatioKaynnissa-metodi estää, ettei pelaaja voi valita ja siirtää uutta pelinappulaa edellisen siirron ollessa vielä käynnissä. OnkoSiirtoja-metodi tarkkailee nimensä mukaisesti, onko pelissä vielä mahdollista tehdä sääntöjen mukaisia siirtoja. Jos ei ole, näytetään loppuanimaatio, eikä pelaaja voi enää siirtää valitsinta tai valita ankkaa. Näytettävä loppuanimaatio riippuu siitä, ovatko siirrot loppuneet kesken vai onko peli pelattu onnistuneesti läpi.

Pelille tehtiin myös etusivu. Uusi kohtaus etusivun rakentamista varten lisättiin Project-ikkunassa siirtymällä kansiorakenteessa Assets -> Scenes, klikkaamalla siellä hiiren oikeaa näppäintä ja valitsemalla Create -> Scene. Etusivulta pelaaja pääsee pelin ohjeisiin, ja etusivulla on myös painike, jolla peli aloitetaan. Pelin olisi tietenkin voinut toteuttaa myös niin, että se avautuu suoraan pelinäkymään ja siellä on mahdollista lukea pelin ohjeet. Etusivu kuitenkin helpottaa pelin jatkokehitystä. Tällä hetkellä peliä voidaan pelata ainoastaan englantilaisella pelilaudalla, mutta jos myöhemmin esimerkiksi haluttaisiin lisätä peliin ranskalainen pelilauta, on päivitys helpompi tehdä, kun on jo olemassa etusivu, jonne vain lisätään valintamahdollisuus, minkälaisella pelilaudalla peli halutaan pelata. Esimerkkikoodissa 12 esitetään ohjelmakoodi, joka tarvittiin etusivun painikkeeseen, jota painamalla pelaaja voi aloittaa pelin.

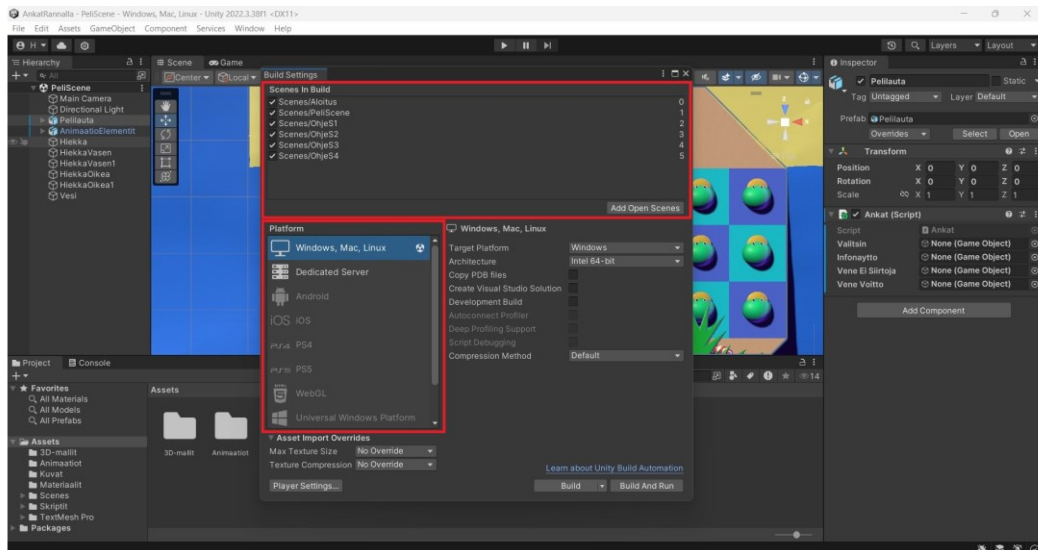
```
using UnityEngine;
using UnityEngine.SceneManagement;

public class Aloita : MonoBehaviour
{
     private void OnMouseOver()
     {
         if (Input.GetMouseButtonDown(0))
         {
             SceneManager.LoadScene(1);
         }
     }
}
```

Esimerkkikoodi 12. Kohtauksesta toiseen siirtyminen hiiren klikkauksella.

Jotta painike toimisi, oli painikkeena toimivalle peliobjektille lisättävä Collider-komponentti. Myös kooditiedosto tuli liittää painikkeen komponentiksi. Ladattava kohtaus eli kohtaus, johon painikkeella haluttiin siirtyä, ilmoitettiin ohjelmakoodissa kohtauksen järjestysnumerolla. Järjestysnumero katsottiin valitsemalla valikkorivillä File -> Build Settings. Mikäli haluttu kohtaus ei ole listalla, kohtaus voidaan tarvittaessa avata Scenes-kansiosta ja painaa tämän jälkeen Add Open Scenes -painiketta. Kohtauksien järjestystä voidaan muuttaa hiirellä raahaamalla.

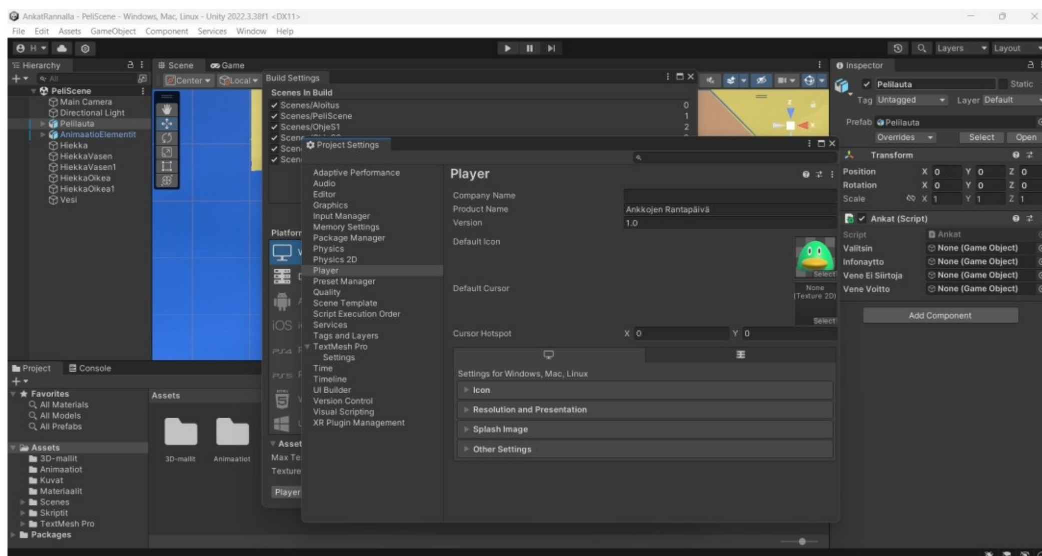
Lopuksi pelinäkymä viimeisteltiin lisäämällä pelilaudan ympärille vesi- ja hiekkaelementtejä. Pelinäkymään myös lisättiin tieto, montako pelinappulaa pelilaudalla on jäljellä sekä painikkeet etusivulle palaamista ja pelin uudelleen käynnistämistä varten.

Kun kaikki oli valmista, peli voitiin kääntää sovellukseksi valitsemalla valikkorivillä File -> Build Settings. Ensimmäisenä oli tarkistettava, että oikea alusta on valittuna ja kaikki halutut kohtaukset tulevat mukaan (kuva 37).



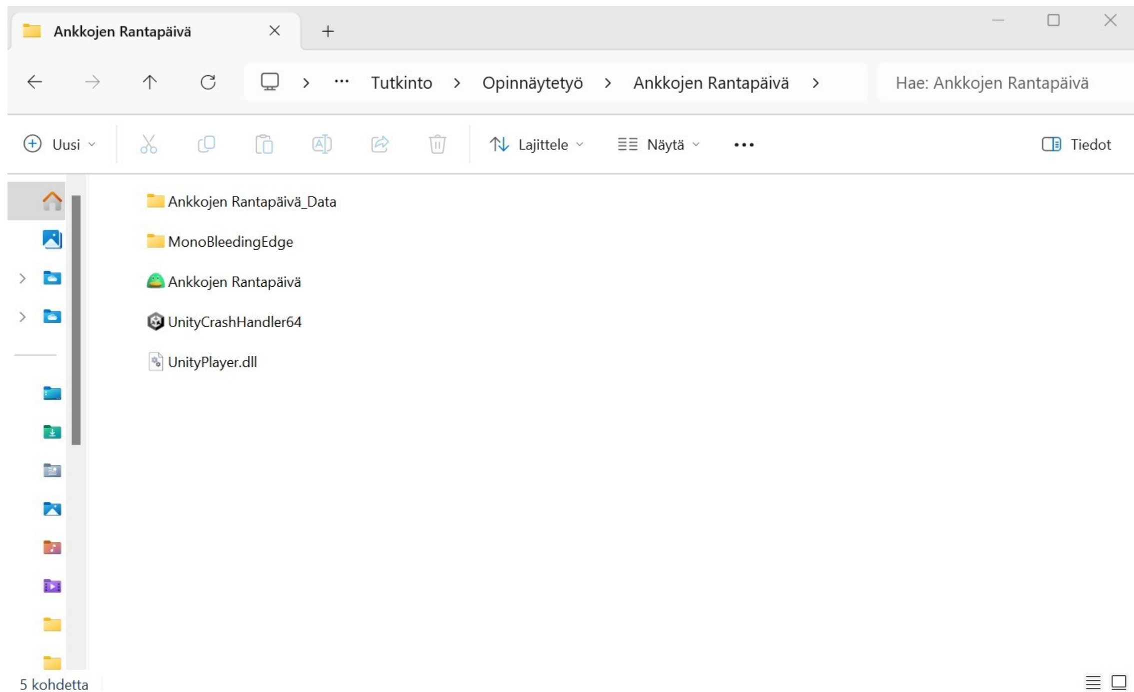
Kuva 37. Alustan ja peliin tulevien kohtausten valinta.

Lisää asetuksia avattiin Player Settings -painikkeella. Tässä ikkunassa annettiin pelin nimi sekä versionumero ja asetettiin sovelluksen kuvake. Resolution and Presentation -valikossa koko näytön tilaksi valittiin ikkunoitu. Näin pelille saatiin tietokoneohjelmista tuttu ikkunointi ja yläpalkki, jolloin pelaaja voi yläpalkin oikeassa reunassa olevilla painikkeilla pienentää tai sulkea sovelluksen. Splash Image -valikossa asetettiin pelille logo ja säädettiin alkunäytön asetuksia. Player Settings -painikkeesta avautuva ikkuna esitetään kuvassa 38.



Kuva 38. Player Settings -painikkeesta avautuva asetusikkuna.

Kun asetukset oli säädetty halutunlaiseksi, ikkuna voitiin sulkea. Tämän jälkeen vain painettiin Build-painiketta Build Settings-ikkunassa ja valittiin, mihin kansioon peli halutaan. Peliä varten kannattaa tehdä oma kansio, joka on nimetty pelin mukaisesti. Tällöin kaikki peliin kuuluva pysyy näppärästi yhtenä kokonaisuutena, eikä mene muiden tiedostojen kanssa sekaisin (kuva 39).



Kuva 39. Ankkojen Rantapäivä -pelin kansio.

Nyt peli voitiin käynnistää tuplaklikkaamalla sovelluksen kuvaketta. Kuvakaappauksia valmiista pelistä esitetään liitteessä 1.

7 Yhteenveto

Tässä insinöörityössä tutustuttiin pelikehitykseen. Aluksi perehdyttiin pelikehityksen vaiheisiin teoriatasolla, jonka jälkeen toteutettiin näiden vaiheiden mukaisesti olemassa olevasta lautapasianssipelistä digitaalinen versio. Tavoitteena oli tehdä pelistä toimiva ja visuaalisesti hyvännäköinen. Lisäksi tavoitteena oli laatia raportti, joka kuvaa pelikehityksen vaiheet selkeästi ja ymmärrettävästi niin, että aloitteleva pelisuunnittelija voi hyödyntää työtä tietolähteenä ensimmäisissä pienissä peliprojekteissaan. Nämä tavoitteet saavutettiin suunnittelemalla pelin ulkoasu ja ohjelmakoodi huolellisesti, kiinnittämällä huomiota selkeään ja ymmärrettävään tekstiin raportissa sekä havainnollistamalla työvaiheita kuvien avulla.

Pelin visuaalisessa toteutuksessa suurimpana haasteena oli erottaa niin sanotut toissijaiset elementit, eli ne, joiden yksityiskohtien suunnitteluun ei ole järkevää paneutua tai käyttää runsaasti aikaa. Esimerkiksi pelilaudan reunalle somisteeksi mallinnetut hiekkaämpärit olivat sen verran pieniä, ettei niiden yksityiskohdat erotu, joten tällaisten elementtien yksityiskohtien toteuttamiseen ei myöskään ollut järkevää käyttää paljon aikaa.

Teoriaosassa haasteena oli materiaalin runsaus, sillä pelikehitys on hyvin laaja aihe ja sitä käsittelevää aineistoa on olemassa paljon. Oli haastavaa pohtia, mitä työhön otetaan mukaan ja mitä ei. Pelikehitysprosessi haluttiin esittää teorian lisäksi myös käytännössä, joten tietoa jouduttiin karsimaan ja tiivistämään, eikä näin ollen ollut mahdollista sukeltaa teoriaan niin syvällisesti kuin mielenkiintoa aiheeseen olisi ollut. Aihe rajatiin itse suunnittelu- ja toteutusprosessiin, eikä tässä työssä sen vuoksi käsitelty pelituotantoon liittyvää mahdollista rahoitusta tai pelin markkinointia ja julkaisua.

Koska kyseessä oli peliprojekti, jatkokehitysmahdollisuuksia jäi tietenkin paljon. Peliin voidaan toteuttaa lisää erilaisia pelilautoja ja teemaan voidaan tuoda mukaan esimerkiksi aina sen hetkiseen vuodenaikaan sopivia elementtejä. Peliin on mahdollista lisätä myös erilaisia saavutuksia. Lisäksi voidaan toteuttaa

vaikkapa pistetaulukko, joka antaa mahdollisuuden vertailla tuloksia kavereiden kanssa, kuten kuka on ratkaissut pelin nopeimmin. Jatkokehitysmahdollisuuksissa on siis käytännössä vain mielikuvitus rajana.

Tämä insinöörityö oli erittäin mielenkiintoinen projekti. Sen lopputuloksena saatiin selkeä raportti pelikehityksen vaiheista ja ennen kaikkea lautapasianssista toteutettiin toimiva ja hauska digitaalinen versio, jonka parissa viihtyvät niin harrastajat kuin satunnaisetkin pelaajat. Projekti syvensi ymmärrystäni pelikehityksestä ja vahvisti Blender- ja Unity-ohjelmistojen osaamistani. Pelikehitystä käsittelevään kirjallisuuteen perehtyminen antoi hyvän käsityksen siitä, kuinka haastava ala on kyseessä. Erityisesti mieleeni jäi Kemppaisen [2, s. 87] ajatus siitä, etteivät eurot ja pelaajien määrä ole ainoat mittarit, joilla pelin arvoa voidaan mitata:

> Jos pelin tekijä saa työstään sisältöä elämäänsä, on peli arvokas.
> Jos peli tuottaa iloa tai merkitystä myös muille, mikäs sen parempi.

Mielestäni nämä lauseet antavat todella hienon näkökulman pelikehitykseen. Olisi mahtavaa, jos etenkin pienien peliprojektien parissa työskentelevät itsenäiset pelisuunnittelijat muistaisivat aina arvioida työtään myös tästä Kemppaisen esittämästä näkökulmasta.

Lähteet

1 Whiter, Barb. 2002. Kodin suuri pelikirja. Jyväskylä: Gummerus.

2 Kemppainen, Jaakko. 2019. Pelisuunnittelijan peruskirja. Rajamäki: Aviador Kustannus.

3 Nacke, Lennart E; Bateman, Chris & Mandryk, Regan L. 2011. BrainHex: Preliminary Results from a Neurobiological Gamer Typology Survey. Verkkoaineisto. <https://www.researchgate.net/publication/220851473_BrainHex_Preliminary_Results_from_a_Neurobiological_Gamer_Typology_Survey>. Luettu 11.7.2024.

4 Järvinen, Aki. 2008. Games without Frontiers: Theories and Methods for Game Studies and Design. Doctoral dissertation study. University of Tampere. Trepo-tietokanta.

5 Saari, Mikko. 2018. Löydä lautapelit. Vantaa: Avain.

6 Adams, Sean. 2017. The Designer's Dictionary of Colour. New York: Abrams.

7 Blender 3.3 Manual. Verkkoaineisto. <https://docs.blender.org/manual/en/3.3/getting_started/about/introduction.html>. Luettu 5.9.2024.

8 Blender 3.3 Manual. Verkkoaineisto. <https://docs.blender.org/manual/en/3.3/interface/window_system/introduction.html>. Luettu 5.9.2024.

9 Blender 3.3 Manual. Verkkoaineisto. <https://docs.blender.org/manual/en/3.3/interface/window_system/areas.html>. Luettu 5.9.2024.

10 Blender 3.3 Manual. Verkkoaineisto. <https://docs.blender.org/manual/en/3.3/interface/window_system/topbar.html>. Luettu 5.9.2024.

11 Blender 3.3 Manual. Verkkoaineisto. <https://docs.blender.org/manual/en/3.3/interface/window_system/status_bar.html>. Luettu 5.9.2024.

12 Blender 3.3 Manual. Verkkoaineisto. <https://docs.blender.org/manual/en/3.3/editors/3dview/introduction.html>. Luettu 5.9.2024.

13 Blender 3.3 Manual. Verkkoaineisto. <https://docs.blender.org/manual/en/3.3/editors/timeline.html>. Luettu 5.9.2024.

14 Blender 3.3 Manual. Verkkoaineisto. <https://docs.blender.org/manual/en/3.3/editors/outliner/introduction.html>. Luettu 5.9.2024.

15 Blender 3.3 Manual. Verkkoaineisto. <https://docs.blender.org/manual/en/3.3/editors/properties_editor.html>. Luettu 5.9.2024.

16 Blender 3.3 Manual. Verkkoaineisto. <https://docs.blender.org/manual/en/3.3/interface/window_system/workspaces.html>. Luettu 5.9.2024.

17 Blender 3.3 Manual. Verkkoaineisto. <https://docs.blender.org/manual/en/3.3/interface/window_system/regions.html>. Luettu 5.9.2024.

18 Blender 3.3 Manual. Verkkoaineisto. <https://docs.blender.org/manual/en/3.3/modeling/introduction.html>. Luettu 5.9.2024.

19 Blender 3.3 Manual. Verkkoaineisto. <https://docs.blender.org/manual/en/3.3/modeling/meshes/primitives.html>. Luettu 5.9.2024.

20 Blender 3.3 Manual. Verkkoaineisto. <https://docs.blender.org/manual/en/3.3/modeling/meshes/structure.html>. Luettu 5.9.2024.

21 Blender 3.3 Manual. Verkkoaineisto. <https://docs.blender.org/manual/en/3.3/scene_layout/object/origin.html#set-origin>. Luettu 5.9.2024.

22 Unity User Manual 2022.3 (LTS). Verkkoaineisto. <https://docs.unity3d.com/2022.3/Documentation/Manual/UsingTheEditor.html>. Luettu 5.9.2024.

23 Unity User Manual 2022.3 (LTS). Verkkoaineisto. <https://docs.unity3d.com/2022.3/Documentation/Manual/ProjectView.html>. Luettu 5.9.2024.

24 Unity User Manual 2022.3 (LTS). Verkkoaineisto. <https://docs.unity3d.com/2022.3/Documentation/Manual/UsingTheSceneView.html>. Luettu 5.9.2024.

25 Unity User Manual 2022.3 (LTS). Verkkoaineisto. <https://docs.unity3d.com/2022.3/Documentation/Manual/overlays.html>. Luettu 5.9.2024.

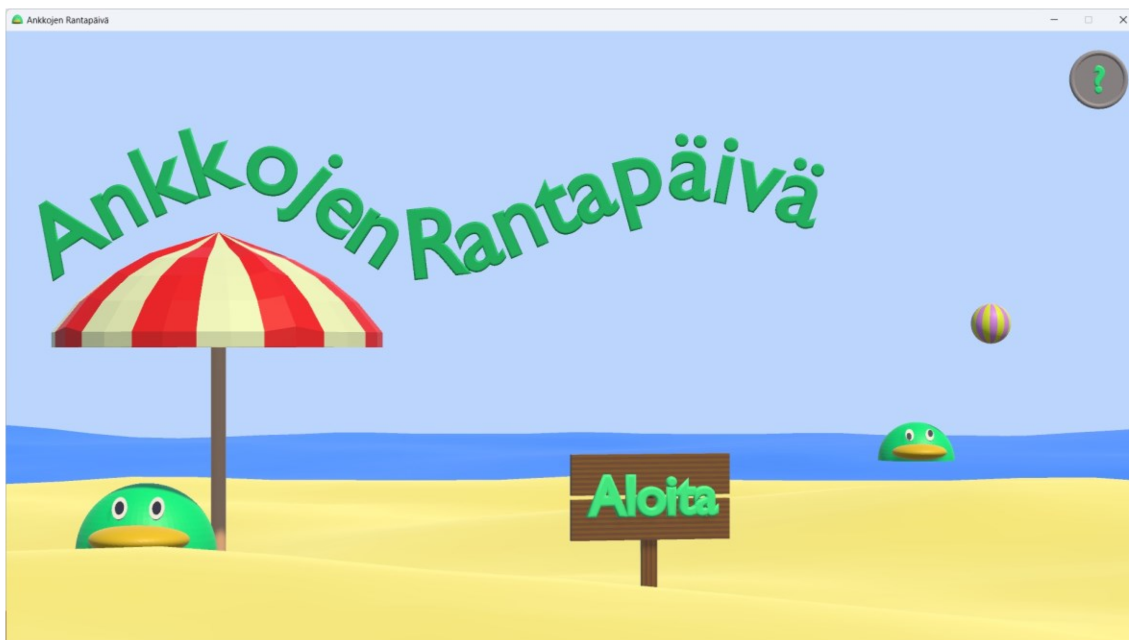
26 Unity User Manual 2022.3 (LTS). Verkkoaineisto. <https://docs.unity3d.com/2022.3/Documentation/Manual/GameView.html>. Luettu 5.9.2024.

27 Unity User Manual 2022.3 (LTS). Verkkoaineisto. <https://docs.unity3d.com/2022.3/Documentation/Manual/default-overlays-reference.html>. Luettu 5.9.2024.

28 Unity User Manual 2022.3 (LTS). Verkkoaineisto. <https://docs.unity3d.com/2022.3/Documentation/Manual/CustomizingYourWorkspace.html>. Luettu 5.9.2024.

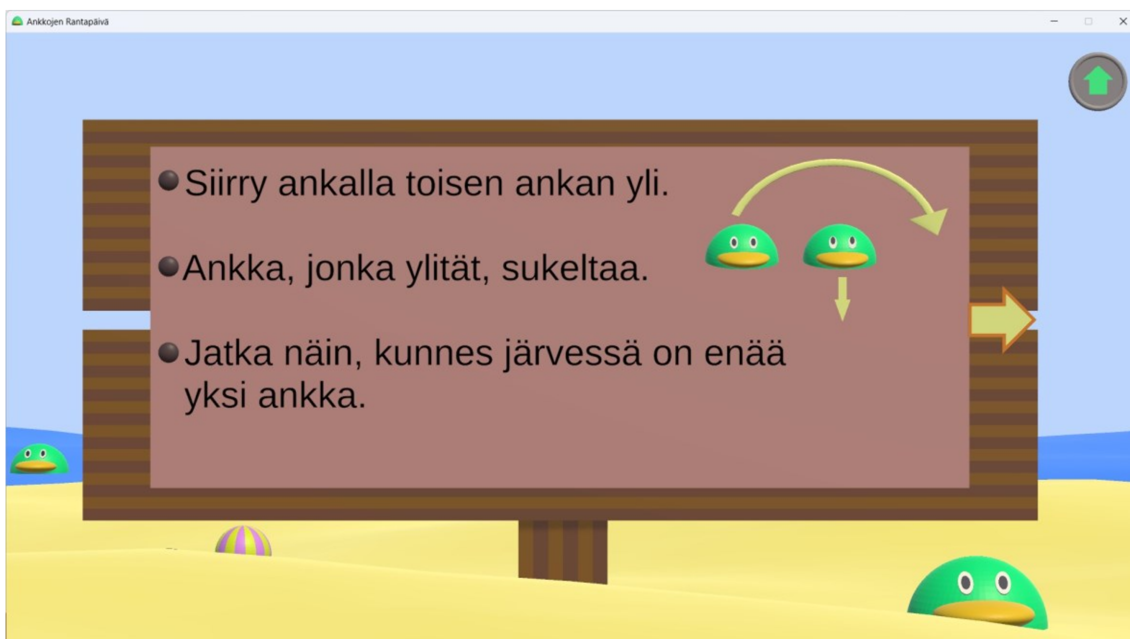
Kuvakaappauksia Ankkojen Rantapäivä -pelistä

Tässä liitteessä esitetään kuvakaappauksia valmiista Ankkojen Rantapäivä -pelistä. Kuvassa 1 on pelin etusivu. Oikean yläkulman kysymysmerkkipainikkeesta avataan ohjeet ja rannalla olevaa kylttiä klikkaamalla aloitetaan peli.



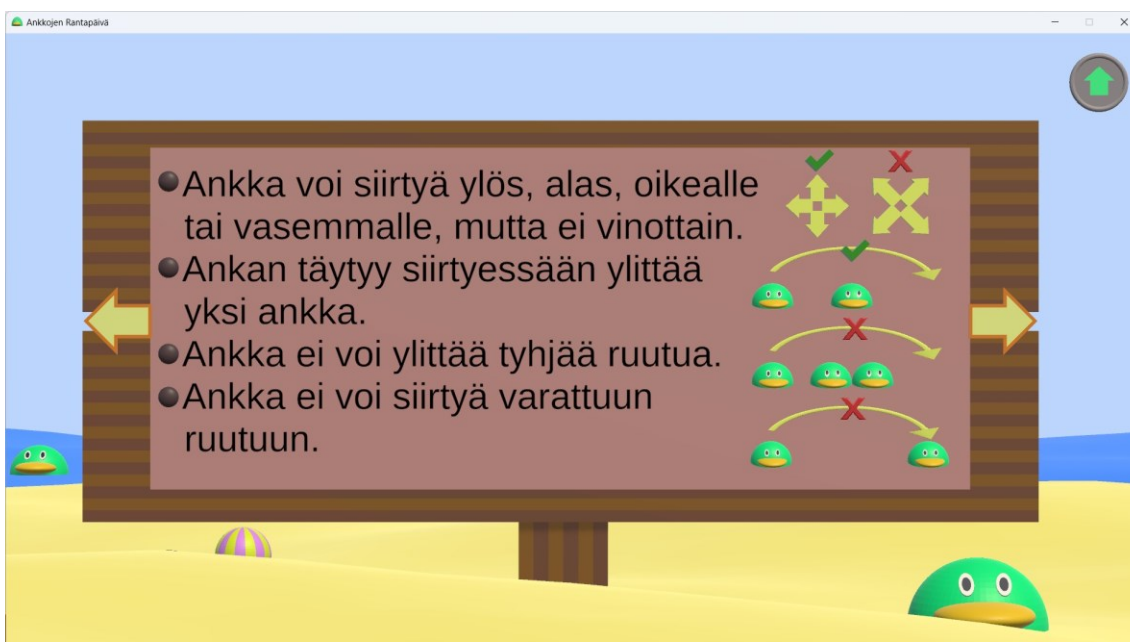
Kuva 1. Etusivu.

Kuvassa 2 esitetään pelin ohjeiden ensimmäinen sivu. Ensimmäisellä ohjesivulla pelaajalle kerrotaan lyhyesti, mitä pelissä tehdään. Seuraavalle sivulle siirrytään oikealle osoittavaa nuolta klikkaamalla. Aloitusnäyttöön voidaan palata oikean ylänurkan kotipainikkeesta.



Kuva 2. Ohjeet, sivu 1/4.

Kuvassa 3 on ohjeiden kolmas sivu. Tällä sivulla käydään läpi pelinappulan siirtämiseen liittyvät säännöt. Edelliselle sivulle voidaan palata vasemmalle osoittavaa nuolta klikkaamalla ja seuraavalle sivulle päästään oikealle osoittavasta nuolesta.



Kuva 3. Ohjeet, sivu 3/4.

Kuvissa 4 ja 5 esitetään pelitilanteita. Kuvassa 4 peli on alkutilanteessa eli siirtoja ei ole vielä tehty.

Kuva 4. Peli alkutilanteessa.

Kuvassa 5 ollaan pelitilanteessa, jossa pelilaudalla on jäljellä 17 pelinappulaa.

Kuva 5. Pelitilanne.

Kuvassa 6 pelaaja on voittanut pelin ja voitosta kertova veneanimaatio on käynnistynyt.

Kuva 6. Voitettu peli.

Pelaaja voi milloin tahansa palata pelinäkymästä etusivulle tai aloittaa pelin uudelleen käyttämällä alalaidan painikkeita.