Tasoeditorin luominen Unity-pelimoottorilla

Salla Salmi

Haaga-Helia ammattikorkeakoulu

Tietojenkäsittely

Opinnäytetyö

2023

Tiivistelmä

Tekijä(t)
Salla Salmi

Tutkinto
Tradenomi

Raportin/Opinnäytetyön nimi
Tasoeditori luominen Unity-pelimoottorilla

Sivu- ja liitesivumäärä
29 + 5

Tämän opinnäytetyön tarkoituksena oli luoda tasoeditori-työkalu 2D-monialustapelin tueksi. Tasoeditorin avulla peliin luodaan uusia tasoja, joita läpäisemällä pelissä edetään. Tasot yhdistetään peliin JSON-tiedostoina, joten tasoeditorilla pitää pystyä myös tallentamaan tasosetit halutunlaisena tiedostona. Myös uusien tasojen testaamismahdollisuus peliin yhdistettynä kuului tasoeditorin tavoitteisiin.

Teoriaosuudessa käydään läpi ohjelmistokehityksen vaiheita, menetelmiä ja teknologioita. Ohjelmistokehityksen vaiheet ja menetelmät vaihtelevat projektin mukaan, joten tässä työssä käyn läpi teorioita, joita itse hyödynsin projektissa. Myös erilaisia teknologioita on useita, joten niistäkin esittelen ne, joita itse käytin. Käyttämäni teknologiat ovat kuitenkin hyvin yleisesti käytettyjä.

Teoriaosuuden jälkeen käyn läpi toteutuksen, jonka tein kolmen sprintin aikana. Toteutin tasoeditorin Unity-pelimoottorilla muokkaamalla sen käyttöliittymää. Tasoeditori yhdistettiin pelin projektiin, jota työstetään myös Unityllä. Ohjelmointikielenä toimi C#, joka on Unityssä yleisesti käytettävä koodikieli. Hyödynsin toteutuksessa myös muita Unityn tarjoamia ominaisuuksia, kuten ScriptableObjecteja, jotka mahdollistavat datan tallentamisen Unity-projektiin.

Lopuksi vielä pohdin toteutuksen onnistumista ja omaa työskentelyäni. Projektin päätyttyä olin erityisen tyytyväinen tasoeditoriin ja sen parissa työskentelyyn, mutta projektin aikana esiintyi kuitenkin myös ongelmia ja haasteita. Suurimpana haasteena oli kirjoittamisen suunnittelu ja aikatauluttaminen. Näitä aiheita avaan vielä enemmän pohdinnassa, jatkokehitysideoiden lisäksi.

Asiasanat
Unity, Tasoeditori, C#, Ohjelmistokehitys

Sisällys

1 Johdanto

Työskentelen ohjelmistosuunnittelijana jo pitkään toimineessa peliyrityksessä 2D-monialustapelin parissa. Peli on ollut jo vuosia toiminnassa ja itse olen työskennellyt sen parissa reilun vuoden. Pelin ideana on kerätä pisteitä ampumalla kupliksi nimettyjä palloja ja niiden avulla tyhjentää pelikenttä. Pelin alareunassa on putken suuaukko, josta kuplat tulevat yksi kerrallaan ja ammutaan ylöspäin kentälle. Kuplat yritetään tähdätä muiden samanväristen viereen ja kun vähintään 3 samanväristä kuplaa on vierekkäin, ne poksahtavat ja tyhjentyvät kentältä. Kun kenttä on kokonaan tyhjä, peli etenee seuraavalle tasolle. Eri tasoilla on eri määrä erivärisiä kuplia, jotka on aseteltu aina eri paikkoihin. Kuplien seassa on myös esteitä, jotka on mahdollista rikkoa vain ampumalla kaikki sen ympärillä olevat kuplat. Pelaajat ansaitsevat pisteitä läpäisemällä tasoja mahdollisimman pitkälle. Pelissä on aina viikoittainen haaste, jonka aikana pyritään kasvattamaan pistesummaa mahdollisimman suureksi. Aina uuden viikon alussa pistemäärät nollaantuvat ja uusi haaste alkaa.

Tämän opinnäytetyön tarkoituksena on luoda tasoeditori-työkalu, jonka avulla pyritään tarjoamaan tehokas tapa uusien tasojen luomiseen. Tasosetit vaihtuvat kaksi kertaa viikoittaisten haasteiden aikana, jotta peli pysyisi mahdollisimman mielenkiintoisena. Tasosetti tarkoittaa JSON-tiedostoa, joka sisältää enintään 60 tason listan ja niiden tiedot. Koska tasoja tarvitaan suuri määrä ja ne vaihtuvat usein, on niiden tarve suuri. Tällä hetkellä uusien tasojen luomiseen ei ole lainkaan kunnon keinoa. Ensin pitäisi selvittää mikä kohta tason kentässä vastaa mitäkin numeroarvoa ja sen jälkeen JSON-tiedosto pitäisi kirjoittaa käsin. Kentän todellisen ulkonäön näkee vasta, kun JSON-tasot liitetään peliin. Tämä on hyvin epätehokas ja monimutkainen tapa, jonka takia tarve tasoeditorille on suuri. Rakentamani tasoeditorin avulla on mahdollista luoda, muokata ja tallentaa uusia tasosettejä.

Peliä kehitetään Unity-pelimoottorilla, joten tasoeditori luodaan myös Unityssä ja yhdistetään pelin projektiin. Unity on suosittu pelimoottori, jota käytetään laajasti erilaisten pelien ja sovellusten kehittämiseen. Yksi Unityn vahvuuksista on sen käyttöliittymän muokattavuus. Toteutan tasoeditorin luomalla uuden työkalun Unityn käyttöliittymään. Luon sen C# kooditiedostojen ja Unityn tarjoamien työkalujen avulla. Ennestään minulla on yli 2 vuoden kokemus Unityn käytöstä, jonka olen saanut opiskelukurssilla tehdystä 3D-pelistä ja tämänhetkisestä työstäni. Kokemukseeni kuuluu pelien rakentaminen, bugien jäljitys ja korjaus sekä projektinhallinta, kuten eri alustaversioiden buildaaminen ja peliin yhdistettyjen ohjelmien päivitys. Minulla ei kuitenkaan ole ennestään kokemusta käyttöliittymän muokkaamisesta tai tämänkaltaisesta projektista. Tasoeditorin tarkkojen vaatimusten avulla pystyn kuitenkin saamaan käsityksen projektin haastavuudesta. Tämän käsityksen ja omien taitojeni perusteella arvioin kykeneväni luomaan

ratkaisun ja pääsemään tavoitteisiin. Loppukäyttäjinä tulee olemaan yrityksen omat työntekijät, joille peli on ennestään tuttu. Tämän vuoksi käyttäjäkokemus ja käytettävyys eivät ole prioriteettien kärjessä, vaan keskitytään enemmän toimivan ratkaisun luomiseen.

Aluksi käsittelen tietoperustaa aiheista, joiden avulla ratkaisu luodaan. Tietoperustan olen kerännyt artikkelien ja teknologioiden omien dokumentaatioiden avulla. Ensin kerron ohjelmistokehityksestä ja sen menetelmistä. Tämän jälkeen käyn läpi ohjelmistokehityksen teknologioita, joita itse käytän. Tietoperustan jälkeen käsitellään tasoeditorin luomisvaihe. Luomisvaihe toteutetaan kolmessa sprintissä. Jokaisesta sprintistä kerrotaan sen tavoite ja kuinka se on toteutettu. Lopussa vielä pohditaan omaa työskentelyä sekä projektin onnistumista ja tulevaisuutta.

2 Ohjelmistokehitys ja sen menetelmät

Ohjelmistokehitys on prosessi, jossa kehitetään ohjelmia tietokoneille ja muille laitteille. Prosessiin sisältyy useita vaiheita, joihin kuuluu esimerkiksi vaatimusmäärittely, suunnittelu, toteutus, testaus ja käyttöönotto. Ohjelmistokehityksen perusperiaatteet ovat samat, vaikka prosessi voi vaihdella eri ohjelmistokehitysyritysten välillä. Ohjelmistokehitys vaatii yleensä laajan osaamisen ohjelmointikielistä ja -tekniikoista sekä hyvää ymmärrystä asiakkaan tarpeista ja vaatimuksista. Ohjelmistokehityksen avulla voidaan luoda monenlaisia ohjelmistoja, kuten mobiilisovelluksia, verkkosivustoja, tietokantaohjelmistoja ja peliohjelmistoja. Ohjelmistokehitysprojektien kesto voi vaihdella muutamasta päivästä useisiin vuosiin riippuen ohjelmiston monimutkaisuudesta ja laajuudesta. (Full Scale, 2022)

2.1 Ohjelmistokehityksen vaiheet

Vaatimusmäärittely on prosessi, jossa määritellään mitä ohjelmistolta vaaditaan. Dokumentaatioon kirjataan ylös, mitä ohjelmistolla on tarkoitus tehdä, millaisia ominaisuuksia siinä tulee olla ja mitä ongelmia sen tulee ratkaista. (Matilainen, 2022). Dokumentaatiossa voidaan myös erikseen määritellä mitä ohjelman ei kuulu tehdä. Vaatimusmäärittelyn avulla varmistetaan, että tulos vastaa haluttua lopputulosta. Sen avulla projektin voi suunnitella pienemmiksi toteutettaviksi osiksi. Vaatimusmäärittelyä on mahdollista kuvata visuaalisesti UML-kaavioiden avulla. UML-kaaviot ovat visuaalisia kuvauksia tietokantojen tai sovellusten rakenteista ja niiden avulla voi yksinkertaistaa monimutkaisia asioita. Luokkakaavio on yleisin UML-kaavio tyyppi, jota käytetään ohjelmistokehityksessä. Sen avulla kuvataan luokat ja niiden väliset suhteet. Jokainen luokka sisältä luokan nimen, määritteet ja metodit tai toiminnot. (Microsoft, 2019)

Vaatimusmäärittelydokumentin pohjalta suunnitellaan ratkaisu. Ohjelmistokehityksen näkymättömyyden takia on sitä vaikea kuvata ilman vertauskuvia. Anu Halme (2000) esittää yleisesti käytetyn talonrakennus vertauksen kirjoituksessaan, sillä kummassakin voi monella eri tavalla päästä samaan lopputulokseen. Tämän takia ohjelman liian yksityiskohtainen suunnittelu saattaa olla tarpeetonta. On kuitenkin tärkeä saada selkeä kuva siitä mitä lähdetään tekemään ja miten päästään haluttuun lopputulokseen. Suunnittelu voidaan jakaa yleis- ja yksityiskohtaiseen suunnitteluun. Yleissuunnitteluun kuuluu järjestelmän kokonaisrakenne, yleiset tietorakenteet, jako alijärjestelmiin sekä osajärjestelmien ja moduulien välinen kommunikointi. Yksityiskohtaisessa suunnitellaan moduulien sisäinen rakenne ja tietorakenteet. (Laine, 1998)

Suunnittelun aikana on hyvä ottaa huomioon myös ohjelman käytettävyys. Käyttäjäkokemus eli UX-suunnittelu tarkoittaa käyttäjän kokemusta ohjelmaa käytettäessä. Ohjelman ominaisuuksien ja visuaalisen ilmeen suunnittelussa tulee ottaa huomioon loppukäyttäjäryhmä ja kuinka ohjelman

käyttökokemus on sille ryhmälle toimiva. Käyttöliittymä eli UI-suunnittelu puolestaan koskee ainoastaan digitaalisia tuotteita ja liittyy siihen mitä käyttäjä näkee näytöllä. UI-suunnittelulla varmistetaan esimerkiksi värien ja sivun asettelun toimivuus. Erityisesti julkisissa ohjelmissa ja sivustoissa pyritään tekemään käyttäjän kokemuksesta mahdollisimman miellyttävä. UX- ja UI-suunnittelu kulkevat käsikädessä ja yhdessä varmistavat kokonaiskäytettävyyden. (Hurja, 2021)

Ohjelmistotestaamiseen kuuluu monia eri osa-alueita. Yksikkötestauksella testataan koodin osia ja integraatiotestauksella miten eri osat toimivat keskenään. Lisäksi testauksella on useita tasoja, joihin kuuluu mm. toiminnallinen testaus, ei-toiminnallinen testaus ja regressiotestaus.

Toiminnallisessa testauksessa varmistetaan ohjelman toimivuus ja että se täyttää tarvittavat vaatimukset. Ei-toiminnallisessa sen sijaan testataan ei näkyviä toimintoja kuten suorituskykyä ja resurssien käyttöä. Regressiotestauksessa tarkistetaan, ettei tehdyt muutokset ole rikkoneet muita ominaisuuksia. Käytettävyystestauksessa tutkitaan, kuinka hyvin toiminnassa oleva ohjelmisto toimii ja täyttääkö sen käytettävyys vaatimukset.

Testauksessa ilmenneiden ongelmien jäljittämiseen käytetään debuggausta eli virheiden jäljitystä. Jäljityksessä syvennytään virheviesteihin, etsitään tietoa ja kavennetaan ongelma-aluetta. Sen yhteydessä on myös hyvä pysähtyä miettimään koodin toimintaa syvemmin ja katsomaan asioita eri näkökulmista. (Cocca, 2022) Debug.Log komennon avulla voi Unity editorissa tulostaa informaatioviestejä virheiden jäljityksen avuksi. (Unity, 2022)

2.2 Ohjelmistokehityksen menetelmät

Ketterä kehitys tarkoittaa ketteriä menetelmiä, joita käytetään ohjelmistokehityksessä. Ketterässä kehityksessä ensisijaisena on toimiva ohjelmisto. Menetelmät sisältävät aktiivista viestintää, yhteistyötä loppukäyttäjien kanssa ja valmiuden muutoksiin. (Ekholm & Lehtonen, 2021) Ketterä kehitys sopii monimutkaisille ja arvaamattomille projekteille. Tarkan suunnitelman sijasta keskitytään tuloksiin pala kerrallaan ja kehitetään työskentelyä projektin edetessä. (Koulutus.fi, 2021) Mikko Korkala kuitenkin varoittaa kirjoituksessaan (2020), että menetelmien raameja noudattaen voi helposti saada illuusion toiminnan olevan automaattisesti ketterää, vaikka raamien ulkopuolelle voi jäädä tärkeitä asioita.

Yksi yleisimmistä ketteristä menetelmistä on Scrum. Scrum prosessissa projekti jaetaan osiksi työjonoon. Työjonon tehtävät jaetaan yleensä viikon tai kahden, mutta myös mahdollisesti pidempään kestäville sprinteille, jonka aikana ne toteutetaan. Sprinttien aikana pidetään useita palavereja, joissa käydään läpi saavutukset ja mitä tehdään seuraavaksi. Sidosryhmille pidetään

myös sprinttikatselmointeja, joissa esitellään siihen asti saadut tulokset. Sprinttien lopussa pidetään retrospektiivi, jonka aikana arvioidaan työskentelyä ja keksitään parannusideoita. (Hietaniemi, 2019)

Jos ongelmalle ei ole saatavilla valmista ratkaisua löydetään se yrityksen ja erehdyksen avulla. Ensin kokeillaan yhtä ratkaisua ja jos se ei toimi kokeillaan toista. Yritys ja erehdys on menetelmä oppia uutta ja sitä hyödynnetään yleisesti ohjelmistokehityksessä. (Gautam, 2023) Ohjelmoinnin osalta menetelmässä on kuitenkin omat sudenkuoppansa. Monesti koodatessa tekee mieli aloittaa työn teko heti ensimmäisestä ideasta. Vaikka koodi toimisi, se ei silti usein ensimmäisellä ratkaisulla kata kaikkia tapauksia. Jos uusia tapauksia lähtee paikkaamaan esimerkiksi ”if” lauseen avulla, tulee koodista helposti paljon monimutkaisempaa kuin sen tarvitsisi olla. Monimutkainen koodi vaikeuttaa ylläpitoa ja mahdollista jatkokehitystä. Yrityksen ja erehdyksen sijaan on parempi harkita tehtävää kokonaisuutena ja järjestää sen sisältö ennen työntekoon ryhtymistä. (Frankel, 2017)

3 Ohjelmistokehityksen teknologiat

Ohjelmistokehitys on monimutkainen prosessi, joka vaatii useita teknologioita ja työkaluja. Näihin teknologioihin kuuluu yleisesti ohjelmointikielet, kehitysympäristöt ja versionhallintajärjestelmät.

3.1 Github

Versionhallinnan avulla ohjelmoija pystyy seuraamaan ja hallitsemaan ohjelmistoprojektin koodiin tehtäviä muutoksia sekä työskentelemään tehokkaammin yhdessä. Versionhallinnan avulla koodille on mahdollista tehdä sivuhaaroja, joihin voi tehdä muutoksia vaikuttamatta viralliseen koodiin. Myös haarojen yhdistäminen onnistuu versionhallinnassa. Git on avoimen lähdekoodin versionhallintajärjestelmä ja Github verkkosivusto toimii sen käyttöliittymänä. Github käyttöliittymä on erittäin käyttäjäystävällinen, jonka takia se sopii aloittelevillekkin koodaajille. Git onkin yksi suosituimmista versionhallintajärjestelmistä. (Kinsta, 2022). Githubissa on myös projektityökalu, joka auttaa organisoimaan ja hallitsemaan projektia Githubissa. Projektiominaisuuden avulla projektin jäsenet voivat luoda taulukoita, joihin voi lisätä projektin tehtäviä, jakaa niitä ja seurata edistymistä. Taulukot ovat usein esimerkkiä ”tehtävät”, ”työn alla” ja ”valmis”. Projektiominaisuus helpottaa projektin hallintaa ja auttaa jäseniä pysymään ajan tasalla projektin etenemisestä. (Github, 2023)

3.2 Unity editori

Unity editori on Unity Technologiesin kehittämä pelimoottori, jolla voi luoda kaksi- ja kolmiulotteisia videopelejä. (Wikipedia). Unity pelimoottorista puhuessa käytetään yleisesti pelkästään nimitystä Unity. Unityyn kuuluu Unity hub, jonka avulla hallitaan Unity käyttöliittymän asennuksia ja luodaan projektit. Unity editori on visuaalinen komponentti, jonka avulla pelit luodaan. Sen käyttöliittymä on helppokäyttöinen jo heti pienen tutkailun jälkeen, jonka takia Unity onkin yksi johtavista pelimoottoreista. Unityn ohjelmointikielenä toimii C#. (Geig, 2022)

3.3 Visual studio

Visual studio on Microsoftin ohjelmointiympäristö, jossa voi käyttää useita ohjelmointikieliä. Se sisältää työkaluja ja ominaisuuksia, jotka tehostavat ohjelmointia. Näihin kuuluu esimerkiksi virheenkorjaus ja koodin täydennys. Koodia kirjoittaessa Visual studiossa se ehdottaa komentoja samalla kirjoittaessa, näyttää virheet ja antaa ehdotuksia niiden korjaamiseksi. Myös luokkien ja eri kooditiedostojen välillä navigointi on helpompaa. Visual studiossa on myös mahdollisuus kirjoittaa koodia muiden kanssa reaaliajassa. (Microsoft, 2023)

3.4 C#

C# on Microsoftin kehittämä ohjelmointikieli, joka toimii .NET:issä. Se juureutuu C-kieliperheestä ja on oliosuuntautunut sekä tyyppiturvallinen. .NET-ohjelmistokehitys sisältää suuren määrän luokkakirjastoja eli valmiiksi kirjoitettua koodia, joka auttaa kehittäjiä luomaan sovelluksia nopeammin ja tehokkaammin. .NET tukee C# lisäksi muitakin ohjelmointikieliä, mutta C# on näistä kaikista suosituin. C# avulla voi kehittää sovelluksia Windowsille, iOS:ille ja Androidille. C# kehitettäessä oli sen tavoite olla samaan aikaan yksinkertainen ja suorituskykyinen. (Codeberry, 2022)

4 Toteutus

4.1 Lähtötilanne

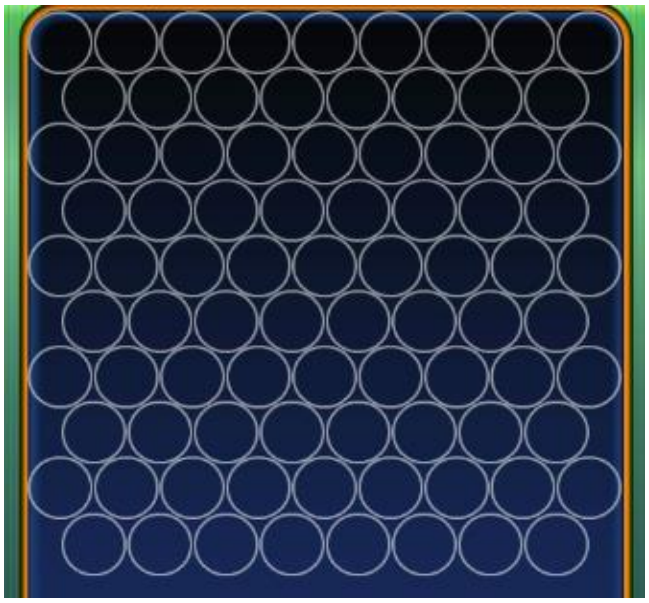
Projektin toimeksiantajana toimii peliyritys, jossa itse työskentelen ohjelmistosuunnittelijana. Valmis tuotos tulee käyttöön yrityksen sisälle ja yrityksen työntekijät tulevat olemaan sen loppukäyttäjiä. Tasoeditori luodaan samankaltaisen menetetyn ohjelman tilalle. Vanha tasoeditori sijaitsi SVN repositoryssä johon kellään ei ole enään pääsyä. Menetetyn ohjelman takia tällä hetkellä uusien tasojen luominen on vajanaista, monimutkaista ja erittäin kustannustehotonta. Tekstitiedostot, joista itse pelin koodi lukee tasot sisältävät useita numerolistoja. Numerot listalla ovat arvoja, jotka kertovat missä kohdissa pelikenttää pallot ja esteet sijaitsevat. Tällä hetkellä nämä tiedostot täytyy kirjoittaa käsin, eikä niitä pysty testata lainkaan.

Kyseinen peli, jota varten tasoeditori luodaan, on kehitetty Unity-pelimoottorilla. Tasoeditori liitetään samaan projektiin, joten tarvittavat ohjelmat ovat jo valmiiksi asennettuna. Ohjelmiin kuuluu Unity-pelimoottori ja Visual Studio. Lähtötilanne projektin aloitukselle on suotuisa, sillä suunnittelun jälkeen pääsen heti itse toteutuksen pariin ilman ylimääräisiä vaiheita.

4.2 Vaatimusmäärittely

Aloitin projektin määrittelemällä vaatimukset dokumentille, jonka myöhemmin hyväksytin esimiehelläni. Keräsin vaatimukset selvittämällä asiakastarpeita ja käyttötapauksia. Koska tasoeditori integroidaan pelin projektiin, piti ottaa huomioon myös siihen liittyvät vaatimukset.

Näkyvien osien vaatimuksiin sisältyy kenttäpohja, kenttäobjektit, painikkeet ja tiedonsyöttöikkunat tason datan muokkaamista varten. Kenttäpohja on alusta, jolle uusi taso rakennetaan ja sen tulisi vastata pelissä käytettävää pohjaa (Kuva 1). Kenttäpohja muodostuisi soluista, joihin kenttäobjektit voidaan asettaa. Kenttäobjekteihin kuuluisi pallo ja este, joiden tulisi myös muistuttaa pelissä käytettäviä objekteja (Kuva 2). Käyttäjä tarvitsee painikkeet kenttäobjektien valintaa ja poistamista varten, tasoeditorin avaamispainikkeen sekä tasosettien tallentamispainikkeen. Kentän rakenteen lisäksi tason dataan kuuluu sen nimi, bnc-arvo, väriarvo ja tyyli. Bnc-arvo on desimaaliluku, joka määrittää kuinka paljon saman värisiä palloja asettuu toistensa viereen. Väriarvo kertoo, kuinka montaa eri palloväriä tasossa käytetään ja tyyli määrittää tasossa käytettävät pallografiikat. Kaikki nämä tarvitsevat muokkauslaatikot, joissa käyttäjä voi muokata niiden arvoja.



Kuva 1

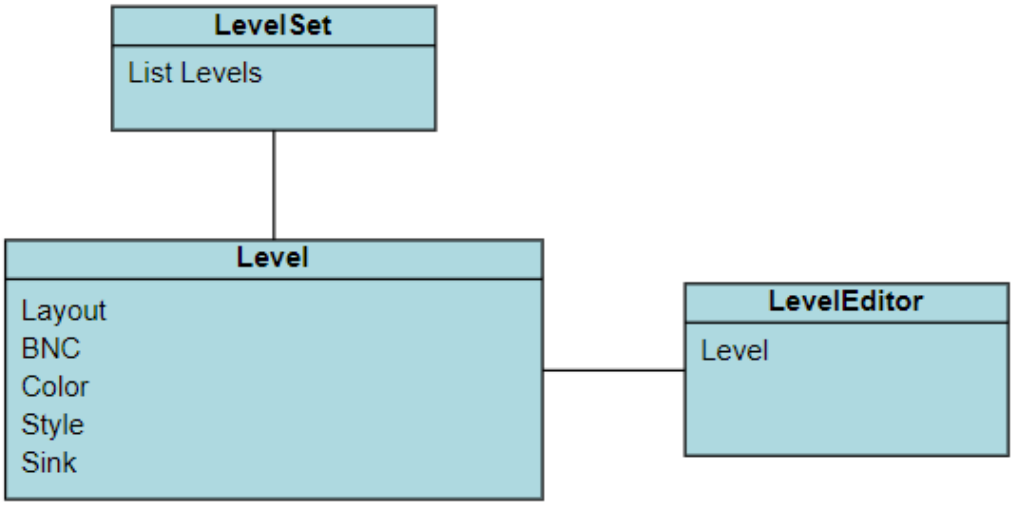
Kuva 2

Muut vaatimukset liittyvät tason tallentamiseen ja testaamiseen. Tason pitää tallentua taustalla automaattisesti, jotta tasoeditorin sulkemisen ja uudelleen avaamisen jälkeen käyttäjä voi jatkaa samasta kohdasta. Tasosetti pitää pystyä tallentamaan myös tiedostona, jossa tasojen data on täysin samanlaisessa JSON-muodossa kuin jo olemassa olevat tasotiedostot (Kuva 3). Kuvassa näkyy yksi taso ja sen tiedot. Todellisuudessa tiedosto sisältää pitkän listan tasoja kuvan kaltaisessa muodossa. Tiedostojen on tärkeää olla samassa muodossa, sillä niiden avulla tasot yhdistetään peliin. Käyttäjä haluaa myös testata luomiaan tasoja. Testaaminen yhdistetään Unity-editorissa toimivaan pelin versioon, jolloin testatessa näkee tason toiminnan niin kuin se toimii oikeassakin pelissä.


```
[{"bnc":0.7,"colors":3,"layout":[[0,0,0,1,0,1,0,0,0],[0,0,0,1,1,0,0,0,3],[0,0,1,1,1,1,1,0,0],[0,0,0,1,1,0,
0,0,3],[0,0,0,1,0,1,0,0,0],[0,0,0,0,0,0,0,0,3],[0,0,0,0,0,0,0,0,0],[0,0,0,0,0,0,0,0,3],[0,0,0,0,0,0,0,0,
0],[0,0,0,0,0,0,0,0,3]],"name":"STELLAR","sink":0,"style":[3,5,8,16,11,9,14,10,4,15,1,2,6,7,12,13,17,18]}]
```


Kuva 3. Yhden tason data.

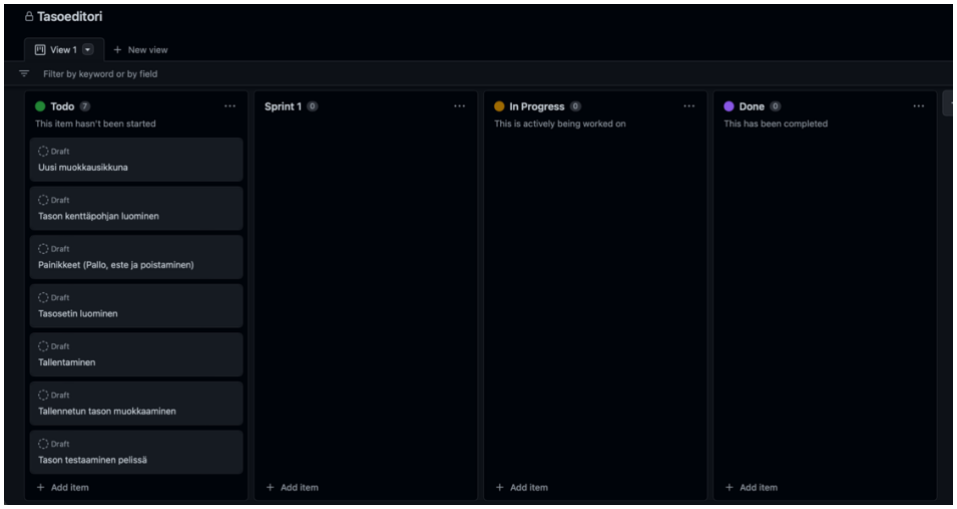
Selvitin vaatimusmäärittelyyn myös luokkien väliset suhteet ja havainnollistin ne UML-kaaviolle (Kuva4). LevelSet eli tasosetti sisältää enintään 60 kappaleen pituisen listan tasoja. Level eli taso sisältää tason datan ja kuuluu tasosettiin. LevelEditor eli tasoeditori muokkaa aina yhtä tasoa kerrallaan. Kaavion on tarkoitus olla suuntaa antava suunnitelma kokonaiskäsityksen tueksi. Tasosetti ja taso luokat toteutetaan Unityn scriptableObject peliobjekteina ja tasoeditori toteutetaan Unityn editor-luokkaa käyttäen.



Kuva 4

4.3 Projektin menetelmä

Projekti toteutetaan Scrum-menetelmää soveltaen. Kokonaiskesto on 6 viikkoa, joka toteutetaan 2 viikon kestoisina sprintteinä. Jaoin vaatimukset osiin ja lisäsin ne työtehtäviksi työjonoon. Ennen sprinttien alkua valitsen niiden aikana työstettävät tehtävät ja siirrän ne työjonossa oikean sprintin alle. Kun alan työstämään jotain, siirrän sen työn alla kohtaan ja seuraavaksi sen valmistuessa valmiit kohtaan. Käytän työjonon tekemiseen apuna Githubin projektiominaisuutta (Kuva 5).



Kuva 5 Github projektiin luotu työjono.

Ensimmäisen sprintin tavoitteena on luoda muokkausikkuna ja sen toiminnot. Toiminnot sisältävät layoutin, painikkeet ja painikkeiden toiminnallisuuden. Kaikki tallentaminen on toisen sprintin tavoitteena. Tallentamiseen kuuluu yksittäisen tason tallentaminen, niiden lisääminen tasosettiin ja tasosetin tallentaminen json-tiedostona. Viimeisessä sprintissä luodaan testausmahdollisuus uusille tasoille sekä testataan itse projektia.

Koska tasoeditori luodaan jo toiminnassa olevaa peliä varten, ovat tavoitteet hyvin tarkat. Tavoitteita rajaa jonkin verran myös itse peliprojekti, johon tasoeditori liitetään. Tarkat tavoitteet on helppo jakaa osiin, vaikka kokemusta tai syvempää tietoa tämänkaltaisesta projektista ei ole.

Valitsin Scrum-menetelmän juuri puuttuvan kokemuksen ja tiedon takia. Uutta oppiessa muutokset ja lisäykset ovat välttämättömiä. Sprinteissä työskentely antaa vapaammat kädet etsiä ratkaisuja, jotta päästäisiin tavoitteisiin. Pienempiin osiin jaetussa työssä on helpompi tietää mistä lähteä liikenteeseen.

4.4 Sprint 1

Ensimmäisen sprintin tavoitteena on luoda uusi muokkausikkuna ja sen sisältö unity editoriin. Muokkausikkunan sisältöön kuuluu kaikki toiminnot, joita uuden tason rakentamiseen tarvitaan, eli kenttäpohja, painikkeet ja niiden toiminnallisuus.

4.4.1 Editori ikkuna

Unity editoria on mahdollista laajentaa omilla muokatuilla tarkasteluikkunoilla ja editoreilla, sekä määrittää kuinka ominaisuudet näytetään laajennuksissa. Ratkaisun työstäminen aloitetaan luomalla uusi editori-ikkuna, johon sisällytetään tasoeditorin toiminnot. Ikkunan luomisen vaiheisiin kuuluu luokan jatkaminen editori-ikkunasta, määrittää mistä valikon kohdasta ikkuna avataan ja ilmentää luokan sisältö ikkunaan. Ikkunaa varten luodaan uusi c# scripti nimeltä ”LevelEditor”. Scriptin sisälle luodaan LevelEditor luokka, joka jatkuu editori-ikkunasta ja luokan sisälle OpenWindow funktio. Funktio hakee GetWindow komennon avulla LevelEditor luokan sisällön näytettäväksi ikkunalla. Ennen funktiota on määritetty valikkopolku, josta editori-ikkuna avataan.

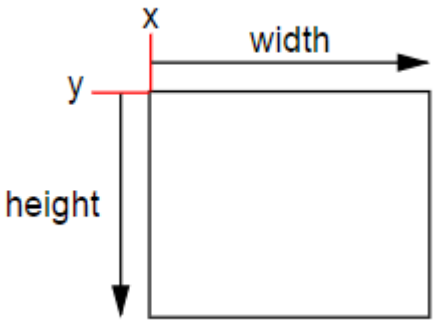
```
using System;
using System.Collections.Generic;
using UnityEditor;
using UnityEngine;

Unity Script | 2 references
public class LevelEditor : EditorWindow
{
    [MenuItem("Window/Level Editor")]
    0 references
    private static void OpenWindow()
    {
        LevelEditor window = GetWindow<LevelEditor>();
        window.titleContent = new GUIContent("Level Editor");

    }
```

Kuva 6 LevelEditor luokan koodista.

4.4.2 Editorin osat

Ikkunan luomisen jälkeen aletaan toteuttamaan sen sisältöä. Tasojen luomista varten tarvitaan taulukkomainen pohja, jolle tasot luodaan. Pelissä tason pohja sisältää 85 solua 10 ja 9 solun riveissä. Rivit on aseteltu lomittain ja joka toinen rivi on 10 solua ja joka toinen 9. Tasoeditorin pohjan tulee vastata pelin pohjan asettelua.

Projektiin luodaan uusi scripti nimeltä ”Tile” joka vastaa yhtä pohjan solua. Luokassa määritetään tile joka sisältää sen sijainnin, koon ja solun tyylin eli grafiikan. Sijainti ja koko määritetään rect rakenteeseen. Rect on 2D-suorakulmio, jonka määrittää X- ja Y-sijainti, leveys ja korkeus. Luokkaan lisätään 2 funktiota Draw ja SetStyle. Draw funktiossa GUI.Box komento luo rect suorakulmion siihen määriteltyjen arvojen mukaan ja asettaa sen tyyliksi määritetyn solun grafiikan. SetStyle funktio asettaa tilen tyyliksi uuden funktiosta saadun tyylin.



Kuva 7 Havainnollistettu kuva Rect suorakulmiosta otettu Unityn sivuilta.

```
using UnityEngine;

Unity Script | 5 references
public class Tile : MonoBehaviour
{
    Rect rect;
    GUIStyle guistyle;

    1 reference
    public Tile(Vector2 position, float width, float hight, GUIStyle defaultstyle)
    {
        rect = new Rect(position.x, position.y, width, hight);
        guistyle = defaultstyle;
    }

    2 references
    public void Draw()
    {
        GUI.Box(rect, "", guistyle);
    }

    0 references
    public void SetStyle(GUIStyle Tilestyle)
    {
        guistyle = Tilestyle;
    }
```

Kuva 8 Tile luokan koodista.

Tile määrittää yhden solun, joten siitä tulee tehdä lista kaikkien solujen yhdistämistä varten. Editori-ikkunan koodiin eli LevelEditor luokkaan lisätään sisäkkäinen lista `List<List<Tile>> tiles;`. Uloin lista on lista pohjan rivejä ja jokainen rivi sisältää listan siihen kuuluvista soluista. Luokkaan luodaan myös uusi tyhjä tyyli ja sille asetetaan projektin EditorItems kansiosta haettu grafiikka solua varten.

OnEnable on funktio, jota kutsutaan, kun objekti on ladattu. LevelEditoriin lisätään OnEnable funktio, jonka aikana tiles listaan asetetaan halutut arvot. Myöhemmin kutsutaan OnGUI funktiota, jonka aikana listalle määritetyt solut piirretään ikkunaan. Listat luodaan sisäkkäin for loopin avulla ja arvot lisätään listalle Add-komennolla. %- merkin avulla voidaan laskea jakojäännös. If ehtolauseessa jakojäännöksen avulla selvitetään pariton rivi. Parittoman rivin pisteet ovat aina puoli askelta eri kohdassa, jolloin solujen piirtyessä rivit ovat lomittain.

```
private void SetUpTiles()
{
    tiles = new List<List<Tile>>();
    for (int i = 0; i < 13; i++)
    {
        tiles.Add(new List<Tile>());

        for (int j = 0; j < 13; j++)
        {
            if (j % 2 == 1)
            {
                tilePos.Set(i * 35 + 17, j * 35 * oddRow);
            }
            else
            {
                tilePos.Set(i * 35, j * 35 * oddRow);
            }
            tiles[i].Add(new Tile(tilePos, 42, 42, empty));
        }
    }
}
```

Kuva 9 SetUpTiles funktiosta

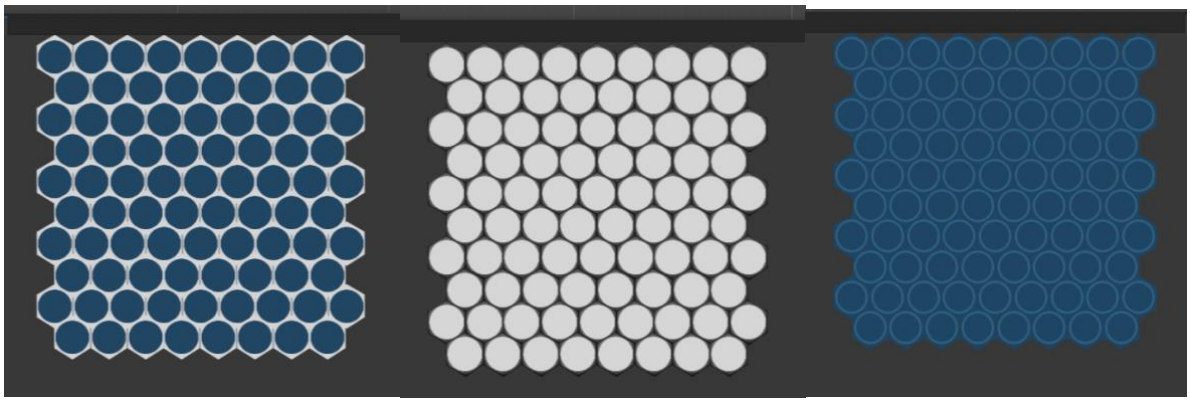
4.4.3 Asettelu

OnGUI funktiota kutsutaan GUI-toimintojen hahmontamiseen ja käsittelyyn. Sitä voidaan kutsua useaan kertaan yhtä freimiä kohden. GUI toiminnot ovat yleisesti käyttäjälle näkyviä asioita. Yksi toiminnoista on Draw jonka avulla tiles listan solut piirretään editori-ikkunaan. Tiles listassa soluja on jokaiselle riville saman verran, mutta if-ehtolauseiden avulla joka toiselta riviltä jätetään yksi piirtämättä. Näin joka toiselle riville saadaan 10 solua ja joka toiselle 9, niin kuin pelin pohjassakin.

```
private void DrawNodes()
{
        for (int i = 4; i < 14; i++)
        {
            for (int j = 0; j < 10; j++)
            {
                if (j % 2 == 0 && i < 13)
                {
                    tiles[i][j].Draw();
                }
                else if (j % 2 == 1 && i < 12)
                {
                    tiles[i][j].Draw();
                }
            }
        }
}
```

Kuva 10 Funktio jonka avulla solut piirretään muokkausikkunaan.

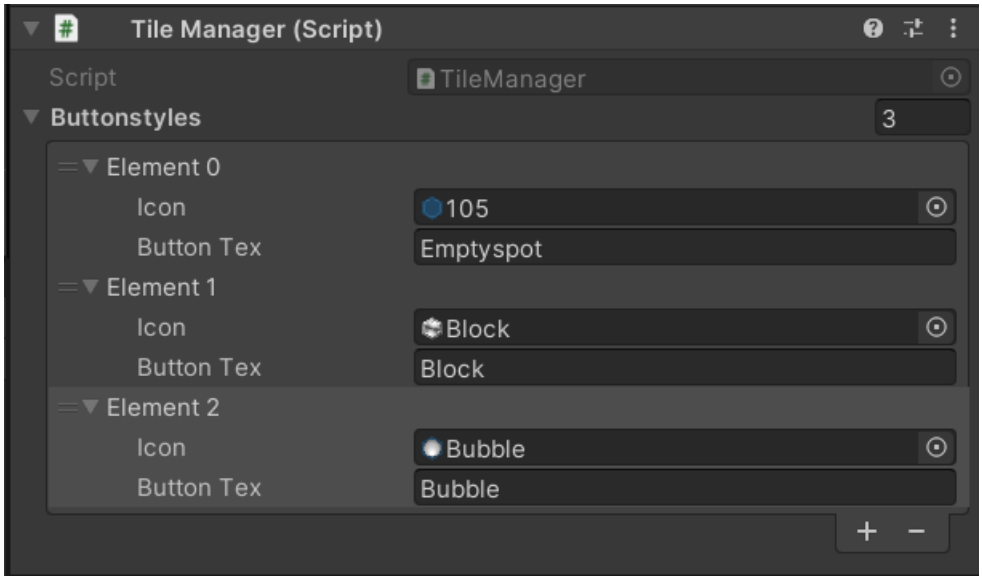
Tason pohjan luonnin aikana soluissa käytetään väliaikaista grafiikkaa. Lopullista grafiikkaa suunnitellessa mietin enemmän käyttäjäkokemusta, sillä kenttäpohja on editorin suurin näkyvä osa. Ensimmäisessä kahdessa versiossa värien kontrastit ovat liian suuret ja epämukavat silmille. Kuvan 11 kaikista oikeanpuolimmaisessa kentässä eli lopullisessa versiossa värit ovat lähempänä toisiaan, mutta palloille tarkoitetut alueet erottuvat silti hyvin.



Kuva 11 Kenttäpohjan ulkoasun eri vaiheet.

4.4.4 Painikkeet

Seuraavaksi siirryn painikkeiden toteutukseen, jota varten luodaan uusi Tilemanager luokka. Luokan sisälle luodaan ButtonStyles niminen lista, joka pitää sisällään painikkeet. Jokainen painike sisältää oman rakenteen, joka on määritetty koodin seuraavassa osassa. Serializable komennon avulla rakenne monistetaan, jolloin on mahdollista luoda useita saman rakenteen omaavia listan osia eli painikkeita. TileManager scriptiä varten luodaan uusi tyhjä peliobjekti, johon se liitetään. Buttonstyles lista piirtyy peliobjektin tarkkailuikkunaan, jossa siihen on mahdollista lisätä ja poistaa listan osia. Lisään listaan kaikki tarvitsemani painikkeet eli tyhjä solu kumittamista varten, este ja pallo. Grafiikat esteelle ja pallolle on tehty aikaisemmin jo peliä varten, josta saan ne myös tasoeditorin käyttöön.



Kuva 12 TileManager peliobjekti, jossa voi lisätä ja poistaa painiketyylejä.

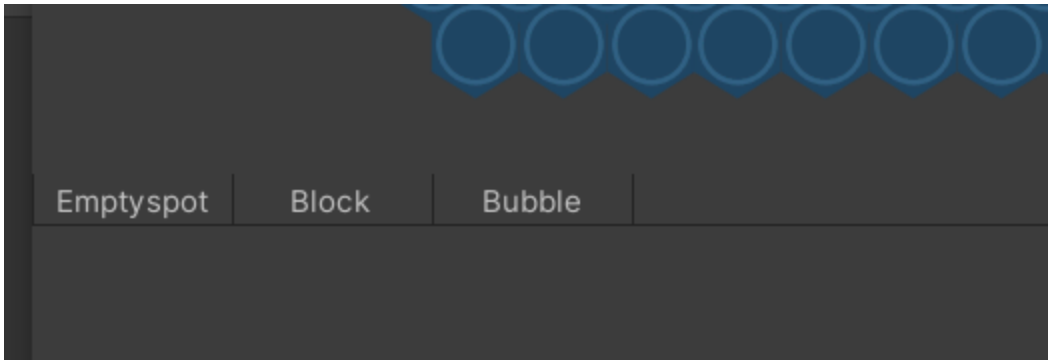
Kun lista painikkeista on luotu, on aika piirtää ne muokkausikkunalle. LeveEditor luokkaan lisätään uusi funktio nimeltä SetUpStyles. Funktiossa etsitään Tilemanager peliobjekti ja käydään sen siältämä Buttonstyles lista läpi. Näin painikkeet saadaan myös LevelEditor luokan käyttöön.

Painikkeiden piirtämistä varten luodaan uusi DrawMenuBar funktio, joka näkyy kuvassa 13. Funktiossa käytetään unityn tarjoamaa Menubar ominaisuutta, jonka avulla painikkeille luodaan oma menupalkki (Kuva 14). Seuraavaksi for-loopin avulla painikkeet käydään läpi ja piirretään menupalkille. Painikkeet luodaan Toggle komennon avulla, joka tarkoittaa, että painike on päällä

niin kauan, kunnes painetaan eri painike päälle. CurrentStyle muuttujaan asetetaan aina päällä olevan painikkeen grafiikka.

```
private void DrawMenubar()
{
    Menubar = new Rect(0, 340, position.width, 300);
    GUILayout.BeginArea(Menubar, EditorStyles.toolbar);
    GUILayout.BeginHorizontal();

    for (int i = 0; i < tilemanager.buttonstyles.Length; i++)
    {
        if (GUILayout.Toggle((currentStyle == tilemanager.buttonstyles[i].TileStyle),
            new GUIContent(tilemanager.buttonstyles[i].ButtonTex),
            EditorStyles.toolbarButton, GUILayout.Width(80)))
        {
            currentStyle = tilemanager.buttonstyles[i].TileStyle;
        }
    }
    GUILayout.EndHorizontal();
    GUILayout.EndArea();
}
```

Kuva 13 DrawMenubar funktio.



Kuva 14 Navigaatiopalkki ja painikkeet.

4.4.5 Tason muuttaminen

Tässä vaiheessa painikkeet on piirretty ikkunalle, mutta niillä ei ole mitään toiminnallisuutta. Kenttäpohjan muokkaamista varten luodaan LevelEditor luokkaan uusi funktio nimeltä ProcessNodes. Aiemmin solujen rivi ja kolumni pistearvoja muokattiin, jotta ne asettuisivat pohjalle lomittain. Myös tässä funktiossa niitä tulee säätää, jotta painikkeiden tekstuurit asettuisivat oikein solujen päälle. If-lauseiden ehtoihin lisätään komennot MouseDown joka aktivoituu hiiren painalluksesta ja MouseDrag joka tarkoittaa hiiren liikuttamista painettuna pohjaan. Jos kumpi tahansa näistä on tosi, tarkistetaan, onko painettu kohta tyhjä vai onko siinä jo pallo tai este. Jos siinä on jotain valmiina, muutetaan se tyhjäksi, jolloin tasoa luodessa objekteja voi kumittaa myös uudelleen klikkaamalla. Jos solu on valmiiksi tyhjä, asetetaan siihen painettuna oleva objekti.

MouseDrag komennon avulla objekteja voi ikään kuin maalata pohjalle, mikä nopeuttaa käyttäjän työtä huomattavasti.

```
public void ProcessNodes(Event e)
{
    int Col = (int)((e.mousePosition.y) / (35 * oddRow));

    if (Col % 2 == 1)
    {
        xPos = 17;
    }
    else
    {
        xPos = 0;
    }

    int Row = (int)((e.mousePosition.x - xPos) / 35);
    apindex = (Row - 4) * 10 + Col;

    if (e.type == EventType.MouseDown)
    {
        if (tiles[Row][Col].guistyle.normal.background.name == "Emptyspot")
        {
            earse = false;
        }
        else
        {
            earse = true;
        }

        if (earse == true)
        {
            tiles[Row][Col].SetStyle(empty);
            savelevel[apindex].val = 0;
            GUI.changed = true;
        }
        else
        {
            if (currentStyle == null)
            {
                currentStyle = empty;
            }
            tiles[Row][Col].SetStyle(currentStyle);
            SetMap(apindex);
            GUI.changed = true;
        }
    }
    if (e.type == EventType.MouseDrag)
    {
        PaintTiles(Row, Col);
        e.Use();
```

Kuva 15 Funktio jonka avulla on toteutettu painikkeiden toiminnallisuus.

Ensimmäisen sprintin jälkeen projektissa on mahdollista avata tasoeditori ikkuna ja muuttaa siinä näkyvää pohjaa painikkeiden avulla.

4.5 Sprint 2

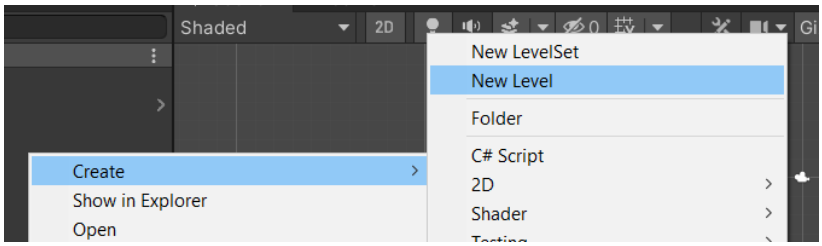
Toisen sprintin tavoitteena on luoda kaikki tallennus ominaisuudet. Tasot ja tasosetit tallennetaan Unityn scriptable objekteina. Scriptable objectit on tietosäiliöitä, joita ei liitetä peliobjekteihin vaan ne tallennetaan projektin resursseihin.

4.5.1 ScriptableObjektien luominen

Tasoille luodaan uusi luokka, joka periytyy ScriptableObjectista (Kuva 17). Ennen luokkaa määritetään valikkopolku (Kuva 18), josta luodaan uusi taso. Luokan sisälle lisätään muuttujat kaikille tason tallennettaville tiedoille. Tasoeditorin kannalta keskeisin on layout lista, johon tallennetaan tason asettelu soluissa.

```
[CreateAssetMenu(fileName = "Level", menuName = "LevelEditor/New level", order = 1)]
public class Level : ScriptableObject
{
    [JsonProperty]
    [SerializeField]
    public double bnc = 0.7;

    [JsonProperty]
    [SerializeField]
    public int color = 3;

    [SerializeField]
    public List<List<int>> layout = new List<List<int>>();

    [JsonProperty]
    [SerializeField]
    public string name;

    [JsonProperty]
    [SerializeField]
    public int sink = 0;

    [JsonProperty]
    [SerializeField]
    public int[] style = { 12, 3, 1, 6, 5, 8, 11, 16, 2, 10, 7, 4, 13, 15, 9, 14, 17, 18 };

    public void SetLayout(List<List<int>> Layout)
    {
        layout = Layout;
    }
}
```

Kuva 17 ScriptableObject joka säilöö tason datan.



Kuva 18 Uuden tason luomista varten tehty valikkopolku.

Myös tasosetit tallennetaan ScriptableObjekteina. Tasosetti luokkaan tallennetaan lista kaikista siihen kuuluvista tasoista.

4.5.2 Tallentaminen JSON-tiedostona

Tasosetin tarkkailuikkunaan lisätään painike, joka kutsuu SaveLevelSet funktiota (Kuva 19). Funktiossa tasosetti tallennetaan Json-tiedostoksi. If-lauseiden avulla lisätään oikeat merkit oikeisiin kohtiin, jotta tiedoston rakenne vastaa täydellisesti aikaisemmin luotujen tasojen rakennetta.

```
[CustomEditor(typeof(LevelSet))]
public class SaveLevelSet : Editor
{
    public override void OnInspectorGUI()
    {
        if (GUILayout.Button("Save LevelSet"))
        {
            SaveLevelset((LevelSet)target);
        }

        DrawDefaultInspector();
    }

    public void SaveLevelset(LevelSet levelSet)
    {
        levelSet.SetLevel();
        if (File.Exists(Application.dataPath + "/" + levelSet.name + ".json"))
        {
            File.Delete(Application.dataPath + "/" + levelSet.name + ".json");
        }

        for (int i = 0; i < levelSet.allLevels.Count; i++)
        {
            string json = JsonConvert.SerializeObject(levelSet.levelsave[i]);
            Debug.Log(json);
            if (i == 0)
            {
                File.AppendAllText(Application.dataPath + "/" + levelSet.name + ".json", "[" + json + "," );

            }
            else if (i == levelSet.allLevels.Count - 1)
            {
                File.AppendAllText(Application.dataPath + "/" + levelSet.name + ".json", json + "]");
            }
            else
            {
                File.AppendAllText(Application.dataPath + "/" + levelSet.name + ".json", json + ",");
            }

        }
    }
}
```

Kuva 19 Luokka, jossa tasosetit tasoineen tallennetaan JSON-tiedostona.

Json-tiedostoon tieto solussa sijaitsevasta objektista tallennetaan numeroarvona. Tyhjän solun arvo on 0, pallon arvo on 1 ja esteen arvo 2. Tätä varten LevelEditor luokkaan luodaan funktio nimeltä SetMap. SetMap funktiota kutsutaan aina kun tasoeditorin pohjaan tehdään muutoksia. Funktiossa tarkistetaan if-ehtolauseiden avulla mikä objekti kyseissä solussa on ja tallennetaan savelevel listaan sen arvo.

LevelEditor luokkaan luodaan myös funktiot DrawList ja GetLevels. DrawList funktiossa tarkistetaan, onko tasolle jo aikaisemmin tallennettua tiedostoa. Jos tiedosto löytyy, kutsutaan GetLevels funktiota. GetLevels funktiossa luetaan tason tiedot tiedostosta ja käännetään ne takaisin listaksi, josta tallennettu taso piirretään tasoeditorin pohjalle sen auetessa. Jos tasolla ei ole tallennettua tiedostoa kutsutaan SetUpList funktiota, jossa luodaan tyhjä lista uudelle tasolle.

```
private void DrawList()
{
    if(File.Exists(Application.dataPath + "/" + level.name + ".text"))
    {
        GetLevels();
    }
    else
    {
        SetUpList();
    }
}
public void GetLevels()
{
    if (count == 0 && File.Exists(Application.dataPath + "/" + level.name + ".text"))
    {
        string json = File.ReadAllText(Application.dataPath + "/" + level.name + ".text");
        savelevel = JsonConvert.DeserializeObject<List<SaveLevelmap>>(json);
    }
}
```

Kuva 20 Funktio jonka avulla tallennettu JSON-tiedosto puretaan tasoeditorissa näytettäväksi tasoksi.

Toisen sprintin tuotoksena tasoeditorilla on mahdollista tallentaa tasosetit JSON-tiedostona ja keskeneräiset sessiot säilyvät myös tasoeditorin sulkeutuessa.

4.6 Sprint 3

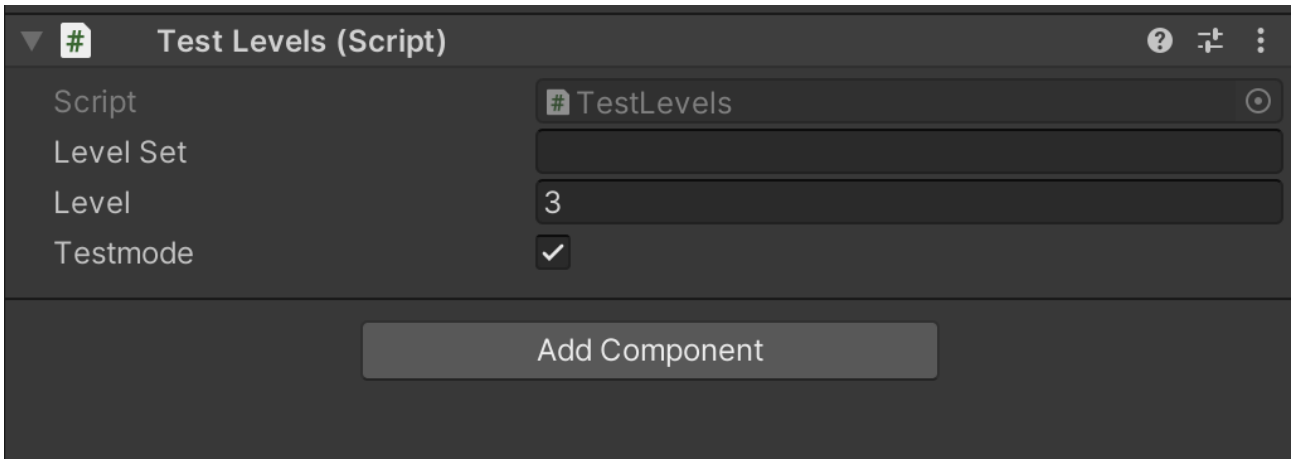
Kolmannen sprintin tavoitteena on testausominaisuus, projektin viimeistely ja tasoeditorin lopputestaus ja julkaisu.

4.6.1 Tason testaaminen

Luon uuden scriptin testausta varten ja lisään siihen aktivointipainikkeen, jonka avulla testauksen voi kytkeä päälle tai pois. Scriptiin lisätään myös tasosetti ja taso muuttujat, joiden avulla käyttäjä voi määrittää mitä haluaa testata. Tasosetti kertoo mitä tasoja halutaan käyttää ja taso muuttujan arvo määrittää mistä tasosta peli aloittaa testauksen aikana. Seuraavaksi muokkaan pelin koodia if-lausekkeiden avulla. Jos testipainike on aktivoituna niin pelin koodi etsii käyttäjän määrittämän tasosetin ja aloittaa määritetystä tasosta, kun editorissa toimiva peli laitetaan päälle. Tasoja läpäistessä koodi etsii seuraavan tason testattavasta tasosetistä, jolloin käyttäjän ei tarvitse valita testattavia tasoja yksi kerrallaan.

```
[CreateAssetMenu(fileName = "filename", menuName = "TestLevel" )]
public class TestLevel : ScriptableObject
{
    [SerializeField]
    public string LevelSet;

    [SerializeField]
    public int Level;

    [SerializeField]
    public bool testmode;

}
```

Kuva 11 TestLevel scripti.



Kuva 12 Testausominaisuus scriptin tarkasteluikkuna.

4.6.2 Projektin viimeistely

Testaustoiminnon jälkeen siivoan projektin ja teen muut viimeistelyt, jonka jälkeen tasoeditori on tämän projektin osalta valmis. Siistimiseen kului jonkin verran aikaa, sillä olin toimintoja luodessa kokeillut erilaisia keinoja, joista oli jäänyt ylimääräisiä resursseja, koodinpätkiä ja kommentteja. Valmiiseen tuotokseen tehdään vielä lopputestaus. Kaikkia ominaisuuksia on testattu erikseen niiden valmistuessa, mutta lopuksi on hyvä testata kokonaisuuden toimivuus. Kokonaisuutta testataan myös käyttämällä sitä ”väärin” ja isommalla kuormituksella. Tällöin voidaan testata, että ratkaisu toimii kaikissa eri käyttötapauksissa ja ettei käyttäjä pysty vahingossa hajottamaan toimintoja. Koska myös pelin koodia muokattiin, pitää testata, ettei muutokset aiheuttaneet ongelmia pelin toimivuuteen. Pelistä buildataan uusi versio testiohjelmaan, jonka jälkeen testataan julkaisumuodossa olevan pelin toimivuus.

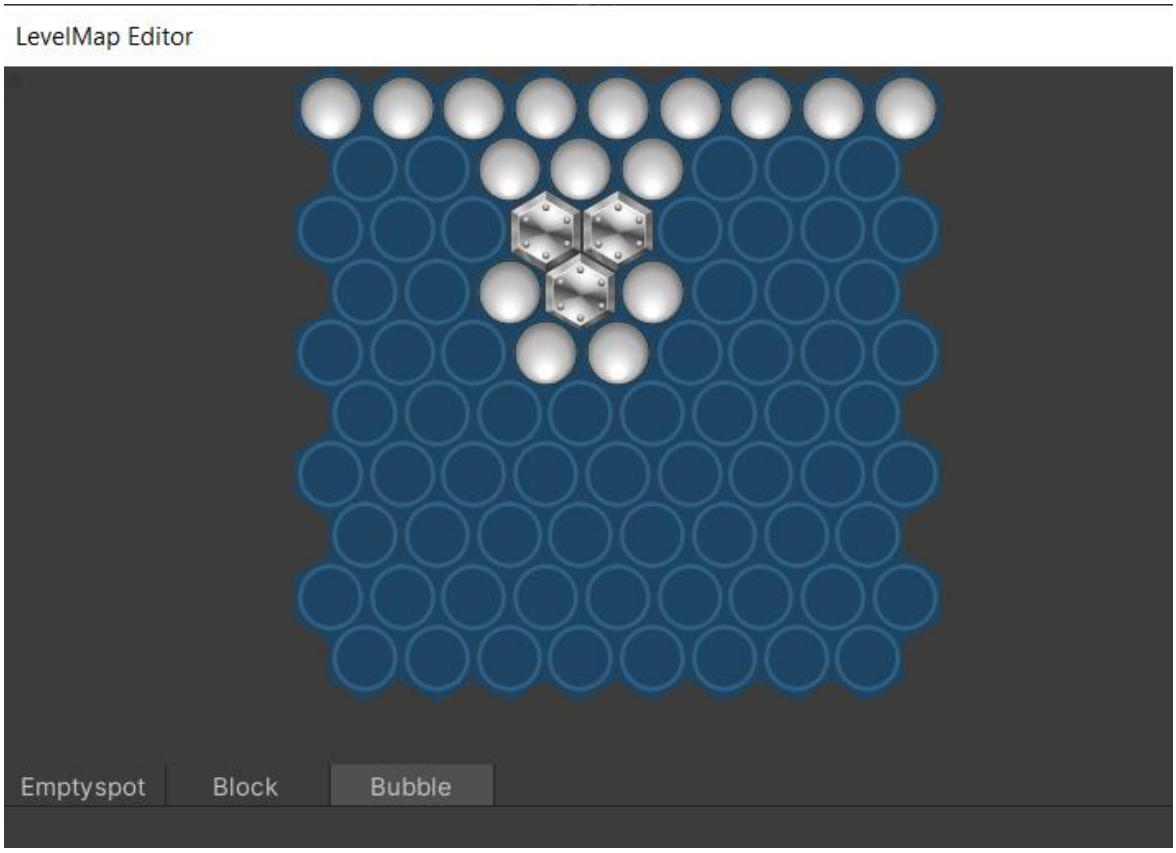
Tasoeditori julkaistaan puskemalla se GitHub versionhallintajärjestelmään, josta muut työntekijä saavat sen myös projekteihinsa. Pusken muutokset versionhallintaan komentorivityökalun avulla. Olen käyttänyt Githubia paljon ja yleensä muutosten puskeminen on hyvin yksinkertaista. Välillä kuitenkin muiden henkilöiden lisäykset projektiin saattavat aiheuttaa yhdistämiskonflikteja. Tasoeditorin julkaiseminen onnistuu kuitenkin ilman ongelmia ja olen saanut toteutuksen tämän projektin osalta valmiiksi.

5 Pohdinta

Työskentelemässäni yrityksessä oli tarve 2D-monialustapelin tasoeditorille. Huomasin hyvän oppimismahdollisuuden ja tarjouduin ottamaan sen tehtäväkseni. Tavoitteena oli kuuden viikon aikana toteutettava tasoeditori-työkalu, jonka avulla on mahdollista luoda, muokata ja tallentaa uusia tasoja. Jaoin ajan kahden viikon sprintteihin, joiden aikana onnistuin luomaan tarpeeseen vastaavan tasoeditorin.

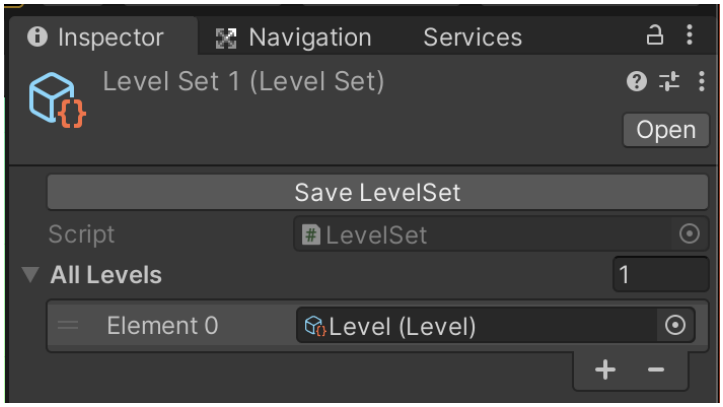
Toteutus toimii täysin Unity-pelimoottorin sisällä ja on liitetty sen käyttöliittymään. Tasoeditori yhdistettiin pelin Unity-projektiin. Pelin Asset kansiossa käyttäjä pystyy luomaan uusia tasosettejä ja tasoja. Kuvassa 13 voidaan nähdä tason tarkkailuikkuna, jossa tason tietoja voidaan muokata. Monissa tasoissa käytetään samoja arvoja, joten laitoin kaikille valmiit oletusarvot työskentelyn nopeuttamiseksi. Tarkkailuikkunasta löytyy myös painike, jonka avulla kuvassa 14 näkyvän muokkausikkunan saa auki. Kuvassa näkyy myös pallot ja esteet, joiden avulla tason kenttä rakennetaan. Objekteja voi lisätä ja poistaa kentältä painamalla tai raahaamalla hiirtä solujen päällä.

Kuva 13 Tason tarkkailuikkuna.



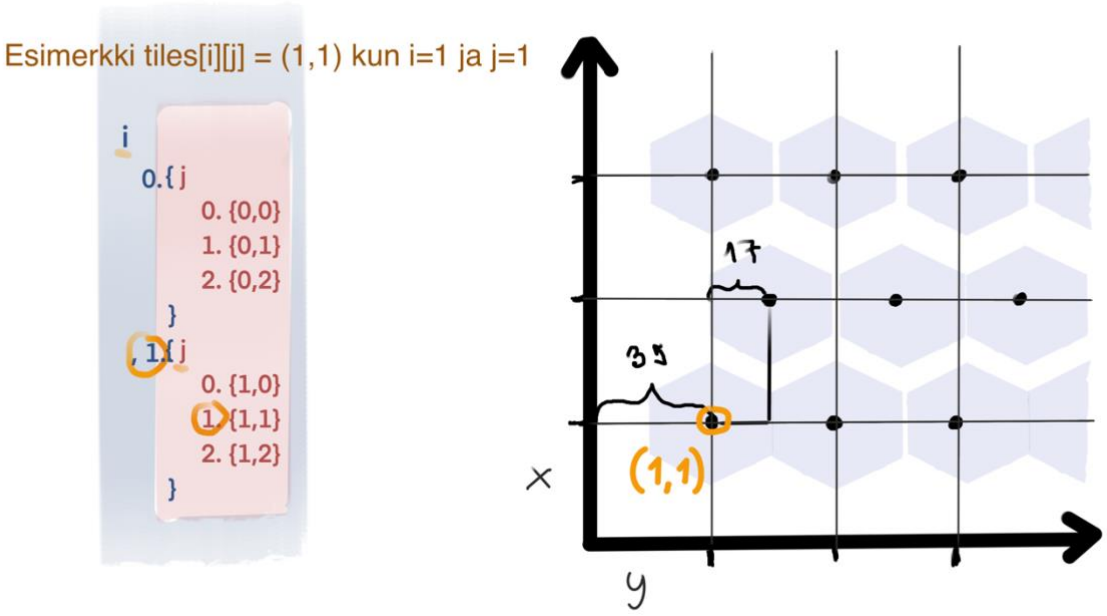
Kuva 14 Tason muokkausikkuna.

Valmiit tasot lisätään haluttuun tasosettiin. Tasosetin tarkkailuikkunassa (Kuva 15) on mahdollisuus muokata tasolistaa lisäämällä ja poistamalla tasoja. Käyttäjä pystyy myös halutessaan muokaamaan helposti tasojen järjestystä. Tämä on tärkeä ominaisuus, sillä järjestys määrää myös järjestyksen pelissä. Tarkkailuikkunasta löytyy myös painike, jota painamalla valmis tasosetti tallennetaan JSON-tiedostona. Testausominaisuudella on myös oma tarkkailuikkuna. Ikkunassa on pois/päälle-painike, jonka avulla testaustilan voi laittaa päälle. Testausikkunaan syötetään testattava tasosetti ja tasonumero, josta halutaan pelin aloittavan.



Kuva 15 Tasosetin tarkkailuikkuna.

Työssäni olen tehnyt paljon koodin lukemista ja bugien korjaamista, mutta en ole saanut lainkaan kosketusta kokonaisten toimintojen tai koodin luomiseen C# kielellä. Aikaisemmista oppimiskokemuksista saadun varmuuden avulla uskoin pystyväni tasoeditorin luomiseen, mutta tiesin ettei se olisi täysin mutkatonta. Yksi suurimmista haasteista oli saada pelikentän solut asetettua oikeille paikoille. Aloitin kentän toteutuksen luomalla sisäkkäisen listan, joka sisälsi solujen sijainnit x ja y akseleilla. Seuraavaksi loin funktiot, joissa listan avulla solut piirretään muokkausikkunaan. Näitä funktioita luodessa oli haastavaa miettiä miten listan saa if-ehtolauseiden avulla avattua niin, että oikea määrä soluja asettuu oikeille paikoille. Tämän luominen vaati paljon aikaa ja kokeilua. Opin, että itselleni erittäin hyvä tapa hahmottaa asia on piirtää siitä konkreettinen esimerkki. Piirsin itselleni esimerkin listan arvoista akseleilla, ja sain lopulta solut asetettua oikeille paikoilleen (Kuva 16).



Kuva 16 Piirretty esimerkki listasta ja sen pisteistä x- ja y-akselilla.

Toinen suuri haaste oli toisen sprintin tallennusominaisuuksien luominen. Niitä tehdessä jouduin kokeilemaan monia eri tapoja ennen, kun löysin toimivan ratkaisun. Useimmat keinot eivät toimineet, koska en saanut jotain tarvittavaa tietoa toisista luokista. Tämän aikana opin paljon luokkien välisistä suhteista ja Unityn ScriptableObjecteista. Uskon näistä opeista olevan paljon hyötyä tulevaisuuden tehtävissäkin. Huomasin oman kehitykseni jo projektin aikana, kun kolmannen sprintin testausominaisuuden luominen onnistui ilman suurempia haasteita. Kolmannelle sprintille oli huomattavasti vähemmän tekemistä kuin kahdelle aikaisemmalle. Olin kuitenkin tyytyväinen ajoitukseeni, sillä minulle jäi tarpeeksi aikaa huolelliselle viimeistelylle ja testaukselle. Nyt tasoeditori on otettu jo käyttöön ja toimii tavoitteiden mukaisesti.

Mielestäni onnistuin tasoeditorin osalta projektissa mainiosti ja olen erittäin tyytyväinen omaan työskentelyyni. Aikataulullisia ongelmia esiintyi kuitenkin opinnäytetyön kirjoittamisen kanssa elämäntilanteen ja suunnittelun vuoksi. Aikataulusuunnitelmassa keskityin liikaa pelkästään tasoeditorin toteuttamiseen, enkä suunnitellut tarpeeksi kirjoittamiseen käytettävää aikaa. Asiaan vaikutti myös paljon se, että tein samanaikaisesti kokopäiväisesti töitä. Tasoeditoria pystyin tekemään työajallani, sillä se kuului työtehtäviini, mutta opinnäytetyön kirjoittaminen piti tehdä vapaa-ajalla. Vapaina aikoina minun oli erittäin vaikea löytää ajatuksia, joiden avulla aloittaisin kirjoittamisen. Opin kuitenkin hyvin konkreettisesti, että parhaiten pääsee vauhtiin, kun vain ryhtyy työskentelemään, vaikka ajatukset eivät olisikaan täysin kasassa. Heti kun pääsee alkukynnyksen yli, työnteko sujuu huomattavasti helpommin. Uskon tämän olevankin yksi opinnäytetyöprojektin tärkeistä opetuksista. Vaikka kirjoittamisen aikataulu venyi huomattavasti, olen tyytyväinen oppimiini asioihin

Koska opin paljon uusia asioita, saattaa koodi myös joissain kohdissa näyttää siltä. Jotkin funktiot sisältävät useita ehtolausekkeita, jotka voisi tulevaisuudessa pilkkoa pienempiin omiin funktioihin. Jatkokehityksessä voisi lisätä myös käyttäjäystävällisyyttä. Mielestäni olisi hyvä, jos kaikki tasosetin tasot näkyisivät listana itse muokkausikkunassa. Tällöin käyttäjän olisi helpompi vaihdella tasojen välillä ja työskentely nopeutuisi. Tällä hetkellä jokaisen tason nimen tulee olla uniikki riippumatta mihin tasosettiin se kuuluu. Tämä voi aiheuttaa ongelmia, jos tasoja on paljon ja useampi ihminen työskentelee niiden parissa. Tällä hetkellä ongelma on ratkaistu yhteisillä nimeämiskäytännöillä, mutta tulevaisuudessa siihen voisi keksiä varmasti turvallisemman ratkaisun. Jatkokehitystä ei ole vielä suunnitteilla, mutta aion varmasti jatkaa sitä, jos työssäni löytyy siihen mahdollisuus.

Lähteet

Ekholm, J. & Lehtonen, U. 2021. Miksi ketterä kehittäminen on tärkeää sinunkin organisaatiollesi – mistä on kysymys ja kuinka pääset alkuun. Luettavissa: https://futurice.com/blog/miksi-kettera-kehittaminen-on-tarkeaa-sinunkin-organisaatiollesi

Frankel, A. 2017. Avoid CodeThinking – The Pitfalls of Trial and Error Coding. Luettavissa: https://www.solutionstreet.com/blog/2017/10/27/avoid-code-thinking-the-pitfalls-of-trial-and-error-coding/

Full Scale blogipostaus. 2022. Everything you need to know about software development. Luettavissa: https://fullscale.io/blog/everything-about-software-development/

Gautam, S. 2023. Thorndike's Trial and Error Theory | Learning | Psychology. Luettavissa: https://www.psychologydiscussion.net/learning/learning-theory/thorndikes-trial-and-error-theory-learning-psychology/13469

GitHub dokumentaatio. 2023. About projects. Luettavissa: https://docs.github.com/en/issues/planning-and-tracking-with-projects/learning-about-projects/about-projects

Halme, A. Ohjelmistokehitys Luettavissa: https://www.itewiki.fi/opas/ohjelmistokehitys/

Hietaniemi, J. 2019. Scrum pähkinänkuoressa. Luettavissa: https://gofore.com/scrum-pahkinankuoressa/

Hurja. 2021. UX- ja UI-suunnittelu – mitä ne ovat ja mikä rooli niillä on verkkosivu- ja ohjelmistoprojektissa? Luettavissa: https://www.hurja.fi/blogi/ux-ja-ui-suunnittelu-mita-ne-ovat/

Kinsta. 2022. What Is GitHub? A Beginner's Introduction to GitHub. Luettavissa: https://kinsta.com/knowledgebase/what-is-github/

Korkala, M. 2020. Ketterä kehittäminen – menetelmiä vai mielenmaisemia? Luettavissa: https://www.eficode.com/fi/blog/kettera-kehittaminen-agile-menetelmia-vai-mielenmaisemia

Koulutus.fi. 2021. Mitä ovat ketterät menetelmät? – Scrum, Lean ja muut tutuksi. Luettavissa: https://www.koulutus.fi/oppaat/projektinhallinta/ketteratmenetelmat-19939

Laine, H. 1998. Ohjelmiston suunnittelu. Luettavissa: https://www.cs.helsinki.fi/u/laine/ot/s98/suunnittelu1.pdf

Matilainen, M. 2022. Millainen on hyvä vaatimusmäärittely hubspotin hankintaa varten? Luettavissa: https://www.salescommunications.fi/blog/millainen-on-hyva-vaatimusmaarittely

Microsoft 365 Team. 2019. Luettavissa: https://www.microsoft.com/fi-fi/microsoft-365/business-insights-ideas/resources/guide-to-uml-diagramming-and-database-modeling

Microsoft. 2023. Visual studio. Luettavissa: https://visualstudio.microsoft.com

Unity user manual 2021.3 (LTS). Editor Windows. Luettavissa: https://docs.unity3d.com/Manual/editor-EditorWindows.html

Unity user manual 2021.3 (LTS). EditorWindoe.OnGui(). Luettavissa: https://docs.unity3d.com/ScriptReference/EditorWindow.OnGUI.html

Unity user manual 2021.3 (LTS). Extending the Editor. Luettavissa: https://docs.unity3d.com/Manual/ExtendingTheEditor.html

Unity user manual 2021.3 (LTS). Rect. Luettavissa: https://docs.unity3d.com/ScriptReference/Rect.html

Unity user manual 2021.3 (LTS). ScriptableObject.OnEnable(). Luettavissa: https://docs.unity3d.com/ScriptReference/ScriptableObject.OnEnable.html

Valerio, R. 2019. How to make a Unity Editor Window (with example code). Luettavissa: https://ricardo-valerio.medium.com/how-to-make-a-unity-editor-window-3cd05440c6c0

W3schools opeturmateriaali. 2023. C# For loop. Luettavissa: https://www.w3schools.com/cs/cs_for_loop.php

Liitteet

Liite 1/1. Vaatimusmäärittely

New Bubbles IQ Level Editor

A tool to create new and modify old level sets. Level sets contain 60 levels, but there should be possibility to have more or less. Level sets are saved in json format.

Features:

- Load level sets
- Create new level sets
- Create and modify level layouts
- Save layout
- Set level version number
- Set level difficulty 1-10
- Testing
- Change level parameters

Parameters:

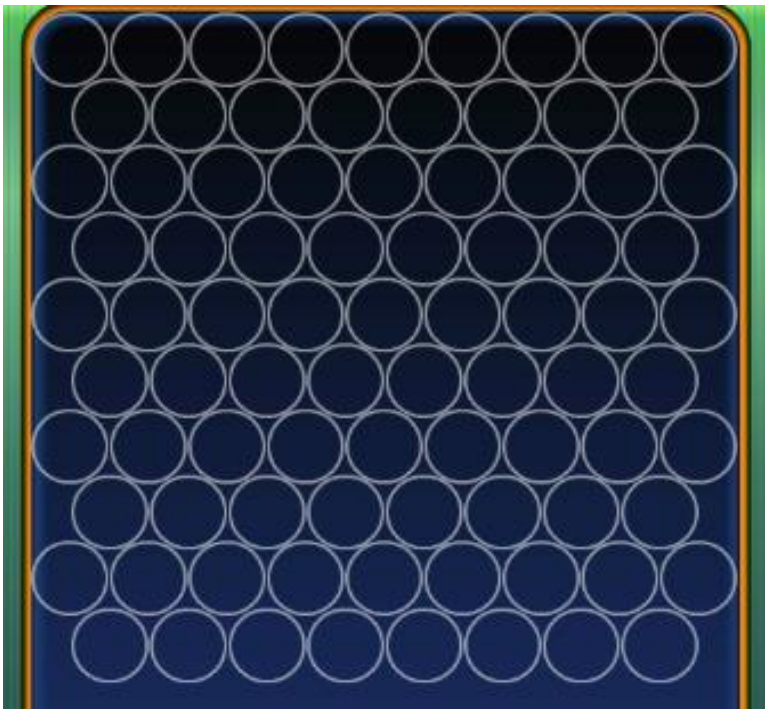
COLORS: Most important parameter. It will directly affect the difficulty of the level. Minimum color amount is 3, max is 16.

BNC: this value determines how much the same colored bubbles "clump" together. Sticking to 0.7 is wise. In rare cases this value can be changed to something else, but then the level needs extensive testing.

(SINK: Determines how much bubbles would sink down after every shot ball. This parameter was deprecated and not used in the old editor. It should be set on 0 and leave if it’s needed later.)

Liite 1/2. Vaatimusmäärittely

Layouts are created on a grid that includes 85 spots.
Height is 10 rows. Every other row is 9 spots and every other is 8 spots.
Layouts are created with bubbles and shields. Shields cannot be placed in the top row.

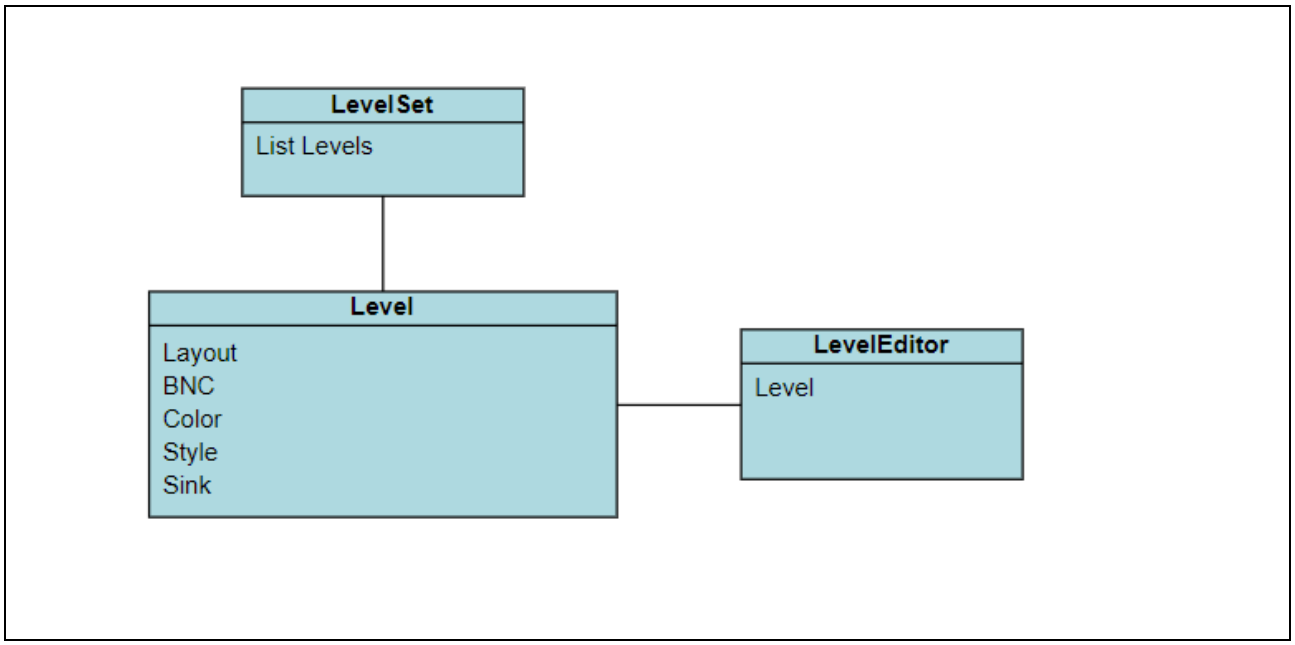


Liite 1/3. Vaatimusmäärittely

- Save in json format

An example of level data:
{
"bnc": 0.7,
"colors": 3,
"layout": [
[0,0,0,1,0,1,0,0,0],
[0,0,0,1,1,0,0,0,3],
[0,0,1,1,1,1,1,0,0],
[0,0,0,1,1,0,0,0,3],
[0,0,0,1,0,1,0,0,0],
[0,0,0,0,0,0,0,0,3],
[0,0,0,0,0,0,0,0,0],
[0,0,0,0,0,0,0,0,3],
[0,0,0,0,0,0,0,0,0],
[0,0,0,0,0,0,0,0,3]
],
"name": "STELLAR",
"sink": 0,
"style": [3,5,8,16,11,9,14,10,4,15,1,2,6,7,12,13,17,18]
}

Relations:


LevelSet
List Levels
Level
Layout
BNC
Color
Style
Sink
LevelEditor
Level

Liite 2. Tasoeditorin Github projekti, josta löytyy tuotoksen kooditiedostot

https://github.com/sallasalmi/LevelEditor/tree/af05321a99e2f3ec8fdf2720f2903dc2f27b758b/My%20project/Assets/LevelEditor