Ville Kajander

Uudelleenkäytettäviä ominaisuuksia ja koodipohjia juoksupeliin

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tieto- viestintätekniikka tutkinto-ohjelma

Insinöörityö

13.12.2022

Tiivistelmä

Tekijä:	Ville Kajander
Otsikko:	Uudelleenkäytettäviä ominaisuuksia ja koodipohjia juoksupeliin
Sivumäärä:	62 sivua
Aika:	13.12.2022
Tutkinto:	Insinööri (AMK)
Tutkinto-ohjelma:	Tieto- ja viestintätekniikka
Ammatillinen pääaine:	Pelisovellukset
Ohjaajat:	Lehtori Antti Laiho Toimitusjohtaja Juha Huhtakallio

---

Insinöörityön tarkoituksena oli tutkia hyper-casual-pelityypin juoksupelien alaluokkaa ja luoda vahva koodipohja tälle genrelle niin, että juoksupelejä pystyisi tekemään mahdollisimman helposti koodipohjaa käyttäen. Insinöörityö toteutettiin yritykselle, jossa toteutetaan mobiili- ja hyper-casual-pelejä. Yksi toteutettavista peleistä oli juoksupeli, joka toimi insinöörityön testipelinä. Testipelissä testattiin toteutuksien toimivuutta.

Työssä havaittiin, että ryhmäsyntyminen tulee olemaan iso osa tämänhetkistä juoksupeligenreä ja että yksi mainontatyyli näkyy ylitse muiden: siinä pelaaja palkitaan mainoksen katsomisesta.

Insinöörintyössä toteutettiin uusi juoksupeli alusta loppuun. Tarkoituksena oli käyttää insinöörityössä tehtyjä toteutuksia ja nähdä niiden toimivuus käytännössä. Juoksutestipelissä keskityttiin vain koodilliseen toteutukseen, eli grafiikka oli minimaalista. Testipelissä huomattiin joidenkin toteutuksien puutteellisuus, ja ne korjattiin. Yksittäisten toteutuksien esillepanossa mitattiin aikaa ja tulokset kirjattiin taulukkoon eri vaiheista. Taulukosta kävi ilmi, että suurin osa toteutuksien esillepanoista kesti vain tunnin, mutta itse juoksupelin ytimen suunnittelu ja toteutuksien puutteet pidensivät testipelin tuotantoaikaa. Testipeliä tehdessä selvisi, että toteutukset todella toimivat ilman koodillista muokkausta ja niiden avulla pystyttiin luomaan kohtuullinen juoksupeli. Testijuoksupelin tekemiseen meni 11,5 tuntia, mikä oli eriomainen tulos.

Testijuoksupelissä ilmeneviä ongelmia pohdittiin, ja tarkasteltiin kriittisesti myös koodillisia toteutuksia. Huomattiin, että ne eivät kuitenkaan vielä sovellu muille kehittäjille ilman järkevämpää dokumentaatiota. Koodipohjan tekemiseen käytetty aika oli tuloksiin nähden järkevä. Koodillisten sovellutuksien pohja on hyvä jatkokehitykselle, ja insinöörityön tarkoitus täyttyi.

Avainsanat: juoksupeli, hyper-casual, koodipohja

Abstract

Author:	Ville Kajander
Title:	Runner’s long-term features and code developments
Number of Pages:	62 pages
Date:	13 December 2022
Degree:	Bachelor of Engineering
Degree Programme:	Information and communications technology
Professional Major:	Game applications
Supervisors:	Antti Laiho, Senior Lecturer
	Juha Huhtakallio, CEO

The goal of the thesis was to study a hyper-casual subclass genre called runner game and to make a code toolbox for it. The purpose of the toolbox was to make it easier to develop runner games. Also, developers did not need to rewrite the code a lot. The thesis was done for a company where mobile and hyper-casual games are developed. One of the hyper-casual games was a runner like game which worked as a test game for this thesis.

This study showed that the group spawning mechanics and one of ad types called rewarded ad had been the most wanted features for a couple of years.

As a result of this project, a new test runner game was made. The purpose of the test game was to use the code implementations of this study in practice. The main focus of the test game was the code and not the graphics, which were minimal. Part of the code was found to be insufficient when developing test game and it had to be fixed. Each individual code implementation was clocked and written down to table. Mostly, the code implementation development time was under one hour. However, some core features and insufficiencies took most of the development time. It turned out that code implementations worked well without touching the code which enabled the test game to be developed. The test runner game was made in 11,5 hours which was a good result.

Problems that were noticed during the development phase were taken up for closer look. It turned out that some code implementations were not ready for other developers to use without better documents. The time used to make code implementations was worth the result. The code implementations were solid structure and could be successfully developed further.

Keywords: Hyper-Casual, runner game, code toolbox

Käsitteet

Prefab Prefab toimii objektille mallina. Yksi Unityn tallennusmuodoista, joka mahdollistaa uusien samakaltaisen objektien luonnin helposti.

Objekti Objekti on esine tai asia pelissä.

Hyper-Casual Hyper-Casual on mobiilipeligenre, joka on yleisesti hyvin helppo pelata ja sopii kaikenikäisille.

ScribtableObject Unityssä oleva objektin tallennusmuoto, joka pysyy scenejenkin välillä ja vähentää mustin käyttöä.

Collider Collider on fysiikan kielessä törmäytin. Collidereita käytetään Unity-pelimoottorissa fysiikoiden laskemiseen esimerkiksi törmäyksissä.

Sisällys

1 Johdanto

Insinöörityössä tutkitaan ja toteutetaan Playstack-nimisessä yrityksessä hyper-casual-pelejä ja keskitytään niiden alaluokkaan nimeltä juoksupeli. Tarkoitus on tutkia juoksupelien toteutustapoja ja sitä, mitkä ovat oleellinen osa itse genreä. Työssä pyritään luomaan vahva koodipohja samantyyppisille peleille, jotta jatkossa saman pelin tekemiseen koodillisesti ei menisi liikaa aikaa.

Ensin tutustutaan itse juoksupeliin genrenä ja siihen, miten lähdetään liikkeelle rakennettaessa juoksutyyppistä peliä. Tarkoitus on pelata juoksupelejä, jotka löytyvät Google Play -kauppapaikasta. Lisäksi tarkastellaan historiaa ja sitä, mistä juoksupelit ovat saaneet alkunsa ja mitkä ovat pelille tärkeitä ja hyödyllisiä mekanismeja, jotka auttavat jatkossa pelityylin rakentamisessa.

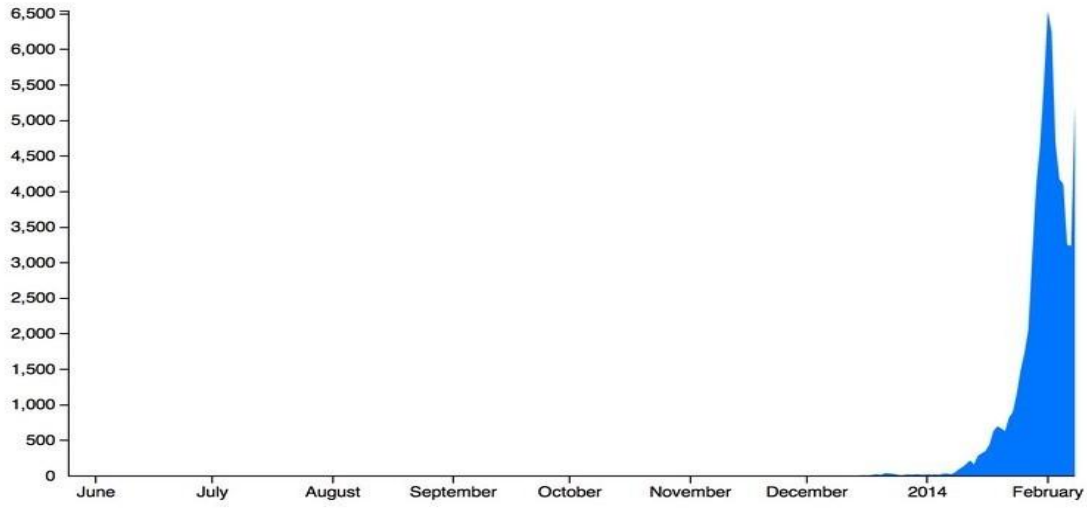
Playtackissä työskentely antaa hyvän käsityksen siitä, kuinka rakentaa hyper-casual-pelejä. Free the Fish -juoksupeli on tämän insinöörityön testipeli. Peliä kehittäessä tarkastellaan eri kehitysvaiheita ja niiden vaikutuksia. Koodilliset toteutukset tuotetaan testipelin sisään. Pelit toteutetaan noin kolmen hengen tiimeissä. Rakennettaessa juoksupeliä päästään näkemään, mikä osa-alue vie eniten aikaa.

Kun koodilliset toteutukset ovat valmiita, luodaan testijuoksupeli. Luotuja toteutuksia ja niiden ominaisuuksia testataan testipelissä. Toteutukset käydään kriittisesti läpi: ovatko ne todella hyödyllisiä juoksugenrelle ja mitkä ovat niiden puutteet. Testipelissä mitataan, kuinka paljon aikaa eri tavoitteiden saavuttamiseen ja itse peliin kuuluu. Pohditaan myös suhdetta toteutuksissa kuluneeseen aikaan ja mitä sillä saavutettiin. Lisäksi vertaillaan toteutuksien yleistä hyödyllisyyttä muissa peleissä, pystyykö toteutuksia käyttämään muissa peleissä vai ovatko ne lukittuja itse juoksupeliin.

2 Hyper-casual genrenä

2.1 Hyper-casual-pelien historiaa

Hyper-casual-pelit ovat aina olleet pelialassa mukana, mutta ne ovat saaneet oman genrenimensä vuonna 2017. Johannes Heinze julkaisi ensimmäisen artikkelin “The Ascendance of Hyper-casual", ja siitä lähtien pelejä on kutsuttu hyper-casualeiksi, vaikka niiden suosio oli nousussa jo vuodesta 2013. [2.] Ensimmäinen mieleenpainuva hyper-casual-peli ympäri maailmaa oli Dong Nguyen kehittämä Flappy Bird. Se julkaistiin vuonna 2013. Suosio ei ollut merkittävää vuoteen, mutta peli sai pientä suosiota jo alkuvuodesta 2014. Siitä alkoi räjähdysmäinen suosio vuoden sisällä, mikä voidaan nähdä kuvassa 1. Suosio oli niin suurta, että itse tekijä päätti ajaa pelin alas sen koukuttavuuden takia ja omista henkilökohtaisista syistään. [2; 3.]



Kuva 1. Flappy Bird -pelin latausmäärien nousu 2014 [1].

Flappy Birdiä oli ladattu yli 50 miljoonaa kertaa, mikä teki siitä suosituimman mobiilipelin vuonna 2014. Itse peli oli ilmainen, eikä siihen liittynyt mitään mikromaksuja ja se tuotti 50 000 dollaria päivässä vain mainostuloilla [5]. Flappy Birdin tuomat tulot, joita myös mainostettiin lehdissä, saivat pelinkehittäjät katsomaan uudelta kannalta hyper-casual-genreä. Pelinkehittäjät kieltäytyivät

sanomasta näitä mainospelejä peleiksi ja väheksyivät aiemmin hyper-casual-pelejä, koska ne saivat tulonsa mainoksia näyttämällä. [2.]

Kuitenkin hyper-casual-genre sai uutta suosiota tulevina vuosina. Yksinkertainen pelityyli, -malli ja -suunnittelu oli kehittäjien mieleen ja sai myönteistä huomiota pelinkehittäjäyhteisössä. Kehittäjät alkoivat toden teolla tutustumaan hyper-casual-genreen suuren tulonlähteen houkuttamana huomattuaan kysynnän olevan suurta tämänkaltaisille peleille. [2.]

Kesti kuitenkin useamman vuoden, kunnes pelisuunnittelijat saivat otteen genrestä ja sen uudenlaisesta mainostuloperiaatteesta, koska mainosten tuominen itse peliin oli oma taiteenlajinsa. Hyper-casual-pelien suosion voi nähdä siitä, että vuonna 2018 ladatuimmista peleistä 62 % oli hyper-casual-pelejä, mikä näkyy myös kuvassa 2. [2.]

Kuva 2. Ladatuimmat uudet pelit vuonna 2018 [2].

Kuvassa 2 voidaan huomata, että kaksi suosituinta uusinta peliä ovat latausmäärissä huomattavasti edellä muita pelejä. Yli 100 miljoonan latauskerran ero kolmanteen peliin on valtava. Voidaan todeta, että kun peli on tarpeeksi hyvä, niin suosio nostaa pelin latauskertoja eksponentiaalisesti.

2.2 Hyper-casual-pelien ominaisuudet

Hyper-casual-pelit ovat helposti lähestyttäviä mille vain ikäryhmälle. Niiden keskeinen piirre on minimaalisuus ja yksinkertaisuus. Niiden nimet ovat lyhyitä, houkuttelevia ja ytimekkäitä. Nimi pystyy jo helposti kertomaan pelin tärkeimmän idean. Houkuttava nimi ja sovelluksen kuvake saavat pelaajat lataamaan pelin, ja pelin voi päästä aloittamaan jo kymmenien sekuntien sisällä, eikä pelaajalle tavallisesti tarvitse antaa opastusta pelimekaniikoista yksinkertaisuuden takia. Pelissä ei ole vaikeita ohjauksia, valikoita tai isoja määriä mekaniikkoja. Perinteisesti hyper-casual-peleissä on vain yksi tai muutama mekaniikka, mikä jakaa helpon käyttökokemuksen. Pelihahmoksi yleensä tuodaan pelkistettyjä ja yksinkertaisia matemaattisia muotoja kuten kuutio, neliö tai pallo. Pelaajan on helppo ottaa peli pelattavaksi muutamien minuuttien ajaksi, koska pelin luonne on sellainen, että tasot kestävät vain noin minuutista viiteen minuuttiin. Koukuttavuus on iso osa peliä, ja yleensä käyttäjän tekee mieli edetä seuraavalle tasolle tai seuraavaan saavutukseen.

Pelin luokittaminen hyper-casualiksi voi välillä olla hankalaa, mutta hyvä muistisääntö on se, että jos peliä pystyy pelaamaan yhdellä sormella, se on tällöin aika varmasti hyper-casual-peli. Nämä pelit ovat hyviä lyhyen vapaa-ajan viihdykkeitä. Peliä esimerkiksi voi hyvin pelata bussimatkalla kouluun, harjoituksiin tai töihin mennessä. [6.] Pelien avulla päästään hetkeksi irti todellisuudesta. Pelinä hyper-casual on hyvin kannustava: vaikka pelissä pelaaja häviää, hän ei tunne aikaa juurikaan menetetyksi, koska tasojen pituudet ovat lyhyitä.

Mainoksien näyttämisellä on tärkeä rooli hyper-casual-peleissä: koska peli on muuten ilmainen, kehittäjien on saatava tulonsa jostain. Mainokset pyritään näyttämään yleensä tasojen alussa tai lopussa, ja niiden pituus on yleensä 30 sekuntia. Kehittäjät määräävät itse, kuinka paljon mainoksia näytetään. Liian

tiheä mainosten näyttäminen jättää pelaajalle vähemmän peliaikaa ja pelaamisesta tulee epämiellyttävää. Mainosten näyttämisille pitää löytää peliajan kanssa tasapaino. Nykyään hyper-casual-peleihin on tullut uusi mainosmekaniikka vanhan lisäksi, missä pelaajalle annetaan pelivaluuttaa tai edistymistä tai uusia elämäpisteitä, kun hän katsoo mainoksen. Tämä on molemmille osapuolille otollinen tilanne, koska kehittäjät saavat rahansa ja pelaaja edistymisensä. Pelaajalle annetaan valta, haluaako hän katsoa mainoksen vai ei.

Hyper-casual-pelejä on monenlaisia, mutta seuraavassa tarkastellaan niiden perusominaisuuksia. Peli 2048 on pulmapeli, jossa pelaaja pyrkii yhdistämään samanmerkkiset palikat yhteen, jolloin yhdistyessä niistä tulee uusi palikka, mikä on yhteenlaskettu summa yhdistyneistä palikoista. Haastavan pelistä tekee se, palikat siirtyvät siihen suuntaan, mihin pelaaja on ne pyyhkäissyt, vain jos niiden edessä on tyhjää tilaa. Pelaajan päämäärä on saada numero 2048 yhdeksi palikaksi. Kuvassa 3 näkyy, miltä 2048 näyttää pelatessa.

Kuva 3. Peli nimeltä 2048 [7].

Stack on reaktiopeli, jossa pelaaja pyrkii olemaan hyvin tarkka painalluksissaan. Uusi palikka alkaa aina joko vasemmalta tai oikealta ja vaihtuu painettaessa

toiseen puoliskoon. Kun pelaaja painaa näyttöä, jämähtää palikka siihen paikkaan ja se osa, joka ei ole pinon pintojen päällimmäisen pinnan ääriviivoilla, leikkautuu pois. Tämä johtaa siihen, että pelaaja ei koskaan pysty olemaan täydellisen tarkka ja pikkuhiljaa pino kärjistyy teräväksi. Pelaaja pyrkii saamaan palikoita pinoon mahdollisimman paljon. [8.] Kuvassa 4 nähdään Stack-peli, jossa on laitettu jo 12 palikkaa.

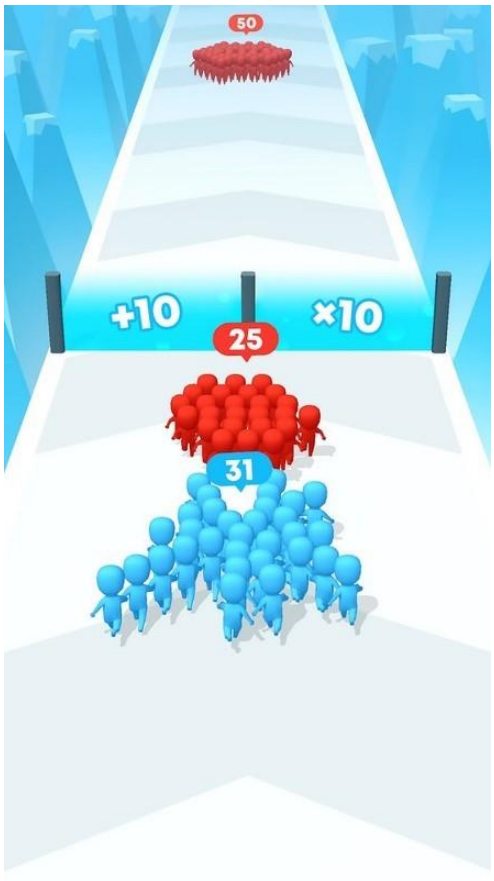
Kuva 4. Peli nimeltä Stack [8].

Flappy Bird on reaktio- ja väistöpeli. Siinä pelaaja antaa sormenpainalluksella voimaa ylöspäin pelaajan lintuun. Peli liikkuu koko ajan eteenpäin, menosuuntana oikea. Jokainen näpäytys sormella antaa pienen voiman ylöspäin, ja pelaaja pyrkii olemaan osumatta vihreisiin putkiin. Jokainen putken ylitys on aina yksi piste edistymistä. Peli jatkuu niin pitkään, kunnes pelaaja osuu tolppaan, putoaa maahan tai osuu kattoon. Dong Nguyen sanoi tämän pelin tekemiseen kuluneen aikaa 20 minuuttia. [9.] Kuvassa 5 nähdään Flappy Birdin pelimaailmaa.

Kuva 5. Peli nimeltä Flappy Bird [9].

Count Masters: Crowd Runner 3D on ketteryys-, väistely- ja juoksupeli. Se on yksi suosituimmista juoksupeleistä kirjoitushetkellä. Pelaaja ohjaa väkijoukkoa, jonka voimalukuun eli pelaajayksikköihin pelaaja vaikuttaa menemällä

matemaattisista porteista. Portit ovat joko laskevia tai nostavia. Pelaaja joutuu taistelemaan esteiden yli tason loppuun. Mitä isomman väkijoukon saa maalin asti, sitä suuremmat pisteet saa ja sen jälkeen pelaaja siirtyy seuraavaan tasoon. Tasot vaikeutuvat edetessä pelissä pidemmälle, ja vastaan voi tulla esimerkiksi päälliköitä, jotka vaativat suuren voimaluvun. Pelaaja pystyy ostamaan pelivaluutalla tai mainoksilla päivityksiä kuten pelaajavoiman alkukertoimia. Kuvassa 6 nähdään ryhmätappelu.



Kuva 6. Pelikuvaa Count Masters: Crowd Runner 3D [10].

Kun tarkastelee edellä mainittuja pelejä, voidaan huomata, että jokaisessa pelissä on vain yksi ydinmekaniikka. Ydinmekaniikka, joka on selkeä ja yksinkertainen.

2.3 Nouseva genretyyppi

Hyper-casual-pelien suosio on noussut jo yli 10 vuoden ajan [2]. Pelien helppo käyttöliittymä ja pelattavuus tuovat huonommatkin pelaajat mukaan genreen. Pelin helppous tuo mukaan myös pienen ikäryhmän pelaajia. Tämä tarkoittaa myös sitä, että ihmiset, jotka eivät kykene tai pysty pelaamaan haastavia pelejä, pääsevät nauttimaan pelaamisen ilosta. [11.] Pelit ovat niin yksinkertaisia, että ne soveltuvat myös kehitysvammaisille, ja sitä kautta ne voivat kehittää heidän reaktionopeuttaan, matemaattista ajattelukykyä tai vaikka lukemaan oppimista.

Toinen syy hyper-casual-pelien suosioon on niiden saatavuus. Tämä tarkoittaa sitä, että pelin kehittäminen on suhteellisen vaivatonta ja nopeaa [2]. Yksi tekijä tai muutaman hengen tiimi pystyy toteuttamaan jonkinlaisen hyper-casual-pelin jopa 1–4 viikon kuluessa. Tekijöiden taitotason ei tarvitse olla korkea, että he saavat tuotettua hyvän, hauskan ja tuottoisan pelin. Näin nopealla kehittämisellä pelimarkkinat täyttyvät nopeaan tahtiin. Nopealla kehityksellä saattaa olla myös yhteys suureen epäonnistumiseen, mikä tarkoittaa sitä, että pelin elinkaari voi olla muutamasta päivästä kuukauteen [12]. Lyhyellä ajanjaksolla ei ole kuitenkaan merkitystä, koska on ehditty saada jo uusi peli markkinoille. Jos peli-tiimi haluaa pitää pelin kuitenkin vielä elossa, pitää siihen tuoda lisää avattavaa sisältöä, kuten esimerkiksi saavutuksia, väripintoja, esineitä tai pelaajahahmon päivityksiä.

3 Juoksija-genren ominaisuuksia

3.1 Genren historia

Juoksija-pelityypin suosio on kasvanut vuosi vuodelta [13]. Juoksupelejä on ollut aina tai niiden variaatioita on esiintynyt peleissä pieninä osioina. Esimerkiksi voidaan ottaa Crash Bandicoot, joka julkaistiin vuonna 1996. Crashissä peli jo itsessään tuntui hyper-casual-juoksupeliltä monissa tasoissa. Näissä tasoissa ei

kuitenkaan ollut pakotettua juoksua eteenpäin kuin muutamissa kohdissa, esim. kun pelaaja hyppää pahkasian selkään väistellen esteitä kuten kuvassa 7. Tästä varmasti on myös lähtenyt vaikutteita nykyisiin juoksupeleihin, ja pakotetut eteenpäin liikkuvat juoksupelit ovat lähteneet nousuun vuodesta 2011 ja ovat olleet siitä lähtien huimassa nousussa. [13.]

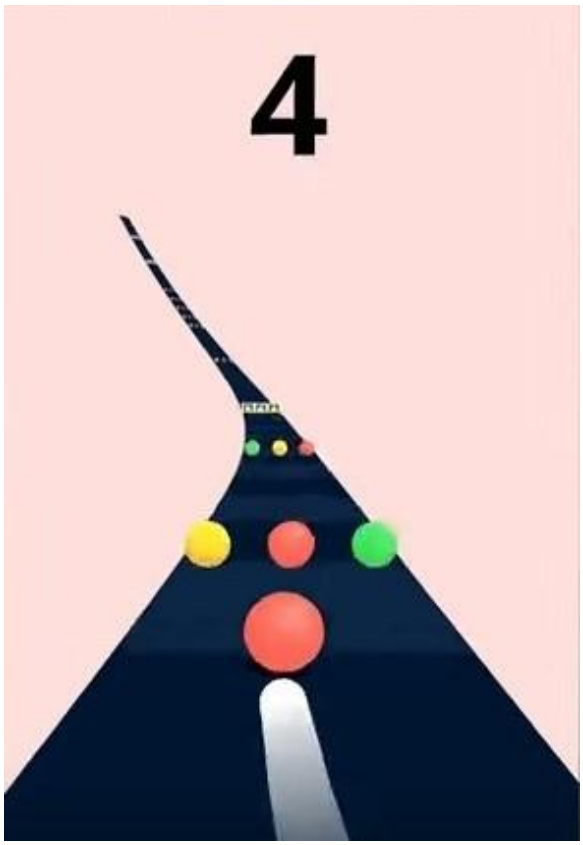
Kuva 7. Crash Bandicoot oli suosittu 2000-luvun molemmin puolin [15].

Moni muistaa hauskan juoksupelin Subway Surfers, jonka SYBO julkaisi vuonna 2011. Tämä hyper-casual-peli on pysynyt jo vuosikymmenen ajan ladatuimpana pelinä. [13.] Peli on pitkällä aikavälillä yksi menestyneimmistä juoksupeleistä. Siinä pelaaja menee eteenpäin automaattisesti keräten kolikoita junaraiteilta, kuten kuvassa 8., väistellen esteitä ja pysyen loitolla poliisista, joka on lähtenyt päähenkilön perään. Pelaaja pystyy itse vaihtamaan radalta toiselle pyyhkäisemällä sormella sivuille, hyppäämään pyyhkäisemällä ylös ja kierimään pyyhkäisemällä alas. Subway Surfers on merkkipaalu siinä, kuinka pelin pystyy pitämään elossa usean vuoden ajan pienillä päivityksillä, hyvillä ja

yksinkertaisella pelattavuudella. Subway Surfer myös erottui ajassaan muista sillä, että se lisäsi toisen tason peliin. Toinen taso toteutettiin junien päällä hyppimisellä, joka toi syvyyttä itse pelimaailmaan. Subway Surfers saa aika ajoin teemakohtaisia päivityksiä, liittyen jouluun, halloweeniin tai esimerkiksi kesään, pysyen ajankohtaisena koko ajan. Pienet haasteet, tehtävät, uudet pelaajahahmot ja päiväkohtaiset tehtävät saavat pelaajan aina palaamaan takaisin surffailemaan.

Kuva 8. Subway Surfers toi toisen juoksutason peliin [15].

Juoksupelit kehittyivät ottaen mallia toisistaan ja pääsivät vuonna 2018 hypercasual-pelien pelatuimpien listalla. Näihin peleihin ilmestyi hyvin yksinkertaiset pelaajahahmot, kuten kuutio, pallo, auto tai ase. [13.] Yksinkertainen pelihahmo voidaan nähdä kuvassa 9. Pelihahmo oli siis korvautunut kokonaan vielä yksinkertaisempaan muotoon. Kun yksinkertaisuus näytti menestyvän, se sai kehittäjätiimit siirtymään tähän tyyliin ja täten vähentämään pelin tuotantoaikaa, rahaa ja resursseja huomattavasti [13]. Juoksupelejä alkoi ilmestyä yhä enemmän juoksuhihnatyyliin. Kehittäjät pyrkivät tuottamaan samantyyppisiä pelejä pienillä muutoksilla ja testaamaan, mistä pelaajat pitävät.



Kuva 9. Color Road on hyvä esimerkki yksinkertaisesta tyylistä [16].

Juoksupelit olivat täydellinen genre hyper-casual-pelimarkkinoille. Niille oli selvästi kysyntää. Juoksupelit olivat yksinkertaisia, ja tasot kestivät noin kahdesta viiteen minuuttia kerrallaan [13]. Tyyli ja yksinkertaisuus jatkuivat vuonna 2019, mutta lisäyksenä syntyivät vanhoista autopeleistä tutut tekoälyvastustajat, jotka toivat uutta otetta ja lisäsivät kilpailun tuntua [13]. Pelaaja kilpaili tekoälyjä vastaan kuten autopeleissä ohittaen aina edellä olevan niin, että oli ensimmäinen maalissa. Tekoälyvastustajat voidaan huomata kuvassa 10. Tämä teki tasoista paljon luistavampia ja pelaajalle koukuttavampia. Sosiaaliset elementit tulivat myös peliin, kuten tulostaulut, tuoden kilpailukykyä muiden pelaajien välille [17]. Tulostaulut toivat kuitenkin mukanaan huijareita, jotka vääristivät pelidataa saaden uskomattomia tuloksia. Nämä sosiaaliset elementit voidaan huomata myös vuosia pystyssä olleessa Subway Runnerissa, joka pysyy ajan tasalla juoksugenressä.

Kuva 10. Aqua Park io -peli, jossa pelaaja kilpailee tekoälyä vastaan [18].

Juoksupelin tyyli pysyi edelleen yksinkertaisena vuonna 2020. Uudenlainen mekaniikka syntyi. Mekaniikalle tuli nimeksi ryhmämekaniikka. Siinä pelaaja kerää itselleen joukon hahmoja, materiaalia tai asioita. [13.] Samalla kun pelaaja kerää niitä, hänen on väisteltävä erilaisia ansoja ja esteitä päästäkseen tason loppuun, joko taistelemaan johtajaa vastaan, toista ryhmää vastaan tai käyttämään saadut koossa olevat yksiköt, olivat ne sitten hahmoja tai materiaalia. Pelaajan keräämät yksiköt käytetään taitotasomittariin, joka määrittelee pelaajan taidot kyseisessä tasossa ja antaa oikean palkinnon. Ryhmämekaniikka nousi suureen suosioon ja on pysynyt tähän päivään asti muuttuen pienin variaatioin.

Vuonna 2021 ryhmämekaniikka sai matemaattiset portit mukaan peliin [13]. Ennen ryhmä kasvoi muutamalla yksiköllä, mutta nyt pystyttiin tuomaan satoja yksiköitä peliin. Tämä toteutus myös näkyy kuvassa 6, joka on sivulla 8. Tämä toi pelaajille yhä enemmän tyydytystä tasoja läpäistäessä. Toinenkin tyyli kasvoi ohessa, tyyli, jossa pelaaja kasvattaa hiuksien pituutta, kynsiä, kehon eri osien tai aseen päivittämistä paremmaksi, mikä tarkoittaa, että pistooli paranee haulikoksi, haulikko konekiväärikseksi ja konekivääri granaatin heittimeksi, kun porteista menee lävitse. Kuvassa 11 voidaan nähdä juoksupeli, jossa kasvatetaan

hiuksien pituutta. Sama tasossa väistely jatkuu ja loppu on suhteellisen sama edelleen.

Kuva 11. Hair Challange -pelissä pelaaja kasvattaa hiuksien pituutta ja väriä [19].

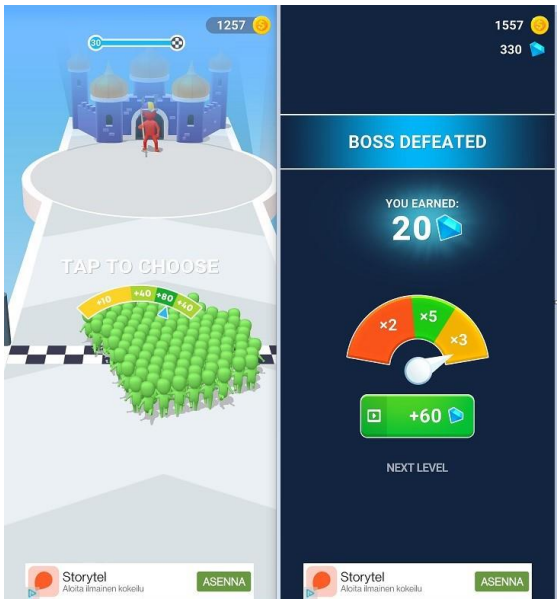
Nykyään mekaniikat pysyvät melkein samana, mutta pelaajahahmo tai ryhmä ampuu jotakin sille loogista asiaa. Tämä voidaan ajatella esimerkiksi niin, että ritsa ampuu kiviä ja kun pelaaja pääsee tarpeeksi pitkälle, ritsa on kehittynyt pistooliksi, joka ampuu pieniä luoteja, ja kun pääsee tykkiin, ampuu tykin panoksia. Tässä tyylissä melkein jokaisen portin välissä on jokin asia, mitä voi ampua tai hajottaa, joko ihmishahmoja, eläimiä tai esimerkiksi laatikoita. Edelleen kuitenkin on ryhmämekaniikka, mutta sitä on höystetty vähän vielä elollisemmaksi tai interaktiivisemmaksi.

3.2 Visuaalinen luonne

Mukavan yksinkertaiset muodot ja mallit pitävät juoksupelin mielenkiintoisena monille ikäluokille. Lämpimät ja yksinkertaiset värit ovat hyvin yleisiä juoksupeleissä. Pienetkin värien muutokset voivat laskea tai nostaa pelin suosiota, joten niiden kanssa pitää olla varovainen. Yleensä pelimaailman tausta on hyvin yksinkertainen tai jopa tyhjä. Pelitaustalla ei ole paljon merkitystä hyvissä juoksupeleissä, mutta käyttämällä taustoissa eri värejä voidaan saada mielenkiintoisempia kenttiä. Esimerkiksi jos peli sijoittuu meren alle ja tausta on sininen, voidaan tunnelmaa muuttaa vaihtamalla taustan väri vihreäksi, jolloin syntyy kuva saastuneesta vedestä. Taustavärillä voi myös näyttää tason haastavuuden. Esimerkiksi hyvin tumma tausta voi tuoda haasteellisimpia vihollisia tai esteitä.

3.3 Juoksupelin palkintojen jakaminen

Palkintojen jakaminen on olennainen osa hyper-casual-genreä. Tämä ei poikkea myöskään juoksupeligenrestä, palkintojen jakaminen vain muuttaa paikkaa. Juoksupelissä yleisin palkintojen jako tapahtuu nykyään tason lopussa. Siihen voi liittyä erilaisia kertoimia, jotka ovat sidoksissa pelaajan suorittamiin tasopisteisiin tai erilliseen valuuttaan. Kertoimet voivat olla sellaisia, missä mitataan pelaajan reaktiotasoa, tai ne ovat avattavissa mainoksen takaa. Kuvassa 12 nähdään reaktiopalkinto ja mainos pelivaluutaksi -palkinto.



Kuva 12. Count Masters: Crowd Runner-pelin palkintoja, joissa vasemmalla reaktiopalkinto ja oikealla mainos pelivaluutaksi -palkinto [10].

Palkintoja kuitenkin voi esiintyä myös vanhaan tapaan kesken juoksun. Esimerkiksi Subway Surfers -pelissä pelaaja joutuu keräilemään kirjaimia sanasta, ja kun sana on saatu kokoon, saa pelaaja kolikoita suhteessa pelattuun aikaan. Palkintoja voi myös esiintyä erilaisina tapahtumina ja niiden suorittamisina. Tapahtumat kestävät tietyn aikaa, esimerkki kuvassa 13. Tapahtumille voidaan antaa suoritusaikaraja, mikä motivoi pelaajaa jatkamaan peliään saadakseen tapahtumapalkinnon. Suoritusaikaraja antaa pelaajalle kiireen tunteen.

Kuva 13. Subway Surfers -palkintoja. Palkinnoissa on päivätapahtumia ja haasteita. [15.]

Tapahtuma voi vaihtua esimerkiksi joka yhdeksäs tunti. Näitä tapahtumia voidaan nähdä kuvassa 13. Tämä tekee sen, että pelaaja voi suorittaa tapahtumahaasteen noin kaksi kertaa päivässä. Pelissä voi olla myös päiväpalkintoja, jotka voi joko lunastaa avaamalla pelin tai suorittamalla pienimuotoisen haasteen ja saada esimerkiksi pelivaluuttaa palkinnoksi. Päiväkohtaiseenkin palkintoon voi lisätä päiväkertoimen. Päiväkerroin nousee jokaisena päivänä esimerkiksi yhdellä kertoimella, mutta jos pelaaja katkaisee päivätapahtumasarjan, kerroin menee takaisin alkuun. Tällaisella tyylillä pelaajaa rohkaistaan avaamaan peli joka päivä ja pelaamaan muutama taso ja ehkä katsomaan pakotettu mainos, joka tuottaa tekijöille tuloa.

3.4 Pelaajan kehityksen esille tuominen

Oman kehityksen näyttäminen on merkittävä osa juoksupelejä. Sillä pidetään pelaaja koukussa peliin ja saadaan pelaaja katsomaan mainoksia mahdollisimman paljon. On yleisesti tärkeää, että pelaaja näkee oman kehityksensä kulun. Juoksupeleissä se tapahtuu tason lopussa, niin että pelaaja saa tietyn määrän valuuttaa. Nykyisellä juoksupeliperiaatteella pelaaja pyrkii saamaan mahdollisimman paljon yksiköitä loppuun, joka antaa sitten tietyn määrän valuttaa. Tätä valuuttaa myös pystyy nostamaan yleensä kertoimilla, jotka ovat avattavissa mainoksen takana.

Nykyisissä juoksupeleissä pelaaja pystyy ostamaan valuutalla päivityspisteitä omaan juoksijaan. Päivitykset ovat yleensä sidoksissa tasoihin. Mitä enemmän päivityksiä pelaaja pystyy ostamaan, sitä helpommin tasot taittuvat pelatessa. Päivityksiä voi esimerkiksi olla pelaajahahmon kaksinkertaistuminen tai valuutan saamiskerroin. Päivitykset toimivat omana erillisinä kehityksen esille tuomisina. Niiden hinta kasvaa päivitystason myötä ja on yhä vaikeampi saada avattua, mutta sillä myös kehittäjät saavat pelattavuusaikaa kasvatettua.

Kun hahmokohtaiset päivitykset eivät riitä, ovat nykyiset juoksupelit lisänneet pieniä minipelejä, värejä tai saavutuslistoja. Pelaajat pystyvät käyttämään valuuttaa minipeleihin tuomaan edistystä tai esimerkiksi ostamaan hahmolle erilaisia värejä tai muotoja.

3.5 Mainonnan esittäminen

Hyper-casual-mainostaktiikka toimii myös juoksupeleissä. Nykyisissä juoksupeleissä mainosten esittäminen voi poiketa hieman. Pakotettu mainos on aina tasojen välissä. Kehittäjät yleensä päättävät, kuinka usein tämä mainos näytetään. Tämä myös riippuu paljolti tasojen pituudesta. Ideaali mainoksen näyttäminen pyritään sijoittamaan noin 2−3 minuuttiin. Sitä alemmaksi pakotetulla mainoksella ei kannata mennä. Myöskään juoksijoissa mainosta ei näytetä kesken juoksun, poikkeus kuitenkin tapahtuu pelaajan kuollessa. Pelaajalle voidaan

antaa vaihtoehto jatkaa juoksuaan, jos hän katsoo 30 sekunnin mainoksen. Tämä on hyvä tapa saada pelaajat katsomaan mainoksen ja saamaan palkkion samalla. Pelaajaa pyritään rankaisemisen sijaista palkitsemaan.

Juoksijoissa esiintyy usein mainoksia myös päivitysten yhteydessä. Jotkut päivitykset voivat maksaa mainoksen. Tähän on kaksi vaihtoehtoa: Joko pelaajalla ei ole tarpeeksi valuuttaa avatakseen seuraavaa päivitystä, jolloin peli ehdottaa mainoksen katsomista, ja mainoksen katsottuaan pelaaja saa päivityksen avattua tai kyseinen päivitystaso pystytään avaamaan vain katsomalla mainos. Jos pelaaja ei mainosta katso, ei hän pääse päivityksessä eteenpäin. [20.]

Juoksupeleissä esiintyy myös päivitysnappeja, jotka antavat suoraan valuuttaa mainoksen katsomalla. Tämä on erittäin edullinen tapa molemmille osapuolille, pelaajalle ja kehittäjille, saada haluamansa. Tässä kuitenkin pitää muistaa, että jos valuuttaa ei saa kulumaan päivityksillä tai muilla pelissä olevilla valuuttakulutusmekaniikoilla, ei pelaaja tule koskaan katsomaan mainosta valuutan takia. Esimerkiksi päivityksen hinnan pitää nousta niin suureksi, että pelaamalla siinä voi kestää pitkään. Mutta pelaaja voi katsoa mainoksen nopeuttaakseen prosessia ja saada mainoksesta pelivaluuttaa, jolla avata päivitys. Tässä kuitenkin tulee ongelma siinä mielessä, että kun päivityksien hinnat jatkavat nousemistaan, jää tällöin valuutan saaminen mainoksesta erittäin vähäiseksi, mikä tarkoittaa, että pelaaja ei mainosta katso. Valuutan saaminen mainoksesta on järkevää nostaa jonkinlaisella kertoimella, joka nostaa valuutan hintaa mainoksen katsottua. Esimerkiksi kun pelaaja katsoo mainoksen, hän saa 2000 pelivaluuttaa ja seuraavalla kerralla mainosta katsomalla hän saakin 3000 pelivaluuttaa.

Jotkut juoksupelit laittavat mainoksia jokaisen napin taakse. Mainoksella on tällöin tietty aikasykli. Tässä kehittäjät voivat helposti tuhota pelin elinkaaren suhteellisen nopeasti. Tämä voi myös olla tarkoitus, kun kuitenkin puhutaan hypercasual-peleistä, jotka tehdään viikoissa. Mainokset myös jossain juoksupeleissä voivat alkaa yhtäkkiä, vaikka pelaaja ei olisi koskenut peliin vähään aikaan. Esimerkiksi puhelimessa voi olla peli auki, ja vaikka pelaaja ei ole puhelimeen koskenut muutamaan minuuttiin, voi mainos siitä huolimatta lähteä käyntiin.

Mainosten jakajien ei tulisi sallia tällaista mainosten näyttölogiikkaa, jossa mainokset lähtevät itsestään käyntiin, koska peliyhtiöhän voisi esimerkiksi ostaa puhelimia ja asentaa pelin niille ja pitää puhelinta vain päällä, mikä toisi mainostuloja. Tätä on kuitenkin aika vaikea varmasti ratkaista automaattisesti ilman manuaalista tarkistusta.

Juoksupeleissä mainosten takia tarvitsee olla valuuttaa tai sellaisia esineitä tai asioita, jotka lisäävät mainoksen katsomiskertoja [21]. Kuten edellä mainittiin, päivitykset ovat hyviä tähän, pienet minipelit, värit, hahmot tai esimerkiksi niin vaikeat tasot, että pelaaja pakotetaan ostamaan päivityksiä päästäkseen läpi tasosta. Mitä piilevämpi valuuttakulutusmekaniikka saadaan aikaiseksi, sitä parempi, koska tällöin pelaaja ei tunne itseään huijatuksi vaan motivoituneemmaksi päästessään eteenpäin pelissä.

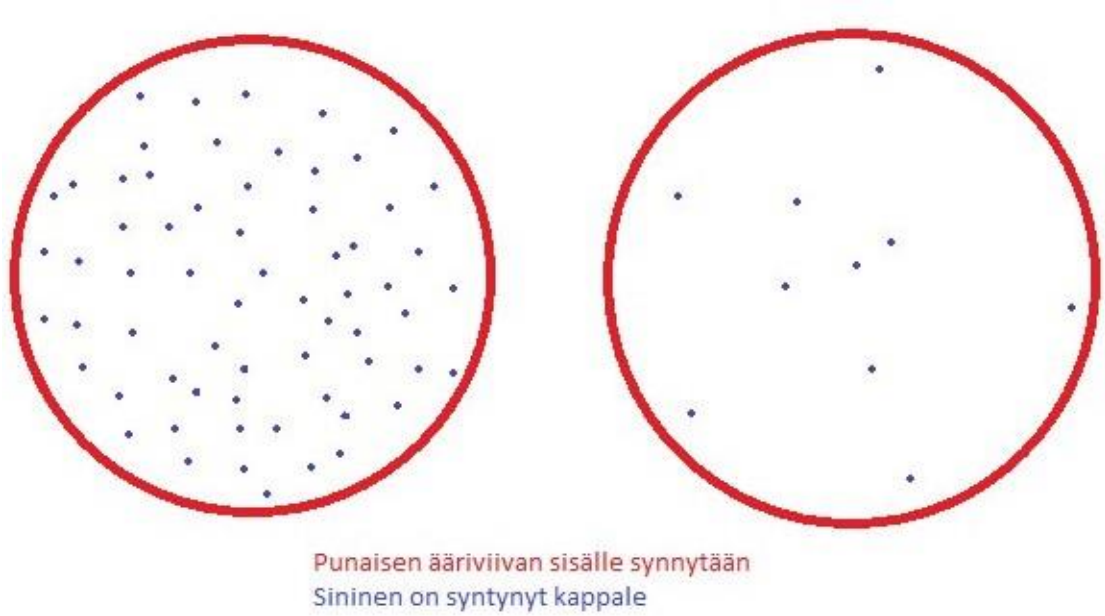
4 Juoksijapelin keskeiset ominaisuudet

4.1 Ryhmässä syntyminen ja organisointi

Nykyaikaisessa juoksupelissä pelaaja kasvattaa vahvuuttaan keräämällä esineitä tai menemällä porttien läpi. Pelaajan vahvuus näkyy yleensä ryhmän koon muuttumisena. Esimerkiksi aluksi pelaajalla on yksi kala ja kun pelaaja pääsee portista läpi, hänellä on kymmenen kalaa ja pelaaja on tällöin 10 kalan vahvuinen.

Ryhmän dynaaminen kasvaminen koodillisesti voi olla hankalaa. Varsinkin kun pyritään luomaan sellainen järjestelmä, joka kävisi yleisesti kaikkiin juoksupeleihin. Hyvin helppo lähestymistapa on tehdä pisteitä, jotka sijoitetaan satunnaisesti keskipisteen ympärille. Näillä pisteillä pitää olla kuitenkin raja esimerkiksi ympyrän sisään. Tähän tapaan on Unity-pelimoottorissa rakennettu metodi. Tässä kuitenkin käy niin, että syntyneistä kappaleista yleensä ei tule hyvännäköisiä. Joko kappaleet ovat liian erillään tai liian lähekkäin. Tämä kuitenkin on

oiva vaihtoehto, jos yritetään luoda massiivista pistepilveä, jossa lähekkäisyydellä ei ole niin väliä, koska se kuitenkin täyttää suuressa määrin luvatun alueen, kuten kuvassa 14 vasemmalla puolella. Ongelma tulee, kun pisteitä on rajoitetusti, mikä tarkoittaa sitä, että rajattu alue on niin suuri, että muutamat kymmenet pisteet näyttävät hyvin erillään olevilta, kuten kuvassa 14 oikealla puolella, vaikka lopputuloksen pitäisi olla ryhmän tai ryppään näköinen.

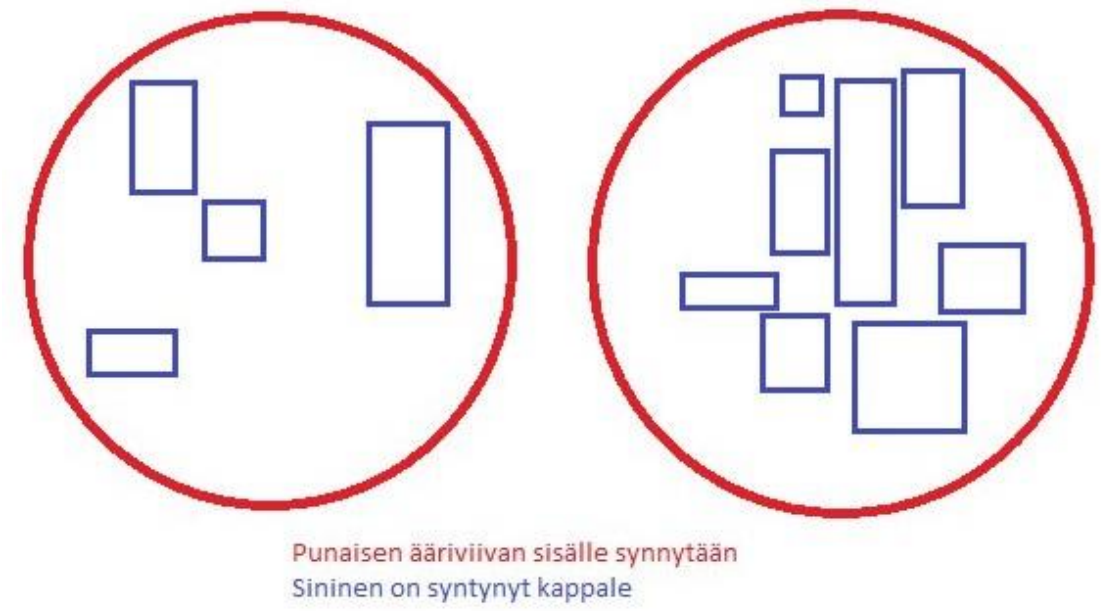


Kuva 14. Vasemmalla ovat tiheästi synnytetyt pisteet ja oikealla vain 10 pistettä.

Tällaisen pistekartan rakentaminen, jossa pisteet ovat mahdollisimman lähellä keskustaa, voi jossain määrin tuntua hankalalta. Hyvin yksinkertainen tapa on tehdä maksimipituus keskipisteestä ja sitten pienentää aloituspituus maksimipisteen kymmenesosaan ja lähteä aalloittain kasvattamaan pituutta esimerkiksi kymmenesosin. Jokaisella aallolla lähetetään tietty määrä pistekyselyitä. Jokainen piste, joka luodaan, tulee poistaa annetusta pisteiden maksimimäärästä. Tällöin saa keskelle hyvin tiiviin ryppään. Tässä tavassa kuitenkin tulee ongelma, koska jotkut pisteet lopussa erkanevat liikaa. Tämän voi korjata niin, että tulevassa aallossa pitää edellisen pituuden muistissa ja varmistaa, että tuleva piste ei ole sen sisällä.

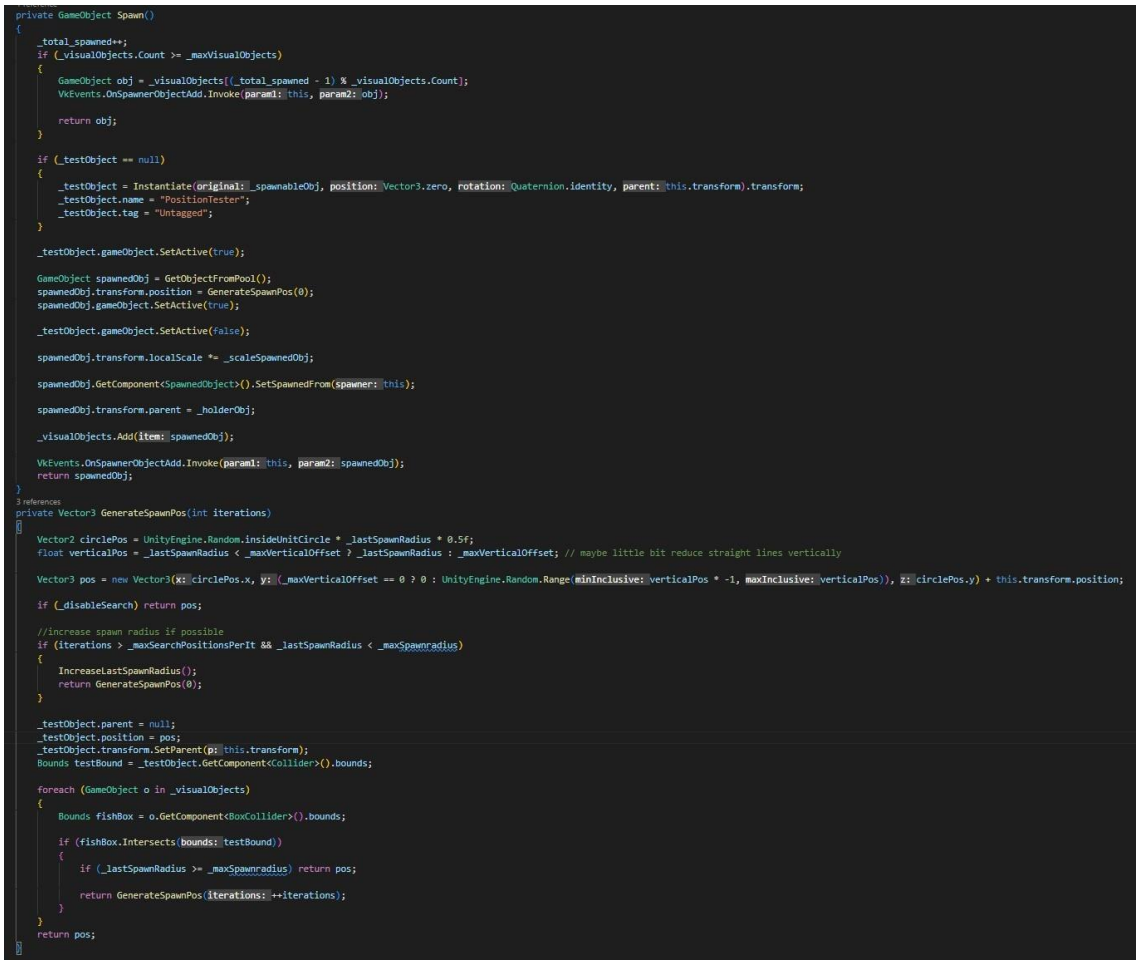
Seuraavaksi lisätään kappaleen kokoon tunnistukset. Tehokkain tapa ja yksinkertaisin olisi testata tulevan pisteen etäisyys toisista pisteistä ja antaa sille minimietäisyys, joka yleensä pyritään laittamaan pisimpään kohtaan luodusta mallista. Jos malli olisi kuutio, minimipituus olisi kuution keskipisteestä kärkeen. Jos yksikin muista luoduista pisteistä on minimietäisyyden sisällä, pistettä ei voida luoda. Sen jälkeen lähdetään etsimään uutta pistettä samasta aallosta. Joskus voi käydä niin, että pistettä ei löydy edellisestä aallosta olleenkaan, jolloin joudutaan siirtymään uuteen aaltoon, jos mahdollista.

Huono puoli pisteen etäisyyden mittaamiseksi toisesta pisteestä on se, että jos kappale ei ole ympyrän muotoinen tai se on yhteen suuntaan hyvin pitkä, tulevat kappaleet ryppäässä olemaan hyvin erillään toisistaan, kuten kuvassa 15 vasemmalla. Unityssä pystyy testaamaan kappaleita, joilla on collider-komponentti, että onko piste niiden sisällä. Tällä tavalla saadaan tarkka, edullinen ja hyvä tarkistusmenetelmä, kuten kuvassa 15 oikealla. Kuitenkin monimutkaisemmat collider-muodot kuten tahkoverkko voivat olla raskaita tarkistaa suurissa määrissä. Kun colliderit ovat kuution muotoisia pisteen tarkistus sen sisältä on suhteellisen nopea prosessi. Collideria apuna käyttäen luodun ryhmäsyntymisen koodillinen toteutus nähdään kuvassa 16.



Kuva 15. Vasemmalla kappaleen origosta laskettu pituus ja kaukaisimpaan kulmaan oikealla käytetty collider-menetelmää.

Edellä mainituilla tavoilla saadaan dynaaminen, tarkka ja pieni rypäs. Ryppään tarkkuus ja ohjelman vaadittava teho kasvavat, mitä enemmän aaltoja ja pisteen tarkistuksia aalto kohti tapahtuu. Tehon kasvettua voi tulla isoja hidastumispiikkejä, jolloin on syytä vaihtaa fysiikkamoottorin laskenta-arvoja peliin sopiviksi tai järjestää ohjelma laskemaan uudelleen syntymispisteet. Pisteiden generointi voidaan myös suorittaa pelin alussa tai tallentaa pelin dataan pistepilvenä. Tallentaminen pistepilvenä voi kuitenkin olla työläs prosessi, jos kappaleiden koot vaihtelevat paljon, jolloin oivallisin tapa on dynaamisesti muuttuva pilvi.



```
private GameObject Spawn()
{
    _total_spawned++;
    if (_visualObjects.Count >= _maxVisualObjects)
    {
        GameObject obj = _visualObjects[(_total_spawned - 1) % _visualObjects.Count];
        VkEvents.OnSpawnerObjectAdd.Invoke(param1: this, param2: obj);

        return obj;
    }

    if (_testObject == null)
    {
        _testObject = Instantiate(original: _spawnableObj, position: Vector3.zero, rotation: Quaternion.identity, parent: this.transform).transform;
        _testObject.name = "PositionTester";
        _testObject.tag = "Untagged";
    }

    _testObject.gameObject.SetActive(true);

    GameObject spawnedObj = GetObjectFromPool();
    spawnedObj.transform.position = GenerateSpawnPos(0);
    spawnedObj.gameObject.SetActive(true);

    _testObject.gameObject.SetActive(false);

    spawnedObj.transform.localScale *= _scaleSpawnedObj;

    spawnedObj.GetComponent<SpawnedObject>().SetSpawnedFrom(spawner: this);

    spawnedObj.transform.parent = _holderObj;

    _visualObjects.Add(item: spawnedObj);

    VkEvents.OnSpawnerObjectAdd.Invoke(param1: this, param2: spawnedObj);
    return spawnedObj;
}
3 references
private Vector3 GenerateSpawnPos(int iterations)
{
    Vector2 circlePos = UnityEngine.Random.insideUnitCircle * _lastSpawnRadius * 0.5f;
    float verticalPos = _lastSpawnRadius < _maxVerticalOffset ? _lastSpawnRadius : _maxVerticalOffset; // maybe little bit reduce straight lines vertically

    Vector3 pos = new Vector3(x: circlePos.x, y: (_maxVerticalOffset == 0 ? 0 : UnityEngine.Random.Range(minInclusive: verticalPos * -1, maxInclusive: verticalPos)), z: circlePos.y) + this.transform.position;

    if (_disableSearch) return pos;

    //increase spawn radius if possible
    if (iterations > _maxSearchPositionsPerIt && _lastSpawnRadius < _maxSpawnradius)
    {
        IncreaseLastSpawnRadius();
        return GenerateSpawnPos(0);
    }

    _testObject.parent = null;
    _testObject.position = pos;
    _testObject.transform.SetParent(p: this.transform);
    Bounds testBound = _testObject.GetComponent<Collider>().bounds;

    foreach (GameObject o in _visualObjects)
    {
        Bounds fishBox = o.GetComponent<BoxCollider>().bounds;

        if (fishBox.Intersects(bounds: testBound))
        {
            if (_lastSpawnRadius >= _maxSpawnradius) return pos;

            return GenerateSpawnPos(iterations: ++iterations);
        }
    }
    return pos;
}
```

Kuva 16. Ryhmäsyntymisen toteutus.

Ryhmässä syntymiselle pitää myös olla mitat helposti saatavilla, jotta sen avulla voidaan laskea pelaajan liikkuvuus tai muita tärkeitä osumakohtia. Tietenkin ryhmäkappaleet ovat listassa, jonka pystyisi läpikäymään aina kysyttäessä ryhmän kokoa. Kuitenkin tällainen kysely esimerkiksi Unityn Update-metodissa olisi suuri suorituskykyongelma pienissäkin ryhmämäärissä, jos useampi olio kutsuu kokoa.

Ryhmäkoko kannattaa laskea vain silloin, kun ryhmän koko muuttuu, ja tallentaa tiedot muistiin, josta ne ovat helposti saatavilla. Ryhmän koon laskemisen voi myös hoitaa epäsynkronisena, jolloin siitä ei koidu ylimääräistä rasitusta pääsäikeeseen, jolloin ei tarvitse myöskään huolehtia ryhmän suuresta koosta. Ryhmänkoon laskeminen pitää kuitenkin vain lopettaa, jos uusi kokomuutos

tapahtuu kesken laskemisen. Kuvassa 17 nähdään ryhmäsyntymisen koon laskeminen.

```
private async Task LoopAsync()
{
    while (!_script_is_disabled)
    {
        if (_sizeHasbeenModified)
        {
            _sizeHasbeenModified = false;
            InitMeasures();
            InitCenterPoint();
            StartCoroutine(routine: CO_AdjustSpawnPositions());

        }

        await Task.Yield();
    }
}
2 references
private void OnSpawnIncrease(GroupSpawner spawner, int value)
{
    if(!spawner.IsSameKind(spawner: this)) return;

    RefreshMeasures();
}
4 references
public void RefreshMeasures()
{
    //this will trigger async method if true
    _sizeHasbeenModified = true;
}
//only be accessed by async loop! Use RefreshMeasure() to trigger this!
1 reference
private void InitMeasures()
{
    GroupSpawnerMeasures newMeasures = new GroupSpawnerMeasures();
    bool isCalled = false;
    for(int i = 0; i < _visualObjects.Count; ++i)
    {
        if(_visualObjects == null || _visualObjects.Count == 0) break;

        if(i >= _visualObjects.Count) break;

        if(_sizeHasbeenModified) break;

        Vector3 visualPosition =_visualObjects[i].transform.position; // 0.2f bc point is center of the object and has more height
        newMeasures.CollectAndSetMeasure(position: visualPosition - transform.position);
        isCalled = true;
    }

    const float heighFix = 0.3f;

    newMeasures.SetMaxVector(vector: newMeasures.GetMaxVector().Add(y:heighFix));
    newMeasures.SetMinVector(vector: newMeasures.GetMinVector().Add(y:-heighFix));

    if(_visualObjects == null || _visualObjects.Count == 0 || !isCalled) newMeasures.SetAllZero();

    _measures = newMeasures;

}
10 references
public Vector3 GetVisualsMaxs() => _measures.GetMaxVector() + transform.position;
9 references
public Vector3 GetVisualsMins() => _measures.GetMinVector() + transform.position;
```

Kuva 17. Ryhmäsyntymisen koon laskeminen.

Laskiessa kokoa ryhmäsyntymisen asioista esimerkiksi kaloista otetaan uloimmat kalat, joiden pisteet kirjataan mittausluokkaan. Laskemisen jälkeen voidaan

helposti kysyä ryhmään syntyneiden ulommaisten kalojen sijainti ilman ylimääräistä laskemista.

4.2 Olioiden varastoiminen

Yksi tärkeimmistä asioista on ohjelmistokoodin roskien keruu varsinkin mobiililaitteilla. Roskien keruu voi vaikuttaa radikaalisti puhelimen muistiin ja suorituskykyyn. Missä vain pelissä, jossa synnytetään samankaltaisia asioita ja tuhotaan niitä, on syytä harkita asioiden varastoimista muistiin tuhoamisen sijaan. Tarkoituksena on olion tuhoutuessa olla käyttämättä Unityn omaa tuhoamismetodia ja sen sijaan lähettää kappale takaisin varastoon ja laittaa se pois päältä odottamaan uutta kutsumista. Kappaleet voidaan tuhota tarvittaessa esimerkiksi latausruudun aikana tai antaa tuhoutua pelin skenaarion mukana vaihtuessa.

Tämänkaltainen toteutus on aivan pakollinen ryhmässä syntymiseen, koska ryhmä pienenee ja suurenee jopa sekunnin välein, jos ei useammin. Tämänkaltaiset tuhoamiskutsut toisivat suuria suorituskykypiikkejä, mikä haittaisi pelaamista oleellisesti. Tähän kannattaa siksi tehdä geneerinen varastojärjestelmä, jonka toteutuksen voi katsoa kuvasta 18.

```
public abstract class ObjectPool<T> : MonoBehaviour where T : Component
{
    2 references
    public T _prefab { private get; set;}

    5 references
    private Queue<T> _objects = new Queue<T>();
    5 references
    private Transform _holder;
    0 references
    private void Awake()
    {
        _holder = new GameObject("Pool").transform;
        _holder.SetParent(p: this.transform);
    }
    1 reference
    public T Get()
    {
        if(_objects.Count == 0) AddObjects(1);
        return _objects.Dequeue();
    }

    2 references
    public void AddObjects(int amount)
    {
        for(int i = 0; i < amount; ++i)
        {
            T newObject = GameObject.Instantiate(original: _prefab);
            newObject.transform.SetParent(p: _holder);
            newObject.gameObject.SetActive(false);
            _objects.Enqueue(item: newObject);
        }
    }

    1 reference
    public void ReturnToPool(T objectToReturn)
    {
        objectToReturn.transform.SetParent(p: _holder);
        objectToReturn.gameObject.SetActive(false);
        _objects.Enqueue(item: objectToReturn);
    }

    1 reference
    public void ClearPool()
    {
        _objects.Clear();
        foreach (Transform child in _holder.transform)
        {
            GameObject.Destroy(obj: child.gameObject);
        }
    }
}
```

Kuva 18. Geneerinen varasto-olion toteutus.

Varastojärjestelmälle on olennaista, että sille annetaan määrä, kuinka monta oliota sen alussa pitää varastossa. Varasto on kuitenkin dynaaminen ja laajenee tarvittaessa. Varastolta pystyy pyytämään oliota, ja varasto antaa olion. Jos oliota ei ole saatavilla, se synnyttää uuden olion. Varasto myös vastaanottaa olion ja automaattisesti sulkee sen.

4.3 Maailman luominen

Juoksupeleissä maailmaa luodaan pelaajan menosuuntaan koko ajan lisää. Näin saadaan tunne loputtomasta tiestä. Maailman luomisprosessin voi ajatella kahtena kappaleena, pelaajan rata ja ympäristön rata. Pelaajaradalle kuuluvat pelaajan liikkuvuuden lisäksi esteet, palkinnot ja pelaajan interaktiiviset objektit.

Ympäristöradalle sijoitetaan kaikki asiat, joihin pelaaja ei juuri pysty vaikuttamaan. Ympäristöradalla luodaan tunnelma, ja sen muuttamisella voi olla suuri merkitys siihen, miltä peli vaikuttaa.

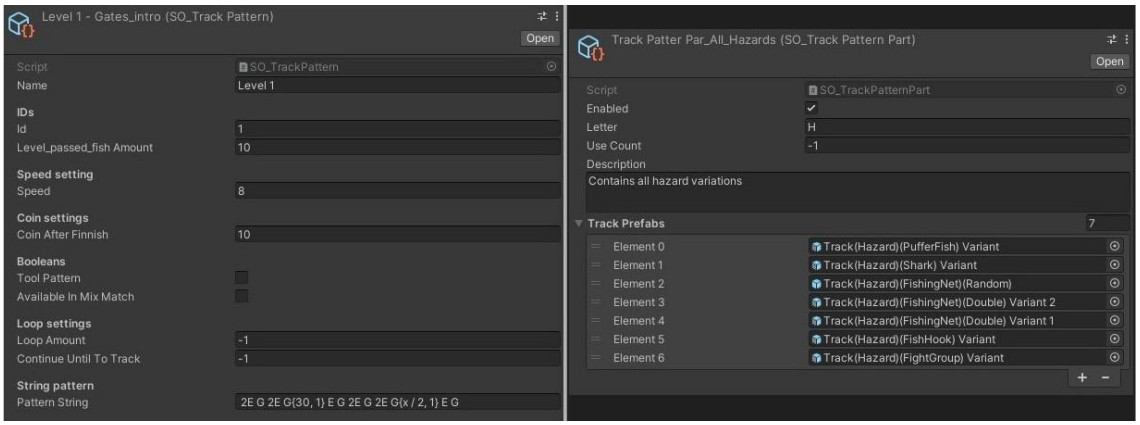
Molemmat radat luodaan paloista, ja radat tuodaan pelaajan näköpiirin niin, että pelaaja ei sitä itse huomaa. Näin pelaaja saa tunteen loputtomasta maailmasta. Molemmille luomisprosesseille luodaan oma manageri, joka hoitaa palojen syntymisen. Molemmille managereille annetaan lista tulevista kopioitavista paloista.

Ympäristömanagerilla on yksinkertainen tehtävä. Se tuo ympäristön palaset listasta järjestyksessä pelaajan näköpiiriin. Kun pelaaja on ohittanut ympäristöpalasen ja on tarpeeksi kaukana siitä, se laitetaan pois päältä ja takaisin varastoon. Pelaajaradan managerin tulee olla monimutkaisempi, koska sillä luodaan pelattavuus. Pelaajaradan tulee olla haasteellinen, palkitseva ja järkevän tuntuinen. Radan luomiseen tarvitaan useampia erilaisia ratapaloja. Esimerkiksi yksi rata voi olla tyhjä, toinen voi olla täynnä esteitä ja kolmannessa palassa on palkintoja. Mitä enemmän erilaisia paloja tehdään, sitä enemmän saadaan variaatiota pelimaailmaan eikä maailma tunnu niin yksitoikkoiselta.

Tällaisissa peleissä pelimaailma voi satunnaisesti generoituna tuntua tylsältä pitemmän päälle. Kun halutaan saada paras vaikutus omaan pelaajakuntaan, on tärkeää, että pystytään tekemään sellaisia maailmoja kuin halutaan suhteellisen helposti. Tällöin otetaan mukaan kaavamanageri. Se hoitaa ja luo ratapalojen järjestystä käyttäen pienempiä kaavapohjaobjekteja. Kaavapohjan luokan voi nähdä kuvassa 19 vasemmalla.

Kaavapohjaa voidaan ajatella tasona. Siihen kuuluu runsaasti vaihtoehtoja, kuten esimerkiksi kartan identifikaatio, nopeus, tunnistetyyppimerkkijono, “onko työkalu vai ei”, “kuuluuko satunaiseen luomiseen” ja “kuinka paljon pelaajalle maksetaan läpipäästyä”. Yksi tärkeimpiä kaavapohjan jäsenmuuttujia on tunnistetyyppimerkkijono, joka määrittää tasossa olevat esteet, haasteet ja palkinnot. Tunnistetyyppimerkkijono koostuu useista tunnistetyypeistä. Jokaisesta

tunnistetyyppistä on tehty oma kaavapohjaosa olio, joka näkyy myös kuvassa 19 oikealla.



Kuva 19. Oikealla kaavapohja ja vasemmalla kaavapohjaosa.

Jokainen ratapala voidaan jakaa omaan tunnistetyyppiin. Esimerkiksi kaikki pelaajalle antavat negatiiviset vaikutukset ovat yhdessä tyypissä, toisessa ovat vain positiiviset ja kolmannessa on vain yksi tyhjä ratapala. Jokaiselle tyypille annetaan jokin tunniste, kuten kirjain. Tunnisteet kerätään tai kirjoitetaan kaavapohjassa olevaan tunniste merkkijonoon peräkkäin ja niiden järjestys kertoo, mikä ratapala tulee seuraavaksi. Kaavamanageri hoitaa kaavapohjan lukemisen, kun kaavapohjan aika koittaa.

Jokainen tyyppi on eritelty, ja niillä voi olla omia parametrejä. Esimerkiksi portti voi olla tunnistekirjaimella G ja H voi olla kaikki esteet, J on koukku ja tyhjä rata tunnisteella E. Näillä saadaan jo yksinkertainen rata tehtyä kuten “EEGHEGJE” eli purettuna: kaksi tyhjää ratapalaa, sitten yksi portti, satunnainen este, yksi tyhjä ratapala, yksi portti, koukkueste ja yksi tyhjä. Radan pituus on seitsemän ratapalaa, ja portit ovat joko negatiivisia tai positiivisia. Saman radan voi tehdä niin, että porteilla on parametrejä, esimerkiksi “EEG{+10,-20}HEGJE”, ja purettuna: kaksi tyhjää ratapalaa, portti, jossa vasemmalla puolella on +10(yksikkö) ja oikealla –20(yksikkö), satunnainen este, tyhjä ratapala ja portti satunnaisilla arvoilla, koukkueste ja lopuksi tyhjä ratapala. Toinen esimerkki voidaan ottaa kuvasta 19 oikealta: merkkijono puretaan pelille ymmärrettävään muotoon näin: ”EEGEEGEGEEGEEGEG”. Tämä purkukoodi esiintyy kuvassa 20. Tällainen

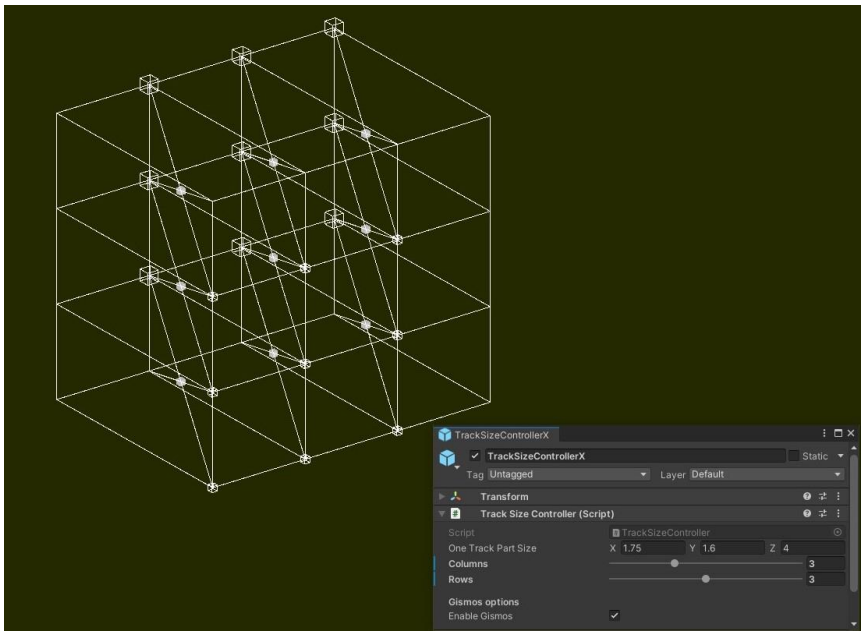
pohja mahdollistaa hyvän laajentamismahdollisuuden tunnistetyypeille ja niiden parametrejä, joita kaavamanageri hoitaa.

```
private string CompilePattern(ref int iterations, SO_TrackPattern so_pattern)
{
    _positionValues = new Dictionary<int, string>();
    string patternStr = so_pattern._patternString.ToUpper();
    ++iterations;
    if (iterations > _maxIterations) { return ""; }
    StringBuilder newPattern = new StringBuilder();
    int multiplier = 1;
    int lastMultiplier = 1;
    StringBuilder newID = new StringBuilder();
    StringBuilder newStr = new StringBuilder();
    bool idSearchEnabled = false;
    bool idStringEnabled = false;
    int index = 0;
    int startIndex = 0;
    for (int charIndex = 0; charIndex < patternStr.Length; ++charIndex)
    {
        char c = patternStr[charIndex];
        if (Char.IsWhiteSpace(c: c)) continue;

        if (Char.IsDigit(c: c) && !idStringEnabled)
        {
            if (idSearchEnabled) { newID.Append(value: c); continue; }
            multiplier = (int)Char.GetNumericValue(c: c);
            continue;
        }
        if (_positionValuesModificationEnabled && c == 'G' && (charIndex + 1 == patternStr.Length || patternStr[charIndex + 1] != '{')) { _positionValues.Add(key: index, ""); }
        if (c == '{' && !idStringEnabled) { idStringEnabled = true; startIndex = index - 1; continue; }
        if (c == '}' && idStringEnabled)
        {
            idStringEnabled = false;
            string theWord = newStr.ToString();
            if (lastMultiplier > 1)
            {
                startIndex -= lastMultiplier - 1;
            }
            for (int i = 0; i < lastMultiplier; ++i)
            {
                _positionValues.Add(key: startIndex + i, value: theWord);
            }
            newStr = new StringBuilder();
            lastMultiplier = multiplier;
            multiplier = 1;
            continue;
        }
        if (c == '[' && !idSearchEnabled) { idSearchEnabled = true; continue; }
        if (c == ']' && idSearchEnabled)
        {
            idSearchEnabled = false;
            int patternID = int.Parse(s: newID.ToString());
            newID = new StringBuilder();
            newStr = new StringBuilder();
            SO_TrackPattern searchPattern = GetPatternByID(id: patternID);
            string newStringPatter = "";
            if (searchPattern != null) { newStringPatter = CompilePattern(iterations: ref iterations, so_pattern: searchPattern); }
            for (int i = 0; i < multiplier; ++i) { newPattern.Append(value: newStringPatter); }
            lastMultiplier = multiplier;
            multiplier = 1;
            continue;
        }
        if (idStringEnabled) { newStr.Append(value: c); continue; }
        for (int i = 0; i < multiplier; ++i) { newPattern.Append(value: c); index++; }
        lastMultiplier = multiplier;
        multiplier = 1;
    }
    string newPatternStr = newPattern.ToString().ToUpper();
    AddPlusSignsIfNeeded(newPatternStr: newPatternStr);
    return newPatternStr;
}
```

Kuva 20. Kaavamanagerin kaavapohjan merkkijonon purku koodillisesti.

Jokainen kaavapohja on mahdollista käyttää myös satunnaisessa luomisessa. Satunnainen luominen voi olla välillä hyvin tylsä, ilman tärkeämpää ohjeistusta, jolloin käsin tehtyjen palojen tekeminen ja niiden satunnainen järjestys peräkkäin voi auttaa asiaa. Isoin ongelma kuitenkin on itse palojen tekeminen niin, että jokainen pala olisi yhteensopiva jokaisen palan kanssa. Palat voivat olla pieniä tai isoja. Tähän voi jatkossa yhdistää tekoälyn laskemaan parhaiten sopivia paloja tai tekemään itse satunnaisia paloja.

Kun tehdään laajaan käyttöön olevaa järjestelmää, pelaaja-alueen mahdollinen skaalaus on erittäin tärkeää. Skaalattavuudella tarkoitetaan sitä, että pelialueet pystytään laajentamaan tarvittaessa isomman kokoisiksi. Kuvassa 22 nähdään pelialueen asettelu laajennusmanagerissa.



Kuva 21. Pelialueen asettaminen halutuksi kooksi.

Esimerkiksi kahden linjan juoksupeli voi joskus tuntua huonolta ja sen laajentaminen esimerkiksi kolmeen linjaan tai vertikaalisten linjojen lisääminen voi muuttaa pelin kulkua huomattavasti. Siksi on tärkeää, että pelialueen mahdollinen laajentaminen olisi mahdollisimman vaivatonta. Kuvassa 22 nähdään sama rata kolmella eri laajennusmahdollisuudella eli vasemmalla 2 x 1, keskellä 2 x 2 ja oikealla 3 x 1.

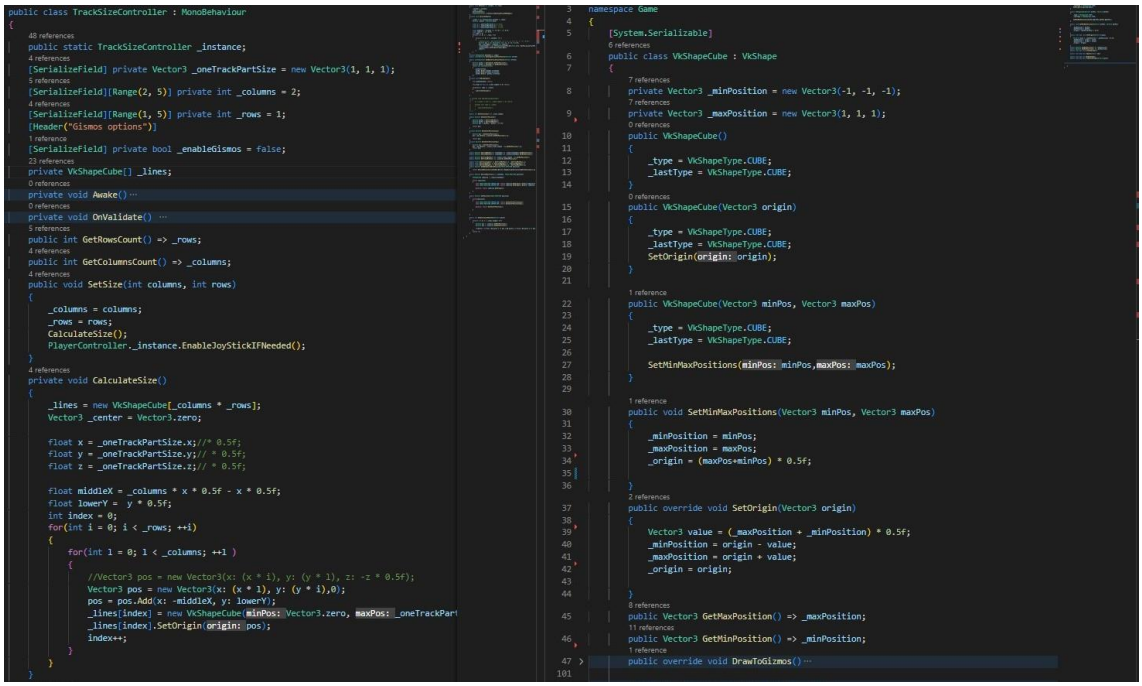
Kuva 22. Sama taso muutettuna kolmeen eri kokoon: vasemmalla 2 x 1, keskellä 2 x 2 ja oikealla 3 x 1.

Pelialueen voi myös laajentaa pelin aikana. Esimerkiksi edellinen oli kahden linjan kokoinen, mutta seuraava taso voikin olla kolmen linjan kokoinen bonustaso. Tällöin tasojen välinen samanlaisuus hajoaa ja tekee pelattavuudesta paljon mukavampaa.

Kun lähdetään tekemään pelialueen laajentamismanageria, on otettava huomioon ratojen koko. Radan mitoitus on järkevä tehdä suorakulmaisilla särmiöillä niin, että yksi suorakulmio vastaa yhtä linjaa, joten kahden linjan juoksupelissä pelialueen radan hahmottaminen tapahtuu kahdella suorakulmaisella särmiöllä.

Yleensä juoksupelien vertikaalinen taso pysyy samassa, mutta suorakulmaisia särmiöitä käyttämällä pystytään pelialueen korkeus myös määrittämään. Laajennusmanagerilla on tiedossa, kuinka monta suorakulmiota on horisontaalisesti ja vertikaalisesti. Managerilla pääsee suorakulmioiden ja niiden mittoihin suoraan käsiksi, mikä helpottaa satunnaisten asioiden generoimista. Esimerkiksi

jos halutaan synnyttää este täysin radan keskelle, pystytään laajentamis-manageria käyttämään apuna ja kysymään siltä radan keskipiste, tai jos halutaan synnyttää asioita radan sisälle, voidaan managerista saada radan sisällä olevan alueen koko. Koodillinen toteutus laajennusmanagerista nähdään kuvassa 23.



```
public class TrackSizeController : MonoBehaviour
{
    48 references
    public static TrackSizeController _instance;
    4 references
    [SerializeField] private Vector3 _oneTrackPartSize = new Vector3(1, 1, 1);
    5 references
    [SerializeField][Range(2, 5)] private int _columns = 2;
    4 references
    [SerializeField][Range(1, 5)] private int _rows = 1;
    [Header("Gismos options")]
    1 reference
    [SerializeField] private bool _enableGismos = false;
    23 references
    private VkShapeCube[] _lines;
    0 references
    private void Awake() ···
    0 references
    private void OnValidate() ···
    5 references
    public int GetRowsCount() => _rows;
    4 references
    public int GetColumnsCount() => _columns;
    4 references
    public void SetSize(int columns, int rows)
    {
        _columns = columns;
        _rows = rows;
        CalculateSize();
        PlayerController._instance.EnableJoyStickIFNeeded();
    }
    4 references
    private void CalculateSize()
    {
        _lines = new VkShapeCube[_columns * _rows];
        Vector3 _center = Vector3.zero;

        float x = _oneTrackPartSize.x;//* 0.5f;
        float y = _oneTrackPartSize.y;// * 0.5f;
        float z = _oneTrackPartSize.z;// * 0.5f;

        float middleX = _columns * x * 0.5f - x * 0.5f;
        float lowerY =  y * 0.5f;
        int index = 0;
        for(int i = 0; i < _rows; ++i)
        {
            for(int l = 0; l < _columns; ++l )
            {
                //Vector3 pos = new Vector3(x: (x * i), y: (y * l), z: -z * 0.5f);
                Vector3 pos = new Vector3(x: (x * l), y: (y * i),0);
                pos = pos.Add(x: -middleX, y: lowerY);
                _lines[index] = new VkShapeCube(minPos: Vector3.zero, maxPos: _oneTrackPar
                _lines[index].SetOrigin(origin: pos);
                index++;
            }
        }
    }
```

```
namespace Game
{
    [System.Serializable]
    6 references
    public class VkShapeCube : VkShape
    {
        7 references
        private Vector3 _minPosition = new Vector3(-1, -1, -1);
        7 references
        private Vector3 _maxPosition = new Vector3(1, 1, 1);
        0 references
        public VkShapeCube()
        {
            _type = VkShapeType.CUBE;
            _lastType = VkShapeType.CUBE;
        }
        0 references
        public VkShapeCube(Vector3 origin)
        {
            _type = VkShapeType.CUBE;
            _lastType = VkShapeType.CUBE;
            SetOrigin(origin: origin);
        }

        1 reference
        public VkShapeCube(Vector3 minPos, Vector3 maxPos)
        {
            _type = VkShapeType.CUBE;
            _lastType = VkShapeType.CUBE;

            SetMinMaxPositions(minPos: minPos,maxPos: maxPos);
        }

        1 reference
        public void SetMinMaxPositions(Vector3 minPos, Vector3 maxPos)
        {
            _minPosition = minPos;
            _maxPosition = maxPos;
            _origin = (maxPos+minPos) * 0.5f;


        }
        2 references
        public override void SetOrigin(Vector3 origin)
        {
            Vector3 value = (_maxPosition + _minPosition) * 0.5f;
            _minPosition = origin - value;
            _maxPosition = origin + value;
            _origin = origin;

        }
        8 references
        public Vector3 GetMaxPosition() => _maxPosition;
        11 references
        public Vector3 GetMinPosition() => _minPosition;
        1 reference
        public override void DrawToGizmos() ···
```

Kuva 23. Koodillinen toteutus laajennusmanagerista.

Tasojen mahdollinen tallentaminen on erittäin hyödyllistä. Tason kaavapohjan tallentaminen molempiin suuntiin mahdollistaa tasojen kehittämistä peliä tehdessä. Kaavapohjien jakaminen pelintekijöiden välillä auttaa pohjien suunnittelussa. Molempiin suuntiin tallentamista voidaan käyttää apuna esimerkiksi automaattisesti luoduilla pohjilla. Automaattisesti luodut pohjat voidaan täten jakaa tutkittavaksi ja paranneltaviksi pelinkehittäjille.

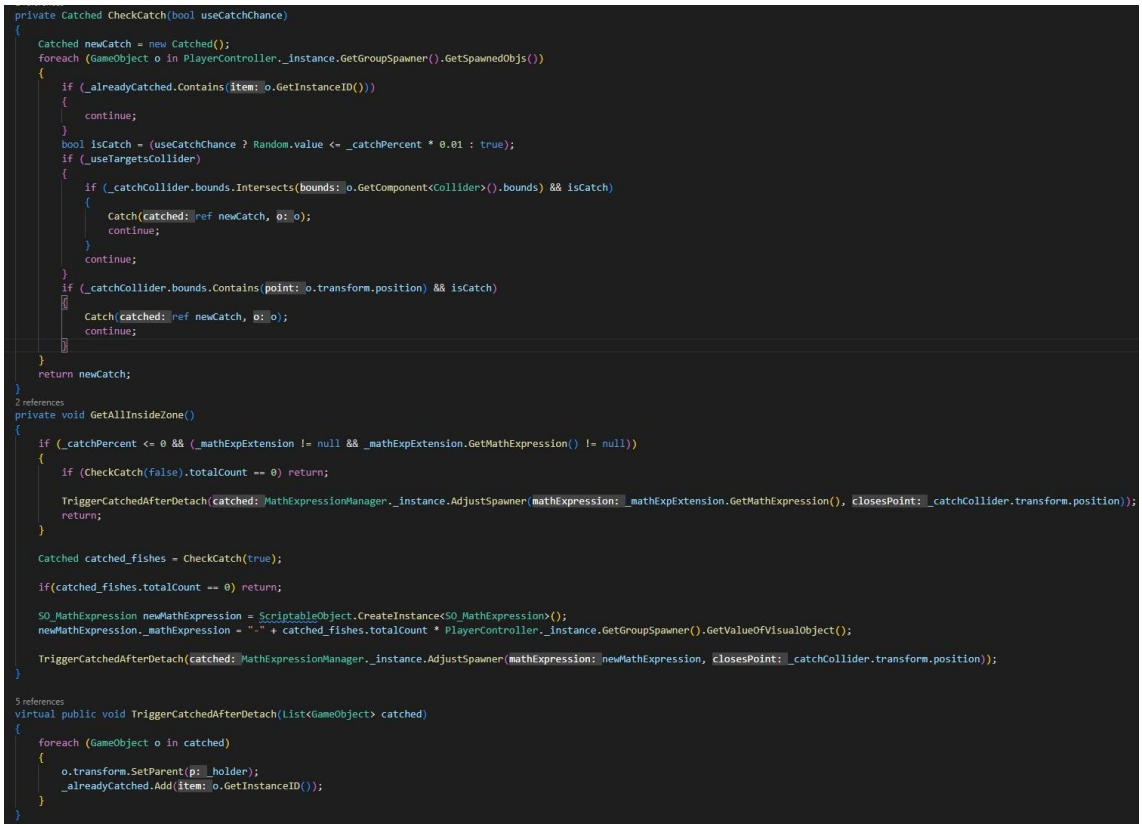
Lyhyesti sanottuna kaavapohjan tallentaminen molempiin suuntiin voidaan toteuttaa tallentamalla kaavapohja tiedostoon. Peli osaa lukea ja kääntää tiedoston kaavapohjaksi. Tähän sopii hyvin JSON eli JavaScript Object Notation. JSON mahdollistaa selkeän tavan ilmaista, mitä kaavapohjaan tulee ihmiselle luettavaan muotoon. Se on myös useissa muissa kielissä käytetty tallennus- ja

datanlähetysformaatti, mikä tekee siitä oivallisen tähänkin asiaan. Tämänkaltainen järjestelmä myös mahdollistaa tulevaisuudessa kaavapohjan lukemisen tai lähettämisen palvelimelta suoraan peliin ilman, että peliä tarvitsee asentaa tai päivittää uusiksi, mikä tekee siitä hyödyllisen.

4.4 Kentällä laukaistavat ominaisuudet

Juoksupeleissä kentällä on useita pelaajille vältettäviä objekteja. Nämä laukaistavat objektit voivat joko lisätä tai vähentää pelaajan tehokkuutta esimerkiksi antamalla lisää voimapisteitä tai viemällä niitä. Esimerkiksi pelaajalla on kalaparvi, jonka määrä kuvastaa pelaajan voimapisteitä. Kun pelaaja menee positiivisesta objektista kuten portista, sen kalaparvi kasvaa, mutta kun taas vastaan tulee hai ja se osuu puoleen parveen, se pienentää parven kokoa puoleen.

Tällainen toteutus voidaan jakaa kahteen komponenttiin. Yksi on nostava ja toinen laskeva ja kumpikin pystytään perimään alemmille lapsille. Tähän tehdään pääluokka, jonka tarkoituksena on ottaa kaikki osutut pelaajan yksiköt kiinni. Tätä pääluokkaa voidaan kutsua pyydysalueeksi. Pyydysalueen ominaisuuksia ovat seuraavat: pyydysalueen collider, ottaako se heti kiinni, kun pystyy, ottaako se kiinni tietyn ajan päästä vai aktivoituuko kiinniotto aivan keskellä pyydysalueen origoa. Pyydysalueen tarkoituksena on olla pääluokka, josta alemmat luokat perivät sen. Pääluokka, joka on napannut pelaajan yksiköt kuten esim. kalat, saadaan irrotettua ja annettua eteenpäin periville lapsille, mutta se ei kuitenkaan irrota niitä ryhmäsynnytysobjektista. Tämä tekee hyödylliseksi sen, että pyydysalue kommunikoi pelaaja komponentin kanssa eikä sen jälkeen tarvitse huolehtia muusta kuin pelaajan voimayksiköistä, eli esimerkiksi kaloista, ihmishamoista tai palloista. Perivät komponentit huolehtivat yksiköiden palauttamisesta, kun ovat tehneet niille sen mitä tarvitsee. Esimerkiksi verkkokomponentti on perinyt pyydysaluepääluokan ja aktivoi verkkoanimaation sekä äänen ja sen jälkeen palauttaa yksiköt takaisin synnyinkomponentille tai tuhoaa ne. Kuvassa 24 nähdään pyydysalueen tärkeimpiä koodillisia metodeja.



```
private Catched CheckCatch(bool useCatchChance)
{
    Catched newCatch = new Catched();
    foreach (GameObject o in PlayerController._instance.GetGroupSpawner().GetSpawnedObjs())
    {
        if (_alreadyCatched.Contains(item: o.GetInstanceID()))
        {
            continue;
        }
        bool isCatch = (useCatchChance ? Random.value <= _catchPercent * 0.01 : true);
        if (_useTargetsCollider)
        {
            if (_catchCollider.bounds.Intersects(bounds: o.GetComponent<Collider>().bounds) && isCatch)
            {
                Catch(catched: ref newCatch, o: o);
                continue;
            }
            continue;
        }
        if (_catchCollider.bounds.Contains(point: o.transform.position) && isCatch)
        {
            Catch(catched: ref newCatch, o: o);
            continue;
        }
    }
    return newCatch;
}
2 references
private void GetAllInsideZone()
{
    if (_catchPercent <= 0 && (_mathExpExtension != null && _mathExpExtension.GetMathExpression() != null))
    {
        if (CheckCatch(false).totalCount == 0) return;

        TriggerCatchedAfterDetach(catched: MathExpressionManager._instance.AdjustSpawner(mathExpression: _mathExpExtension.GetMathExpression(), closesPoint: _catchCollider.transform.position));
        return;
    }

    Catched catched_fishes = CheckCatch(true);

    if(catched_fishes.totalCount == 0) return;

    SO_MathExpression newMathExpression = ScriptableObject.CreateInstance<SO_MathExpression>();
    newMathExpression._mathExpression = "-" + catched_fishes.totalCount * PlayerController._instance.GetGroupSpawner().GetValueOfVisualObject();

    TriggerCatchedAfterDetach(catched: MathExpressionManager._instance.AdjustSpawner(mathExpression: newMathExpression, closesPoint: _catchCollider.transform.position));
}

5 references
virtual public void TriggerCatchedAfterDetach(List<GameObject> catched)
{
    foreach (GameObject o in catched)
    {
        o.transform.SetParent(p: _holder);
        _alreadyCatched.Add(item: o.GetInstanceID());
    }
}
```

Kuva 24. Kooditoteutus pyydysalueen oleellisimmista metodeista.

Muutama yleinen perintävariaatio, jota melkein jokainen peli pystyy käyttämään, ovat imurit ja välittömät kuolemakomponentit, jotka perivät pääluokan pyydysalue. Imurikomponentti tekee nimensä mukaisesti, eli sen tarkoituksena on imeä pelaajan yksiköt kohti imurille annettua vetopistettä. Imurikomponentti on siitä viisas, että se laukaisee metodin, kun kaikki yksiköt ovat päässeet vetopisteeseen. Tätä komponenttia voi käyttää esimerkiksi hain tai valaan suussa, joka imee pelaajan kalat. Imurikomponentille voi olla käyttöä myös erilaisissa putkissa, tuulissa tai mustissa aukoissa. Imurikomponentti on siis tarkoitus periä eteenpäin. Kuvassa 25 nähdään imuriluokan keskeisimpiä metodeja.

```
public override void TriggerCatchedAfterDetach(List<GameObject> GetAllCatched)
{
    base.TriggerCatchedAfterDetach(catched: GetAllCatched);
    foreach (GameObject o in GetAllCatched)
    {
        Rigidbody rb = o.GetComponent<Rigidbody>();

        if (rb == null) rb = o.AddComponent<Rigidbody>();

        rb.useGravity = false;
        rb.constraints = RigidbodyConstraints.FreezeRotation;

        Collider collider = o.GetComponent<Collider>();

        if (collider != null) collider.isTrigger = _inVacuumCollidersSetToTrigger;

        _allCatched.AddLast(value: rb);
    }
    if (!_coroutineRuning) StartCoroutine(routine: VacuumActivated());
}
1 reference
IEnumerator VacuumActivated()
{
    _coroutineRuning = true;
    LinkedList<GameObject> vacuumed = new LinkedList<GameObject>();
    while (_allCatched.Count > 0)
    {
        LinkedListNode<Rigidbody> node = _allCatched.First;
        while (node != null)
        {
            LinkedListNode<Rigidbody> next = node.Next;
            Rigidbody rigidbody = node.Value;
            GameObject o = rigidbody.gameObject;

            float distance = o.transform.DistanceTo(target: _vacuumMounth);

            Vector3 dir = o.transform.DirectionTo(destination: _vacuumMounth);
            Vector3 force = dir * _vacuumPower / distance;

            if (o.transform.DistanceTo(target: _vacuumMounth) < _disableRbDistance) //|| Vector3.Dot(totalizerDir, o.transform.position) < 0)
            {
                rigidbody.velocity = Vector3.zero;
                Destroy(obj: node.Value); //destroying rigidbody
                _allCatched.Remove(node: node);
                vacuumed.AddLast(value: o);
            }
            else
            {
                node.Value.AddForce(force: force, mode: ForceMode.Acceleration);
            }

            Vk_Utilities.SpeedControl(rigidbody: ref rigidbody, maxSpeed: _vacuumPower);
            node = next;
        }
        yield return null;
    }
    _coroutineRuning = false;

    if (vacuumed.Count > 0) AllVacuumed(vacuumed: vacuumed);

}
```

Kuva 25. Kooditoteutus imurista, jonka perii luokan pyydysalue.

Toinen komponentti on välitön kuolema. Se kertoo jo heti, mikä on komponentin periämmäinen tarkoitus. Komponentti käyttää pääluokan pyydysalueen laukaistua metodia, jolla se saa otetut yksiköt kiinni ja palauttaa ne takaisin yksikön synnyttäneelle komponentille. Tätä voi käyttää esimerkiksi kaikissa objekteissa, jotka tekevät niistä vaarallisia, esimerkiksi hain vartalo, kivieste tai vaikka pallokala. Tässä voidaan huomata, että esimerkiksi hailla on kaksi pääluokan perimää komponenttia: Hain suu, jossa on imurikomponentti, ja hain vartalo, jossa on äkkikuolemakomponentti. Tämänkaltaisilla yhdistämisillä saadaan elävämpiä esteitä,

joilla on useampi ominaisuus. Kuvassa 26 nähdään pääluokan välitön kuolema toteutus ja kuvassa 27 estehainluokan koodillinen toteutus, jossa se perii imuriluokan.

```
public class HazardInstaDeath : CatchZone
{
    2 references
    [SerializeField] private AnimationEventTrigger effectTrigger;
    5 references
    public override void TriggerCatchedAfterDetach(List<GameObject> GetAllCatched)
    {
        base.TriggerCatchedAfterDetach(catched: GetAllCatched);

        foreach (GameObject o in GetAllCatched)
        {
            //Destroy(o);
            o.GetComponent<SpawnedObject>().ReturnToSpawner();
        }
        if (effectTrigger != null)
            effectTrigger.TriggerEffect(0);
    }
}
```

Kuva 26. Kooditoteutus välittömästä kuolemasta.

```
public class HazardShark : Vacuum
{
    3 references
    private Animator _animator;
    2 references
    protected override void Awake()
    {
        base.Awake();
        _animator = this.GetComponentInChildren<Animator>();
    }
    3 references
    protected override void AllVacuumed(LinkedList<GameObject> vacuumed)
    {
        base.AllVacuumed(vacuumed: vacuumed);

        foreach (GameObject o in vacuumed)
        {
            //Destroy(o);
            o.GetComponent<SpawnedObject>().ReturnToSpawner();
        }
        _animator.SetTrigger("Bite");
    }
    4 references
    protected override void TriggerFirstEntered(GameObject obj)
    {
        base.TriggerFirstEntered(obj: obj);

        _animator.SetTrigger("PrepareAttack");
    }


}
```

Kuva 27. Hailuokkatoteutus, joka perii imuriluokan.

Laukaistaviin kappaleisiin kuten haihin, kalaverkkoon, koukkuun tai ohjukseen tarvitaan myös jonkinlainen sijainti ja liikkuminen. Manuaalisesti sijainnin tai

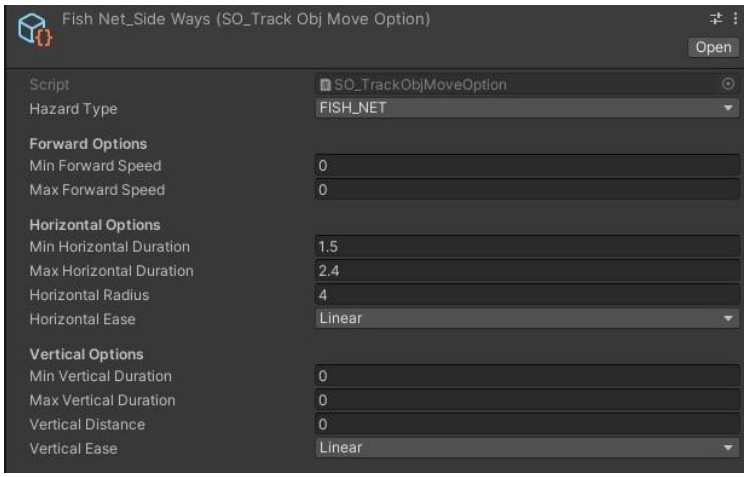
liikkumisen laittaminen voi olla työlästä, mutta sitä ei saa kuitenkaan sulkea pois, koska joskus halutaan asialle tietty esitystapa. Automaattisella asettamisella on hyvät ja huonot puolensa. Kun puhutaan sijainnista ja otetaan esimerkiksi kalaverkko, sen asettaminen johonkin satunnaiseen pisteeseen horisontaalisti on hyvä idea, mutta kun kalaverkkoja synnytetään useampi, ne voivat mennä peräkkäin, jolloin kahden esteen väistely muuttuu yhdeksi esteeksi. Tämä on kuitenkin pieni olemassa oleva sattuma, joka voidaan ajan myötä joko pitää tai muokata esimerkiksi niin, että kalaverkko katselee edellisen kalaverkon syntymispisteen sijainnin eikä sijoita asioita sen perään. Lisäksi automaattisen sijaintikomponentin pitää osata kommunikoida laajentamismanagerin kanssa. Esimerkiksi jos objekti halutaan täysin keskelle rataa, voidaan sijainti kysyä laajentamismanagerilta. Laajentamismanagerilta kysytään myös radan koon ääripäät, jotta annettu sijainti ei mene radan yli.

Unityn kauppapaikasta löytyy käytännöllinen paketti, kuten DOTween. Se on melkein missä projektissa tahansa yleishyödyllinen paketti, jonka avulla pystyy tekemään kappaleiden, värien tai yksittäisien arvojen muutoksia tietyssä ajassa. Esimerkiksi sillä pystytään yhdellä rivillä tekemään niin, että este liikkuu sivulta toiselle tasaisesti annetussa ajassa tai että väri muuttuu sinisestä punaiseen ja vihreään tietyssä ajassa. Tätä pakettia hyödyntämällä pystytään tekemään liikkumisjärjestelmä, johon lukeutuvat liikkumismanageri, kontrolleri, liikkumisasetukset, scriptableobject ja olio liikkumiskomponentti. Kuvassa 28 nähdään liikkumisolion toteutus.

```
public void ApplyMovement()
{
    if (PlayerController._instance.IsMovementDisabled())
    {
        StartCoroutine(routine: WaitingForPlayerMovement());
        return;
    }
    if(_useTween){
        _anchor = new GameObject("TrackMovement_Anchor").transform;
    } else if (_anchor == null) _anchor = new GameObject("TrackMovement_Anchor").transform;
    _anchor.position = transform.position;

    //order matters!
    CalculateHorizontalStuff();
    CalculateVerticalStuff();
}

1 reference
private void CalculateHorizontalStuff()
{
    if (_horizontalRange <= 0 || _horizontalDuration <= 0) return;

    float maxPossibleX = TrackSizeController._instance.GetTrackMaxPos().x;
    float minPossibleX = TrackSizeController._instance.GetTrackMinPos().x;
    if(!_useTween){
        if(_dirValue == 0) _dirValue = Random.value < 0.5 ? -1 : 1;
        else _dirValue*-1;
    } else
        _dirValue = Random.value < 0.5 ? -1 : 1;

    Vector3 endPosition = new Vector3(x: Mathf.Clamp(value: transform.position.x + _horizontalRange * _dirValue, min: minPossibleX, max: maxPossibleX), y: transform.position.y, z: transform.position.z);
    Vector3 startPosition;
    if(_useTween){
        startPosition = new Vector3(x: Mathf.Clamp(value: transform.position.x + -(_horizontalRange * _dirValue), min: minPossibleX, max: maxPossibleX), y: transform.position.y, z: transform.position.z);
        _anchor.DOMove(endValue: endPosition, duration: _horizontalDuration).SetEase(ease: _horizontalEase).SetLoops(-1, loopType: LoopType.Yoyo);
    } else {
        _targetPos = endPosition;
        startPosition = new Vector3(x: Mathf.Clamp(value: transform.position.x, min: minPossibleX, max: maxPossibleX), y: transform.position.y, z: transform.position.z);
        _horizontalSpeed = _horizontalRange / _horizontalDuration;
    }
    _anchor.position = startPosition;
}
0 references
private void FixedUpdate()
{
    if (GameManager._instance.gameState != GAME_STATE.PLAY) return;

    if (_anchor != null)
    {
        this.transform.position = new Vector3(x: _anchor.position.x, y: _anchor.position.y, z: transform.position.z);
        if(!_useTween && (_horizontalRange > 0 || _horizontalDuration > 0)){
            if(Mathf.Abs(f: _targetPos.x - _anchor.transform.position.x) <= 0.05f){
                ApplyMovement();
            } else {
                _anchor.transform.position += new Vector3(x: _horizontalSpeed * _dirValue * Time.deltaTime, 0, 0);
            }
        }
    }
    if (_forwardSpeed <= 0) return;

    this.transform.position += new Vector3(0, 0, z: -_forwardSpeed * Time.deltaTime);
}

1 reference
private void CalculateVerticalStuff()
{
    if (_verticalDistance <= 0 || _verticalDuration <= 0) return;

    _anchor.DOMove(endValue: _anchor.position + new Vector3(0, y: _verticalDistance, 0), duration: _verticalDuration).SetEase(ease: _verticalEase).SetLoops(-1, loopType: LoopType.Yoyo);
}
```

Kuva 28. Kooditoteutus liikkumisoliosta.

Manageri hoitaa ja lukee kaikki pelissä olevat liikkumisasetukset. Sen tarkoituksena on antaa jokaiselle kontrollerille sen tyyppinen liikkumisasetus. Asetuksessa on kaikki liikkumiseen olevat tarpeelliset tiedot, esimerkiksi etenemisnopeus, horisontaali aika ja horisontaalin matkan pituus. Nämä näkyvät kuvassa 29.



Kuva 29. ScriptableObject-liikkumisasetusolio.

Kun liikkumisasetuksia on useita samalle tyypille, manageri ottaa satunnaisesti jonkun niistä. Managerille pystyy myös asettamaan tasolle ominaisia liikkumisasetuksia, jos esimerkiksi halutaan, että tasossa esteet näyttäisivät hyppivän tai vaikka kaikki esteet liikkuvat sivuttain tasaisesti. Kontrolleri siis kyselee managerilta asetuksen ja asettaa asetuksen datan liikkumisoliolle. Liikkumisolio laskee tarvittavat asiat ja toteuttaa liikkumisen DOTweeniä käyttäen tai ilman sitä. DOTweenin voi laittaa pois myös päältä, mutta kaikkia liikkumisominaisuuksia ei tällöin pysty käyttämään. Kun DOTweeniä käyttää, liikkuminen ei pysty olemaan dynaamista, mikä tarkoittaa, että kun DOTweenille on annettu arvot, ei arvoja pysty muuttamaan tekemällä uutta DOTween-oliota. Tähän saadaan ratkaisu niin, että luodaan näkymätön tyhjä haamukappale pelialueelle ja asetetaan sille DOTween-liikkumisparametrit ja sen jälkeen annetaan liikkumisolion kopioida sen liikettä.

4.5 Nopeasti ratkottavat pulmat

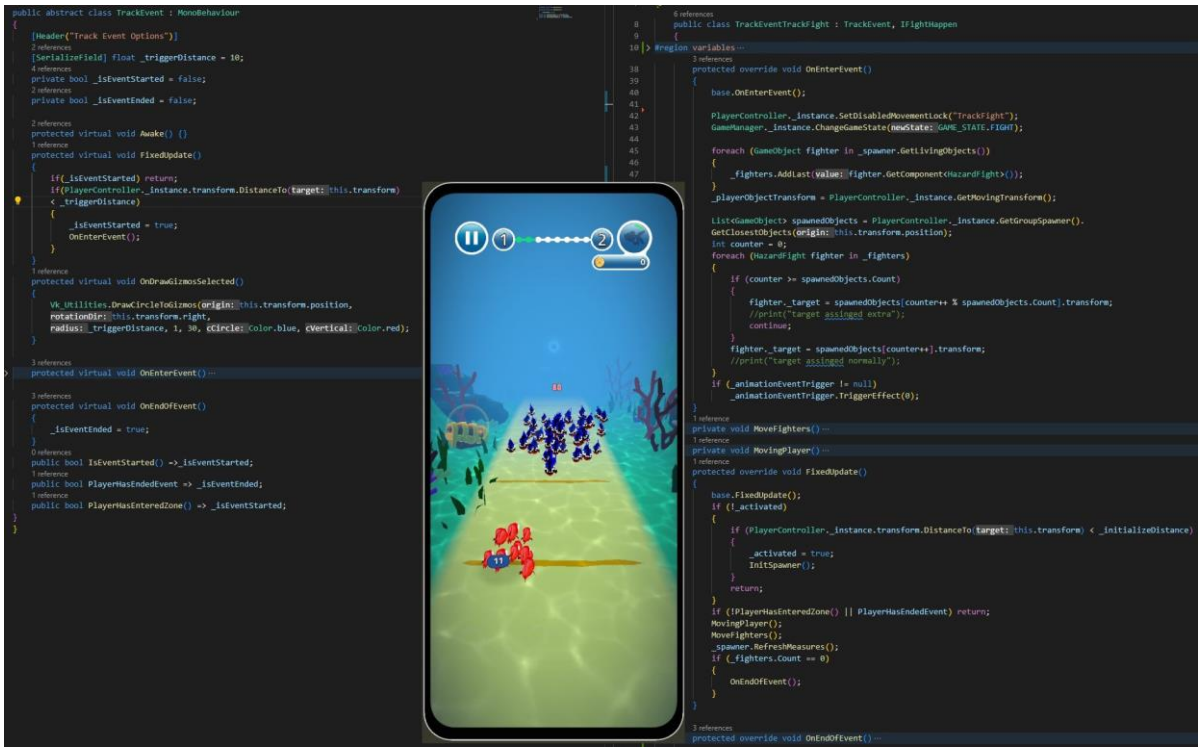
Nykyisissä juoksupeleissä matematiikan ja siihen liittyvän nopean laskemisen taidot ovat lisääntymässä. Yhä useammassa juoksupeleissä näkee portteja, joissa on jokin matemaattinen funktio, kaava tai laskutoimitus. Esimerkiksi juoksupeleissä pelaajaradalla voi esiintyä portteja, joissa lukee +3, log(4) tai / 2. Kun pelaaja menee näistä porteista läpi, sen voimataso nousee tai laskee

matemaattisen yhtälön myötä. Pelissä tulee siis olla matemaattinen ilmaisukomponentti, joka kommunikoi oman managerin välityksellä. Manageri hoitaa kaiken laskemisen ja matemaattisten funktioiden kaavojen tai laskutoimituksen tuomisen komponentteihin. Aina kun komponentti alustetaan, se lähettää managerille tiedon siitä ja manageri antaa komponenteille tarvittavat tiedot. Komponenteilla on myös ominaisuus, jolla voidaan aktivoida kyseinen matemaattinen funktio, joka komponenteilla on.

4.6 Ratatapahtumat

Nykyaikaisissa juoksupeleissä esiintyy jonkinlaista taistelua tai tapahtumia kesken juoksun. Se voi olla esimerkiksi tappelu pomoja vastaa, ryhmää vastaan tai vaikka jonkinlainen torni, joka ampuu pelaajaa kohti.

Tämänkaltaisille tapahtumille voidaan luoda ratatapahtumaluokka. Luokalla on aktivointi pituus. Kun pelaaja astuu pituuden sisälle, se laukaisee tapahtuman. Tämä on oiva tapa tehdä tapahtumia radalle. Näihin voi esim. yhdistää yllä mainitun ryhmäsyntymisluokan. Tällä tavoin saadaan tehtyä esimerkiksi tappelu pelaajan ja tapahtuman välillä. Kuvassa 30 voidaan nähdä koodillinen toteutus ratatapatumasta ja ratatappelutapahtumasta.



Kuva 30. Koodillinen toteutus ja keskellä näkyy tapahtuma pelissä.

Kuvassa 30 näkyy ratatapahtuman ja pääluokan välinen perintäyhteys. Ratatapahtuma pääluokkaan on helppo lisätä oleellisimpia tapahtumametodeja, kuten esimerkiksi onko tapahtuma alkanut tai päättynyt tai kuinka kauan pelaaja on viettänyt aikaa tapahtumassa.

4.7 Pelaajan liikkuminen

Liikkuminen on olennainen osa juoksupeliä. Sen tarkoitus on olla hyvin yksinkertainen, mutta toimiva. Liikkuminen ei saa olla tökkivää tai muuten tuntuu siltä, että pelaajalla ei ole siitä kontrollia. Esineiden kerääminen ja väistely on olennainen osa juoksupeliä, ja liikkumisen on oltava sulavaa ja herkkää. Kuvassa 31 nähdään koodillinen toteutus pelaajan liikkumisesta.

```
public bool UpdatePosition(Vector3 pos, Vector3 lastPos)
{
    float maxPossibleX = TrackSizeController._instance.GetTrackMaxPos().x;
    float minPossibleX = TrackSizeController._instance.GetTrackMinPos().x;
    Vector3 worldPoint = Camera.main.ScreenToWorldPoint(new Vector3(x: pos.x, y: pos.y, z: _camDistance));
    Vector3 lastWorldPoint = Camera.main.ScreenToWorldPoint(new Vector3(x: lastPos.x, y: lastPos.y, z: _camDistance));
    float xDir = worldPoint.x - lastWorldPoint.x;
    Vector3 minVisualPos = GetGroupSpawner().GetVisualsMins();
    Vector3 maxVisualPos = GetGroupSpawner().GetVisualsMaxs();
    float xPos;
    if(minVisualPos.x < minPossibleX)
    {
        minPossibleX += Math.Abs(value: minPossibleX - minVisualPos.x);
        xPos = Mathf.Clamp(value: transform.position.x + xDir * GameManager._instance.touchMovementMultiplier, min: minPossibleX, max: maxPossibleX);
        if(Mathf.Abs(f: transform.position.x - xPos) < 0.025 && xDir < 0)
            xPos = transform.position.x;
    }
    else if(maxVisualPos.x > maxPossibleX)
    {
        maxPossibleX -= Math.Abs(value: maxPossibleX - maxVisualPos.x);
        xPos = Mathf.Clamp(value: transform.position.x + xDir * GameManager._instance.touchMovementMultiplier, min: minPossibleX, max: maxPossibleX);
        if(Mathf.Abs(f: transform.position.x - xPos) < 0.025 && xDir > 0)
            xPos = transform.position.x;
    }
    else
        xPos = Mathf.Clamp(value: transform.position.x + xDir * GameManager._instance.touchMovementMultiplier, min: minPossibleX, max: maxPossibleX);

    if(_enableVerticalMovements)
    {
        float yOffset = 0.2f; // Stops fish from going inside ground
        float maxPossibleY = TrackSizeController._instance.GetTrackMaxPos().y;
        float minPossibleY = TrackSizeController._instance.GetTrackMinPos().y + yOffset;
        float yDir = worldPoint.y - lastWorldPoint.y;
        float yPos;
        if(minVisualPos.y < minPossibleY)
        {
            minPossibleY += Math.Abs(value: minPossibleY - minVisualPos.y);
            yPos = Mathf.Clamp(value: transform.position.y + yDir * GameManager._instance.touchMovementMultiplier, min: minPossibleY, max: maxPossibleY);
            if(Mathf.Abs(f: transform.position.y - yPos) < 0.025 && yDir < 0)
                yPos = transform.position.y;
        }
        else if(maxVisualPos.y > maxPossibleY)
        {
            maxPossibleY -= Math.Abs(value: maxPossibleY - maxVisualPos.y);
            yPos = Mathf.Clamp(value: transform.position.y + yDir * GameManager._instance.touchMovementMultiplier, min: minPossibleY, max: maxPossibleY);
            if(Mathf.Abs(f: transform.position.y - yPos) < 0.025 && yDir > 0)
                yPos = transform.position.y;
        }
        else
            yPos = Mathf.Clamp(value: transform.position.y + yDir * GameManager._instance.touchMovementMultiplier, min: minPossibleY, max: maxPossibleY);

        transform.position = new Vector3(x: xPos,y: yPos, z: transform.position.z);
    }
    else
    {
        transform.position = new Vector3(x: xPos, y: transform.position.y, z: transform.position.z);
    }
    return true;
}
```

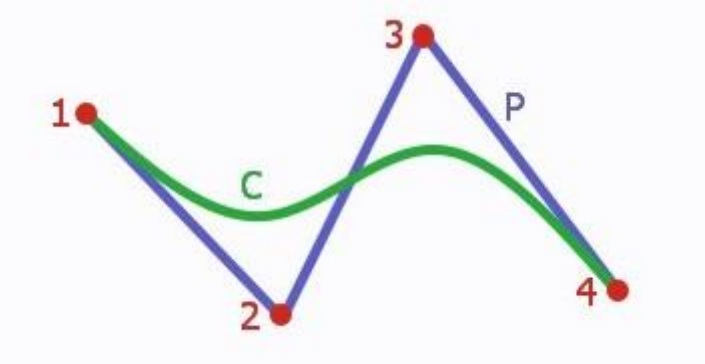
Kuva 31. Pelaajan liikkumistoteutus.

Liikkumismetodi kommunikoi pelaajan yksikkösyntymisen ja maailman laajentamismanagerin kanssa. Niistä saadaan tarkat tiedot ryhmä syntymisen koon ja maailman koon ääripäistä. Näiden kautta pystytään laskemaan liikkumisalue ja rajoittamaan sitä tarvittaessa.

4.8 Bézier-käyrä

Bézierin käyrä on yksinkertainen matemaattinen tapa tehdä käyriä. Yleensä sitä käytettiin kuvan käsittelyssä, ja se on saanut nimensä Pierre Bézierin mukaan, joka työskenteli 1960-luvulla Renaultin autotehtaalla suunnittelijana [22].

Bézierin käyrä ei ole monimutkainen. Se mahdollistaa tasaisesti kääntyvän käyrän kahden tai useamman pisteen välillä. Kuvassa 32 nähdään visuaalinen malli Bézier-käyrästä.



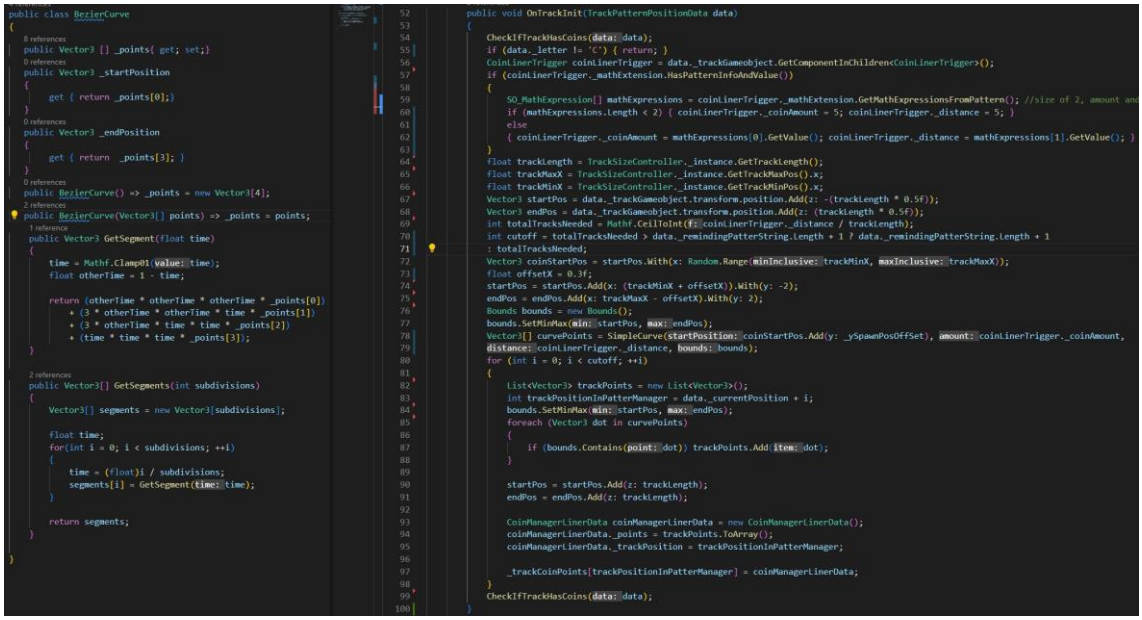
Kuva 32. Bézier-käyrämalli [23].

Juoksupeleihin käytetään kolmannen asteen Bézier-käyrää. Siinä siirrellään neljää pistettä niin, että muodostuu järkevän näköinen käyrä ajassa t. Jokainen piste vaikuttaa omalla painollaan muodostuneeseen käyrään. Ensimmäisellä pisteellä on suurin vaikutus alkuun, toisella pisteellä puolessa välissä käyrää jne. [23.]

Bézier-käyrä on loistava tapa tehdä juoksupeliin kaikenlaisia käyriä. Käyriä voidaan käyttää esim. tekemään kolikkojonoja, jota pelaaja seuraa kuvassa 33. Tämä kolikoiden syntyminen hoidetaan niin, että sillä on erillinen manageri, joka hoitaa kolikoiden synnyttämisen.

Kuva 33. Bézier-käyrä käytössä pelissä.

Bézier-käyrän koodillisen toteutuksen voi nähdä kuvassa 34. Bézier-käyrän koodillinen toteutus on suhteellisen yksinkertainen, mutta sen asentaminen esimerkiksi ratapaloihin on monimutkaisempaa, koska jotkin pisteet saattavat olla esteiden tai asioiden sisällä.



```
public class BezierCurve
{
    8 references
    public Vector3 [] _points{ get; set;}
    0 references
    public Vector3 _startPosition
    {
        get { return _points[0];}
    }
    0 references
    public Vector3 _endPosition
    {
        get { return  _points[3]; }
    }
    0 references
    public BezierCurve() => _points = new Vector3[4];
    2 references
    public BezierCurve(Vector3[] points) => _points = points;
    1 reference
    public Vector3 GetSegment(float time)
    {
        time = Mathf.Clamp01(value: time);
        float otherTime = 1 - time;

        return (otherTime * otherTime * otherTime * _points[0])
            + (3 * otherTime * otherTime * time * _points[1])
            + (3 * otherTime * time * time * _points[2])
            + (time * time * time * _points[3]);
    }

    2 references
    public Vector3[] GetSegments(int subdivisions)
    {
        Vector3[] segments = new Vector3[subdivisions];

        float time;
        for(int i = 0; i < subdivisions; ++i)
        {
            time = (float)i / subdivisions;
            segments[i] = GetSegment(time: time);
        }

        return segments;
    }
}
```

```
public void OnTrackInit(TrackPatternPositionData data)
{
    CheckIfTrackHasCoins(data: data);
    if (data._letter != 'C') { return; }
    CoinLinerTrigger coinLinerTrigger = data._trackGameobject.GetComponentInChildren<CoinLinerTrigger>();
    if (coinLinerTrigger._mathExtension.HasPatternInfoAndValue())
    {
        SO_MathExpression[] mathExpressions = coinLinerTrigger._mathExtension.GetMathExpressionsFromPattern(); //size of 2, amount and
        if (mathExpressions.Length < 2) { coinLinerTrigger._coinAmount = 5; coinLinerTrigger._distance = 5; }
        else
        { coinLinerTrigger._coinAmount = mathExpressions[0].GetValue(); coinLinerTrigger._distance = mathExpressions[1].GetValue(); }
    }
    float trackLength = TrackSizeController._instance.GetTrackLength();
    float trackMaxX = TrackSizeController._instance.GetTrackMaxPos().x;
    float trackMinX = TrackSizeController._instance.GetTrackMinPos().x;
    Vector3 startPos = data._trackGameobject.transform.position.Add(z: -(trackLength * 0.5f));
    Vector3 endPos = data._trackGameobject.transform.position.Add(z: (trackLength * 0.5f));
    int totalTracksNeeded = Mathf.CeilToInt(f: coinLinerTrigger._distance / trackLength);
    int cutoff = totalTracksNeeded > data._remindingPatterString.Length + 1 ? data._remindingPatterString.Length + 1
    : totalTracksNeeded;
    Vector3 coinStartPos = startPos.With(x: Random.Range(minInclusive: trackMinX, maxInclusive: trackMaxX));
    float offsetX = 0.3f;
    startPos = startPos.Add(x: (trackMinX + offsetX)).With(y: -2);
    endPos = endPos.Add(x: trackMaxX - offsetX).With(y: 2);
    Bounds bounds = new Bounds();
    bounds.SetMinMax(min: startPos, max: endPos);
    Vector3[] curvePoints = SimpleCurve(startPosition: coinStartPos.Add(y: _ySpawnPosOffSet), amount: coinLinerTrigger._coinAmount,
    distance: coinLinerTrigger._distance, bounds: bounds);
    for (int i = 0; i < cutoff; ++i)
    {
        List<Vector3> trackPoints = new List<Vector3>();
        int trackPositionInPatterManager = data._currentPosition + i;
        bounds.SetMinMax(min: startPos, max: endPos);
        foreach (Vector3 dot in curvePoints)
        {
            if (bounds.Contains(point: dot)) trackPoints.Add(item: dot);
        }

        startPos = startPos.Add(z: trackLength);
        endPos = endPos.Add(z: trackLength);

        CoinManagerLinerData coinManagerLinerData = new CoinManagerLinerData();
        coinManagerLinerData._points = trackPoints.ToArray();
        coinManagerLinerData._trackPosition = trackPositionInPatterManager;

        _trackCoinPoints[trackPositionInPatterManager] = coinManagerLinerData;
    }
    CheckIfTrackHasCoins(data: data);
}
```

Kuva 34. Koodillinen toteutus Bézier-käyrästä.

Bézier-käyrä on ohjelmoitu toimimaan kaavaratapalamanagerin kanssa. Se myös tietää maailman koon laajentumismanagerista. Kehittäjät voivat laittaa kaava pohjaan merkkijonoalueelle kirjaimen C. Tämä kirjain tarkoittaa, että tulossa on kolikkojono, joka on 10 kolikkoa ja 10 yksikköä pitkä. Haluttaessa kirjaimen perään voidaan laittaa parametrit C{x, y}, jossa x tarkoittaa, kuinka monta kolikkoa on ja y kuinka pitkä kolikkojono tulee olemaan. Lopuksi kolikkomanageri hoitaa kolikoiden ripottelemisen Bézier-käyrälle tasaisesti ja asettaa ne pelimaailmaan ottaen huomioon myös esteet, niin että se ei synnytä kolikoita esimerkiksi kiven, kalaverkon tai hain sisään.

5 Toteutuksien esillepano luotaessa uutta juoksupeliä

5.1 Testijuoksupelin tarkoitus

Insinöörityötä varten oli tarkoitus luoda testijuoksupeli, jossa mitataan, miten paljon toteutuksien esillepano vie aikaa. Pelissä oli tarkoitus keskittyä vain koodilliseen puoleen, joten grafiikka tuli pitää minimissä. Grafiikka tuli esittää niin, että siitä on ymmärrettävissä, mitä pelissä tehtäisiin, eli värimaailmaan tai malleihin ei panostettaisi. Pelin käyttöliittymäänkään ei juuri keskityttäisi. Käyttöliittymä on oma osaamislajinsa, ja siihen yleensä tarvitaan suuri määrä suunnittelua sekä yhden tai useamman henkilön täysiaikainen panostus.

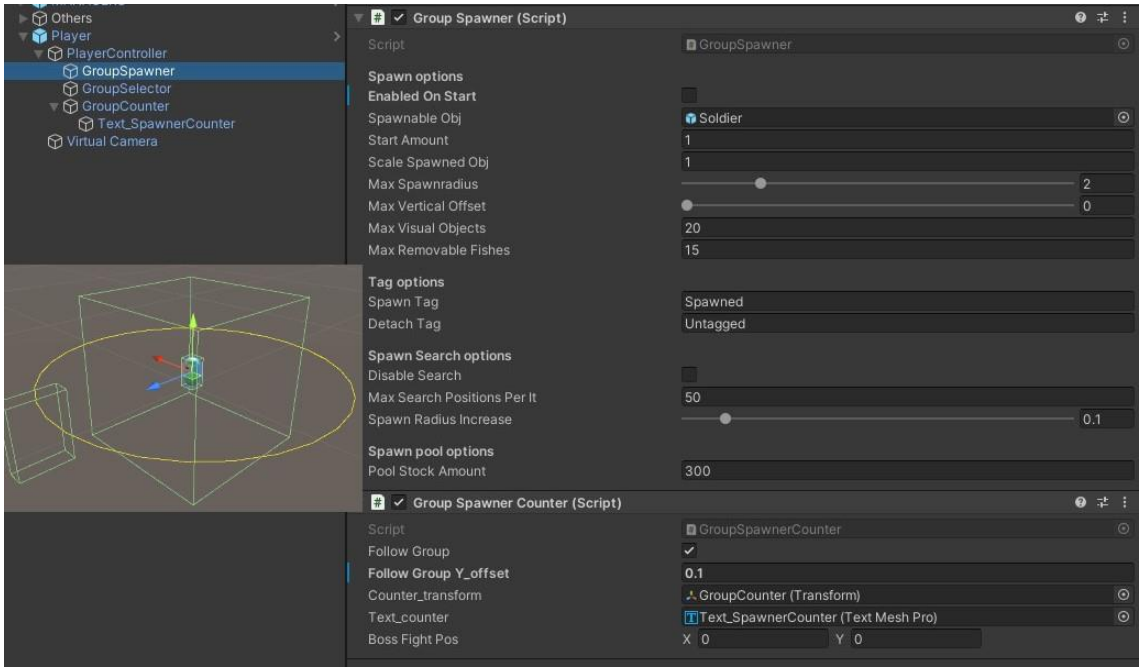
Testipelissä tarkastellaan koodillisia toteutuksia kriittisesti: mitkä asiat sujuivat ja mitkä tarvitsevat korjausta tai joudutaanko toteutuksia jättämään pois, onko toteutuksien pääidea toiminut ja onko se ideaalia juoksupeleille sekä pystyykö ulkopuolinen ohjelmoija ottamaan koodilliset toteutukset suoraan käyttöön vai ei.

5.2 Pelin tuottaminen ja toteutuksien käyttö

Testijuoksupeliä lähdettiin rakentamaan tekemällä tyhjä projekti Unityssä ja laittamalla toteutuksien koodilliset skriptit projektiin. Toteutuksissa oli hieman muunneltavaa niin, että koodia jouduttiin karsimaan. Tämä tarkoittaa, että skripteissä saattoi esiintyä tarpeettomia koodipätkiä tähän testipeliin.

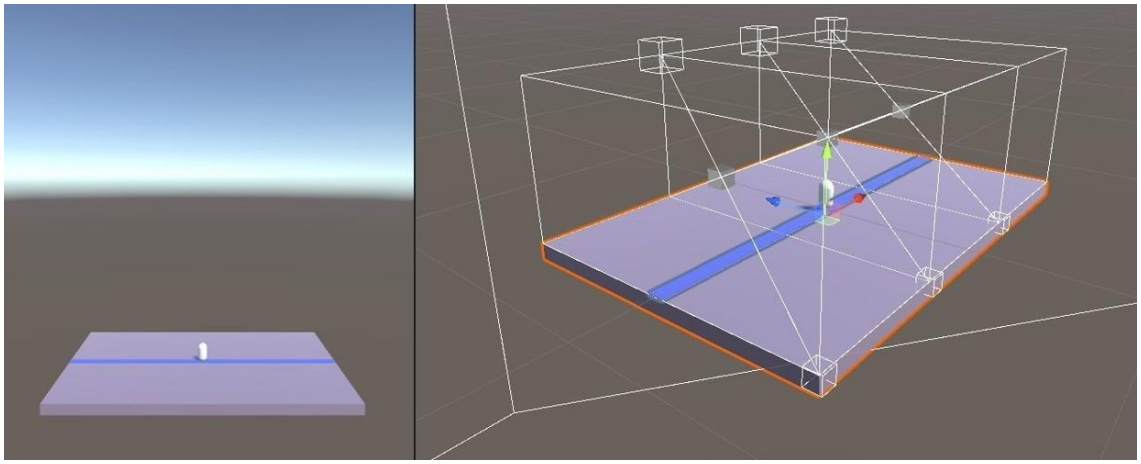
Tarpeettomat koodipätkät johtuivat yleensä siitä, että koodilliset toteutukset olivat käytössä oikeassa juoksupelissä, jota oli rakennettu useamman kuukauden ajan ja kehittäjät olivat lisänneet asioita toteutuksiin. Onneksi toteutuksia ei muutettu vaan asioita oli vain lisätty ja ne oli helppo ottaa pois. Toteutuksien tuominen projektiin muokkauksineen vei noin tunnin.

Kun kaikki toteutukset oli tuotu peliin, lähdettiin rakentamaan pelaajaa. Tarkoituksena oli luoda pelaaja ja saada se liikkumaan ensiksi. Pelaajalle oli annettava pelaajakontrolleri, joka ohjaa pelaajan liikettä ja kameraa. Ryhmäsyntyminen asennettiin lapsiin, samoin sille olennaiset osat kuten ryhmäsyntymisen laskentalisäosa, joka näyttää, kuinka paljon ryhmässä on yksiköitä, sekä pelaajavalitsinlisäosa, joka tarkkailee ryhmän kokoa ja sen mittasuhteita. Arvojen laittamiseen kohdalleen ja sen päättämiseen, kuinka ryhmäsyntyminen halutaan esittää, meni aikaa noin 40 minuuttia. Pelaajan liikkuminen horisontaalisesti ja eteenpäin toimi hyvin asennuksien jälkeen. Kuvassa 35 nähdään oleellisimmat osat pelaajan asennuksesta.



Kuva 35. Unity-editorissa oleva pelaajan hierarkia.

Kun pelaaja saatiin liikkumaan ja ryhmässä syntyminen asennettua, oli aika luoda pelaajarata. Ensin luodaan tyhjä ratapala-prefab ja tehdään siitä sen kokoinen ja näköinen kuin halutaan. Tämän jälkeen lisätään laajentamismanageri, jolla päätetään, kuinka monta linjaa ja kuinka iso rata on 3D-maailmassa. Sitten rata-prefablle luodaan kaavapohjaosaolio ja annetaan sille tunnistetyyppi ”E”. Tähän meni vain 20 minuuttia. Toteutuksen suunnittelua voidaan nähdä kuvassa 36.

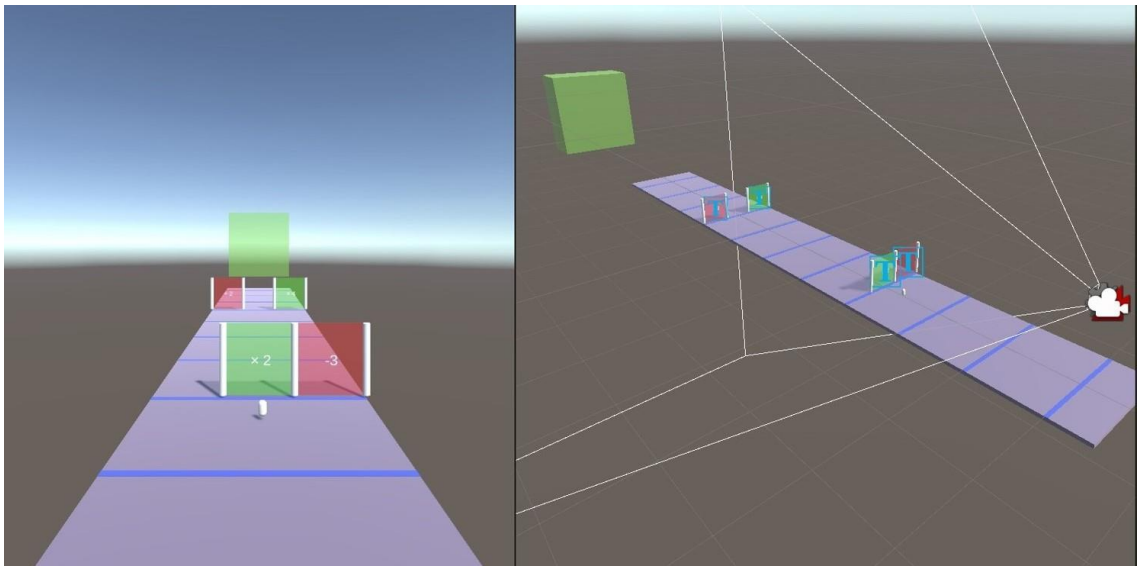


Kuva 36. Rata-prefabia luotaessa valkoiset viivat tulevat ratalaajentamismanagerista.

Seuraavaksi otetaan käyttöön ratalaajentamis- ja kaavamanageri. Tähän lisätään ratatuottaja, joka hoitaa rata-prefabien laiton paikoilleen. Kaavamanageri tarvitsee kaavapohjatasoja, joten tehdään niitä muutamia. Nämä ovat Unityn Scriptable Objecteja, joten ne ovat klikkauksien päästä valmiita. Kaavapohjiin voidaan tällä hetkellä antaa vain merkkijono tunnistetyypille pelkkää E-kirjainta käyttäen. Ratatuottaja tarvitsee vielä radan päättymispalasen, joten sille annetaan vihreä suorakulmainen objekti, joka kuvastaa radan loppusekvenssiä. Radan luonti testataan, ja se toimii.

Seuraavaksi siirrytään radalle luotaviin asioihin. Luodaan uusi ratapala, johon laitetaan porttikontrolleri. Suunnitellaan portin ulkonäkö ja annetaan sille colliderit ja tekstipaikat. Tehdään portista prefab ja annetaan sille portti- ja matematiikkakomponenttiskriptit ja laitetaan niihin tarvittavat arvot. Ratapalalle annetaan

porttikontrolleri, joka saa portti-prefabin. Porttikontrolleri hoitaa porttien asettelun kommunikoiden ratalaajentamismanagerin kanssa. Se voi esimerkiksi laittaa yhden portin keskelle rataa tai vaikka kolme porttia ratalaajentamismanagerin linjamäärien mukaan. Tuodaan myös porttimanageri, joka kirjaa jokaisen porttikontrollerin syntymisen. Testataan porttien ja radan tuottaminen. Luonti toimii mutkitta, mikä näkyy myös kuvassa 37. Pelaajaradan luontiin, portteihin ja porttien toimintaan kului aikaa noin kolme tuntia.

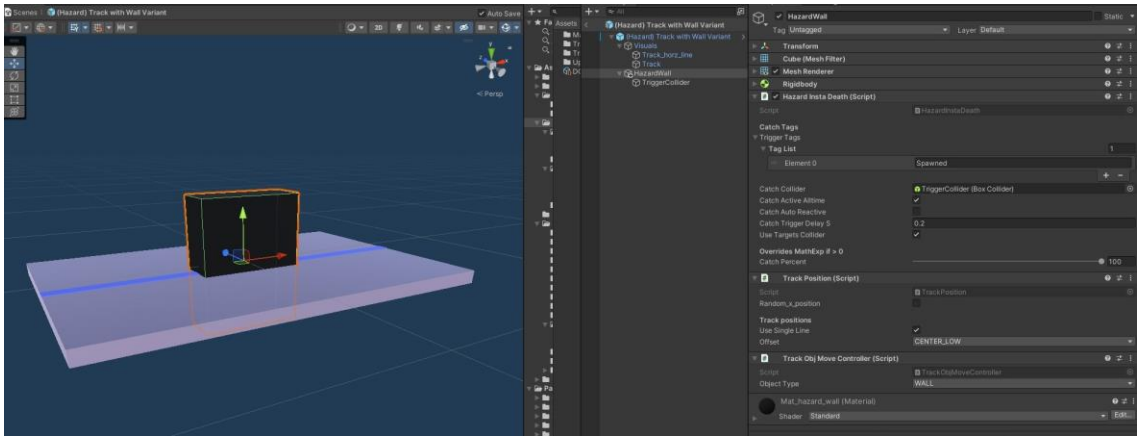


Kuva 37. Pelaajaradan luominen portteineen.

Ympäristö näyttää tosi tyhjältä, joten on aika sen luomiseen. Tuodaan ympäristömanageri ja tehdään ympäristö Unityn sisäisellä ympäristönluomisohjelmalla. Tehdään ympäristöstä prefab ja laitetaan se ympäristömanageriin ja kerrotaan managerille, kuinka iso ympäristö on. Ympäristömanageri hoitaa loput ympäristön asettamisesta ja tuhoamisesta. Nyt on ympäristö asennettu, ja siihen kului aikaa 30 minuuttia

Ympäristön ja porttien asennusten jälkeen lisätään enemmän haastetta radalle. Lisätään esteitä, kuten seiniä, tulikuumia pilareita ja laavaa. Jokaiselle esteelle tehdään omat ratapalat niin kuin tehdessä porttiratapalaa tai tyhjää ratapalaa. Jokainen este on uniikki omalla tavallaan, mikä tarkoittaa, että joidenkin esteiden halutaan olevan tietyssä paikassa, liikkuvan tai imevän pelaajayksiköitä.

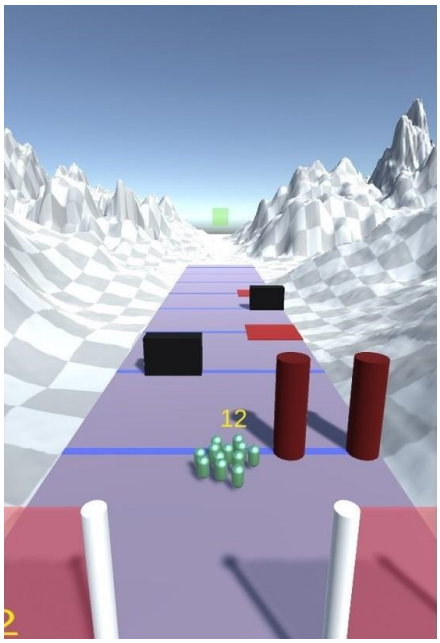
Kuitenkin jokaiselle esteelle on ominaista periä pyydysaluekomponentti. Seinäeste voidaan ottaa tarkempaan katseluun. Seinään lisätään välitön kuolema -komponentti, joka perii pyydysalueen luokan. Kuvassa 38 voidaan nähdä seinälle asennettu välitön kuolema -komponentti. Seinälle myös annetaan ratasijaintikomponentti, jolla pystytään määrittämään esteen sijainti eri paikoissa. Jos sijaintikomponenttia ei olisi, seinä pysyisi aina siinä kohdassa, mihin se prefabissa on laitettu, mikä voi olla ominaista joillekin esteille.



Kuva 38. Seinäratapalan prefab.

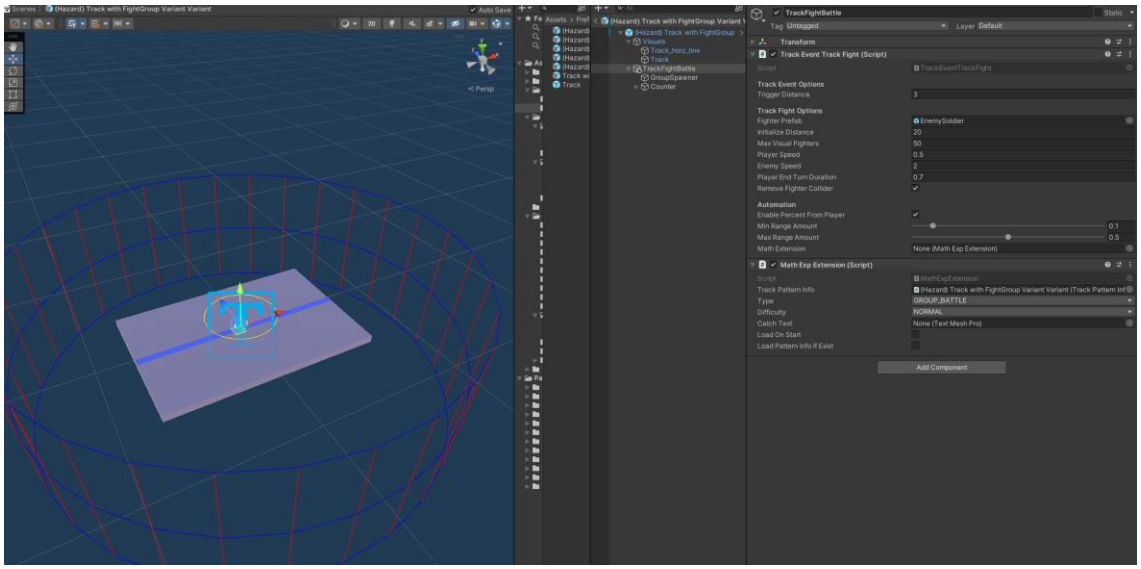
Seinälle haluttiin kolme ominaisuutta. Ensimmäinen ominaisuus on, että seinä syntyy johonkin sijaintiin. Ensimmäisen ominaisuuden hoitaa sijaintikomponentti. Toinen ominaisuus on, että seinä laitetaan liikkumaan sivuttain tiettynä ajanjaksona. Tehdään seinälle kaksi Unityn liikkumisasetus-ScriptableObjectia, joista toinen on kolmas ominaisuus. Annetaan seinälle liikkumiskontrolleri, joka kommunikoi liikkumisasetusmanagerin kanssa. Liikkumisasetusmanageri hoitaa liikkumisasetuksien tuomisen tiedostoista kontrollereille. Toinen ja kolmas ominaisuus ovat melkein samat, mutta kolmas ominaisuus on vaikeampi seinä, joka liikkuu sivuttain nopeampaan tahtiin. Nyt seinissä on kolme ominaisuutta. Seinät voivat olla paikoillaan jossain kohtaa radalla ja liikkua sivuttain kahdella eri nopeudella. Seinän liikkumisasetuksia voidaan tehdä kuinka paljon tahansa muutamalla napinpainalluksella.

Lopuksi annetaan esteille omat tunnistetyypit ja luodaan vielä oma yhteinen tunnistetyyppi, esimerkiksi H, jossa on kaikki esteet. Jos tämä H on tunnistetyyppimerkkijonossa, se ottaa satunnaisesti jonkin esteen. Ajallisesti kolmen eri esteen asentamiseen meni vain 30 minuuttia ja seinäesteen liikkumisasetuksien laittoon noin 15 minuuttia. Kuvassa 39 nähdään esteet toiminnassa pelimaailmassa.

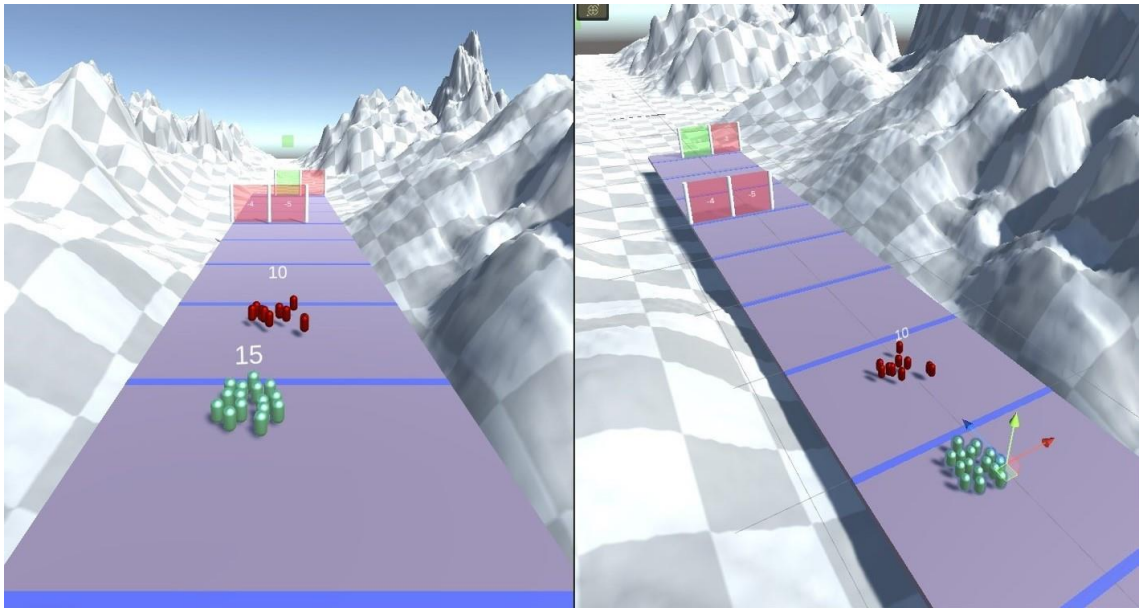


Kuva 39. Esteiden testaus.

Nyt kun muutama este on asennettu ja toiminnassa, on aika ratatapahtumalle. Asennetaan tappeluratatapahtuma. Luodaan sama pohja kuin porteille tai tyhjälle ratapalalle, mutta lisätään ratapalaan tappelutapahtuma. Sille on ominaista periä ratatapatumapääluokka, ohjata tappelijoita pelaajan yksiköitä vastaan ja kommunikoida omassa ryhmässä syntymiskomponentin kanssa. Pääluokka aktivoi tapahtuman, kun pelaaja menee tarpeeksi lähelle. Tämän ratatapahtuman asentamiseen meni 25 minuuttia, mutta jos tapahtumassa ei olisi käytetty valmiiksi tehtyä ratatapahtumaa, olisi aikaa kulunut huomattavasti enemmän. Tämä tapahtuma on kuitenkin hyvin yleinen nykyisissä juoksupeleissä. Kuvassa 40 nähdään tappeluratatapahtuma-prefabin tekemistä ja kuvassa 41 tapahtuma pelin aikana.



Kuva 40. Ratatappelutapahtuman ratapala-prefab.

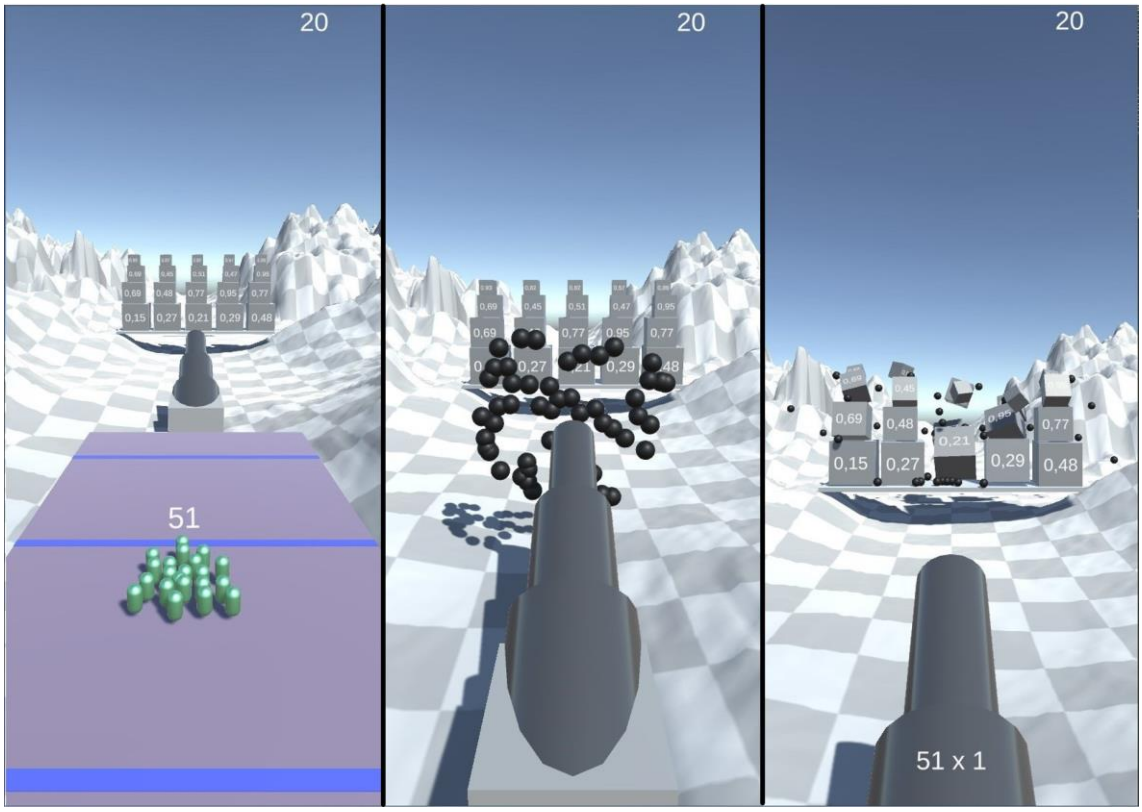


Kuva 41. Tappelutapahtuma pelissä.

Kun rata on täytetty haasteilla, on loppusekvenssin aika. Radoille suunniteltiin nopeasti toteutettava lopetus. Kun pelaaja pääsee radan loppuun, menevät pelaajan yksiköt eli tässä pelissä kapselihahmot tykkiin sisään. Tykin eteen syntyy tykinkuulia saman verran kuin pelaajan yksiköitä, satunaisesti tykinsuun ympärille ympyrämäisesti. Menee hetki ja tykinkuulat laukaistaan pöytää kohti, jossa

on laatikoita erilaisilla kertoimilla. Laukaisusuunta on hieman satunnainen, ja jotkut kuulat voivat mennä pöydän ohi.

Tarkoituksena on saada mahdollisimman moni laatikko pudotettua pöydältä alas, jolloin pelaaja saa laatikossa olevan kertoimen lisättyä yhteiskertoimeen, joka lopuksi kerrotaan pelaajan yksiköiden määrällä, josta syntyvät pelaajan rahavarat. Mitä vähemmän pelaajalla on yksiköitä, sen pienempi mahdollisuus sillä on kaataa laatikot. Koko tapahtuma on automaattinen, eikä pelaajan tarvitse kuin odottaa. Koko loppusekvenssiin kului noin kolme tuntia aikaa. Kuvassa 42 nähdään loppusekvenssi.

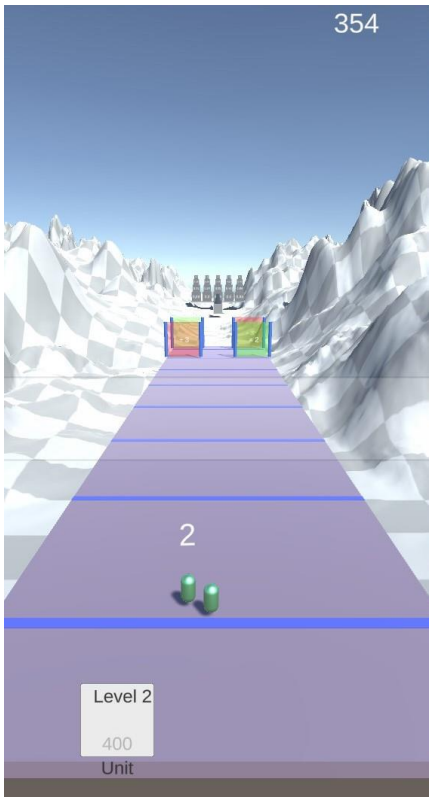


Kuva 42. Testipelin loppusekvenssi.

Loppusekvenssin jälkeen on aika siirtyä uuteen tasoon, jonka hoitaa tasomanageri. Sen tarkoitus on vaihtaa tapahtumapaikka, nollata kaikki singletonit, jotka eivät tuhoudu tapahtumapaikkaa vaihdettaessa, ja kertoa ratakaavamanagerille seuraava taso. Jokaisesta managerista löytyy tapahtumapaikan nollausmetodi, jonka tasomanageri kutsuu.

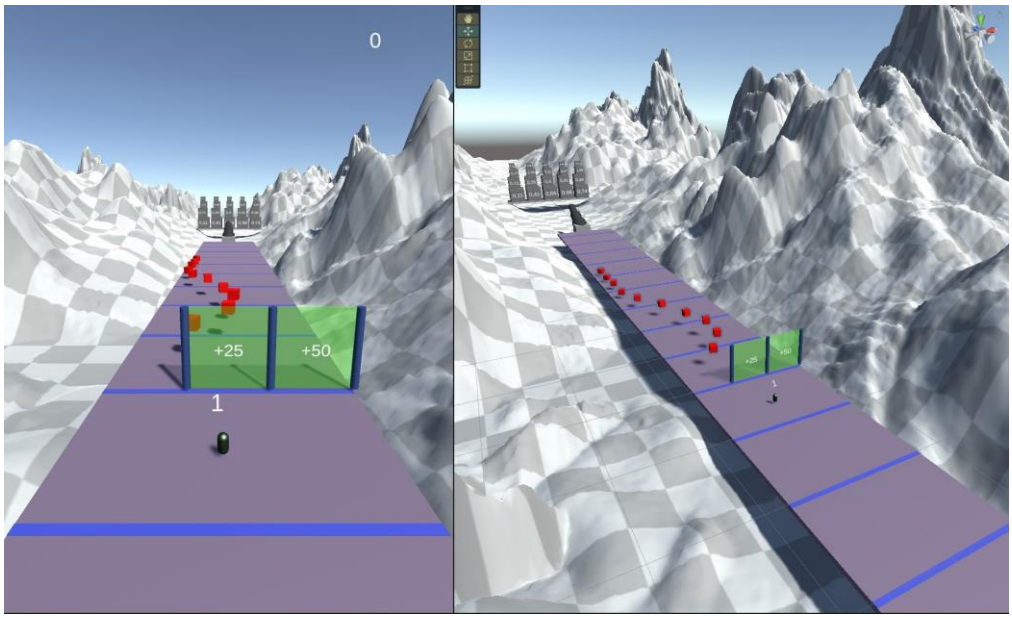
Peli siis toimii nyt sulavasti silmukassa, jossa pelaaja suorittaa tason väistellen esteitä tason loppuun, missä pelaaja saa rahansa. Uusi taso alkaa, ja kun käsin tehdyt tasot loppuvat, tekee kaavamanageri uusia tasoja yhdistelemällä muita tasoja. Tämänkin ominaisuuden voi muuttaa helposti valmiilla metodeilla niin, että tasot ovat täysin satunnaisia. Koska rahaa ei käytetä mihinkään, tarvitsee peli rahan käytölle jonkin ominaisuuden.

Luodaan päivitysmanageri ja päivityspohja. Päivityspohja asennetaan käyttöliittymään nappina, ja päivitysmanageri hoitaa pelaajarahan laskemisen ja toteutuksen, kun pelaaja aktivoi päivitysnapin. Päivitys on pelaajayksiköiden alkumäärän lisääminen. Pelaaja käyttää rahaa ostaakseen enemmän yksiköitä pelin alkuun, koska jokainen taso alkaa yhdellä yksiköllä. Mutta jos pelaaja ostaa kaksi yksikköpäivitystä, hänellä on tässä ja seuraavilla tasoilla kolme yksikköä. Päivitysjärjestelmään meni aikaa noin yksi tunti. Kuvassa 43 nähdään yksikköpäivitysnappi käyttöliittymässä, joka on jo tasolla 2. Kuvasta voidaan huomata, että pelaajalla on tällöin kaksi kapselia pelaajayksiköitä.



Kuva 43. Yksikköpäivitysnappi tasolla kaksi käyttöliittymässä.

Lopuksi lisättiin vielä kolikkojonot, joissa pelaaja voi saada vähän lisätienestiä. Tehdään taas ratapala, joka toistui jo monessa aiemmassa kohdassa. Ratapalalla annetaan komponentti, jossa säilytetään kolikkojonon data eli esimerkiksi 10 kolikkoa ja 10 yksikköä pitkä kolikkojono. Tämä kuitenkin voi muuttua, jos kaavapohjamanagerissa on tässä kohtaa annettu ratapalalle erilaiset parametrit ja ne on asetettu tähän dataan. Kun ratapala syntyy, hoitaa kolikkomanageri kolikoiden laittamisen datan mukaan. Kolikkomanagerille piti myös tehdä kolikkoprefab, jota se asettaa radalle. Prefabille on ominaista, mitä kolikon kerääminen antaa, eli tässä tilanteessa se antaa yhden rahayksikön. Kuvassa 44 voi nähdä kolikkojonototeutuksen pelissä. Kolikkojonojen tekemiseen meni vain 20 minuuttia.



Kuva 44. Kolikkojonot pelissä.

Lopputuloksena on sujuva juoksupeli ilman grafiikkaa. Pelaaja pystyy käyttämään tasoissa saatuja rahoja päivitykseen. Jokainen taso on omalla laillaan uniikki. Juoksupelille saa paljon lisäsisältöä, kun lisätään esteitä, esteiden liikkumista tai vaikka päivitysnappeja. Tason lopetuksia voi lisätä, jos haluaa, ja mainokset voidaan asettaa haluttuihin kohtiin.

5.3 Toteutuksen huomioita

Testipeliä tehdessä tasojen, liikkumisasetuksien tai taso-osien hakeminen Unityn omalla resource loadilla ei ollut järkevää, koska asennusprosessin aikana ei ollut tietoa, mistä kansioista kyseiset toteutukset hakevat tietonsa. Tämä osoittautui haasteeksi esimerkiksi radanluomisprosessissa. Tulevaisuudessa asiat olisi järkevämpää laittaa Untityn tarkastelijaan prefabeina, ja tarkastelijan kentät paljastavat, mitä asioita pitää manageriin laittaa, jotta se toimii.

Yksi suurimmista ongelmista ja eniten aikaa vievistä asioista testipeliä tehdessä oli kokonaisuuksien yhtäläisyys eli se, mitkä komponentit kommunikoivat toistensa kanssa. Esimerkiksi radanluomisprosessissa tarvittavat osat olivat ratapala-prefab, kaavapohja, kaavapohjatunnistetyyppiosa, radanluomisobjekti ja

laajentumismanageri. Yhtäläisyysongelma ilmeni myös portteja tehdessä, johon tarvittiin neljä erilaista asiaa lukuun ottamatta colliderien laittamista tms.

Liikkumisasetuksissa on vielä parannettavaa. Erilaisille esteille pitäisi tehdä niiden äärirajat, josta liikkumisasetus osaa katsoa liikkumistilavuuden. Jos kehittäjä ei liikkumisasetuksia säädellessään tiedä, kuinka iso seinäeste on, voi seinä liikkua yli pelaajaradan rikkoen pelin tunnelman ja tuoda keskeneräisen pelin vaikutelman.

Yhtäläisyys eri toteutuksien välillä on epäselvää, eli toteutuksien aktivointijärjestys pitäisi olla tiedossa. Jotkut toteutukset eivät lue dataa, jos se esimerkiksi on aktivoitu vasta sille tärkeän toteutuksen jälkeen. Näitä kuitenkin oli vain muutama. Aktivointijärjestystä koskeva ongelma esiintyi esimerkiksi tasojen vaihtuessa. Ryhmäsyntyminen oli jostain syystä jäänyt aktivointiprosessin taakse, mikä tuotti ongelmia. Tunnin etsimisen jälkeen virhe huomattiin ja korjattiin. Insinöörityön toteutuksia tehdessä pyrittiin minimoimaan aktivointijärjestyksien merkitys, mutta näin laajassa kokonaisuudessa ei aina pystynyt virheitä estämään.

5.4 Ajallinen hyöty toteutuksia käyttäessä

Toteutuksia on paljon, ja niihin meni aikaa noin kolme kuukautta suunnitellessa, tehdessä ja parannellessa. Kolmen kuukauden aika on suhteellisen pitkä, kun verrataan hyper-casual-genreen, mutta todellisuudessa arvo ja aika, jonka sillä saavuttaa, on kaiken sen arvoista. Taulukkoa 1 tarkastellen voidaan huomata ratatuottajan ja loppusekvenssien vieneen eniten aikaa. Muussa toteutuksien esillepanossa kesti yleensä vain alle tunnin. Kun asiat laskee taulukosta yhteen, voidaan todeta, että koko testijuoksupelin tekemiseen meni 11,5 tuntia insinöörityön toteutuksia käyttäen. Saavutus oli merkittävä. Peli toimii koodillisesti, mutta tarvitsee käyttöliittymän ja grafiikkaa. Grafiikan tuominen vaatii oman aikansa, ja käyttöliittymän tekeminen on oma prosessinsa. Käyttöliittymänkin voisi toteuttaa uudelleen käytettäväksi juoksupeleissä, koska jokaisessa juoksupelissä se on lähes samanlainen.

Taulukko 1. Testijuoksupelin vaiheisiin kuluneet ajat.

Vaiheet	Aika (h)
Toteutuksien tuominen	1,0
Pelaaja ja ryhmäsyntyminen	0,7
Ratapalan luonti	0,3
Ratatuottaja ja siihen portit	3,0
Ympäristön luonti	0,5
Esteiden luonti	0,8
Tappeluratatapahtuman asennus	0,4
Loppusekvenssin suunnittelu ja toteutus	3,0
Pelaajapäivitys ja alkeellinen käyttöliittymä	1,0
Kolikkojonojen luonti	0,3

Testipeliä tehdessä aikaa kului eniten etsimiseen ja siihen, mitkä luokat kommunikoivat toistensa kanssa sekä loppusekvenssin suunnitteluun. Jos näihin olisi ollut valmiit dokumentit ja suunnitelmat, olisi tuotantoaika lyhentynyt testipelissä muutamilla tunneilla, mitä voi verrata noin 15 % – 30 % nopeammaksi ajaksi. Koodillisten toteutusten pohja on hyvä uusille implementaatioille ja jatkokehitykselle, mikä johtaa vielä lyhyempään pelituotantoaikaan.

5.5 Toteutuksien tuominen muihin peleihin

Muutamat toteutukset toimivat muissa peleissä, mutta suurin osa toteutuksista on sidottu loputtomaan radan syntymiseen ja sen kommunikoimiseen. Koodia muuttamalla minkä vain toteutuksen saa mille vain pelille sopivaksi, mutta ilman koodin muuttamista toteutuksia on vähän.

Ryhmäsyntymisen toteutus voidaan laittaa melkein mihin vain peliin, jossa halutaan luoda parvia, tasopisteitä tai objekteja tasossa. Esimerkiksi testipelissä olisi voitu käyttää loppusekvenssissä ryhmäsyntymistä tykinkuulien syntymiseen.

Laajentamismanageri on yleishyödyllinen kokojen mittaaja. Sitä voidaan käyttää erilaisten kokojen tekemiseen ja laskemisen. Jos halutaan luoda alue, se voidaan jakaa kuinka moneen osaan vain laajentamismanagerilla ja kysyä

yksittäisten linjojen sijainteja. Sillä pystyy myös esimerkiksi tekemään taulukon 2D-maailmassa, jos haluaa.

Bézierin-käyrä on myös yleishyödyllinen toteutus. Jos tarvitaan erilaisia käyriä, ne saadaan aika varmasti Bézierin käyrällä. Tätä voi käyttää esimerkiksi vuoristoradan luomiseen tai vaikka objektin nopeuden säätämiseen tietyissä ajanjaksoissa.

5.6 Toteutuksien yhteenveto

Toteutuksien esillepano toimi melkein odotetusti. Juoksutestipeli saatiin valmiiksi, eikä toteutuksia tarvinnut juuri muuttaa. Testipelin rakentamisessa aikaa vei eniten pelin suunnittelu ja toteutuksien yhtäläisyyksien etsiminen. Suunnittelulla tarkoitetaan sitä, mitkä esteet halutaan peliin, minkälainen loppusekvenssi on, mitä rahalla tehdään tai mikä pelin päätarkoitus on.

Toteutuksissa selvisi muutamia ongelmia, ja ilman tarkkaa perehtymistä toteutuksien käyttö oli haasteellista. Tämä tarkoittaa sitä, että toteutusten ja niiden yhtäläisyyksien dokumentointi on jopa melkein pakollista. Pieni testipeli dokumentaation ohessa vahvistaisi ymmärrystä toteutuksien käytöstä.

Vaikka testipeliä tehdessä oli pieniä ongelmia, oli pelin tuottamisaika erinomainen. Testijuoksupeli pystyttiin tuottamaan toiminnalliseksi noin 11,5 tunnissa. Muutamat toteutukset toimivat myös muissa peleissä, mikä on positiivista.

6 Yhteenveto

Hyper-casual-pelejä lähdettiin insinöörityössä tutkimaan odottaen, että ne ovat hyvin yksinkertaisia toteuttaa. Hyper-casual-kehitykseen vaikuttavat monet asiat, jotka eivät ole koodillisesti riippuvaisia. Esimerkiksi pelin pääidean suunnittelu, graafinen puoli ja käyttöliittymän suunnittelu osoittautuivat paljon työläämmiksi, kuin arvioitiin, ja veivät yllättävän paljon aikaa. Oli kyseessä sitten

juoksu- tai hyper-casual-peli, nämä kolme osa-aluetta vaativat runsaasti osaamista.

Juoksupelit ovat kehittyneet huimasti vuosien varrella ja tuoneet monenlaisia pelejä markkinoille. Monet pelit kuitenkin hukkuvat markkinapaikalle, mutta useamman pelin tekeminen tuo suuremman todennäköisyyden päästä joskus pelatuimpien listalle.

Yksi huomattavan tärkeä ominaisuus insinöörityötä tehdessä ja juoksupelejä tutkiessa oli, kuinka pitää pelaaja koukussa peliin. Tämä asia on elintärkeä ominaisuus, koska mitä pidempään pelaaja saadaan pelaamaan peliä, sitä enemmän se tuottaa mainostuloja. Vaikka mainosten näyttämislogiikkaa ei tehty insinöörityössä, sen toteutus kuitenkin nähtiin Free The Fish -testipelissä. Tämän myötä tehtiin johtopäätös, että mainosten näyttämisen tahti, niiden implementaatio ja suunnittelu eivät olleet yksinkertainen prosessi. Jokaisen mainoksen näyttämisellä on tarkoituksensa, miksi kyseinen mainos tulee. On pelejä, jotka työntävät mainoksia vähän väliä välittämättä pelattavuudesta lainkaan, jolloin ne asettavat itse pelin nopeaan elinkaaren hiipumiseen.

Toteutuksien luominenkaan ei ollut nopea prosessi. Toteutuksien tuottaminen ja parantelu jatkui koko pelin kehityksen ajan. Aika ajoin tuli tilanteita, jolloin tarvittiin jokin ominaisuus, ja loppujen lopuksi toteutuksien koodilliset pituudet nousivat tuhansiin riveihin. Tässä insinöörityössä pystyttiin näyttämään vain murto-osa koodillisista toteutuksista.

Toteutuksien esillepano uudessa testipelissä oli vaikuttava näky. Toteutukset todella toimivat koskematta koodiin. Toimiva testijuoksupeli tuotettiin 11,5 tunnissa. Koodillisten toteutuksien puutteet tulivat myös selkeästi esille ja tarkoittivat sitä, että kehitystä tarvitaan. Dokumentaation tekeminen on seuraava askel kohti parempaa kehitysympäristöä. Insinöörityön pääperiaatteet toteutuivat, ja valmiiksi saatiin vahva kooditoteutuspaketti, jolla pääsee tekemään tarvittaessa melkein minkälaista juoksupeliä tahansa.

Lähteet

1 Williams, Zach. Flappy Bird. Verkkoaineisto. Zachwill. <https://zachwill.com/flappy-bird/> Luettu 30.10.2022.

2 Calderon, Christian. 2019. KEYNOTE: A Brief History of Hyper Casual. Verkkoainesto. Youtube. <https://www.youtube.com/watch?v=8MVEN24OONM>. Luettu 30.10.2022.

3 Flappy Birdin taival katkesi kuin kanan lento – syy yllättää. 2014. Verkkoaineisto. Ilta-Sanomat. <https://www.is.fi/digitoday/art-2000001825753.html>. Luettu 8.12.2022.

4 Kännykkäpelistä tuli liian hyvä – kokeile itse! 2014. Verkkoaineisto. Iltalehti. <https://www.iltalehti.fi/digi/a/2014021018025836>. Luettu 8.12.2022.

5 Flappy Bird creator removes game from app stores. 2014. Verkkoaineisto. BBC. <https://www.bbc.com/news/technology-26114364>. Luettu 30.10.2022.

6 Hanicar, Matija. 2022. Hyper Casual Games. Mobile Gaming's Most Profitable Genre? Verkkoaineisto. Udonis. <https://www.blog.udonis.co/mobile-marketing/hyper-casual-game>. Luettu 20.11.2022.

7 2048. 2014. Gabriele Cirullin.

8 Stack. 2016. France: Ketchapp.

9 Flappy Bird. 2013. Vietnam: Gears.

10 Count Masters: Crowd Runner 3D. 2021. Freeplay Inc.

11 Shtoyer, Roy. 2019. TabTale on the state of hyper-casual and how it's found success. Verkkoaineisto. PocketGames. < https://www.pocketgamer.biz/comment-and-opinion/69941/tabtale-on-the-state-of-hyper-casual-and-how-its-found-success/>. Luettu 20.11.2022.

12 Cui, Tom. 2022. State of hyper-casual games in 2022 by Tom Cui (Sensor Tower). Verkkoaineisto. Youtube. <https://www.youtube.com/watch?v=T5cYlhBz1A4>. Luettu 20.11.2022.

13 SteinBach, Rae. 2022 .The past, present, and future of hyper-casual runner games. Verkkoaineisto. Supersonic. <https://supersonic.com/blog/the-

past-present-and-future-of-hyper-casual-runner-games/>. Luettu 1.11.2022.

14 Crash Bandicoot. 1996. United States: Naughty Dog.

15 Subway Surfers. 2012. Denmark: SYBO.

16 Color Road. 2018. France: Voodoo.

17 Heinze, Johannes. 2017. The ascendance of hyper-casual part one: Drawing the lines between the different gaming genres. Verkkoaineisto. Pocket Gamer. <https://www.pocketgamer.biz/comment-and-opinion/65256/the-ascendance-of-hypercasual/>. Luettu 20.11.2022.

18 Aquapark.io. 2019. France: Voodoo.

19 Hair Challenge. 2021. Rollic Games.

20 Karnes, KC. Hyper Casual Games Marketing & Monetization Strategies – Gamify. Verkkoaineisto. Gamify. <https://www.gamify.com/gamification-blog/marketing-on-hyper-casual-games-101>. Luettu 20.11.2022.

21 Hunter, Chay. 2022. Six features that turn a hyper-casual dud into a hybrid-casual hit. Verkkoaineisto. GameAnalytics. <https://gameanalytics.com/blog/six-features-hyper-casual-dud-hybrid-casual-hit/>. Luettu 20.11.2022.

22 Cederberg, Janne. Bézier-käyrät, kuvankäsittely ja tuotesuunnittelu. Verkkoaineisto. OpetusTV. <https://opetus.tv/2014/10/bezier-kayrat-kuvankasittely-ja-tuotesuunnittelu-osa-1/>. Luettu 5.11.2022.

23 What is a NURBS. 2006. Verkkoaineisto. Rw-Desinger. <http://www.rw-designer.com/NURBS>. Luettu 7.11.2022.