Uudelleenkäytettävyys hyperkasuaalisten pelien kehityksessä

Jesse Ahonen

Aleksi Kateisto

Opinnäytetyö, AMK
Toukokuu 2021
Liiketalouden ala
Tradenomi, tietojenkäsittelyn tutkinto-ohjelma

Jyväskylän ammattikorkeakoulu
University of Applied Sciences

jamk

Kuvailulehti

Ahonen, Jesse & Kateisto, Aleksi

Uudelleenkäytettävyys hyperkasuaalisten pelien kehityksessä

Jyväskylä: Jyväskylän ammattikorkeakoulu. Toukokuu 2021, 39 sivua

Liiketalouden ala. Tradenomi, tietojenkäsittelyn tutkinto-ohjelma. Opinnäytetyö AMK.

Julkaisun kieli: suomi

Verkkojulkaisulupa myönnetty: kyllä

Tiivistelmä

Älypuhelinten yleistymisen myötä mobiilipelit ovat nousseet suureksi osaksi peliteollisuutta. “Hyper-casual” on kasvavassa mobiilipeliteollisuudessa hiljattain syntynyt termi, jolla tarkoitetaan erittäin yksinkertaista, helposti omaksuttavaa ja lyhyeen pelikokemukseen tarkoitettua peliä. Kyseiset pelit ovat myös verrattain nopeita tuottaa, tehden niistä houkuttelevan vaihtoehdon varsinkin indie-pelifirmoille.

Toimeksiantajana toimiva Zaibatsu Interactive on mobiilipeleihin keskittyvä, Keski-Suomen suurin pelialan yritys. Heidän tavoitteenaan oli selvittää keinoja tuottaa uusia hyperkasuaalisia pelejä entistä pienemmällä vaivalla ja kehitysajalla, käyttäen aikaisemmin tuottamiaan pelejä pohjana uusille kokemuksille. Tätä varten tutkimuksessa selvitettiin kehittämistutkimuksen keinoin vanhan pelin koodin pohjalta rakennettavien muunnelmien kehityksen haasteita, pääosassa ohjelmoinnin sekä pelisuunnittelun osalta. Lisäksi tutkittiin koodin uudelleenkäytettävyyden teoriaa ja sen soveltamista Unity-pelimoottorissa.

Hankittua teoriaa ja toimeksiantajan ohjeita käyttäen kehitettiin yhden kantapelin koodin pohjalta kaksi uutta peliä Unity-ympäristössä. Peleistä luotiin eroavia mahdollisimman pienillä muutoksilla, keskittyen pääosin pelin koodiin ja hieman visuaaliseen ilmeeseen.

Tutkimuksessa saatiin selville tietoa uudelleenkäytettävyydestä pelikehityksessä ja hyperkasuaalisten pelien muokkaamisessa. Lisäksi tuloksena syntyi kaksi toisistaan ja kantapelistä eroavaa, täysin pelattavissa olevaa peliprototyyppiä.

Avainsanat (asiasanat)

Peli, mobiili, indie, Android, Unity, koodi, uudelleenkäytettävyys

Muut tiedot (salassa pidettävät liitteet)

jamk

Description

Ahonen, Jesse & Kateisto, Aleksi

Reusability in the development of hyper-casual games

Jyväskylä: JAMK University of Applied Sciences, May 2021, 39 pages

Business administration. Degree programme in business information systems. Bachelor’s thesis.

Permission for web publication: Yes

Language of publication: Finnish

Abstract

Since the rise of the smartphone, mobile gaming has become a huge part of the gaming industry at large. "Hyper-casual" is a new term that has recently surfaced from this market, meaning a game that is extremely simple, easy to grasp and meant for short gaming sessions. Hyper-casual games are also relatively quick to produce, making them an attractive option especially for indie companies.

The commissioner, Zaibatsu Interactive, is the largest gaming company in Central Finland, with a focus on mobile gaming. Their desire was to find ways to produce hyper-casual games with less effort and development time, using their previously produced games as bases for new experiences. To that end, the study uses design research methods to find out the challenges of developing new games using the code of a previous game as a base, mainly in terms of programming and game design. In addition, the research covered the theory of code reusability and its implementation in the Unity game engine.

With the acquired theory and the guidance of the commissioner, two new, different games were developed from the code of a single base game using Unity. The created games were developed to be as different as possible, with as few changes as possible. The changes focused mainly on code, with small modifications to the visual look of the game.

The results revealed information about reusability in game development and in the modification of hyper-casual games. Additionally, two playable game prototypes were created, which differ both from each other and from the base game.

Keywords/tags (subjects)

Game, mobile, indie, Android, Unity, Code, Reusability

Miscellaneous (Confidential information)

Sisältö

Sanasto ... **3**
1 Johdanto ... **5**
2 Tutkimusasetelma ... **6**
2.1 Tutkimuksen tavoitteet ja rajaukset ... 6
2.2 Tutkimuskysymykset ... 6
2.3 Tutkimus- ja kehittämismenetelmät ... 7
2.4 Tutkimuksen rakenne ... 7
3 Teoreettinen viitekehys ... **8**
3.1 Teoriapohja ja käytetyt aineistot ... 8
3.2 Avainkäsitteet ... 9
4 SOLID-periaatteet ... **9**
5 Kantapeli: Drift Royale ... **11**
5.1 Pelin kuvaus ... 11
5.2 Kantapelin koodin tarkastelu ... 12
6 Peli-ideoiden valinta ... **13**
6.1 Peli-Idea #1: Ihminen romurallin keskellä ... 13
6.2 Peli-Idea #2: Monster Truck ... 14
6.3 Peli-idea #3: Tavaran kuljetus ... 14
6.4 Peli-idea #4: Rallipeli ... 15
6.5 Peli-idea #5: Esteajelu ... 16
6.6 Peli-idea #6: King of the Hill ... 17
7 Valittu idea #1: Survival Derby ... **17**
7.1 Alustavat suunnitellut muutokset ... 18
7.2 Muutosten toteutus ja havainnot ... 19
8 Valittu idea #2: King of the Hill ... **23**
8.1 Alustavat suunnitellut muutokset ... 23
8.2 Muutosten toteutus ja havainnot ... 25
9 Vastaukset tutkimuskysymyksiin ... **35**
9.1 Mitä uudelleenkäytettävyys on pelinkehityksessä? ... 35
9.2 Miten uudelleenkäytettävyys toteutetaan hyperkasuaalisten pelien kehityksessä? ... 36
9.3 Miten samaa ydintä käyttävien pelien välillä luodaan riittävästi eroa? ... 36
10 Pohdinta ... **37**
10.1 Ajatuksia tuloksista ... 37

10.2 Tutkimuksen luotettavuus ja jatkokehitys 38
Lähteet 39

Kuviot

Kuvio 1. Drift Royale-pelin mainoskuva Google Play-kaupassa 11
Kuvio 2. SphereVehicle-skriptin OnCollisionEnter-metodin sisällä oleva tarkistus 19
Kuvio 3. Ansaitun rahan laskeminen ScoreView-skriptissä 20
Kuvio 4. Kohteen valinta ja satunnainen vaihto EnemyCar-skriptissä 21
Kuvio 5. Pelaajan liikkuminen ja väistö-ominaisuus uuden PlayerCar-skriptin sisällä 22
Kuvio 6. Lopullinen Survival Derby-versio Unityn sisällä 23
Kuvio 7. KotH-alue pelin sisällä 25
Kuvio 8. KotH-alueen liikkumisen logiikka ZoneObjectController-skriptissä 26
Kuvio 9. HandleDead-metodin logiikka 27
Kuvio 10. CountScores-metodin pisteenlasku-osuus 27
Kuvio 11. Pisteet pelinäkymän alaosassa 28
Kuvio 12. CountScores-metodin kruunuobjektin aktivointi ja deaktivointi 29
Kuvio 13. ZoneOccupationStatus-metodin logiikka 30
Kuvio 14. Peliobjektien hallinta enemiesInZone-listassa 31
Kuvio 15. Tekoälyn kohteen valinta KotH-alueen ollessa tyhjä 32
Kuvio 16. Tekoälyn viholliskohteen valinta 33
Kuvio 17. Ajastimen toiminnan logiikka 34
Kuvio 18. King of the Hill-pelimuunnelman lopullinen versio 35

Sanasto

Collider

Unity-pelimoottorissa peliobjekteihin lisättävä komponentti, jonka avulla ohjelma saa tietoa mm. siitä, kun yksi colliderin omaava objekti osuu toiseen.

Frame

Vastaa yhtä ruudulla näytettävää kuvaa kuvasarjasta, jonka nopealla päivittymisellä luodaan peleissä visuaalinen liike.

Funktio/metodi/aliohjelma

Koodin osa, joka suorittaa tietyn määritellyn toiminnon.

Luokka

Olio-ohjelmoinnin käsite. Templaatti, jonka pohjalta luodaan olioita, määritellen niille tiedot ja toiminnallisuudet.

Olio

Olio-ohjelmoinnissa luokan ilmentymä. Voi sisältää mm. muuttujia sekä metodeja.

Prefab

Unity-pelimoottorissa tallennettu peliobjekti, josta voidaan luoda kopioita. Jos prefabin ominaisuuksia muutetaan, muuttuvat myös luotujen kopioiden ominaisuudet.

Shader

Peliobjektien materiaaleihin liitettäviä skriptejä, jotka sisältävät algoritmeja sekä laskentoja, joiden avulla lasketaan väri jokaiselle renderöitävälle pikselille.

Skripti

Tiedosto, jossa koodi sijaitsee.

Trigger

Laukaisimia, joiden avulla koodissa voidaan laukaista eri tapahtumia, kuin esimerkiksi metodeja.

Unity

Unity Technologies-yrityksen kehittämä pelimoottori, jolla voidaan luoda 2D- ja 3D-pelejä sekä ohjelmistoja.

1 Johdanto

Mobiilipelit ovat nousseet viime vuosikymmenen aikana erittäin suureksi osaksi peliteollisuutta, kiitos älypuhelinten yleistymisen. Suurin osa maailman indie-pelifirmoista aloittaa tai keskittää kokonaan toimintansa mobiilipelien tuottamiseen. Viime vuosina mobiilipeliteollisuudesta on noussut esiin uusi termi: hyper-casual.

Hyperkasuaalisella pelillä tarkoitetaan pientä, mekaniikoiltaan hyvin yksinkertaista, useimmiten mobiililaitteella tai selaimella pelattavaa peliä, joka on erittäin helppo oppia ja jossa yksittäinen pelisessio on yleensä hyvin lyhyt. Yleensä mekaniikkana toimii vain yksi napautus tai klikkaus. Hyperkasuaalisten pelien monetisaatio perustuu lähes kokonaan pelin sisälle laitettaviin mainoksiin. (Fang, 2019)

Hyperkasuaaliset pelit ovat houkutteleva genre varsinkin indie-pelifirmoille, koska verrattuna muihin genreihin ne ovat nopeita ja yksinkertaisempia tuottaa. Vaikka yksittäisen käyttäjän tuoma rahamäärä on pieni, se kompensoituu kyseisten pelien erittäin suurella käyttäjäkunnalla (Heinze, 2017).

Opinnäytetyön tavoitteena on tutkia Unity-pelimoottorilla luotavien hyperkasuaalisten pelien kehittämistä tilanteessa, jossa yhden pelin lähdekoodin pohjalta luodaan useita eri pelejä. Toteutamme tutkimuksen käyttämällä toimeksiantajan toimittamaa Drift Royale-mobiilipeliä “ytimenä”, jonka pohjalta luomme kaksi uutta peliä. Ydinpelin pohjalta luotujen pelien tulisi jakaa ainakin 80 % lähdekoodista ytimen kanssa, mutta ollen silti tarpeeksi erilaisia houkutellakseen pelaajia. Tutkimuksessa selvitämme vastaantulevia ongelmia sekä kehitämme toimivaa ratkaisua pelien kehittämiseen, joka vastaa toimeksiantajamme tarpeisiin.

Aiheen toimeksiantaja Zaibatsu Interactive on vuonna 2014 perustettu suomalainen pelialan yritys, joka tuottaa palveluita sekä sisältöä pelikehityksessä, pelillistämisessä ja ohjelmistokehityksessä.

Hyperkasuaalisista peleistä ei kirjoitushetkellä löydy suurta määrää kirjallisuutta, joten hyödynnämme suureksi osaksi verkkolähteitä, toimeksiantajan osaamista, sekä omia havaintojamme kehityksen aikana. Pyrimme myös työssämme kokoamaan tietoa paremmin saataville asiasta kiinnostuneille.

2 Tutkimusasetelma

2.1 Tutkimuksen tavoitteet ja rajaukset

Aiheen valinta pohjautuu toimeksiantajan tarpeeseen tutkia aihetta sekä aikaisemman tutkimustiedon vähäiseen määrään.

Tutkimuksen tavoitteena on selvittää pelikehityksen näkökulmasta, kuinka saadaan kehitettyä useita yksinkertaisia pelejä käyttäen suurimmaksi osaksi samaa lähdekoodia. Jos saman lähdekoodin pohjalta pystytään kehittämään useampia erilaisia pelejä, sillä voidaan parantaa tuottavuutta ja nopeuttaa kehittämistä.

Tutkimus pohjautuu Unity-pelimoottorin ympärille, koska se on toimeksiantajan pääasiallinen kehitysalusta pelikehityksessä.

Tutkimustuloksista saatuja tietoja voidaan soveltaa hyperkasuaalipelien kehityksessä. Usean pelin rakentaminen samaan koodipohjaan nopeuttaa ja helpottaa kehitystä. Koska hyperkasuaalisissa peleissä ajatuksena on, että pelin elinikä on suhteellisen lyhyt, auttavat tutkimuksen tulokset toimeksiantajaa rakentamaan nopeasti useita eri hyperkasuaalisia pelejä, saaden niitä näin markkinoille tiheämpään tahtiin.

2.2 Tutkimuskysymykset

- Mitä uudelleenkäytettävyys on pelikehityksessä?
- Miten uudelleenkäytettävyys toteutetaan hyperkasuaalisten pelien kehityksessä?
- Miten samaa ydintä käyttävien pelien välille luodaan riittävästi eroa?

2.3 Tutkimus- ja kehittämismenetelmät

Tutkimus toteutetaan kehittämistutkimuksena, koska tavoitteena on kehittää toimeksiantajan hyperkasuaalisten pelien tuotantoa sekä ajankäytön että kehitysmenetelmien osalta. Kehittämistutkimukselle on ominaista lähestymistapa, jossa ensin kartoitetaan syklisesti nykytilanne ja määritellään ongelma, josta siirrytään tutkimukseen ja muutoksen toteuttamiseen omana syklinään. (Kananen, 2015, 39-42.)

Osana tutkimusta kehitämme kaksi peliprototyyppiä muokkaamalla toimeksiantajan toimittaman "kantapelin" lähdekoodia. Näiden prototyyppien pohjalta saadaan konkreettinen näkökulma kehittämiseen uudelleenkäytettävyyden näkökulmasta. Prototyypit kehitetään Unity-kehitysalustalla.

Laadullisen tutkimuksen menetelmät ovat käytössä, koska aiheesta ei ole tehty paljon aikaisempia tutkimuksia ja siitä pyritään saamaan parempi ymmärrys havainnoinnin ja toimeksiantajan haastatteluiden avulla. (Kananen, 2015, 34.)

2.4 Tutkimuksen rakenne

Tutkimusta varten toimeksiantaja on antanut käytettäväksemme valmiin Drift Royale-pelin lähdekoodin. Tutkimme ja muokkaamme pelin koodia niin, että samasta ytimestä rakentuu kaksi uutta erilaista peliä mahdollisimman pienin muokkauksin. Kehityksen aikana selvitämme omien kokemustemme, toimeksiantajan palautteen sekä löytämiemme lähteiden perusteella mitkä ovat tällaisessa tapauksessa vastaan tulevat ongelmat sekä parhaat käytänteet niiden ratkaisemiseen.

Kantapelin lisäksi toimeksiantaja on ehdottanut useaa eri ideaa muunneltujen pelien pohjaksi. Aluksi tutkimme jokaista ideaa ja pisteytämme ne idean helppouden, uudelleenkäytettävyyden ja eroavaisuuden mukaan. Pisteiden avulla valitsemme ideoista kaksi parasta, joita lähdemme kehittämään.

Valintojen jälkeinen kehitys tapahtuu tiiviisti yhteistyössä toimeksiantajan kanssa. Muutamme ominaisuuksia vähän kerrallaan, dokumentoiden muutokset ja käyden säännöllisesti keskustelua toimeksiantajan kanssa muutosten toimivuudesta pelisuunnittelun näkökulmasta.

Kehittämistutkimuksen syklisen lähestymistavan mukaan muutoksia jatketaan, kunnes kaikki osapuolet ovat tyytyväisiä pelien tilaan.

Lopuksi tarkastelemme valmiita pelejä, sekä kehityksessä vastaan tulleita ongelmia. Arvostelemme myös alkuperäistä lähdekoodia uudelleenkäytettävyyden kannalta ja ehdotamme muutoksia tulevien pelien koodiin, mikäli uudelleenkäytettävyys on haluttua.

3 Teoreettinen viitekehys

3.1 Teoriapohja ja käytetyt aineistot

Tutkimusmenetelmän osalta teoreettiseen viitekehykseen sisältyy kehittämistutkimuksen teoria ja kirjallisuus.

Prototyypit kehitetään Unity-kehitysalustalla ja siinä käytämme hyödyksi ensisijaisesti Unityn omaa virallista dokumentaatiota, sekä siihen liittyvää kirjallisuutta ja artikkeleita. Lisäksi hyödynnämme tarvittaessa muita sähköisiä lähteitä kuten opetusvideoita.

Tutkimuksen aihe liittyy keskeisesti uudelleenkäytettävyyden teoriaan. Teoriaa löytyy ohjelmistokehityksen sekä pelikehityksen aihepiiristä, joista molempia voidaan hyödyntää ja soveltaa aiheessamme. Pelikehityksen puolelta aineistoa löytyy huomattavasti vähemmän, joten tarpeeksi kattavan teoriapohjan saadaksemme hyödynnämme myös ohjelmistokehitykseen liittyvää teoriaa.

Uudelleenkäytettävyyden osalta ohjelmistokehitykseen on olemassa useita eri periaatteita ja teorioita. Tähän tutkimukseen valitsimme hyvin yleisesti käytössä olevat “SOLID”-periaatteet ja pääaineistoksi Matthias Nobackin Principles of Package Design, jossa kuvattuja SOLID-periaatteita ja uudelleenkäytettävyyden menetelmiä tulemme tutkimaan Unityn näkökulmasta ja käyttämään niitä soveltaen omassa koodissamme mahdollisuuksien mukaan.

Pelisuunnittelun teoriassa käytämme suurimmaksi osaksi hyödyksi toimeksiantajan kokemusta ja tietoa aiheesta. Suunnittelemme prototyypit iteratiivisesti yhdessä toimeksiantajan kanssa, muuttaen ominaisuuksia heidän toiveidensa mukaan.

3.2 Avainkäsitteet

Tutkimuksen aikana esiin nousseita tärkeitä käsitteitä ovat:

- Hyperkasuaaliset pelit
- Mobiilipelit
- Komponentti
- Koodin uudelleenkäytettävyys.

Hyperkasuaalinen peli on terminä hyvin uusi, eikä sille ole olemassa tarkkaa virallista määritelmää. Tässä tutkimuksessa termillä tarkoitetaan mobiilipeliä, joka on erittäin yksinkertainen, helposti omaksuttava ja lyhyihin pelisessioihin suunniteltu.

Mobiilipeleillä tarkoitetaan pelejä, joita pelataan Android- tai iOS-käyttöjärjestelmää käyttävillä mobiililaitteilla, kuten tableteilla tai älypuhelimilla.

Komponentti on osa järjestelmää, joka on vastuussa tietystä toiminnallisuudesta. Pelikehityksessä esimerkiksi pelaajan liikkumiseen liittyvä koodi voi olla oma komponenttinsa. Unity-pelimoottorissa komponentti voi tarkoittaa myös peliobjektiin liitettävää skriptiä, jolla objektiin saadaan uusia toiminnallisuuksia.

Koodin uudelleenkäytettävyydellä tässä tutkimuksessa tarkoitetaan mahdollisuutta käyttää pelissä käytettyä koodia uudestaan uusissa peleissä ilman muutoksia tai hyvin pienin muutoksin, vähentäen kehitykseen vaadittavaa aikaa ja virheiden mahdollisuutta.

4 SOLID-periaatteet

Tässä tutkimuksessa tulemme osittain käyttämään uudelleenkäytettävyyden saralla viitekehyksenä SOLID-periaatteita. SOLID on lyhenne, joka tulee viidestä eri ohjelmistoarkkitehtuuriperiaatteesta. Yksittäiset periaatteet ovat useiden eri ohjelmisto-alan ihmisten kehittämiä, mutta ne liitti yhteen SOLID-järjestelmäksi amerikkalainen ohjelmistokehittäjä ja kirjailija Robert C. Martin. Kyseisten periaatteiden käytön tarkoituksena on tehdä koodista uudelleenkäytettävämpää, helppolukuisempaa ja vähemmän altista virheille (Martin, 2003). SOLID-periaatteiden ei ole

tarkoitus olla tiukkoja sääntöjä, vaan ennemminkin ohjeita, joita kannattaa soveltaa mahdollisuuksien mukaan. Käymme seuraavaksi lyhyesti läpi kaikki viisi periaatetta.

Single Responsibility Principle (SRP)

Yhdellä luokalla tulisi olla vain yksi syy sen muutokseen, eli yhden luokan vastuulla tulisi olla vain yksi asia (Noback, 2018, 19). Pelikehitykseen liittyvä esimerkki voisi olla pelaajan hahmoon liittyvät luokat. Sen sijaan, että yhdessä luokassa sijaitsee sekä hahmon vahingoittamiseen, että liikkumiseen liittyvät koodit, tulisi ne jakaa kukin omaan luokkaansa. Tätä periaatetta käyttämällä luokat säilyvät pienempinä ja täten helppolukuisempina ja helpommin testattavina (Noback, 2018, 25).

Open-Closed Principle (OCP)

Luokan toimintoja tulee pystyä laajentamaan muokkaamatta sitä, toisin sanoen luokan tulee olla avoin laajentamiselle, mutta suljettu muokkaamiselle. Jos tarvitaan uusia toiminnallisuuksia, niitä ei pidä toteuttaa muokkaamalla vanhaa toteutusta, vaan kirjoittamalla uutta koodia, joka käyttää hyväkseen vanhaa. Tämän periaatteen tarkoituksena on tehdä järjestelmästä helppo laajentaa tarvittaessa, ilman että se vaikuttaa olemassa olevaan koodiin. (Noback, 2018, 27.)

Liskov Substitution Principle (LSP)

Perivien luokkien tulee olla vaihdettavissa kantaluokkiensa kanssa (Noback, 2018, 47). Toinen tapa esittää asia on sanoa, että jos aliluokka syrjäyttää kantaluokan toiminnallisuutta odottamattomalla tavalla, joka rikkoo yhteensopivuutta lopun koodin kanssa, ei tätä periaatetta ole silloin noudatettu (Villalobos, 2019).

Interface Segregation Principle (ISP)

Rajapinnoista tulee tehdä spesifejä niitä käyttäville luokille, sekä “hienojakoisia”, eli pieniä määriä metodeja sisältäviä. Eli rajapintoja kannattaa luoda useita ja tehdä niistä spesifejä, jotta niitä käyttävät luokat eivät ole riippuvaisia metodeista, joita ne eivät tarvitse. (Noback, 2018, 70.)

Dependency Inversion Principle (DIP)

Riippuvuus pitäisi olla abstraktioissa, ei konkretioissa. Korkean tason komponenttien ei pitäisi riippua matalan tason komponenteista, vaan molempien tulisi riippua abstraktioista, kuten

esimerkiksi rajapinnoista. Tätä periaatetta käyttämällä koodista tulee joustavampaa, koska se vähentää riippuvuuksia komponenttien välillä. (Villalobos, 2019.)

5 Kantapeli: Drift Royale

5.1 Pelin kuvaus

Tutkimuksessa tuotettavien pelimuunnelmien kantapelinä toimii Zaibatsu Interactiven aiemmin tuottama Drift Royale-mobiilipeli. Pelissä pelaaja ohjaa autoa ylhäältä kuvatussa perspektiivissä kentässä, joka hitaasti vajoaa maahan reunoilta. Ohjaus tapahtuu näpäyttämällä ruudun vasenta tai oikeaa reunaa, jolloin auto kääntyy haluttuun suuntaan.

Kuvio 1. Drift Royale-pelin mainoskuva Google Play-kaupassa

Pelissä on mukana tekoälyn ohjaamia autoja, jotka hyökkäävät sekä pelaajan että toistensa kimppuun. Sekä pelaaja että tekoäly voi hyökätä joko törmäämällä muihin autoihin, tai keräämällä kentälle satunnaisesti syntyviä erilaisia aseita.

Jokaisella autolla on elinvoima, joka vähenee törmäyksissä tai aseen osuessa. Kun auton elinvoima loppuu, tai jos auto putoaa kentältä, tippuu kyseinen auto pelistä pois. Pelaajan tavoitteena on olla viimeinen kentälle jäävä, ehjä auto. Pelejä voittamalla pelaaja kerää kolikoita, joilla voi ostaa itselleen erilaisia autoja.

5.2 Kantapelin koodin tarkastelu

Koska yksi tutkimuksen tavoitteista on pitää koodin muutokset mahdollisimman vähäisenä, tarkastelimme ensin kantapelin koodipohjaa löytääksemme muunnelmien tekemisen kannalta olennaisimmat osat. Käymme seuraavaksi läpi tärkeimmät skriptit.

SphereVehicle.cs

Autojen kantaluokka sijaitsee tässä skriptissä. Luokasta löytyvät mm. ohjaukseen, vahingon ottamiseen ja aseisiin liittyviä toiminnallisuuksia. SOLID-periaatteiden mukaisesti voisi olla parempi jakaa näitä toiminnallisuuksia useampiin luokkiin. Lisäksi projektista löytyvät erilliset CarObject.cs, joka vaikuttaa liittyvän autoista lähteviin efektiin ja ääniin, sekä Car.cs, jossa on myös ohjaukseen liittyvää koodia. Asioiden hajauttaminen näin ei välttämättä ole ongelma, mutta paremmat nimeämiskäytännöt skripteille parantaisivat koodin ymmärrettävyyttä.

EnemyCar.Cs

Edellisen lapsiluokka, jossa sijaitsee tekoälyn liikkuminen sekä kohteen valinta. Suurella todennäköisyydellä tekoälyä joudutaan muokkaamaan lähes missä tahansa muunnelmassa, joten tämä skripti tulee olemaan tärkeässä osassa.

LevelController.cs

Sisältää kentän luomiseen ja tuhoamiseen liittyvät koodit. Muunnelmissa joudumme vääjäämättä muokkaamaan kenttää tai poistamaan/lisäämään sen ominaisuuksia, jotta eri peligenre toimisi.

GarageController.cs

Sisältää uusien autojen ostoon ja valintaan liittyvät koodit. Tämän työn kannalta olennaisesti myös muuttujat, jotka määräävät eri autojen ominaisuudet, kuten massan, vauhdin ja elinvoiman.

Muunnelmien kannalta nämä ovat olennaisimmat osat, mutta pieniä muokkauksia mahdollisesti tarvitaan myös muualla koodikannassa. Muutosten pitämiseksi vähäisenä ja suuremman refaktoroinnin välttämiseksi keskitämme muutoksemme kuitenkin mahdollisimman paljon yllä mainittuihin skripteihin.

6 Peli-ideoiden valinta

Kaksi tuotettavaa peli-ideaa valitaan kuuden toimeksiantajan toimittavan idean joukosta. Valinta tapahtuu vertailemalla ideoiden hyviä ja huonoja puolia, keskittyen kriteereinä varsinkin toteutuksen helppouteen ja vaadittavien muutosten määrän. Lopullinen ratkaisu tehdään toimeksiantajan suostumuksella, kun vertailu on tehty.

6.1 Peli-Idea #1: Ihminen romurallin keskellä

Idean kuvaus

Peli, jossa pelaaja on jalankulkija keskellä tekoälyn käymää romurallia. Pelimoodeja voi olla esimerkiksi esineiden kerääminen, tietyn ajan verran selviäminen tai tiettyyn pisteeseen kentässä pääseminen.

Vaaditut muokkaukset/lisäykset/poistot

Ohjaus voi säilyä ennallaan ja toimii varsinkin hyvin jos hahmosta tehdään räsynukkemainen, jonka raajat heilahtelevat kun pelaaja liikuttaa hahmoa, jolloin pelistä saadaan humoristisemman näköinen. Tekoäly vaatii pientä muokkausta, jotta viholliset eivät hyökkää suoraan pelaajaan. Saattaa olla tarpeen kirjoittaa koodia joka varmistaa että vihollisten romuralli tapahtuu kuitenkin aina pelaajan lähettyvillä. Eri pelimoodit vaativat hyvin pientä lisäystä koodiin, esim. kerättävät esineet ja niistä saadut pisteet, ajastin, ja/tai ”maali” johon pelaaja pyrkii. Pelaajan koodista poistettaisiin aseiden käyttö.

Hyvät puolet

- Hyvin erilainen peli
- Minimaalista muokkausta vaativa
- Sopii useampaan pelimoodiin
- Humoristinen.

Huonot puolet

- Mahdollisesti vähemmän tyydyttävää pelaajalle ilman aseita
- Saattaa olla turhauttavaa, jos tekoäly sattuu luomaan liikaa mahdottomia tilanteita.

6.2 Peli-Idea #2: Monster Truck

Idean kuvaus

Peli, jossa tekoäly ajaa tavallisilla autoilla, kun taas pelaaja ohjaa valtavaa monster truckia. Pelaajan tehtävänä on liiskata vastustajat ennen kuin ne tekevät liikaa vahinkoa pelaajan autoon. Tämä idea on myös mahdollista kääntää siten, että pelaaja olisi pikkuauto, jonka tehtävä on selviytyä monster truckien keskellä.

Vaaditut muokkaukset/lisäykset/poistot

Ohjausta ei tarvitse muuttaa, mutta pikkuautojen nopeus ja/tai kääntyvyys tulisi olla hieman monster truckia parempi. Tekoälyn tulisi valita kohteeksi aina pelaaja, tai muutoin ajaa karkuun. Käänteisessä moodissa tekoäly pysyisi suurimmaksi osaksi ennallaan. Monster truckeissa joko ei olisi aseita ollenkaan, tai aseet vaihdettaisiin vihollisia hidastaviin/pysäyttäviin, jotta tuhoaminen tapahtuisi vain ajamalla päälle.

Hyvät puolet

- Hieman erilainen kokemus lähes täysin samoilla mekaniikoilla
- Kaksi hyvin erilaista pelimoodia helposti toteutettavissa
- Asioiden murskaaminen aina tyydyttävää ja koukuttavaa, jos audiovisuaalinen puoli on hyvin tehty.

Huonot puolet

- Mahdollisesti liiankin samanlainen
- Monster truckin ja autojen väliset fysiikat voivat vaatia paljon työtä.

6.3 Peli-idea #3: Tavaran kuljetus

Idean kuvaus

Tämän pelin ajatuksena on pelaajan ohjaama rekka, jolla tulee kuljettaa tavaroita tiettyyn pisteeseen kentässä. Kun pelaaja ottaa osumaa vihollisista tai törmää, tavarat voivat tippua kyydistä. Mitä enemmän ehjiä tavaroita pelaaja saa maaliin, sitä suuremmat pisteet hän ansaitsee.

Vaaditut muokkaukset/lisäykset/poistot

Ohjaus voi toimia alkuperäisellä tavalla, tosin rekan kääntyvyyttä/nopeutta saattaa joutua säätämään parhaan pelikokemuksen saamiseksi. Kyydissä oleville tavaroille ja niiden vaikutukselle pisteisiin pitää kirjoittaa omat koodinsa. Tekoälyn tulisi tähdätä pelaajaan ja sen hyökkäysten tahtia pitää rajoittaa niin, ettei kokemus ole turhauttava. Kenttä pitää uudistaa enemmän lineaariseksi, josta löytyy alku ja loppu, mahdollisesti esteitä.

Hyvät puolet

- Hyvin erilainen päämäärä ja täten pelikokemus
- Helppo luoda erilaisia variaatioita jatkossa (Laiva, lentokone, jne.)
- Tarvittaessa progressio voidaan toteuttaa tekemällä haastavampia kenttiä.

Huonot puolet

- Yhteen kenttään kyllästyy helposti
- Useamman kentän teko pitää tehdä käsin tai kenttiä satunnaisesti luovalla järjestelmällä
- Rekka ja tavarat vaatisivat isompia muutoksia.

6.4 Peli-idea #4: Rallipeli

Idean kuvaus

Tässä muunnelmassa pelistä rakennettaisiin tyypillinen kilpa-ajopeli, jossa ajetaan tekoälyä vastaan kilpaa tietty määrä kierroksia rataa pitkin.

Vaaditut muokkaukset/lisäykset/poistot

Pelaajan ohjaus säilyisi ennallaan, mutta aseet poistettaisiin kokonaan. Tekoälyn koodi tulisi kirjoittaa suurelta osin uusiksi rata-ajoa varten. Lisäksi vaikeustason säätäminen sopivaksi vaatisi paljon testausta ja tekoälyn yksityiskohtaista muokkausta. Kenttä pitää uudistaa ralliradaksi, jonka tulisi olla tarpeeksi haastava ja mielenkiintoinen ajaa.

Hyvät puolet

- Yksinkertainen konsepti jossa pystytään käyttämään jonkin verran samaa koodia
- Täysin erilainen pelikokemus.

Huonot puolet

- Ratojen tekeminen ja vaikeustason säätäminen
- Tekoälyn muokkaus.

6.5 Peli-idea #5: Esteajelu

Idean kuvaus

Pelaaja ajaa loputonta rataa pitkin esteitä väistellen, tavoitteena on päästä mahdollisimman pitkälle ennen törmäystä. Pelaajan kulkiessa eteenpäin kenttä vajoaa maahan takanapäin, pakottaen nopean vauhdin.

Vaaditut muokkaukset/lisäykset/poistot

Auton ohjaus säilyisi lähestulkoon ennallaan, mutta kääntyvyyttä joudutaan muokkaamaan jonkin verran tiukemmaksi. Tämä pelimoodi ei vaatisi tekoälyä eikä aseita, joten molemmat poistettaisiin kokonaan käytöstä. Mielenkiintoa lisättäisiin tekemällä kentälle satunnaisesti syntyviä “power-up” esineitä, joita keräämällä pelaaja saa erinäisiä kykyjä, esim. nopeamman vauhdin. Lisäksi vaaditaan järjestelmä, joka luo kenttää loputtomasti lisää. Tarvitaan myös keino vaikeuttaa peliä jatkuvasti sitä enemmän, mitä pidemmälle pelaaja pääsee.

Hyvät puolet

- Ei tekoälyn tuomia haasteita
- Mainittujen lisäysten ulkopuolella kaikki muu uudelleenkäytettävissä tai poistettavissa
- Täysin eri genre.

Huonot puolet

- Nopeampi tahti tarvitsee hyvin tiukan ja sulavan ohjauksen
- Pelin jatkuva vaikeutuminen vaatii oman toteutuksensa.

6.6 Peli-idea #6: King of the Hill

Idean kuvaus

Pelissä pyritään pysymään rajatulla alueella mahdollisimman pitkään, taistellen tekoälyä vastaan joka pyrkii samaan. Alueella ollessa karttuu pisteitä ja ensimmäisenä tiettyyn pistetavoitteeseen päässyt voittaa.

Vaaditut muokkaukset/lisäykset/poistot

Sekä ohjaus, aseet, että kenttä voidaan uudelleenkäyttää suoraan. Myös tekoäly pysyisi suurelta osin samana. Tekoälyyn lisättäisiin logiikka, jolla se yrittää pysytellä pistealueen sisällä, mutta jatkaen silti taistelua. Itse kenttä pysyisi samana, mutta reunojen tuhoutuminen otettaisiin pois ja lisättäisiin satunnaisiin paikkoihin syntyvä, liikkuva pistealue. Sekä pelaaja että tekoäly säädettäisiin syntymään uudelleen kentälle tietyn ajan päästä tuhoutumisesta, jolloin peli päättyisi vain pistetavoitteen täyttyessä.

Hyvät puolet

- Ei vaadi suuria muutoksia tai lisäyksiä, suurin muutos on tekoälyn muokkaus
- Käytännössä kaikki uudelleenkäytettävissä.

Huonot puolet

- Peli voi olla haastava saada erottumaan tarpeeksi kantapelistä.

7 Valittu idea #1: Survival Derby

Ensimmäiseksi toteutettavaksi ideaksi valittiin “ihminen romurallin keskellä”, jolle annoimme työnimen Survival Derby. Tässä ideassa näimme eniten potentiaalia suureen eroavaisuuteen pienillä muutoksilla.

Pelaajan päämääränä on kerätä kentälle ilmestyviä esineitä jalankulkijana, samalla kun tekoäly käy Drift Royale-taistelua hänen ympärillään. Pelaaja kuolee hyvin pienestä vahingon määrästä, jolloin peli loppuu ja pelaaja saa kolikoita kerättyjen esineiden määrän mukaan.

Sekä pelaajan että vihollisen aseet poistettu käytöstä, tekoäly taistelee törmäyksillä ja tähtää myös pelaajaan. Ase-pickupit voidaan muuntaa suoraan vain pelaajan keräämiksi esineiksi.

7.1 Alustavat suunnitellut muutokset

Pelaaja

Jotta pelikokemuksesta saadaan nopea ja haastava, pelihahmon tulee kestää vain yksi osuma vihollisten autoista. Koska alkuperäisessä pelissä kaikki autot jakavat samat elinvoiman määräävät health-muuttujat, annetaan pelaajalle erillinen maxHealth-muuttuja ja tarkistetaan SphereVehicle-skriptin SpawnCar-funktiossa autoa luotaessa, onko luotava ”auto” pelaaja. Pelaajan hahmosta poistetaan aseiden käyttöön tarkoitettu VehicleWeapons-skripti ja muualta koodista viittaukset siihen. Lisäksi selvitämme toimeksiantajan kanssa tapahtuvalla pelitestauksella, tarvitseeko pelaajan nopeutta tai ohjattavuutta muokata hyvän pelikokemuksen saavuttamiseksi.

Viholliset

Viholliset pysyvät suurimmaksi osaksi ennallaan. Jotta peli ei lopu vihollisten tuhotessa toisiaan, tehdään vihollisista kestävämpiä nostamalla maxHealth-muuttujan arvoa. Toinen vaihtoehto on luoda vihollinen takaisin kentälle, kun se on tuhottu, käyttäen olemasaolevaa SpawnCar-funktiota. Kuten pelaajalta, vihollisten autoista poistetaan VehicleWeapons-skriptit ja viittaukset siihen muista skripteistä. EnemyCar-skriptin SetDestination-funktiosta poistetaan pickupeihin liittyvä osa, koska emme halua, että viholliset yrittävät jahdata esineitä.

Esineet

Koska VehicleWeapons-skriptit poistettiin, ei esineitä voi enää poimia. Tehdään tätä varten pelaajalle erillinen PlayerPickups-skripti, johon laitetaan Trigger Collider-tarkistus esineitä varten. Kun pelaaja osuu esineeseen, annetaan tietty määrä kolikoita, jotka ansaitaan pelin lopussa. Nämä voidaan lisätä winCoins-muuttujaan ScoreView-skriptissä, korvaten nykyisen tappoihin ja voittoon perustuvan laskun.

Muut asiat

Koska emme halua, että peli päättyy muuten kuin pelaajan hahmon ottaessa osuman autosta, pysäytetään kentän vajoaminen ottamalla LevelController-skriptistä pois SinkCoroutine. Lisätään kentän reunoille seinät, jotka estävät autoja putoamasta.

7.2 Muutosten toteutus ja havainnot

Muunnettiin GarageController-skriptissä Player-prefabin health-muuttuja arvoon 1, jotta pelaaja kestää vain yhden osuman. Lisäksi sen sijaan, että vihollisten elinvoimaa muutetaan, muutettiin törmäyksen tarkistus niin, että vain pelaaja voi ottaa vahinkoa. Tämä tehtiin lisäämällä if-lause olemassaolevaan koodiin, joka tarkastaa onko osuttu auto pelaaja.

```
293            if (this is PlayerCar)
294            {
295                Debug.Log("coll magnitude" + collision.relativeVelocity.magnitude);
296
297                int damage = (int)(carConfig.CollisionDamage.Evaluate(collision.relativeVelocity.magnitude) * multiplier);
298                bool kill = TakeDamage(damage);
299                otherVehicle.AddDamageDealt(damage);
300                if (kill)
301                {
302                    otherVehicle.AddKills(1);
303                }
304            }
```

Kuvio 2. SphereVehicle-skriptin OnCollisionEnter-metodin sisällä oleva tarkistus

Koska pelaaja kestää vain yhden osuman ja viholliset ovat tuhoutumattomia, ei elinvoimaa tarvitse nähdä. Poistettiin Healthbar-skripti käytöstä tarpeettomana, sekä kommentoitiin pois viittaukset siihen SphereVehicle, Selector ja CarObject-skripteistä. Lisäksi poistettiin CarObject-prefabista HealthBarCanvas-lapsiobjekti.

Tässä muunnelmassa vain pelaajan on tarkoitus kyetä poimimaan esineitä. Tätä varten ei tarvittu erillistä uutta skriptiä, sen sijaan tarkastamme onko esineeseen osunut hahmo pelaaja. Tämä tapahtuu if-lauseella SphereVehicle-skriptin OnTriggerEnter-metodissa.

Pisteenlaskua varten luotiin GameController skriptin GameSettings-luokkaan uudet muuttujat CollectedItemsCount, jota nostetaan yhdellä kun pelaaja poimii esineen, sekä MoneyPerItem, jolla määrätään esineen arvo. ScoreView-skriptissä laskemme tuloksen käyttäen vanhaa AnimateScreen-metodia.

```
62          int winCoins = (int)Mathf.Ceil(GameController.Instance.GameSettings.CollectedItemsCount
63                                          * GameController.Instance.GameSettings.MoneyPerKill);
64          if (winCoins > 0)
65          {
66
67              yield return StartCoroutine(LerpMoney(0, winCoins, 1.0f));
68
69              yield return new WaitForSecondsRealtime(0.5f);
70              bonusList.SetActive(true);
71              yield return new WaitForSecondsRealtime(0.2f);
72
73              PendingMoney = winCoins;
74              yield return new WaitForSecondsRealtime(0.3f);
75              if (AdController.Instance.IsRewardedLoaded())
76              {
77                  doubleButton.SetActive(true);
78              }
79          }
```

Kuvio 3. Ansaitun rahan laskeminen ScoreView-skriptissä

Muutettiin loppuruudun “Player Kills” teksti muotoon “Items Collected” ja siihen haetaan arvo CollectedItemsCount-muuttujasta. Tekstin viereinen ikoni tarvitsee muuttaa sopivammaksi jatkokehityksessä.

Koska vihollisautot eivät voi tuhoutua, eikä aseita ole, poistettiin käyttöliittymästä näihin liittyvät CarsAlive, CarsDead ja CurrentPickup-objektit tarpeettomana. Tilalle lisättiin laskuri, joka näyttää kerättyjen esineiden määrän pelin aikana. Tämäkin tarvitsee uuden ikonin.

Lisätty seinä kentän alareunassa toisinaan peittää pelaajan näkymän hahmosta. Lopullisen seinän tulisi olla läpinäkyvä. Jonkinlainen “aita” seinän sijaan toimisi hyvin.

Koska viholliset eivät hae Pickup-esineitä, joissain peleissä syntyy ruuhkia, jossa viholliset vain kiertävät toisiaan yrittäen osua. Muutosta kokeiltu pakottamalla viholliset ottamaan vain pelaajan kohteeksi. Tämä toimii hieman paremmin, mutta luo suman pelaajan taakse. Ratkaisuna asetettiin vain osa vihollisista valitsemaan pelaajan kohteeksi, kun taas loput asetettiin ajamaan kohti satunnaisesti valittua esinettä, joka vaihtuu viiden sekunnin välein.

```
44      private void SetDestination(float pX, float pZ)
45      {
46          if (!TargetsPlayer)
47          {
48              if (targetVehicle == null && targetPickUp == null)
49                  RefreshTargetPickup();
50
51              Vector3 position = new Vector3(pX, 0, pZ);
52              navMeshAgent.SetDestination(position);
53
54              StartCoroutine(SwapTarget());
55          }
56          else
57              navMeshAgent.SetDestination(PlayerController.Instance.PlayerVehicle.transform.position);
58      }
59
        2 references
60      private IEnumerator SwapTarget()
61      {
62          yield return new WaitForSeconds(5);
63          RefreshTargetPickup();
64          StartCoroutine(SwapTarget());
65      }
66
        2 references
67      private void RefreshTargetPickup()
68      {
69          if (LevelController.Instance.Pickups.Count < 1) return;
70
71          List<GameObject> orderedPickups = LevelController.Instance.Pickups;
72          orderedPickups.Sort((x, y) =>
73          {
74              float distX = Vector3.Distance(transform.position, x.transform.position);
75              float distY = Vector3.Distance(transform.position, y.transform.position);
76              return distX.CompareTo(distY);
77          });
78          targetPickUp = orderedPickups[Random.Range(0, orderedPickups.Count - 1)];
79          SetDestination(targetPickUp.transform.position.x, targetPickUp.transform.position.z);
80      }
```

Kuvio 4. Kohteen valinta ja satunnainen vaihto EnemyCar-skriptissä

Koska peli ei erottele pelaajan ja vihollisten autojen välillä, vaatisivat ohjauksen muutokset perusteellisia muutoksia koodiin ja/tai huomattavan määrän uutta koodia. Toinen vaihtoehto on tehdä nykyisestä ohjauksesta temaattisesti sopivaa. Esim. Pelaaja voisi ajella ostoskärryllä, jolloin pienet nopeuden muutokset olisivat riittäviä.

Toimeksiantajan kanssa käydyn palaverin perusteella muutettiin pelin ohjausta seuraavalla tavalla: Pelaaja liikuttaa hahmoa Joystickin kaltaisella napilla vasemmassa alareunassa. Tätä varten käytettiin Unity Asset Storesta löytyvää ilmaista "Virtual Plug and Play Joystick"-pluginia. Lisäksi peliin lisättiin syöksymis-/väistöominaisuus, jossa toista nappia painamalla hahmo syöksyy nopeasti menosuuntaan. Syöksyä voi nopeasti kolme kertaa peräkkäin, kunnes täytyy odottaa syöksyjen latautumista skriptissä määritellyn ajan verran.

Ohjauksen ominaisuudet vaativat tähän mennessä suurimman muutoksen koodiin, koska pelaajalta tuli poistaa kaikki “auton” ominaisuudet ja tehdä täysin uusi koodi liikkumista ja syöksymistä varten. Tämä toteutettiin muuttamalla SphereVehicle-skriptissä oleva liikkumisen koodi vain vihollisten käyttöön ja kirjoittamalla pelaajalle oma uuden PlayerCar-skriptin sisälle.

```
23      protected override void Update()
24      {
25          base.Update();
26          PlayerSteering();
27      }
28
        1 reference
29      private void PlayerSteering()
30      {
31          Vector3 move = new Vector3(Input.GetAxis("Horizontal"), 0, Input.GetAxis("Vertical"));
32
33          rbody.AddForce(move * PlayerSpeed);
34
35          if (Input.GetButtonDown("Jump") && CurrentDodges > 0)
36          {
37              rbody.AddForce(move * DodgeSpeed, ForceMode.Impulse);
38              CurrentDodges--;
39              Invoke("RegenDodges", DodgeRegenSpeed);
40          }
41
42      }
43
        0 references
44      private void RegenDodges()
45      {
46              CurrentDodges++;
47      }
48
```

Kuvio 5. Pelaajan liikkuminen ja väistö-ominaisuus uuden PlayerCar-skriptin sisällä

Pelaajan hahmo muutettiin myös autosta ihmishahmoksi. Tässä käytimme “SuperCyan Character Pack”-pluginin ilmaista versiota. Muutos toteutettiin vaihtamalla yksi auto-prefab uuteen malliin ja poistamalla siitä autoon liittyvät efektit. Pieni muutos EnemyCar-skriptiin estää vihollisten luomisen tätä prefabia käyttäen. Lisäksi pelaajan hahmon Rigidbody-komponentin asetuksista muutettiin hahmo kevyemmäksi ja humoristisemman lopputuloksen saavuttamiseksi hahmo laitettiin lentämään osuman saatuaan. Jatkokehityksessä hahmo tarvitsee animaatiot liikkumiseen, sekä efektit väistöliikeeseen.

Kuvio 6. Lopullinen Survival Derby-versio Unityn sisällä

8 Valittu idea #2: King of the Hill

Toiseksi toteutettavaksi peli-ideaksi valittiin King of the Hill. Pelissä tavoitteena on pysytellä mahdollisimman pitkään merkityn KotH-alueen sisällä ja pyrkiä pitämään muut poissa alueelta. Jokaiselle KotH-alueella olevalle autolle kertyy pisteitä, ja minuutin aikana eniten pisteitä kerännyt auto voittaa pelin. Viholliset sekä pelaaja ovat tuhottavissa, joka mahdollistaa häviämisen myös tuhotuksi tulemisella. Pelaajalle se mahdollistaa myös vaihtoehtoisen tavan voittaa pelin tuhoamalla vihollisia. Vihollisten pisteet nollaantuvat niiden tuhoutuessa ja tuhoutuminen on pysyvä. Pelaajan tuhoutuessa peli päättyy pelaajan häviöön.

8.1 Alustavat suunnitellut muutokset

Pelin kulku

Aluksi suunnitelmissa oli toteuttaa pelin kulku niin, että pelin tavoitteena on saavuttaa määritelty pistemäärä ja peli loppuu automaattisesti, kun pelaaja tai jokin vihollisista pääsee tavoitteeseen. Pistemäärän saavuttamiseen asti pelaaja sekä viholliset olisivat tuhoutuessaan syntyneet uudestaan pelialueelle.

Kehitysvaiheessa todettiin, että on yksinkertaisempaa sekä pelillisesti järkevämpää tehdä peliin rajattu aika pelisessiolle ja luopua uudelleensyntymisestä. Kyseisellä toteutuksella pelisessioiden kesto saadaan helposti pysymään lyhyinä ja maksimikesto vakiona. Tätä voidaan perustella sillä, että tavanomaisesti hyperkasuaalisten pelien sessiot ovat kestoltaan lyhyitä. Tällä suunnitelmalla itse toteutus on myös huomattavasti yksinkertaisempi tehdä, koska pystytään luopumaan uudelleensyntymisen sekä siihen liittyvien osien toteutuksesta.

KotH-alue

KotH-alueelle luodaan oma objekti, johon liitetään ZoneObjectController-skripti, joka liikuttaa alueobjektia pelialueen sisällä. Alueen jatkuvan liikkeen avulla saadaan lisättyä peliin dynamiikkaa.

Autojen interaktio KotH-alueen kanssa

Luodaan ZoneInteractionController-skripti, joka hallitsee keskitetysti autojen ja KotH-alueen välistä vuorovaikutusta. Autoille tehdään trigger collider, joka tunnistaa, kun kukin auto on KotH-alueen sisällä.

Ajastin

Luodaan pelaajalle näkyvä ajastin, joka käynnistyy pelin alkaessa ja ajan mennessä nollaan peli loppuu automaattisesti.

Vihollisten tekoäly

Muokataan EnemyCar-skriptissä sijaitsevaa tekoälyn logiikkaa niin, että viholliset pyrkivät KotH-alueelle sekä yrittävät tuhota muita alueen sisällä olevia autoja. Jotta peli ei keskity pelkästään alueen sisälle, säilytetään vihollisilla mahdollisuus hyökätä myös alueen ulkopuolella oleviin autoihin.

Pistelasku

Luodaan pistelasku, joka ylläpitää tietoa johdossa olevan vihollisen ja pelaajan pistemäärästä. Pistemäärät tehdään näkyväksi pelaajalle. Lisäksi lisätään peliin visuaalinen kruunuobjekti, joka tulee näkyväksi peliä johtavan auton yläpuolelle.

Kenttä

Kentän vajoaminen otetaan pois käytöstä poistamalla LevelController-skriptistä SinkCoroutine- sekä SinkLevelHexagon-metodit.

8.2 Muutosten toteutus ja havainnot

Peliin luotiin KotH-aluetta varten objekti, johon lisättiin shader ulkonäköä varten sekä Trigger Collider-komponentti, jonka avulla voidaan tulkita objektiin kohdistuvia osumia muilta objekteilta.

Kuvio 7. KotH-alue pelin sisällä

Peliin haluttiin tuoda lisää dynaamisuutta luomalla KotH-alueelle ZoneObjectController-skripti, jonka avulla alue pysyy jatkuvassa liikkeessä. Näin mahdollistettin, että alue voi siirtyä mihin tahansa pisteeseen pelialueen sisällä sen sijaan, että sijainti pysyisi staattisena. Skriptiin on asetettu raja-arvot, jonka sisällä alue voi liikkua. Alueelle asetetaan SetNewDestination-metodissa sattumanvarainen määränpää raja-arvojen sisältä. Alue liikkuu määränpäätä kohti jokaisella framella MoveToDestination-metodin avulla, kunnes määränpää on saavutettu, jolloin asetetaan uusi.

Kentän vajoamisen toiminnallisuus toi alkuperäiseen ydinpeliin oman dynamiikkansa, mutta se päätettiin jättää pois tästä muunnelmasta, koska se saattaisi luoda mekaniikasta liian sekavan tai

luoda liiallista samankaltaisuutta ydinpelin kanssa. Toiminnallisuus poistettiin LevelController-skriptistä poistamalla SinkCoroutine- ja SinkLevelHexagon-metodit.

```
void Update()
{
    if (gameControllerScript.gameRunning)
    {
        if (transform.position != goalPos)
        {
            MoveToDestination();
        }

        else if (transform.position == goalPos)
        {
            SetNewDestination();
        }
    }
}

1 reference
void MoveToDestination()
{
    transform.position = Vector3.MoveTowards(transform.position, goalPos, speed);
}

1 reference
void SetNewDestination()
{
    goalPos = (new Vector3(Random.Range(xNegBoundary, xPosBoundary), 0, Random.Range(zNegBoundary, zPosBoundary)));
}
```

Kuvio 8. KotH-alueen liikkumisen logiikka ZoneObjectController-skriptissä

Jokaiselle autolle luotiin ZoneTrigger-niminen lapsiobjekti, joka nimensä mukaisesti toimii triggerinä. Objektiin lisättiin skripti, jossa tunnistetaan OnTriggerStay- ja OnTriggerExit-metodien avulla aina, kun auto on KotH-alueen sisällä tai poistuu alueen sisältä. Triggerin avulla tallennetaan muuttujaan tieto siitä, onko auto alueen sisällä. Lisäksi suoritetaan pisteiden lisääntyminen ja tallennus sekä auton tuhoutuessa pisteiden nollaus.

GameController-skriptiin lisättiin myös muuttuja nimeltä gameRunning, johon tallennetaan tieto siitä, onko peli käynnissä. Tähän muuttujaan tehdään viittauksia useissa metodeissa, joita on tarkoitus ajaa vain pelin ollessa käynnissä.

ZoneInteractionController-skripti oli suurin peliin tehtävä lisäys. Skriptiin on keskitetty kaikki KotH-alueeseen liittyvien interaktioiden toiminnot. Kyseisten toimintojen pohjalle tarvitaan tieto pelissä olevista autoista. Vihollisautojen objektit ovat EnemyController-objektin lapsiobjekteja, joten ZoneInteractionController-skripti sijoitettiin kyseiseen objektiin, koska lapsiobjekteihin on helppo päästä käsiksi parent-objektin kautta.

ZoneInteractionController-skripti sisältää peliobjektilistan nimeltä enemyCars, johon lisätään pelin alkaessa kaikki EnemyController-objektin aktiiviset lapsiobjektit eli kaikki pelissä olevat vihollisautot. Pelin edetessä listasta täytyy poistaa tuhoutuneet viholliset, joten luotiin HandleDead-metodi, jossa tarkastellaan jatkuvasti kunkin vihollisauton EnemyCar-skriptin aktiivisuutta. Vihollisauton tuhoutuessa edellä mainittu skripti deaktivoituu, jolloin metodi poistaa vihollisauton listasta.

```
private void HandleDead()
{
    for (int i = 0; i < enemyCars.Count; i++)
    {
        enemyAlive = enemyCars[i].gameObject.GetComponent<EnemyCar>().isActiveAndEnabled;

        if (gameRunning)
        {
            if (!enemyAlive)
            {
                enemyCars[i].GetComponentInChildren<ZoneTrigger>().enabled = false;
                enemyCars[i].transform.GetComponent<EnemyCar>().crown.SetActive(false);

                if (carsInZone.Contains(enemyCars[i]))
                {
                    carsInZone.Remove(enemyCars[i]);
                }

                enemyCars.RemoveAt(i);
            }
        }
    }
}
```

Kuvio 9. HandleDead-metodin logiikka

Olennaisessa osassa pelissä ovat KotH-alueelta kertyvät pisteet, joiden pohjalta määritetään kuka johtaa peliä. ZoneInteractionController-skriptiin tehtiin keskitetty metodi nimeltä CountScores pisteiden laskua varten. Skriptissä haetaan pelaajan sekä jokaisen aktiivisen vihollisen ZoneTrigger-skriptin sisältämä KotHScore-muuttuja, johon on tallennettu autolle kertyneet pisteet. Kaikkien autojen pistemääriä verrataan keskenään ja korkein vihollisauton pistemäärä tallennetaan enemyTopScore-muuttujaan sekä pelaajan pistemäärä playerScore-muuttujaan.

```
private void CountScores()
{
    playerScore = playerZoneTriggerScript.kothScore;

    tempHighScore = playerScore;

    tempEnemyTopScore = 0;

    if (tempHighScore > 0)
    {
        kingOfTheHill = playerObject;
    }

    for (int i = 0; i < enemyCars.Count; i++)
    {
        enemyScore = enemyCars[i].transform.GetChild(0).GetComponent<ZoneTrigger>().kothScore;

        if (enemyScore > tempHighScore)
        {
            tempHighScore = enemyScore;

            kingOfTheHill = enemyCars[i];
        }

        if (enemyScore > tempEnemyTopScore)
        {
            tempEnemyTopScore = enemyScore;
        }
    }

    enemyTopScore = tempEnemyTopScore;
```

Kuvio 10. CountScores-metodin pisteenlasku-osuus

Pelin näkymään haluttiin lisätä pelaajan oma pistemäärä sekä johtavan vihollisen pistemäärä. Editorissa lisättiin GameView-objektin alle ContenderScore- ja PlayerScore-objektit, joihin lisättiin Text Mesh Pro UGUI-komponentit, joiden avulla saadaan teksti skriptin kautta näkyväksi ruudulle. GameView-skriptissä haetaan ZoneInteractionController-skriptistä pisteet, jotka liitetään tekstielementteihin. GameView-skriptiin tehtiin myös toinen pieni lisämuutos, jossa pelaajan pistetekstin väriä muutetaan vihreäksi silloin, kun pelaaja johtaa pisteissä ja punaiseksi silloin, kun jokin muu auto on johdossa.

Kuvio 11. Pisteet pelinäkymän alaosassa

Lisäksi peliin haluttiin jokin visuaalinen indikaattori, josta voidaan nähdä mikä autoista johtaa peliä. Indikaattoriksi valittiin kruunua esittävä 3D-objekti, joka lisättiin jokaisen automallin prefabiin.

CountScores-metodissa on GameObject-tyypin muuttuja nimeltä kingOfTheHill, johon asetetaan pisteiden laskun yhteydessä eniten pisteitä hallitsevan auton peliobjekti. Peliobjekti on irrallinen automallin prefab-instanssista, johon kruunu on sijoitettu, mutta pelaajan ”PlayerCar” ja vihollisautojen ”EnemyCar” skripteihin pystyttiin lisäämään muuttuja, joka on viittaus peliobjektille kuuluvaan prefab-instanssiin ja sen sisältämään kruunuobjektiin. Muuttujaan viittaamalla voidaan CountScores-metodissa aktivoida kruunuobjekti peliä johtavalle autolle, sekä deaktivoida se sen vaihtuessa. Kruunu on nähtävissä aiemmin esitetyssä kuviossa 7.

```
if (kingOfTheHill == playerObject)
{
    playerObject.transform.GetComponent<PlayerCar>().crown.SetActive(true);

    for (int i = 0; i < enemyCars.Count; i++)
    {
        enemyCars[i].transform.GetComponent<EnemyCar>().crown.SetActive(false);
    }
}

else
{
    playerObject.transform.GetComponent<PlayerCar>().crown.SetActive(false);

    for (int i = 0; i < enemyCars.Count; i++)
    {
        if (enemyCars[i] == kingOfTheHill)
        {
            enemyCars[i].transform.GetComponent<EnemyCar>().crown.SetActive(true);
        }

        else
        {
            enemyCars[i].transform.GetComponent<EnemyCar>().crown.SetActive(false);
        }
    }
}
```

Kuvio 12. CountScores-metodin kruunuobjektin aktivointi ja deaktivointi

Tekoälyn ohjaamille autoille tarvittiin tieto kaikista KotH-alueen sisällä olevista autoista, jotta niitä pystyttiin asettamaan tekoälyn kohteeksi. Se toteutettiin tekemällä ZoneInteractionController-

skriptiin viimeisenä toiminnallisuutena ZoneOccupationStatus-metodi. Metodi hakee autojen ZoneTrigger-skriptistä InsideZone-nimisen boolean-muuttujan joka sisältää tiedon siitä, onko auto KotH-alueen sisällä vai ulkona. Metodi sisältää carsInZone-peliobjektilistan, johon lisätään kunkin auton peliobjekti silloin, kun ne menevät KotH-alueen sisälle ja niiden poistuessa alueelta ne poistetaan listasta.

```
private void ZoneOccupationStatus()
{
    playerInsideZone = playerObject.transform.GetChild(0).GetComponent<ZoneTrigger>().insideZone;

    if (playerInsideZone && !carsInZone.Contains(playerObject))
    {
        carsInZone.Add(playerObject);
    }

    else if (!playerInsideZone && carsInZone.Contains(playerObject))
    {
        carsInZone.Remove(playerObject);
    }

    for (int i = 0; i < enemyCars.Count; i++)
    {
        enemyInsideZone = enemyCars[i].transform.GetChild(0).GetComponent<ZoneTrigger>().insideZone;

        if (enemyInsideZone && !carsInZone.Contains(enemyCars[i]))
        {
            carsInZone.Add(enemyCars[i]);
        }

        else if (!enemyInsideZone && carsInZone.Contains(enemyCars[i]))
        {
            carsInZone.Remove(enemyCars[i]);
        }
    }
}
```

Kuvio 13. ZoneOccupationStatus-metodin logiikka

Vihollisautojen EnemyCar-skriptissä haetaan ZoneInteractionController-skriptistä aiemmin mainittu carsInZone-peliobjektilista. Sitä ei voida käyttää tekoälyssä sellaisenaan, koska se voi sisältää minkä tahansa auton peliobjektin eli myös tekoälyn itsensä. Tästä aiheutuisi bugi, jossa vihollinen voisi ottaa itsensä kohteeksi. Ratkaisuna tehtiin EnemyCar-skriptiin lisäys, jossa skriptiin luotuun enemiesInZone-peliobjektilistaan lisätään kaikki peliobjektit carsInZone-listasta, poislukien kyseisen skriptin instanssin omistavan objektin. enemiesInZone-listasta myös poistetaan peliobjekteja sitä mukaa, kun ne poistuvat carsInZone-listasta.

```
if (enemiesInZone.Count != 0)
{
    for (int i = 0; i < enemiesInZone.Count; i++)
    {
        if (!zoneIntCtrlScript.carsInZone.Contains(enemiesInZone[i]))
        {
            enemiesInZone.Remove(enemiesInZone[i]);
        }
    }

    for (int i = 0; i < zoneIntCtrlScript.carsInZone.Count; i++)
    {
        if (!enemiesInZone.Contains(zoneIntCtrlScript.carsInZone[i]) && zoneIntCtrlScript.carsInZone[i] != this.gameObject)
        {
            enemiesInZone.Add(zoneIntCtrlScript.carsInZone[i]);
        }
    }
}

else if (zoneIntCtrlScript.carsInZone.Count != 0 && enemiesInZone.Count == 0)
{
    for (int i = 0; i < zoneIntCtrlScript.carsInZone.Count; i++)
    {
        if (zoneIntCtrlScript.carsInZone[i] != this.gameObject)
        {
            enemiesInZone.Add(zoneIntCtrlScript.carsInZone[i]);
        }
    }
}
```

Kuvio 14. Peliobjektien hallinta enemiesInZone-listassa

Koska pelin keskeinen elementti on KotH-alue, täytyi tekoälyä muokata niin, että se hakeutuu alueelle. Ensimmäinen muutos, jolla vaikutettiin tähän oli logiikaltaan sellainen, että jos alueen sisällä ei ole yhtään autoa, tekoäly hakeutuu alueelle. Tällä varmistettiin se, että alueelle pyrkii jatkuvasti jokin autoista. Käytännön toteutuksessa Zone-objektin eli KotH-alueen sisälle sijoitettiin kolmeen eri pisteeseen objektit, jotka eivät ole pelissä näkyviä. Niiden tarkoitus on sijainnin välitys tekoälylle EnemyCar-skriptissä.

Tekoäly sijaitsee SetDestination-metodissa, johon tehtiin lisäys, jossa KotH-alueen ollessa tyhjä tekoälyn kohteeksi asetetaan yksi alueen sisältämien objektien sijainneista. Kun tekoäly saavuttaa kohteen, se ottaa uudeksi kohteeksi seuraavan objektin sijainnin alueen sisältä. Tekoäly kiertää alueen objektien sijainteja niin pitkään, kunnes alueelle saapuu toinen auto tai jos tekoälyllä ei enää ole käytössä asetta, jolloin se lähtee hakemaan uutta. Aseen hakeminen säilytettiin tekoälyn ensimmäisenä prioriteettina, kuten kantapelissäkin, koska se on pelin toiminnan ja etenemisen kannalta olennaista.

```
if (enemiesInZone.Count == 0)
{
    if (targetKothPoint == null || targetKothPoint == "Pos1")
    {
        targetKothPoint = "Pos1";
        X = kothPosOne.x;
        Z = kothPosOne.z;
        Vector3 heading = new Vector3(transform.position.x, 0, transform.position.z) - new Vector3(X, 0, Z);
        float distanceToTarget = heading.magnitude;

        if (distanceToTarget < destinationReachedTreshold)
        {
            targetKothPoint = "Pos2";
        }
    }

    else if (targetKothPoint == "Pos2")
    {
        targetKothPoint = "Pos2";
        X = kothPosTwo.x;
        Z = kothPosTwo.z;
        Vector3 heading = new Vector3(transform.position.x, 0, transform.position.z) - new Vector3(X, 0, Z);
        float distanceToTarget = heading.magnitude;

        if (distanceToTarget < destinationReachedTreshold)
        {
            targetKothPoint = "Pos3";
        }
    }

    else if (targetKothPoint == "Pos3")
    {
        targetKothPoint = "Pos3";
        X = kothPosThree.x;
        Z = kothPosThree.z;
        Vector3 heading = new Vector3(transform.position.x, 0, transform.position.z) - new Vector3(X, 0, Z);
        float distanceToTarget = heading.magnitude;

        if (distanceToTarget < destinationReachedTreshold)
        {
            targetKothPoint = "Pos1";
        }
    }
}
```

Kuvio 15. Tekoälyn kohteen valinta KotH-alueen ollessa tyhjä

Alkuperäinen tekoälyn logiikka perustuu siihen, että kun auto on saanut aseen, se asettaa kohteekseen lähimmän vihollisauton. Jos tekoälyssä olisi käytetty ydinpelin alkuperäistä viholliskohteen valintaa, ei peli silloin keskittyisi KotH-alueella kilpailuun, joten tekoälyyn tarvittiin vielä lisämuutoksia. Tekoälyn ei kuitenkaan haluttu keskittyvän pelkästään alueella taisteluun, koska se olisi tehnyt vihollisten käyttäytymisestä ennalta-arvattavan. Ratkaisuna siihen tehtiin alkuperäisen viholliskohteen valinnan rinnalle toinen logiikka, jossa tekoäly ottaa kohteeksi vain KotH-alueen sisällä olevia vihollisia. Näiden kahden logiikan välille tehtiin randomisointi, jolloin tekoäly valitsee sattumanvaraisesti toisen kahdesta vaihtoehdosta.

Uutena lisätty logiikka toimii niin, että EnemyCar-skriptiin luodusta enemiesInZone-peliobjektilistasta valitaan kohteeksi sattumanvarainen peliobjekti ja se pysyy tekoälyn kohteena niin pitkään kuin se on KotH-alueen sisällä tai kunnes tekoäly lähtee hakemaan uutta asetta. Ennen jokaista kertaa, kun tekoälyn kohde on muuttumassa, tehdään sattumanvarainen valinta, käytetäänkö alkuperäistä vai uutena luotua vihollisko hteen valinnan logiikkaa.

```
targetKothPoint = null;
int rand = Random.Range(0, 2);

if (targetInZone != null || rand == 0 && !targetSetOther)
{
    if (targetInZone != null && enemiesInZone.Contains(targetInZone))
    {
        X = targetInZone.transform.position.x;
        Z = targetInZone.transform.position.z;
    }

    else
    {
        int choiceFromList = Random.Range(0, enemiesInZone.Count);
        targetInZone = enemiesInZone[choiceFromList];
        X = targetInZone.transform.position.x;
        Z = targetInZone.transform.position.z;
    }
}

else if (targetSetOther || rand == 1)
{
    targetSetOther = true;
    Vector3 target = MoveToTargetVehicle();
    X = target.x;
    Z = target.z;
}
```

Kuvio 16. Tekoälyn vihollisko hteen valinta

Pelisession kestoa haluttiin rajata lyhyeksi, joten peliin kehitettiin ajastin. GameView-objektin alle lisättiin ajastimelle objekti nimeltä GameTimer, sekä sille Text Mesh Pro UGUI-komponentti. GameView-skriptiin tehtiin ajastin, jonka määritetyn ajan loputtua peli lopetetaan kutsumalla PlayerController-skriptin EndGame-metodia. Metodille välitetään tieto, onko pelaaja voittanut vai

hävinnyt pelin. Ydinpelissä pelin päätyttyä häviöön, tulee näkyviin teksti ”WRECKED!”, joka ei tässä tapauksessa sopinut tilanteeseen, jossa peli päättyy ajan loppumiseen. Skriptiin tehtiin lisäys, jossa pelin päättyessä ajan loppumiseen ja pelaajan häviöön, tulee alkuperäisen tekstin tilalle ”LOST!”. Ajan vähiin käymistä indikoimaan tehtiin myös lisäys, jossa ajastimen numerot muuttuvat viimeisen kymmenen sekunnin kohdalla punaisiksi.

```
public void GameTimer()
{
    gameTime -= Time.deltaTime;
    int seconds = Mathf.FloorToInt(gameTime);
    gameTimerText.text = string.Format("{0}", seconds);

    if (seconds < 11 && !colorChanged)
    {
        gameTimerText.color = new Color32(255, 0, 0, 255);
        colorChanged = true;
    }

    if (seconds <= 0)
    {
        if (zoneInteractionCtrlScript.playerScore >= zoneInteractionCtrlScript.enemyTopScore)
        {
            PlayerController.Instance.EndGame(true);
        }

        else if (zoneInteractionCtrlScript.enemyTopScore > zoneInteractionCtrlScript.playerScore)
        {
            loseText.text = "LOST!";
            PlayerController.Instance.EndGame(false);
        }
    }
}
```

Kuvio 17. Ajastimen toiminnan logiikka

Pelin loputtua pelaajalle kertyy kolikoita perustuen siihen, kuinka hyvin pelaaja on pärjännyt pelissä. Alkuperäistä kolikoiden laskun sopivuutta testattiin tähän pelimuunnelmaan ja se todettiin toimivaksi pienellä lisäyksellä. Kolikoiden lasku pidettiin muilta osin samana, mutta pelaajan voittaessa tulee hänelle aiempaan verrattuna 20 kolikkoa enemmän. Sillä pyrittiin siihen, että pelin voittamiseen pyrkiminen olisi kannattavampaa kuin kolikoiden kerääminen vain vihollisia tuhoamalla, riippumatta siitä voittaako pelin. Laajemman käyttäjätestauksen avulla kolikoiden laskun toteutuksen sopivuudesta olisi ehkä saatu vielä parempi käsitys.

Kuvio 18. King of the Hill-pelimuunnelman lopullinen versio

9 Vastaukset tutkimuskysymyksiin

9.1 Mitä uudelleenkäytettävyys on pelinkehityksessä?

Pelikehityksessä uudelleenkäytettävyyttä mietittäessä koodin puolella pätevät suurelta osin samat säännöt ja periaatteet kuin muussakin ohjelmistokehityksessä. Koska tämän tutkimuksen tapauksessa uudelleenkäytettävyyttä haluttiin tutkia juuri siitä näkökulmasta, että samasta pelistä luotaisiin useita erilaisia, nousi pelisuunnittelun osuus vahvasti esille. Jotta erilaiset pelit toimivat omina julkaisuinaan, tulee niiden erottua toisistaan tarpeeksi, ettei pelaaja huomaa uudelleenkäytettyä ydintä.

Pelikehityksessä uudelleenkäytettävyys jakautuu täten kahteen osaan, pinnan alla tapahtuvaan tekniseen toteutukseen, joka pitää suunnitella mahdollistamaan helpot muutokset, sekä näkyviin ominaisuuksiin eli pelimekaniikkoihin ja audiovisuaaliseen ilmeeseen. Hyvin suunniteltu, uudelleenkäytettävyyden periaatteita käyttävä koodi mahdollistaa monet eri muunnelmat, kun taas pelaajalle näkyvissä olevat ominaisuudet piilottavat oikein suunniteltuina hyvinkin samanlaiset tekniset ratkaisut.

9.2 Miten uudelleenkäytettävyys toteutetaan hyperkasuaalisten pelien kehityksessä?

Tutkimuksessa nousi vahvasti esiin tarve suunnitella uudelleenkäytettävyyden varalle jo varhaisessa vaiheessa. Koska kantapelinä toimi jo olemassa oleva peli, jota ei aiemmin suunniteltu muunnelmia varten, tuli kehityksessä vastaan suurempia koodimuutoksia vaativia seikkoja. Jos nämä seikat otetaan huomioon jo ennen kantapelin luomista, tekee se muunnelmien luomisesta myöhemmin nopeampaa ja tehokkaampaa.

Hyvänä esimerkkinä toimii Drift Royalen autojen ominaisuudet. Alkuperäinen peli ei tee eroa pelaajan ja vihollisten liikkumisen välille, joka vaikeuttaa muunnelmia, joissa pelaajan hahmon halutaan käyttäytyvän erilailla. Uudelleenkäytettävää kantapeliä suunnitellessa onkin hyvä ajatella jo valmiiksi mahdollisten muunnelmien haluttuja genrejä, jotta vastaavilta tilanteilta vältyttäisiin. SOLID-periaatteiden noudattamisessa voi olla apua myös tässä. Tutkimuksen kehitysvaiheessa huomattiin myös, että tilanteessa jossa kantapeliä ei ole suunniteltu uudelleenkäytettävyyden kannalta, on myös muunnelmaa vaikeampi kehittää uudelleenkäytettävyyden periaatteita noudattaen.

Hyperkasuaalisten pelien tapauksessa suuri etu uudelleenkäytettävyyden puolesta on niiden yksinkertaisuus. Mitä yksinkertaisempi peli on, sitä helpompi siitä on tehdä muunnelmia, koska ominaisuuksia on aina parempi lisätä kuin muokata. Otetaan esimerkiksi takaa kuvattu “endless runner”-genren peli, jossa pelaajan tarvitsee vain näpäyttää ruutua hypätäkseen esteiden yli. Mekaniikkojen lisääminen kyseiseen peliin olisi hyvin helppoa muuttamatta vanhoja, mutta jos ohjaus olisi alun perin nelisuuntainen erillisellä hyppynapilla, vaatisi esimerkiksi kuvakulman muuttaminen mahdollisesti muutosta myös ohjaukseen. Tämä ei tietysti tarkoita, että kaikista uudelleenkäytettävistä peleistä tulisi tehdä mahdollisimman yksinkertaisia, mutta jos uudelleenkäytettävyys on päällimmäinen määränpää, on yksinkertaista kantapeliä hyvä tavoitella.

9.3 Miten samaa ydintä käyttävien pelien välillä luodaan riittävästi eroa?

Pelimuunnelmia kehitettäessä nousi esiin yllättävänä seikkana se, kuinka vähän muutoksia lopulta tarvittiin uuden pelikokemuksen luomiseksi. Tärkein muutos pelimekaniikkojen kannalta oli antaa

pelaajalle uusi ja erilainen tavoite, sillä se muuttaa pelaajan suhtautumista peliin ja täten saa pelikokemuksen tuntumaan erilaiselta, vaikka muut perusmekaniikat olisivatkin hyvin samanlaisia.

Mietittäessä muutosten pitämistä mahdollisimman vähäisenä, pitää nostaa esille pelin ohjaus. Alkuperäisen ohjauksen soveltuvuus eri genreihin vaikuttaa huomattavasti uudelleenkäytettävyyteen ja täten mahdollisiin luotaviin eroihin pelien välillä. Tutkimuksen kantapelin tapauksessa ohjattavuus toimii ilman suurempia muutoksia tietysti ylhäältä kuvatuissa rallipeleissä tai muissa vähemmän tarkkaa kääntyvyyttä vaativissa genreissä, mutta muunlaiset genret vaativat perusteellisempaa muutosta koodiin. Yleisenä sääntönä voisi pitää, että mitä yksinkertaisempi peli on pelata, sitä helpommin se kykenee useampaan erilaiseen muunnelmaan.

Tässä tutkimuksessa keskityimme koodiin ja tekniseen toteutukseen, joten pelien audiovisuaalinen puoli jäi taka-alalle. Mainittava on kuitenkin, että malleja, tekstuureja ja efektejä muuttamalla pelistä saa hyvin erituntuisen, mutta myös asiat kuten eri asioihin keskittyvä käyttöliittymä tai kamerakulman muutos jo itsessään muuttavat pelikokemusta huomattavalla tavalla. Uusien pelien äänimaailma ja musiikki ovat myös asioita, joihin jatkokehityksessä tulisi kiinnittää huomiota.

10 Pohdinta

10.1 Ajatuksia tuloksista

Tutkimuksen päällimmäisenä tavoitteena oli luoda annetun kantapelin pohjalta kaksi mahdollisimman paljon eroavaa muunnelmaa mahdollisimman pienin muutoksin. Tässä onnistuimme ja kehityksen aikana esiin syntyneet ongelmat tuottivat tietoa hyperkasuaalisten pelien uudelleenkäytettävyyden tarpeista. Myös SOLID-periaatteiden tarkastelu toi uutta näkökulmaa uudelleenkäytettävien pelien suunnitteluun.

Yhteistyö toimeksiantajan kanssa pelien muutoksia suunnitellessa oli erittäin hyödyllistä, koska pystyimme keskittymään muutosten tekemiseen ja raportointiin, sen sijaan että pelisuunnittelu veisi tutkimukselta aikaa. Lisäksi toimeksiantajan kokemus pelialalta auttoi ohjaamaan muutokset oikeaan suuntaan.

Tutkimuksen tärkein oppi on kantapelin suunnittelun tärkeys. Kantapelin koodi oli melko hajanaista ja kommentoinnin puuttuminen vaikeutti muutosten tekoa. Uudelleenkäytettävyys tulisi ottaa huomioon jo alkuvaiheessa, jotta tulevaisuudessa kehitettävät muunnelmat olisivat helppo ja nopea toteuttaa ilman, että hyvistä koodausperiaatteista luovutaan. Kun koodikanta on vankalla ja joustavalla pohjalla, voidaan muunnelmia suunnitellessa keskittyä vähemmän teknisiin ongelmiin ja enemmän pelien erotteluun toisistaan mekaniikkojen sekä audiovisuaalisen ilmeen kautta. Lisäksi koodikannan ymmärtämisen ja täten ajankäytön kannalta järkevintä olisi, että muunnelmien kehityksessä mukana olisi tiiviisti ainakin osa alkuperäisen pelin ohjelmoijista.

10.2 Tutkimuksen luotettavuus ja jatkokehitys

Vaikka toimeksiantajan mielipiteet ja kommentit mobiilipelialan ammattilaisyrityksenä tuovat tutkimukselle luotettavuutta, olisi sitä voinut lisätä suorittamalla pelitestausta laajemmalla yleisöllä. Tätä kautta olisi saatu tietoa varsinkin pelien erottuvuuden onnistumisesta, mutta se olisi vaatinut myös suurempaa panosta pelien ulkoasun muokkaamiseen ja tämä ei ollut toimeksiantajan tavoitteena tämän tutkimuksen puitteissa.

Myös tutkijoiden vähäinen aikaisempi kokemus SOLID-periaatteiden käytöstä saattaa vaikuttaa tuloksiin, sillä niiden täysin oikeaoppinen käyttö vaatii huomattavan paljon aikaa ja opettelua. Suuremmalla kokemuksen määrällä olisi mahdollista saada enemmän konkreettista tietoa koodipohjan muokkaamiseen uudelleenkäytettäväksi.

Vaikka alkuperäinen tavoite liittyi spesifisti toimeksiantajan tarpeisiin, uudelleenkäytettävyyden teoria ja tutkimuksen tulokset sen osalta voidaan yleistää ainakin hyperkasuaalisten mobiilipelien osalta. Lisäksi SOLID-periaatteiden hyödyt soveltuvat myös muuhun peli- sekä ohjelmistokehitykseen.

Tutkimuksessa keskityimme enemmän tekniseen toteutukseen ja koodikannan muokkaamiseen. Jatkotutkimukset voisivat keskittyä tarkemmin juuri erojen tuottamiseen pelien välille, ottaen näkökulmaksi enemmän sisällöntuotannon, sekä pelaajien psykologian ja käyttäen laajempaa pelitestausta hyväkseen. Jatkossa voisi myös olla hyödyllistä luoda kantapeli, joka jo alusta alkaen on suunniteltu tämän tutkimuksen kaltaiseen uudelleenkäytettävyyteen. Kantapelin kehityksen tutkimuksessa nousisi esille konkreettisemmin alkupään suunnittelun tarpeet ja toteutus.

Lähteet

Fang, E. 2019. How Hyper-Casual Gaming Took Over China (And The World). https://www.forbes.com/sites/forbestechcouncil/2019/04/19/how-hyper-casual-gaming-took-over-china-and-the-world/.

Heinze, J. 2017. The Ascendance of Hyper-Casual Part Two: What defines the genre? https://www.pocketgamer.biz/comment-and-opinion/65390/hyper-casual-part-two-what-defines-hyper-casual/.

Heinze, J. 2018. Hyper-casual: Mobile gaming's newest genre. https://blog.applovin.com/hyper-casual-mobile-gamings-newest-genre/

Kananen, J. 2015. Kehittämistutkimuksen kirjoittamisen käytännön opas: miten kirjoitan kehittämistutkimuksen vaihe vaiheelta. https://janet.finna.fi/Record/janet.290348.

Kananen, J. 2015. Opinnäytetyön kirjoittajan opas: Näin kirjoitan opinnäytetyön tai pro gradun alusta loppuun. https://janet.finna.fi/Record/janet.283510.

Martin, R. C. 2003. Principles of OOD. http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod.

Noback, M. 2018. Principles of Package Design: Creating Reusable Software Components. http://common.books24x7.com.ezproxy.jamk.fi:2048/toc.aspx?bookid=144483.

Unity User Manual. https://docs.unity3d.com/Manual/index.html.

Villalobos, J. O. 2019. S.O.L.I.D. Principles. https://www.brainstobytes.com/tag/s-o-l-i-d-principles/