Kustomisaation kehittäminen mobiilipelissä

Jarkko Niskanen

Opinnäytetyö, AMK
Syyskuu 2023
Tietojenkäsittely ja tietoliikenne
Tieto- ja viestintätekniikan tutkinto-ohjelma

jamk | Jyväskylän ammattikorkeakoulu
University of Applied Sciences

jamk

Kuvailulehti

Niskanen, Jarkko

Kustomisaation kehittäminen mobiilipelissä

Jyväskylä: Jyväskylän ammattikorkeakoulu. Syyskuu 2023, 50 sivua.

Tieto- ja viestintätekniikan tutkinto-ohjelma. Opinnäytetyö AMK.

Julkaisun kieli: suomi

Julkaisulupa avoimessa verkossa: kyllä

Tiivistelmä

Pelialalla kilpailu on runsasta, ja uudentyyppiselle sisällölle on aina tarvetta. Tutkimuksellisella kehittämistyöllä selvitettiin kehitettävän mobiilipelin vastaavia toteutuksia ja pyrittiin löytämään uusia kustomisaatiototeutuksia. Selvitetyistä kustomisaatiototeutuksista etsittiin toistuvia teemoja, kehityskohteeksi valittiin tarrojen asettaminen ajoneuvoon ajallisesti kannattavimpana vaihtoehtona, ja tätä varten selvitettiin eri teknologioita toteutusta varten.

Tarrojen asetteluun löydettiin useampi teknologiavaihtoehto, Unityn tarjoamia sekä kolmannen osapuolen kehittämiä. Jokainen teknologiavaihtoehto käsiteltiin ja mahdollisuuksien mukaan kokeiltiin kehitettävässä kohteessa. Valittu teknologia oli jo ennestään hyödynnetty ajoradoilla, kuten seinään projektoitu spraymaalaus.

Työn tuloksena kehitettiin tarrojen asettelu Unityllä, joka muuntaa käyttäjän syötteet ajoneuvon paikalliseen koordinaatistoon, laskee piirtoalueen syvyyden, sekä yhdistää luodut tarrat yhdeksi peliobjektiksi. Tarrojen data muunnettiin JSON muotoon ja tallennettiin mobiililaitteelle salattuna. Yhdistämisen tuoma suorituskykymuutos mitattiin ruudunpäivitystestillä, jossa samalla varmistettiin, ettei tarrojen implementaatio heikentänyt pelin suorituskykyä.

Kehitystyön perusteella mobiilipelikehityksessä suuria objektimääriä tulee välttää, sillä ruudunpäivitysnopeus alkaa laskemaan hyvin nopeasti. Unityllä kehittäessä objektien skaalauksia ei lähtökohtaisesti tule muuttaa, sillä se tuo ongelmia myöhemmässä kohtaa, pahimmillaan suorituskykyongelmia. Käyttöliittymää suunnitellessa on hyvä huomioida mahdollisimman selkeät symbolit sekä pelaajan ensimmäinen kokemus uudesta ominaisuudesta.

Avainsanat (asiasanat)

Unity, pelikehitys, kustomointi, projektio

Muut tiedot (salassa pidettävät liitteet)

jamk

Description

Niskanen, Jarkko

Improving customization in a mobile game

Jyväskylä: JAMK University of Applied Sciences, September 2023, 50 pages.

Degree Programme in Information and Communication Technology. Bachelor´s thesis.

Permission for open access publication: Yes

Language of publication: Finnish

Abstract

There is a lot of competition in video game industry, and companies are always looking for new type of content. The aim of the development work was to look up what customization options other games has to offer compared to the mobile game being developed and find new type of customization. Repeating features were searched within gathered data and decal system was selected to be implemented as the most reasonable feature in development time. Multiple different technologies were examined for projection technology.

There were multiple different types of decal projection technologies, Unity's offered and third-party developed. Each technology option was examined and tested, if possible, selected technology were already implemented in the game as decorations within racetracks such as spray-painting in a wall.

As a result, decal projection system was implemented with Unity, system automatically converts user given inputs to a local coordinate system, calculates the depth of the projection and combines all decals into a single mesh. Decal data was converted into JSON format, which was encrypted and stored locally to the device. Mesh combining performance was tested with different frames per second test scenarios which at the same time confirmed that decal system did not reduce the performance of the game.

Based on development work, large amounts of game objects should be avoided in mobile game environment, since it will affect performance very quickly. Game object scale chances should also be avoided to prevent problems later, in the worst-case scenario it can cause performance issues. When designing user interface, it is good to consider symbols that are clear as possible and first-time user experience is not confusing.

Keywords/tags (subjects)

Unity, game development, customization, projection

Miscellaneous (Confidential information)

Sisältö

1 Johdanto ... 4
2 Tutkimusasetelma ... 5
2.1 Tutkimusongelma ... 5
2.2 Tavoite ja tutkimuskysymykset ... 5
2.3 Tutkimusmenetelmät ... 6
3 Kustomisaatio ... 7
3.1 Kustomisaatio yleisesti ... 7
3.1.1 Kustomisaatiotilanteet ... 7
3.1.2 Kosmeettiset esineet ... 7
3.2 Kustomisaation tärkeys ... 8
3.2.1 Pelaajien uskollisuus ... 8
3.2.2 Monetisaatio ... 9
4 Kustomisaatio Unityssä ... 10
4.1 Skriptaaminen ... 10
4.2 Visuaalisuus ... 12
4.3 Projektio ... 12
4.4 Omat luokat ... 13
4.5 Resurssit ... 14
4.6 Rajapinta ... 14
4.7 Singleton ... 16
4.8 Tapahtumat ... 17
4.9 JSON ... 18
5 Alkukartoitus ... 18
5.1 Boom Karts ... 18
5.2 Muiden mobiilipelien kustomisaatiot ... 19
5.3 Kustomisaatioiden vertailu ... 20
5.4 Kustomointitarpeet ... 21
5.5 Projektioteknologian valinta ... 21
6 Tarrojen toteutus ... 22
6.1 Projektion suunnittelu ... 22
6.2 Projektion kohdistaminen ... 24
6.3 Projektion skaalaus ja manuaalinen rotaatio ... 26
6.4 Maailman koordinaatistosta paikalliseen muuntaminen ... 28

6.5 Käyttäjän syötteiden havaitseminen 29
6.6 Tarran peilaaminen 30
6.7 JSON data 31
6.8 JSON datan salaus 34
6.9 Singletonin toteuttaminen 35
6.10 Tarrojen yhdistäminen 36
7 Tulokset 39
7.1 Projektiototeutuksen lopputulos 39
7.2 Suorituskykytesti 40
7.3 Käyttöliittymävaihtoehtojen vertailu 42
8 Pohdinta 44
Lähteet 47
Liitteet 50

Kuviot

Kuvio 1. Unityn inspector näkymä 11
Kuvio 2. Vehicle ja VehicleManager luokat 13
Kuvio 3. VehicleManager inspector näkymässä 13
Kuvio 4. Rajapinnan hyöty silmukassa 15
Kuvio 5. Singleton luokan määrittäminen Unityssä 16
Kuvio 6. MonoSingleton luokan määritys ja käyttö 17
Kuvio 7. Tapahtuman luominen ja käyttäminen 18
Kuvio 8. Ajoneuvoon valittu trail tehoste 19
Kuvio 9. Boom Kartsin ja muiden mobiilipelien kustomisaatiovaihtoehdot 20
Kuvio 10. UML-kaavio tarran muuttujista 23
Kuvio 11. Rotaation asettaminen objektiin 24
Kuvio 12. Törmäyksen sijainnin ja syvyyden laskeminen 25
Kuvio 13. Syvyyden laskenta 26
Kuvio 14. Rotaation ja skaalauksen laskeminen 27
Kuvio 15. Sijainnin ja rotaation muuntaminen paikalliseen koordinaatiojärjestelmään 28
Kuvio 16. Käyttäjän syötteessä käytetyt korutiinit 29
Kuvio 17. Projektion kohdistaminen Update funktiossa 30
Kuvio 18. JSON datan optimointiin käytetty struktuuri 32
Kuvio 19. JSON datan tallentaminen ja lataaminen 33

Kuvio 20. JSON dataan käytetyt luokat 34
Kuvio 21. DecalManager singletonin käyttö 36
Kuvio 22. Visuaalisten komponenttien nollaus ja decalRootin skaalauksen muuntaminen 37
Kuvio 23. Tarrojen yhdistämisen skaalausten laskeminen 38
Kuvio 24. Skaalausten normalisointi ja palauttaminen 38
Kuvio 25. Kaksi objektia eri skaaloilla 39
Kuvio 26. Rakettitarra ajoneuvossa 40
Kuvio 27. Ruudunpäivitystestit tarrojen yhdistämisen kanssa 41
Kuvio 28. Käyttöliittymän ensimmäinen versio eri tilanteissa 43
Kuvio 29. Käyttöliittymän viimeinen versio eri tilanteissa 44

1 Johdanto

Nykypeleissä on pelaajille usein tarjolla erilaisia kustomisaatiovaihtoehtoja, pelihahmolle voidaan vaihtaa asusteita, tai ajoneuvoon voidaan vaihtaa renkaita. Pelihahmon asusteita voidaan vaihtaa erilaisissa tilanteissa, kuten erillisessä kustomisointihuoneessa tai jopa pelikokemuksen keskellä. Kustomisaatiototeutus riippuu paljon pelistä, free-to-play mallin peleissä kustomisaation avulla peliä voidaan monetisoida. (Rovira 2013.)

Yhä useampi peli on nykyään ilmaiseksi pelattavissa, joskin pelissä tulee vastaan mikrotransaktioita tai mainoksia, näitä kutsutaan free-to-play mallin peliksi. Mallin tavoitteena on tarjota kehitetty peli ilmaiseksi, ja tulon lähteenä ovat mikrotransaktiot sekä mainokset. Mobiilipeleissä hyödynnetään muitakin keinoja, kuten päivittäistä palkintoa, pelaaja saa esimerkiksi avattavan arkun palaamalla päivittäin takaisin peliin. Mikrotransaktiot ovat pelin sisäisiä pieniä maksuja, joilla pelaaja saa peliin lisäsisältöä, kuten kosmeettisen esineen. (Luban 2011; Colagrossi 2021.)

Kosmeettisia esineitä esiintyy nykyään monissa peleissä, näiden perusideana on olla visuaalisia muutoksia, jotka eivät välttämättä tuo mitään lisäetuja. Puhutaan myös puhtaasti kosmeettisista esineistä, jolloin pelaaja saa ainoastaan visuaalisen muutoksen. Pelaajat saavat kosmeettisia esineitä useimmiten pelaamalla peliä, tai mikrotransaktioiden kautta. Tyypillisiä kosmeettisia esineitä ovat erilaiset asusteet pelihahmolle. (Böffel, Würger, Müsseler & Schlittmeier 2022.)

Toimeksiantaja Zaibatsu Interactive Oy kehittää free-to-play mallin mobiilipeliä, pelissä hyödynnetään mikrotransaktioita sekä mainostusta. Mikrotransaktioihin lukeutuu myös puhtaasti kosmeettisia esineitä, jotka tuovat pelaajille visuaalisia muutoksia. Zaibatsu pyrkii jatkuvasti kehittämään peliä parempaan suuntaan, viime aikoina kustomisaation kehittäminen on ollut puheenaiheena, joten tämä valittiin tutkimustyön aiheeksi.

Tutkimustyön tavoitteena oli kehittää pelin kustomisaatiota toteuttamalla uudenlaisia ominaisuuksia. Työ aloitettiin selvittämällä kustomisaatioon liittyvää teoriaa sekä tapoja toteuttaa kustomisaatiota Unity pelimoottorissa, tämän jälkeen kartoitettiin muiden peliyritysten tapoja toteuttaa kustomisaatiota peleissään. Kartoitukseen valittiin Boom Kartsin kaltaisia mobiilipelejä, saadun tiedon pohjalta valittiin esille nousseista vaihtoehdoista toteutettavaksi tarrojen asettelu ajoneuvoihin.

2 Tutkimusasetelma

2.1 Tutkimusongelma

Työn toimeksiantaja on Zaibatsu Interactive. Zaibatsu on vuonna 2014 perustettu Keski-Suomessa toimiva peli- ja koodistudio, pelikehityksen lisäksi yritys tarjoaa osaamistaan myös muiden web- ja mobiilikehittäjien projekteihin. (Game development studio 2023.)

Toimeksiantaja pyrkii tuomaan Boom Kartsiin jatkuvasti uutta pitääkseen peliä tuoreena, tämä tapahtuu säännöllisillä päivityksillä. Kustomisaatiomahdollisuuksia tuodaan peliin säännöllisesti, mutta suurimmaksi osaksi ne pohjautuvat aikaisempiin ratkaisuihin, esimerkiksi ajoneuvoon lisätään uusi maalipinta. Peliin halutaan jotain uutta, erityisesti kustomisaatioon. Pelialalla kilpailu on runsasta, vastaavia pelejä löytyy lähes aina, joten uudentyyppisiin toteutuksiin on hyvä kartoittaa, mitä vastaavissa toteutuksissa on.

2.2 Tavoite ja tutkimuskysymykset

Tutkimustyön pyrkimys on hankkia tietoa kustomisaatiosta, erityisesti erilaisia tapoja toteuttaa sitä Unityllä, kartoittaa tilannetta alalla sekä luoda selvitys kehityskohteista. Selvitetyistä kohteista tarkastellaan suosittuja sekä toistuvia kohteita, joiden perusteella valitaan toteutettavia kohteita. Kehittämistyön tarkoituksena on toteuttaa valittu kohde peliin ja tuloksissa mitata, ettei pelin suorituskyky laske ja että käyttöliittymämuutokset ovat mahdollisimman käyttäjäystävällisiä.

Tutkimuskysymykset:

1. Miten mobiilipeleissä toteutetaan kustomisaatiota?
2. Mitä kustomisaatiotarpeita Boom Kartsiin kohdistuu?
3. Miten kustomisaatiotarpeet toteutetaan Boom Kartsiin?

Ensimmäiseen kysymykseen vastataan keräämällä Google Play:n kautta löytyvistä peleistä kustomisointimahdollisuuksia. Toiseen kysymykseen vastataan tarkastelemalla ensimmäisen kysymyk-

sen pohjalta kerättyä tietoa, onko jotain suosittua tai toistuvaa kustomisaatiota, jota Boom Kartsissa ei ole. Viimeiseen kysymykseen vastataan toteutuksella, toteutuskohde valitaan toisessa kysymyksessä esille nousseiden kehityskohteiden perusteella.

Toteutuksessa pyritään optimoimaan koodia mahdollisimman paljon, jotta suorituskyky ei laskisi sekä pyrkimään uudelleenkäytettävyyteen. Suorituskykymuutokset mitataan ruudunpäivitysnopeustesteillä, jotta nähdään ettei pelin suorituskyky ole heikentynyt. Jokainen uudentyyppinen kustomisaatiototeutus tulee sisältämään jonkinlaisen käyttöliittymän, joten on tärkeää arvioida Zaibatsun tiimin kesken, mikä käyttöliittymä on käyttäjäystävällisin.

Työn lopputuloksena syntyy paranneltu tuote sekä selvitys kehityskohteista jatkokehitystä varten, pyrkimyksenä on myös yrittää kehittää sisällöntuotannon työprosessia, jos tulee ilmi jokin tehokkaampi tapa implementoida.

2.3 Tutkimusmenetelmät

Tutkimusmenetelmä opinnäytetyössä on tutkimuksellinen kehittämistyö. Työn teoriaosuuden tarkoituksena on tuottaa lisätietoa kustomisaatiosta käsitteenä sekä toteuttamisesta työn tekijälle ja toimeksiantajalle. Erityisesti Unityn käytänteet ohjaavat työn toteutusta, teoriaosuudessa perustellaan myös, miksi kustomisaatiota kannattaa kehittää.

Tutkimuksellisessa kehittämistoiminnassa tavoitteena on tuottaa tietoa aidossa toimintaympäristössä, pääpaino on kuitenkin kehittämisessä. Käytännön ongelmat ohjaavat kysymysten asettamiseen ja tiedontuotantoon, tällöin ei ole kyse pelkästään tiedon soveltamisesta. Kehittämistoiminnan tavoitteena ei ole ainoastaan ratkaista käytännön ongelmia, vaan pyritään myös perustelemaan ratkaisut sekä kuvaamaan ongelmat tarkasti, tällöin työn tulokset voidaan nostaa yleiselle tasolle. Työn tulokset dokumentoidaan, analysoidaan sekä arvioidaan kriittisellä näkökulmalla, kyseenalaistaminen on myös osa kehittämistoimintaa. (Toikko & Rantanen 2009.)

Valittu kehityskohde perustellaan tutkimuksen kautta, aineistonkeräämisessä hyödynnetään Google Playn kautta löytyviä mobiilipelejä, joita verrataan kehitettävään kohteeseen. Kehityskohteen rajaamiseen vaikuttaa myös työn laajuus, onko kohde ajallisesti järkevää toteuttaa, varsinaiseen toteutukseen verrataan eri teknologioita ja perustellaan valittu teknologia.

3 Kustomisaatio

3.1 Kustomisaatio yleisesti

Nykypäivänä monet pelit tarjoavat hahmon kustomisointia, ajopeleissä on odotettavaa, että ajoneuvoa voi muokata vaihtamalla puskuria, renkaita tai konepeltiä. Hahmon kustomisoinnissa on usein tarjolla erilaisia vaatekappaleita ja hahmon kehon muotoja voi muokata. Kustomisaatio tuo pelaajille vapauden luoda omannäköisensä hahmon, mutta tämä voi johtaa myös väärinkäyttöön. Esimerkiksi ajopeleissä, jossa voi luoda omia teippauksia, pelaajat voivat luoda tämän avulla ei toivottuja symboleja. Kustomisaation suunnittelu ja toteuttaminen riippuu täysin toteutettavasta pelistä ja pelialustasta, tietokonepeleissä on usein jopa annettu pelaajille mahdollisuus luoda omia resursseja, joita jaetaan yhteisön sisällä. (Rovira 2013.)

Kustomisaatio käsitteenä voidaan myös pilkkoa tarkempiin käsitteisiin. Bycerin (2017) mukaan personalisaatiolla tarkoitetaan pelihahmon visuaalisia muutoksia, jotka eivät vaikuta pelattavuuteen, tämä voi olla esimerkiksi pelihahmon asusteiden vaihtamista. Kustomisaatio ja personalisaatio eroavat toisistaan siten, että kustomisaatiolla vaikutetaan myös pelattavuuteen, esimerkiksi ajoneuvon kiihtyvyyteen. (Bycer 2017.)

3.1.1 Kustomisaatiotilanteet

Hahmon luominen syö tietokoneen laskentaresursseja. Pelit toteuttavat luomisen eri tavoin välttääkseen epämiellyttävää pelikokemusta. Helpoin tapa luoda hahmo on tehdä se latausruudun aikana, koska silloin pelissä ei tapahdu mitään reaaliaikaista. Toinen tapa on luoda hahmo erillisessä kustomointihuoneessa. Tällöin editointi tapahtuu reaaliajassa, mutta resurssit ovat käytettävissä pelkästään hahmon editointia varten, tämä mahdollistaa korkeamman tekstuurilaadun. On myös mahdollista luoda hahmo pelin aikana, tämä on vaikea tapa, sillä kustomointi tapahtuu pelikokemuksen keskellä ja minkäänlaista tökkimistä ei saisi tapahtua. (Rovira 2013.)

3.1.2 Kosmeettiset esineet

Kosmeettisista esineistä on tullut valtava bisnes, vuonna 2022 kosmeettisten esineiden markkinoiden koon arvioitiin olevan noin 50 miljardia dollaria. Yhä useampi yritys tarjoaa kehittämänsä pelin

ilmaiseksi, ja luottavat sen varaan, että pelaajat ostavat muun muassa kosmeettisia esineitä. Hyvänä esimerkkinä tästä on Epic Gamesin julkaisema Fortnite Mobile. Fortnite Mobile on free-to-play mallin mobiilipeli, ja mikrotransaktiot kohdistuvat kosmeettisiin esineisiin. Free-to-play:tä sekä mikrotransaktiota käsitteenä tarkastellaan kappaleessa 3.2.2. Pelaajat käyttivät Fortnite Mobileen 44,3 miljoonaa dollaria huhtikuussa vuonna 2020. (Chapple 2020; Naysmith 2022.)

Puhtaasti kosmeettisten esineiden ideana on pelkästään visuaalinen muutos, pelaaja ei saa siitä mitään lisähyötyä. Tästä huolimatta pelaajat ovat valmiita maksamaan kosmeettisista muutoksista, tätä ilmiötä tarkastellaan kappaleessa 3.2.2 kohdassa ostokäyttäytyminen. Puhtaasti kosmeettinen muutos poistaa kritiikin esimerkiksi pay-to-win väitteistä. Pay-to-win (P2W) tarkoittaa nimensä mukaisesti sitä, että ostamalla pelin sisäisiä tavaroita, pelaaja saa merkittävän lisähyödyn esimerkiksi kilpailullisessa tilanteessa. (Böffel, Würger, Müsseler & Schlittmeier 2022.)

3.2 Kustomisaation tärkeys

3.2.1 Pelaajien uskollisuus

Liao, Cheng ja Teng tutkivat pelihahmon viehättävyyden ja kustomointimahdollisuuksien vaikutusta pelaajien flow:hun ja uskollisuuteen. Flow:lla tarkoitetaan täydellisen keskittymisen kokemusta, joka sisältää myös sisäistä nautintoa. Pelaajille tämä on varmasti tuttu kokemus, ajantajun menettää, kun pääsee flow-tilaan. Tutkimuksessa käytettiin verkkokyselyä, johon saatiin 2095 vastausta. Verkkokyselyissä on riskinä väärien vastausten antaminen tai useita vastauksia samalta käyttäjältä, tuloksista poistettiin toistuvat sähköpostiosoitteet sekä pelit, joita ei ole olemassa. Lisäksi tuloksista poistettiin vastaukset, joissa ilmoitettiin viikoittaiseksi peliajaksi yli 168 tuntia vedoten tarkkaavaisuuden puutteeseen sekä lopuksi poistettiin pelit, joissa on käytössä ensimmäisen persoonan kamera, sillä tutkimus käsittelee pelihahmon visuaalisuutta. Ensimmäisen persoonan peleissä omaa hahmoa ei erityisesti nähdä itse pelin aikana. Karsinnan jälkeen vastauksia jäi jäljelle 1944 kappaletta. (Liao, Cheng & Teng 2019.)

Kyselyn tuloksina saatiin selville seuraavat asiat:

- Hahmoon identifioituminen sekä hahmon viehättävyys vaikuttaa positiivisesti pelaajan uskollisuuteen
- Hahmon kustomisaatio vaikuttaa positiivisesti hahmoon identifioitumiseen

- Hahmoon identifioituminen vaikuttaa positiivisesti flow:hun
- Flow vaikuttaa positiivisesti pelaajan uskollisuuteen
- Hahmon kustomisaatio ei vaikuta suoraan pelaajan uskollisuuteen

Hahmon kustomisaatio ei siis suoraan vaikuta uskollisuuteen, mutta hahmoon identifioituminen vaikuttaa. Kustomisaation vaikutusta pitäisi tarkastella tarkemmin hahmoon identifioitumisen ja flow:n kautta. Tätä yhteyttä pitäisi tarkastella lisätyöllä, jotta johtopäätöksiä voidaan vetää. Tutkimus tukee kuitenkin ideaa, että pelihahmon kustomisointimahdollisuuksilla on merkitystä pelaajan uskollisuuteen. (Liao ym. 2019.)

3.2.2 Monetisaatio

Free-to-play

Free to play (F2P) mallin ideana on tarjota videopeli ilmaiseksi, käyttäjän ei tarvitse maksaa tuotteesta tai tuotteen käytöstä. Mallin tavoitteena on tehdä tuloja pelin sisäisillä ostoilla ja mainoksilla. Ostot voivat tuoda pelaajalle lisähyötyä, esimerkiksi ajopeleissä tehokkaampi moottori tai tavara voi olla täysin kosmeettinen, esimerkiksi erilainen ajoneuvon teippaus. Mallissa hyödynnetään muitakin keinoja, jotta pelaaja palaa peliin takaisin. Yksi usein käytetty keino on päivittäinen palkinto, peliin palaamalla pelaajalle annetaan esimerkiksi palkintoarkku. (Luban 2011.)

Mikrotransaktiot

Mikrotransaktioita esiintyy tyypillisesti free-to-play peleissä. Mikrotransaktiot ovat pelin sisäisiä pieniä maksuja, jolla pelaaja saa esimerkiksi kosmeettisen esineen tai avattavan arkun. Mikrotransaktiota hyödynnetään myös pelin sisäisissä tavaroissa, jotka ovat jokaisen pelaajan saavutettavissa. Esimerkiksi tämä voi olla uuden hahmon avaaminen, jonka voi avata pelin sisäisellä valuutalla. Valuuttaa keräämällä hahmon avaamiseen kuluisi useita pelitunteja, kun taas pienellä mikrotransaktiolla hahmon saa avattua heti. (Colagrossi 2021.)

Ostokäyttäytyminen

Kordyaka ja Hribersek tutkivat pelaajien ostokäyttäytymistä League Of Legends pelissä. Peli sisältää kosmeettisia esineitä, kuten erilaisia hahmojen ulkoasuja sekä erilaisia hahmoja, hahmojen ulkoasuilla saa ainoastaan hahmolle erilaisen ulkonäön. Tutkimuksessa väitettiin muun muassa, että pelaajien itsepresentaatiolla sekä virtuaaliseen ryhmään identifioitumisella on positiivinen vaikutus ostokäyttäytymiseen. Tutkimus käsitteli 209 osallistujan vastauksia, ja näistä ilmeni, että osallistujista 96 prosenttia on ostanut hahmon ulkonäköön vaikuttavan asun ja 6 prosenttia on ostanut hahmon. Tämä tukee tutkimuksen väitettä, että pelaajat ostavat kosmeettisia esineitä itsepresentaation vuoksi. (Kordyaka & Hribersek 2019.)

Tutkimuksen perusteella itsepresentaatio ja identifioituminen vaikuttaa suoraan pelaajien virtuaalisten esineiden ostokäyttäytymiseen. Identifioitumisen merkityksellisyyttä lisäämällä on mahdollista lisätä virtuaaliesineiden ostoa. Tämä voisi olla esimerkiksi pelaajaprofiilien tuomista muuallekin kuin vain peliin liittyvään statistiikkaan. (Kordyaka & Hribersek 2019.)

Kustomisaation kehittämisen näkökulmasta kosmeettisia esineitä on teknisesti helppoa toteuttaa, ja ne soveltuvat hyvin free-to-play malliin. Puhtaasti kosmeettisilla esineillä voidaan myös poistaa pay-to-win kritiikki, sillä esineillä ei saa kilpailullista hyötyä, Kordyakan ja Hribersekin tutkimuksen perusteella pelaajat ovat valmiita ostamaan kosmeettisia esineitä.

4 Kustomisaatio Unityssä

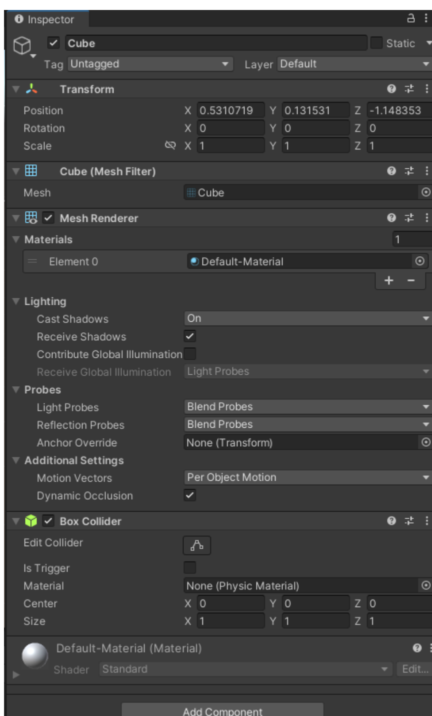
Unity on Unity Technologiesin vuonna 2005 julkaisema monialustainen pelimoottori, jolla voidaan kehittää 2D, 3D sekä virtuaalitodellisuuspelejä. Pelikehitys Unityllä on ilmaista, ja tarvittaessa Unityyn voi ostaa myös maksullisia lisenssejä lisäominaisuuksia varten. Unityllä on tehty valtavasti pelejä, vuonna 2021 kaikista julkaistuista peleistä Unityn osuus oli yli 50 prosenttia. Unityn osuus oli 70 prosenttia tuhannen parhaan mobiilipelien joukossa. Unity tarjoaa nykyään pelimoottorinsa lisäksi muitakin palveluja, kuten palvelimien ylläpitoa. (Schardon 2023; Welcome to Unity n.d.)

4.1 Skriptaaminen

Unityllä kehittäminen tapahtuu skriptien avulla, skripteillä suoritetaan pelissä tapahtuvaa logiikkaa, kuten pelaajan liikuttamista. Luodut skriptit toimivat komponentteina, joita voi lisätä objekteihin. Skriptejä ohjelmoidaan C# kielellä, ja olennaisinta näillä on, että ne perivät MonoBehaviour

luokan. MonoBehaviour on Unityn kehittämä luokka, tämän perivä luokka voi implementoida esimerkiksi Start ja Update funktiot. Start funktiota kutsutaan objektin luomisen yhteydessä, ja Update funktiota kutsutaan jokaisella päivitysruudulla. MonoBehaviour luokka tarjoaa lisäksi paljon muitakin tapahtumatyyppisiä funktioita, kuten fysiikkalaskennoissa FixedUpdate. (Scripting n.d; Creating and Using Scripts n.d.)

Hyvin tyypillisesti skriptien halutaan kommunikoivan keskenään, tähän on olemassa monia tapoja. Yksinkertaisimmillaan voidaan määritellä skriptin sisälle uusi muuttuja, joka toimii referenssinä toiselle skriptille. Muuttujan luominen tapahtuu samalla periaatteella kuin C#-kielellä muutenkin, muuttujalle määritellään näkyvyys, tyyppi ja nimi. Näkyvyys voi olla julkinen tai muuttuja voi sisältää attribuutin SerializeField ja olla yksityinen, jolloin muuttuja on esillä Unityn inspector näkymässä, inspectorin kautta voidaan raahata objekti luotuun muuttujaan, jolloin skriptillä on referenssi toiseen skriptiin. Inspector on näkymä, joka sisältää objektin tiedot sekä siihen kytketyt komponentit (ks. Kuvio 1). (Scripting n.d; Creating and Using Scripts n.d.)



Kuvio 1. Unityn inspector näkymä

Unity tarjoaa GetComponent kutsun, jolla voidaan hakea objektista tietty komponentti. Referenssi voi siis olla pelkästään GameObject-tyyppinen, ja objektin komponentit haetaan GetComponent

kutsulla, mutta kyseisen kutsun suorittaminen syö laskentatehoa. Tätä kutsua ei suositella ajettavan useita kertoja peräjälkeen etenkään silmukoissa, sen sijaan voidaan hakea komponentti ennen silmukkaa ja säilyttää sitä paikallisessa muuttujassa. GameObject referenssillä on kuitenkin myös etunsa, se sisältää kaiken objektiin liittyvän datan, esimerkiksi objektin sijainnin ja rotaation. (GameObject.GetComponent n.d.)

4.2 Visuaalisuus

Skriptaamisessa peliobjektien visuaaleja hallinnoidaan pääasiallisesti komponenttien kautta. MeshRenderer komponentilla hallinnoidaan objektin materiaaleja sekä valotukseen liittyvää dataa ja MeshFilter komponentissa säilytetään verkkorakenne, tämä data sisältää objektin mallinnuksen. Koska kummatkin ovat komponentteja, näistä voi luoda referenssejä muihin skripteihin ja näiden arvoja voidaan muuttaa. Tämä on tarpeen, jottei visuaalisesti erilaisia objekteja tarvitse luoda useita kappaleita, parhaimmillaan riittää yksi objekti, jonka visuaaleja muutetaan tarpeen mukaan, ja objektista voidaan luoda prefab, tästä tarkemmin kappaleessa 4.5. (Mesh Filter component n.d; Mesh Renderer component n.d.)

4.3 Projektio

Ajopelit tarjoavat usein mahdollisuuden luoda oman teippauksen ajoneuvoon. Tämä tuo pelaajalle rajattomat mahdollisuudet luovuuteen, mutta lisää myös väärinkäyttöä. Unityssä tarrojen projektointi onnistuu jokaisella piirtoputkella. Piirtoputkia on tällä hetkellä saatavilla Built-in Render Pipeline, Universal Render Pipeline (URP) sekä High Definition Render Pipeline (HDRP). Piirtoputken valinta riippuu siitä, minkälaista peliä ollaan kehittämässä, URP on paras valinta, jos peli toteutetaan usealle alustalle. (Decals and projectors n.d; Render pipelines introduction n.d.)

Built-in piirtoputkelle on käytettävissä projector komponentti. Projector komponentti projektoi siihen asetetun tekstuurin jokaiseen objektiin, jonka projektori kohtaa, toimintatapa on hyvin lähelle sama kuin oikealla projektorilla. Eri tasoilla voidaan myös erikseen määritellä, jos projektointia ei haluta tietylle tasolle. Tätä voi hyödyntää esimerkiksi renkaissa, projektiota ei välttämättä haluta kohdistuvan näihin. Projector komponentti käyttää light ja multiply shadereita, jotka löytyvät Unityn Standard Assets paketista. Pakettia ei Unityn puolesta ole enää saatavilla, mutta tarvittaessa sen löytää netistä etsimällä. (Projector component n.d.)

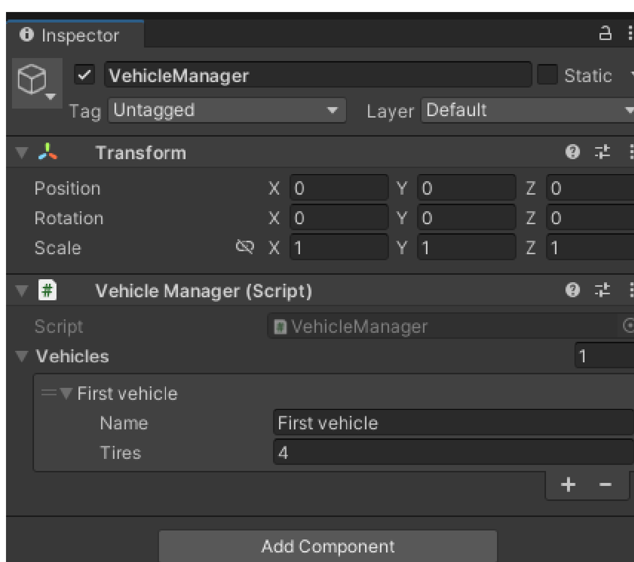
4.4 Omat luokat

Unity ei pakota perimään MonoBehaviour luokkia, käyttäjä voi määritellä myös oman luokkansa. Kehittäjä voi tehdä omia luokkiaan projektin sisälle, ja luoda näistä olioita skriptien sisälle. Omat luokat mahdollistavat olio-ohjelmoinnin tuomat edut, kuten modulaarisuuden. Olio-ohjelmoinnin tavoin luokista voi luoda olioita, joiden parametrit määritellään koodissa, mutta jos luokasta luodut oliot halutaan määritellä Unityn inspector näkymässä, luokalle täytyy määrittää attribuutti System.Serializable. Tämä kertoo Unitylle, että luokan voi serialisoida. Kehittäjä voi esimerkiksi tehdä luokan Vehicle (ks. Kuvio 2), joka määrittelee nimen ja renkaiden lukumäärän. Tämän jälkeen luokkaa voi hyödyntää esimerkiksi VehicleManager luokassa (ks. Kuvio 2), joka perii MonoBehaviour luokan sekä määrittelee listan, jonka tyyppinä on Vehicle. Tämä pitää koodin puhtaana ja mahdollistaa Vehicle luokan käytön esimerkiksi komponentissa (ks. Kuvio 3). (Script serialization n.d; Amlin 2022.)

```
[System.Serializable]
2 references
public class Vehicle
{
    public string Name;
    public int Tires;
}
```

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

Unity Script (1 asset reference) | 0 references
public class VehicleManager : MonoBehaviour
{
    public List<Vehicle> Vehicles = new List<Vehicle>();
}
```

Kuvio 2. Vehicle ja VehicleManager luokat



Kuvio 3. VehicleManager inspector näkymässä

4.5 Resurssit

Unityllä on mahdollista luoda omia resursseja sekä käyttää Unityn omia resursseja. Omien resurssien käyttäminen helpottaa esimerkiksi konfiguraatioiden hallintaa, kehittäjä voi luoda pelinsisäiset tavarat suoraan projektin tiedostoihin ja hallita näiden konfiguraatiota inspector näkymän kautta. Skripteihin voi tämän jälkeen määritellä luodut tavarat esimerkiksi listan sisään tai hakemalla ne suoraan kansiosta. ScriptableObject luokka mahdollistaa datan tallentamisen, ja CreateAssetMenu attribuutti tuo kyseisen luokan Unityn projektinäkymään haettavaksi. (AssetDatabase.CreateAsset n.d; ScriptableObject n.d.)

Peliobjekteistakin on mahdollista luoda uudelleenkäytettävä resurssi, tällöin puhutaan prefabista. Prefab on ennalta määritelty mallikappale, jota voidaan kloonata ja muokata tarpeen mukaan. Pelimaailmassa tämä voi olla esimerkiksi puu, puusta on olemassa yksi prefab, jota luodaan pelimaailmaan useita kappaleita. Peliobjektin muuntaminen prefabiksi on Unityllä hyvin helppoa, peliobjekti raahataan hierarkianäkymästä projektitiedostoihin. Prefab säilyttää kaikki objektiin liitetyt komponentit sekä lapsiobjektit, tyypillinen käyttötarkoitus on ennalta määriteltyjen objektien luominen skenaarioon. Prefabien hyötynä on myös työn tehostaminen, kun prefabia muokataan, se päivittyy kaikkialle missä sitä käytetään. Näin ollen ei tarvitse esimerkiksi tehdä samoja muutoksia useisiin skenaarioihin. Prefabien ei myöskään tarvitse olla identtisiä, sen voi tarvittaessa yliajaa omaksi variantiksi. (Prefabs n.d.)

4.6 Rajapinta

Rajapintoja voi hyödyntää luokkien määrittelyn yhteydessä, se on kokoelma metodien allekirjoituksia ja ominaisuuksia. Rajapintaa voidaan käyttää erilaisille objekteille, joilla on samankaltaisia metodeja. Esimerkiksi ovi ja ikkuna, kummatkin voidaan avata mutta nämä ovat hyvin erilaisia keskenään. Rajapinta mahdollistaa näin ollen saman metodikutsun eri objekteille, silloinkin kun objektit ovat eri luokkaa. (Interfaces 2022.)

Otetaan esimerkiksi kaksi eri moottorityyppiä, toinen on bensiinimoottori ja toinen dieselmoottori. Luokkien luomisessa nämä voi nimetä esimerkiksi CombustionEngine ja DieselEngine. Eri moottoreita halutaan säilöä listassa, joten Unity kehittämisessä voidaan hyödyntää resurssien luomista.

Moottorit perivät ScriptableObject luokan, ja ylle lisätään attribuutti [CreateAssetMenu(menuName = "Engines/Moottorityyppi")]. (AssetDatabase.CreateAsset n.d; ScriptableObject n.d; Interfaces 2022.)

Jos kehittäjä haluaa myöhemmin säilöä erityyppisiä objekteja listassa, voidaan käyttää esimerkiksi Unityn ScriptableObject luokkaa. Listan läpikäymiseen voidaan käyttää silmukkaa, mutta listan elementillä ei voida suoraan kutsua moottorien metodeja. Vaihtoehtona on tehdä listan elementistä tyyppimuunnos haluttuun moottoriluokkaan. Jatkokehityksen kannalta tämä johtaa helposti bugeihin, jos myöhemmin määritellään esimerkiksi ElectricEngine luokka, tyyppimuunnos pitäisi muistaa lisätä kaikkiin moottoriin liittyviin kutsuihin. Tämän vuoksi rajapinnat ovat hyödyllisiä, kehittäjä voi esimerkiksi määrittää IEngine rajapinnan, joka määrittelee StartEngine metodin. Tämän jälkeen IEngine voidaan määrittää kumpaankin luokkaan, ja näin ollen luokkiin on varmistettu StartEngine metodin määritys. Kehittäjä voi nyt luoda silmukan, joka tyyppimuuntaa listan elementin IEngine rajapinnaksi, ja voi kutsua StartEngine metodia. Uusien moottorityyppien luomisessa riittää IEngine rajapinnan implementointi, jolloin luokka toimii sellaisenaan mainitussa silmukassa (ks. Kuvio 4). (Interfaces 2022.)

```
public class EngineManager : MonoBehaviour
{
    public List<ScriptableObject> Engines = new List<ScriptableObject>();

    Unity Message | 0 references
    void Start()
    {
        StartEngines();
    }

    1 reference
    private void StartEngines()
    {
        // Käy läpi moottorit
        for (int i = 0; i < Engines.Count; i++)
        {
            CombustionEngine combustionEngine = Engines[i] as CombustionEngine;
            DieselEngine dieselEngine = Engines[i] as DieselEngine;

            // Jos polttomoottorimuuttuja ei ole tyhjä, käynnistä
            if (combustionEngine)
            {
                combustionEngine.StartEngine();
            }

            // Jos dieselmoottorimuuttuja ei ole tyhjä, käynnistä
            if (dieselEngine)
            {
                dieselEngine.StartEngine();
            }
        }
    }
}
```

```
public class EngineManager : MonoBehaviour
{
    public List<ScriptableObject> Engines = new List<ScriptableObject>();

    Unity Message | 0 references
    void Start()
    {
        StartEngines();
    }

    1 reference
    private void StartEngines()
    {
        // Käy läpi moottorit
        for (int i = 0; i < Engines.Count; i++)
        {
            // Castaa IEngine rajapinnaksi
            IEngine engine = Engines[i] as IEngine;

            // Käynnistä moottori
            engine.StartEngine();
        }
    }
}
```

Kuvio 4. Rajapinnan hyöty silmukassa

4.7 Singleton

Tyypillisesti pelit sisältävät jonkinlaisen GameManagerin pelin hallintaan. GameManager pitää huolta pelin kulusta, esimerkiksi pelin käynnistyksen yhteydessä tapahtuvista asioista. GameManager instansseja ei luoda useita, sillä se voi rikkoa pelilogiikan ja useat managerit taistelevat keskenään pelin kulusta. Tästä syystä käytetään singletoneja, tällä varmistetaan, että luokasta on olemassa vain yksi instanssi kerrallaan. Singletoneja voi hyödyntää missä tahansa luokassa, tyypillisesti erilaisissa managereissa. Kehittäjä voi esimerkiksi luoda ItemManager luokan, johon määritellään pelissä esiintyvät tavarat, ja myöhemmin käyttöliittymää luodessa voidaan kutsua ItemController luokasta esimerkiksi ItemManager.Instance.GetItemWithId(id). ItemController olisi tässä tapauksessa luokka, josta voi olla useita kopioita. Yksinkertaisimmillaan singletonin voi luoda alla olevan kuvion mukaisesti (ks. Kuvio 5). Pelit koostuvat useista skenaarioista, useimmiten peli kulkee päävalikon kautta kenttiin. Singletonit eivät automaattisesti kulje skenaarioiden välillä, Unity tarjoaa tätä varten DontDestroyOnLoad kutsun, nimensä mukaisesti tämä kutsu ei tuhoa objektia skenaarion vaihtuessa. (Thorn 2014, luku 4, GameManager and Singletons.)

```
Unity Script (1 asset reference) | 1 reference
public class ItemManager : MonoBehaviour
{
    // Luodaan staattinen referenssi singletonille, tätä voi kutsua nyt missä tahansa luokassa ItemManager.Instance
    public static ItemManager Instance;

    // Awake event funktiota kutsutaan Unityssä ensimmäisenä, singletonit luodaan tyypillisesti täällä
    Unity Message | 0 references
    private void Awake()
    {
        // Jos singletonia ei ole, määritellään instanssiin tämä luokka
        if (Instance == null)
        {
            Instance = this;
        }
        // Jos singletoni on olemassa, poistetaan tämä luokka jottei duplikaatteja synny
        else
        {
            Destroy(this);
        }
    }
}
```

Kuvio 5. Singleton luokan määrittäminen Unityssä

Singletonin luomisessa voi hyödyntää perintää, jolloin vältetään kirjoittamasta samaa pohjaa yhä uudelleen. Tätä varten luodaan MonoSingleton luokka, joka määritellään abstraktiksi, jotta luokka on tarkoitettu mallipohjaksi. Luokan periytyvyyteen määritellään MonoBehaviour where T : MonoSingleton<T>, näin ollen minkä tahansa luokan voi määrittää MonoSingleton luokaksi (ks. Kuvio 6), DecalManager on tässä erillinen skripti. Toimintaperiaate on täysin sama kuin aiemmin, mutta pitää koodin siistimpänä. Perivä luokka ei kuitenkaan voi tämän jälkeen käyttää Awake funktiota, muutoin instanssin luominen yliajetaan. (Martin 2020.)

```
using UnityEngine;

Unity Script | 2 references
public abstract class MonoSingleton<T> : MonoBehaviour where T : MonoSingleton<T>
{
    private static T _instance;

    0 references
    public static T Instance
    {
        get
        {
            if (_instance == null)
            {
                Debug.Log(typeof(T).ToString() + " is null");
            }
            return _instance;
        }
    }

    Unity Message | 0 references
    private void Awake()
    {
        _instance = this as T;
    }
}
public class DecalManager : MonoSingleton<DecalManager>
{
```

Kuvio 6. MonoSingleton luokan määritys ja käyttö

4.8 Tapahtumat

Skriptien väliseen kommunikaatioon ei aina tarvitse luoda referenssiä tai singletonia, on myös mahdollista luoda tapahtuma. Skriptissä kutsutaan tapahtumaa, ja toinen skripti voi kuunnella kyseistä tapahtumaa. Tapahtumat ovat tärkeitä pelinkulkuun liittyvässä logiikassa, usein käytetään esimerkiksi OnGameStarted, OnGameRestart ja OnGameEnded nimisiä tapahtumia, GameManager voi kutsua esimerkiksi OnGameStarted, ja jokainen tätä tapahtumaa kuunteleva saa tiedon, että peli on aloitettu ja voi toimia sen mukaisesti. (Thorn 2014, luku 3.)

Tapahtumien välillä voidaan myös välittää dataa, tämä voi esimerkiksi olla ostotapahtumissa, tapahtuman mukana viedään päivitetty valuuttamäärä. Yksinkertaisesti tapahtuman voi määritellä alla olevan kuvan mukaisesti (ks. Kuvio 7). ShopManager nimiseen skriptiin määritellään OnCurrencyChanged tapahtuma, jota kutsutaan Invoke funktiolla. Invoke funktiota kutsuttaessa on tärkeää käyttää kysymysmerkkiä, jotta kutsuminen jätetään tekemättä, jos mikään skripti ei kuuntele tapahtumaa. Tapahtuman määrittelyn jälkeen sitä voidaan alkaa kuuntelemaan missä tahansa, tyypillisesti OnEnable ja OnDisable funktioiden yhteydessä. (Onur 2021.)

```
public class ShopManager : MonoSingleton<ShopManager>
{
    public static event Action<int> OnCurrencyChanged;

 OnCurrencyChanged?.Invoke(10);

Unity Message | 0 references
private void OnEnable()
{
    ShopManager.OnCurrencyChanged += CurrencyChanged;
}

Unity Message | 0 references
private void OnDisable()
{
    ShopManager.OnCurrencyChanged -= CurrencyChanged;
}

2 references
private void CurrencyChanged(int newAmount)
{
    Debug.Log("Currency changed: " + newAmount);
}
```

Kuvio 7. Tapahtuman luominen ja käyttäminen

4.9 JSON

JavaScript Object Notation (JSON) on yleisesti käytetty formaatti, joka on helppolukuista ja kevyttä dataa, jota hyödynnetään usein datan siirtoon clientin ja palvelinpuolen kanssa. JSON perustuu objekteihin, joille annetaan nimi ja arvo. Voidaan esimerkiksi määrittää {"testimuuttuja": 5}, vasemmalle puolelle määritetään nimi, ja oikealle arvo. Unity tukee JSON dataa tarjoamalla JsonUtility luokan, jonka avulla voidaan muuttaa dataa JSON:iksi ja takaisin. Takaisin muuntaessa täytyy kuitenkin käyttää luokassa [System.Serializable] attribuuttia ja muuttujatyyppien täytyy olla tuettuja. Luokka johon JSON dataa puretaan, täytyy rakenteeltaankin vastata täsmälleen JSON datan rakennetta. (Introducing JSON n.d; JsonUtility n.d; JsonUtility.FromJson n.d.)

5 Alkukartoitus

5.1 Boom Karts

Boom Karts on Fingersoftin julkaisema free-to-play karting moninpeli, jota Zaibatsu kehittää yhteistyössä Fingersoftin kanssa. Boom Kartsissa kisataan muita pelaajia vastaan, ja pelin edetessä pelaaja saa käyttöönsä uusia ajoneuvoja ja parempia osia. Pelissä on liigoja, joita ylöspäin kivutaan liigakisoissa. (Welcome to Boom Karts 2021.)

Ajoneuvolle ja hahmolle on saatavilla erilaisia kustomisaatiovaihtoehtoja. Lisäksi pelaaja voi valita kisojen aikana näytettäviä hymiöitä. Hahmon muokkaamiseen käytettäviä asusteita saa ilmaiseksi sekä mikrotransaktioilla. Jotkin asusteet on mahdollista avata kertaluonteisessa tapahtumassa, esimerkiksi vuoden 2022 kesätapahtumista pelaaja pystyi saamaan kesäteemaisia asusteita. Uudelle pelaajalle tarjotaan muutamia asusteita eri asustekategorioihin, peliä pelaamalla saadaan palkintoarkkuja ja näistä tulee satunnaisesti asusteita.

Ajoneuvojen parannusosat ovat suurimmaksi osaksi ei-visuaalisia muutoksia. Muutokset kohdistuvat kisoihin, esimerkiksi moottoriparannus lisää huippunopeutta. Visuaalisia muutoksia ovat maalipinnan sekä trailin vaihtaminen, näitä voi saada arkuista sekä Boom passista, trailit ovat takarenkaista tulevia tehosteita (ks. Kuvio 8). Eri ajoneuvoja on kahdeksan, joihin on omat maalipintansa, trailit ovat käytössä jokaisessa ajoneuvossa, joten niitä ei tarvitse avata toiselle ajoneuvolle uudestaan.

Kuvio 8. Ajoneuvoon valittu trail tehoste

5.2 Muiden mobiilipelien kustomisaatiot

Tutkittavaksi otettiin mobiilipelejä, jotka olivat Boom Kartsin kaltaisia, jokainen peli ladattiin ja tarkasteltiin kustomisaatiovaihtoehtoja. KartRider Rush+ mobiilipelissä eri ominaisuuksia avautui myöhemmin, eikä internetistä löytynyt informaatiota, mitä ominaisuus tekee. Muutama ominaisuus jäi näin ollen selvittämättä, sillä se olisi vaatinut kohtuuttoman paljon pelitunteja. Saadut tiedot taulukoitiin (ks. Kuvio 9), huomioitavaa on myös, että Starlit Kart Racing hahmot ja ajoneuvot

olivat kytketty toisiinsa, joten määrät ovat samoja. Lisäksi StarLit Kart Racingissa ajoneuvojen parannusosat oli kytketty profiiliin, eli esimerkiksi kiihtyvyyden parantaminen kohdistui jokaiseen ajoneuvoon.

Boom Karts	Beach buggy racing	Beach buggy racing 2	KartRider: Drift	Mario Kart	Starlit Kart Racing	KartRider: Rush+
Hahmo	**Hahmo**	**Hahmo**	**Hahmo**	**Hahmo**	**Hahmo**	**Hahmo**
Ihonväri	Eri hahmoja (10)	Eri hahmoja (15)	Eri hahmoja (27)	Eri hahmoja (235)	Eri hahmoja (20)	Eri hahmoja (31)
Hiusten väri		Hahmoille eri skinejä	Hahmoille eri skinejä		Hatut	Lentävä lemmikki(8)
Pään muoto			Hahmon animaatiot		Ajoneuvon parannusosat	Lemmikki(20)
Kasvot			Emotet		(ei visuaalisia)	Hatut
Hiukset					(kytketty profiiliin)	
Parrat						
Päähineet						
Hatut						
Maskit						
Kypärät						
Paidat						
Kädet						
Ekstra (esimerkiksi reppu)						
Housut						
Kengät						
Emotet						
Ajoneuvo	**Ajoneuvo**	**Ajoneuvo**	**Ajoneuvo**	**Ajoneuvo**	**Ajoneuvo**	**Ajoneuvo**
Eri ajoneuvot (8)	Eri ajoneuvot (10)	Eri ajoneuvot (60)	Eri ajoneuvot (37)	Eri ajoneuvot (286)	Eri ajoneuvot (20)	Eri ajoneuvot (173)
Maalipinta	Maalipinnan väri	Maalipinnan väri	Maalipinnan väri	Liidokit	Maalipinnan väri	Rekisterikilpi
Trailit	Maalipinnan kuvio	Maalipinnan kuviointi ja kuvioinnin väri	Maalipinnan toinen väri (rungon väri)		Boosteri	Spoileri
Parannusosat (ei visuaalisia)	Maalipinnan kuvion väri	Eturenkaat	Maalipinnan oma teippaus			Driftmoji
	Parannusosat (ei visuaalisia)	Takarenkaat	Renkaat			Parannusosat (ei visuaalisia)
		Jarrujen väri	Boosterit			
		Ajoneuvon koristeet	Rekisterikilvet			
		18 ennaltamäärättyä sijaintia	Parannusosat (ei visuaalisia)			
		Sijainteihin saatavilla erilaisia koristeita	Ilmapallot			
		Parannusosat power uppeihin (ei visuaalisia)				

Kuvio 9. Boom Kartsin ja muiden mobiilipelien kustomisaatiovaihtoehdot

5.3 Kustomisaatioiden vertailu

Taulukkoa tarkastelemalla (ks. Kuvio 9) nähdään, että Boom Kartsin hahmon kustomointimahdollisuudet ovat aivan ylivertaiset verrattuna vastaaviin toteutuksiin. Toistuva teema muissa peleissä oli ennalta määritellyillä hahmoilla pelaaminen, Boom Karts tarjoaa pelaajalle mahdollisuuden luoda omannäköisensä hahmon ja tarjolla on runsaasti erilaisia vaatekappaleita.

Toisaalta Boom Kartsin ajoneuvon kustomoinnissa ei ole hirveästi vaihtoehtoja. Visuaalisen kustomisaation osalta pelaaja voi valita vain maalipinnan sekä trailin. Maalipinnoissa on toki isojakin eroja, erityisesti Boom passista saatavista. Boom pass on noin kuukauden välein vaihtuva kausi, jonka ostamalla pelaaja saa eksklusiivisia esineitä. Boom passin ajoneuvoihin tulevat palkintomaalipinnat ovat usein mallinnettu uniikiksi, kun taas arkuista saatavat ilmaiset maalipinnat ovat enimmäkseen väri- sekä teippausmuutoksia.

Boom Kartsin kaltaisissa peleissä oli huomattavissa ajoneuvojen kattava kustomisaatio. Ainoastaan Mario Kart Tour ei tarjonnut ajoneuvojen visuaalisia muutoksia, pelkästään eri ajoneuvoja ja eri liidokkeja. Toistuvana ominaisuutena oli lisäksi maalipinnan muutokset, väriä ja kuviointia pystyi vaihtamaan ja jopa luomaan oman tarroituksen. Erilaisia ajoneuvoja oli vastaavissa peleissä myös runsaasti, Mario Kart Tour tarjosi jopa 286 eri vaihtoehtoa.

5.4 Kustomointitarpeet

Ajoneuvon syvällisempään kustomointiin oli selkeästi tarvetta. Ajoneuvo on kuitenkin kisojen aikana eniten näkyvillä, itse hahmosta nähdään lähinnä hiukset. Hahmolla on joitakin animaatioita, jotka tuovat sitä enemmän esille, esimerkiksi jos pelaaja osuu ohjuksella toiseen pelaajaan, tällöin näytetään animaatio, jossa hahmo kääntyy katsomaan kameraa kohti iloitsemaan osumaa. Pelaajat näkevät muiden pelaajien hahmot kokonaan vasta kisan päätyttyä palkintokorokkeella. Pelin maalipintojen värit ovat ennalta määrättyjä, joten värin vaihtaminen vaatisi graafikoita ensin muuttamaan jokaisen ajoneuvon eri tekstuurit valkoiseksi, jotta maalipintaan voi lisätä uuden värin, tämä olisi kohtuuttoman suuri työmäärä. Muiden pelien toteutuksissa tuli esille mahdollisuus asettaa omia tarroja ja näin syventää pelikokemusta luomalla persoonallinen teippaus autoon, tämä koettiin ajallisesti järkevänä toteuttaa, toteutuskohteeksi rajautui näin ollen tarrojen toteutus.

5.5 Projektioteknologian valinta

Ensimmäisenä kokeiltiin Unityn built-in renderöintiputkeen kehitettyä projektorikomponenttia tekstuurien projektointiin. Unity ei enää tarjonnut komponentille vaadittuja shadereita, mutta nämä löytyivät netistä helposti. Komponentissa tuli kuitenkin vastaan rajoituksia, se ei tukenut objektin skaalausta. Boom Kartsin ajoneuvoissa käytettiin objektin skaalausta, esimerkiksi hyppyristä laskeutuessa ajoneuvo ”litistyi” hiukan. Tämä oli melko huomaamaton animaatio, mutta visuaalisesti tärkeä. Projektori ei osannut seurata tätä skaalausta, ja lopputuloksena projektiot liikkuivat epätoivotulla tavalla. Tämä ilmiö oli kuin oikeassa projektorissa, jos valkokangasta pienennetään, itse kuva säilyy samankokoisena. Toisena rajoituksena tuli objektin oma väritys, projektoidut tekstuurit eivät olleet puhtaasti omissa väreissään. Objektien tekstuurien täytyisi olla valkoisia, jottei

projektioon kohdistu värimuutoksia. Tämä paketti jätettiin pois, sillä se ei tarjonnut sopivaa ratkaisua ja ongelmien ratkaisu olisi ollut kohtuuttoman työlästä, lisäksi pakettia ei enää Unityn puolelta tuettu.

Llockham-Industriesin luoma Dynamic Decals paketti tarjosi ratkaisut kumpaankin ongelmaan, projektiot olivat puhtaasti omissa väreissään ja projektiot seurasivat ajoneuvon skaalausta, paketti tarjosi metallisen sekä heijastavan pinnan, joka sopi toteutukseen. Projektion koon muuttaminen tapahtui skaalausta muuttamalla, joten se voi olla objektihierarkiassa missä kohdassa tahansa, skaalaus seurasi normaalisti parent objektien skaalaa. Paketissa tuli kuitenkin vastaan ongelmia, paketti toimii oikein hyvin, jos projektissa käytetään Unityn tarjoamaa varjostusta. Boom Kartsissa varjostukset hoidettiin omalla tavalla, jolloin paketin metallinen sekä heijastava shader eivät toimineet odotetusti. Jäljelle jäi shadereita, jotka eivät sopineet toteutukseen, esimerkiksi valaisematon shader. Tämä piirtyi pelissä oikein, mutta ei reagoinut valomuutoksiin, sillä se oli tarkoitettu esimerkiksi graafiseen käyttöliittymään.

Pelissä oli ennenkin hyödynnetty tarrojen asetteluja. Kilparadoista löytyi projektioita, jotka lisäsivät yksityiskohtia, esimerkiksi spraymaalaus seinässä. Samaa projektiokomponenttia kokeiltiin myös ajoneuvoihin, ja se toimi oikein hyvin. Komponentissa oli tarjolla piirtotasojen määrittely, joilla voitiin rajata, mille tasoille projektio tehdään. Tasoina voi esimerkiksi olla ”tie” ja ”ajoneuvo”, tällöin voitiin määrittää komponenttiin ainoastaan ”ajoneuvo” taso. Suorituskyvyn kannaltakin tämä komponentti sopi todella hyvin, projektio tapahtui tarvittaessa vain tarran luomisen yhteydessä ja siitä luotiin verkkorakenne MeshFilter komponenttiin. Tämä ratkaisi kummatkin ongelmat, tarrat pysyivät omissa väreissään ja projektio tuki skaalauksen muutoksia, lisäksi yhteensopivuusongelmia ei jouduttu pohtimaan.

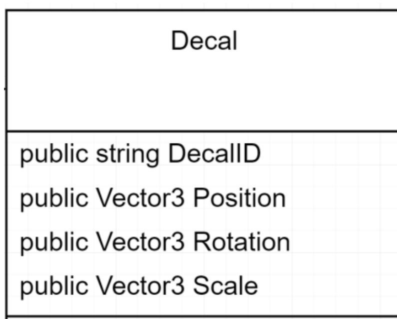
6 Tarrojen toteutus

6.1 Projektion suunnittelu

Toteutukseen valittu teknologia oli Boom Kartsin oma tarrakomponentti. Tämä takasi yhteensopivuuden, sillä komponentti oli ollut pitkään jo pelissä käytössä. Luokkien suunnittelussa täytyi ottaa huomioon, minkälaista dataa palvelinpuolelle lähetetään. Datan tuli olla mahdollisimman pientä ja optimoitua, mitään tarpeetonta ei tule lähettää palvelinpuolelle. Tarrojen asettelussa tuli vastaan

kolme välttämätöntä vektoria: sijainti, rotaatio sekä skaalaus. Rotaatio on Unityn peliobjekteissa kvaterniotyyppisenä, mutta sen voi säilöä eulerin kulmina, jotka ovat ymmärrettävämpiä. Eulerin kulmia tallennettaessa ei tarvinnut tässä tapauksessa huolehtia gimbalin lukosta, sillä rotaatio oli alun perin kvaterniomuodossa. Myöhemmässä vaiheessa tarralle määriteltiin myös peilaaminen, tästä tarkemmin kappaleessa 6.6.

Vektoreiden lisäksi tarra täytyi jollakin muuttujalla identifioida. Tämän voi tehdä muun muassa merkkijono- tai kokonaislukutyyppisenä muuttujana, merkkijonotyyppinen on käyttäjäystävällinen, sillä se kertoo suoraan mikä tarra on kyseessä. Datan koko kuitenkin kasvaa merkkijonoa käytettäessä, mutta tätä ei koettu ongelmaksi tässä kohtaa. Muuttujien määrittelyn jälkeen luotiin UML-kaavio selventämään toteutusta (ks. Kuvio 10). Tähän luokkaan ei säilötty tarraan liittyviä materiaaleja tai tekstuureja, sillä ne eivät olisi olleet tuettuja JSON formaatissa. Nämä hoidettiin toisella luokalla, tunnisteella haettiin tarran visuaaliset elementit.



Kuvio 10. UML-kaavio tarran muuttujista

Tarran kohdistaminen ajoneuvoon vaati skriptin, tätä varten luotiin DecalMouseController, joka peri MonoBehaviour luokan. Tarroja ei muokattu muualla kuin päävalikossa, näin ollen tarran kohdistamisessa ei ollut tarvetta skenaarioiden välillä kulkevaan komponenttiin. Komponentin tehtävänä oli laskea UML-kaaviossa mainitut vektorit (ks. Kuvio 10), havaita tarran vaihtaminen ja peilaus sekä lähettää näitä tietoja eteenpäin. Lisäksi komponentissa säilöttiin tarran muokkaamiseen liittyviä muuttujia, kuten skaalausrajoituksia.

6.2 Projektion kohdistaminen

Projektio täytyi saada osoittamaan ajoneuvoon käyttäjän haluamalle paikalle. Raycast tarjosi ratkaisun tähän, tällä voitiin lähettää kamerasta säde, jonka lähtöpisteenä oli hiiren sijainti ruudulla. Tietokoneella ja mobiililaitteella hiiren ja kosketuksen sijaintitiedot toimivat samalla tavalla, joka helpotti myös testaamista. Objektissa täytyy olla törmäysalue, jotta säteen törmäys rekisteröidään, esimerkiksi BoxCollider tai MeshCollider komponentti, monimutkaisiin objekteihin MeshCollider on yleispätevä komponentti. Raycast palauttaa törmäyksen jälkeen törmäysdataa RaycastHit-tyyppisenä, joista olennaisimpia ovat törmäyksen sijainti maailman koordinaatistossa sekä törmäysobjektin pinnasta poispäin osoittava normaalivektori. Tarran täytyi osoittaa pintaa kohti, joten normaalivektorin kääntäminen onnistui negaatiolla, eli lisäämällä miinusmerkki vektorin eteen.

Tarrassa täytyi olla lisäksi käyttäjän määrittämä lisärotaatio, joten lopullinen rotaatio laskettiin seuraavan kuvion mukaisesti (ks. Kuvio 11). Unityn tarjoamalla LookRotation funktiolla voitiin kohdistaa projektio annettua suuntaa kohti, tässä tapauksessa pintaa kohti. Tämä kerrottiin käyttäjän määrittämällä Z-akselin rotaatiolla, jolloin saatiin kaksi kvaterniota yhdistettyä. Laskettu rotaatio muunnettiin decalRoot objektin paikalliseen koordinaatistoon, tästä tarkemmin luvussa 6.4. DecalRoot objekti tulee esiintymään useasti työn aikana, decalRoot objekti luotiin jokaisen ajoneuvon sisälle tyhjänä objektina, jolloin se sisälsi vain Transform komponentin oletusarvoilla.

```
Quaternion normalRotation = Quaternion.LookRotation(-hitNormal);
Quaternion rotation = normalRotation * Quaternion.Euler(0, 0, angle);
Quaternion localSpaceRotation = Quaternion.Inverse(Kart.decalRoot.rotation) * rotation;
```

Kuvio 11. Rotaation asettaminen objektiin

Tarralle täytyi määrittää myös syvyys, ja se ei voinut olla ennalta määritelty. Jos syvyys olisi esimerkiksi aina yhden metrin, projektio voi jakautua osiin merkittävillä syvyyseroilla. Tämä voi tapahtua projektion osuessa puoliksi ajoneuvon oveen, toinen puoli voi osua vastakkaiseen oveen ja loput projektiosta piirtyy sinne. Raycast ei tarjonnut poistumispistettä, jonka avulla syvyyden laskeminen olisi onnistunut suoraan. Tätä varten luotiin toinen raycast, joka laski poistumispisteen (ks. Kuvio 12). Ensimmäisen törmäyspisteen jälkeen ajettiin silmukkaa sata kertaa, jonka sisällä haettiin uutta

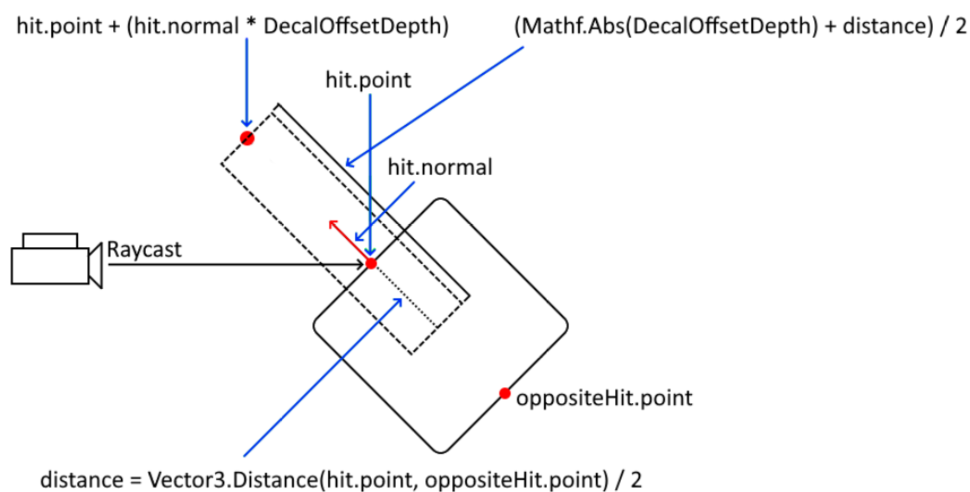
törmäyspistettä. Sata iteraatiota oli lähinnä arvio, jonka jälkeen poistumispisteen etsiminen olisi turhaa, tällöin syvyydeksi asetettiin määritelty maksimisyvyys.

```
private Vector3 MoveDecal()
{
    Ray ray = GarageController.Instance.garageCamera.ScreenPointToRay(Input.mousePosition);

    if (Physics.Raycast(ray, out RaycastHit hit, Mathf.Infinity))
    {
        if (!hit.transform.GetComponent<KartShaker>())
        {
            return Vector3.zero;
        }

        hitNormal = hit.normal;

        RaycastHit oppositeHit = new RaycastHit();

        Vector3 rayDirection = hit.normal;
        Vector3 rayStart = hit.point;

        for (int i = 0; i < depthCalculationIterations; i++)
        {
            if (Physics.Raycast(rayStart, rayDirection, out RaycastHit exitPoint, 1f))
            {
                if (exitPoint.transform.GetComponent<KartShaker>())
                {
                    oppositeHit = exitPoint;
                    break;
                }
            }
            rayStart += -rayDirection * 0.01f;
        }

        if (oppositeHit.point != null)
        {
            float distance = Vector3.Distance(hit.point, oppositeHit.point) / 2;
            scaleDepth = (Mathf.Abs(decalOffsetDepth) + distance) / 2;
            scaleDepth += 0.05f;
        }
        else
        {
            scaleDepth = maxScaleDepth;
        }

        earlierPos = hit.point + (hit.normal * decalOffsetDepth);

        return earlierPos;
    }
    return Vector3.zero;
}
```

Kuvio 12. Törmäyksen sijainnnin ja syvyyden laskeminen

Syvyyslaskennassa (ks. Kuvio 12) uuden törmäyspisteen lähtökohtana on ensimmäinen törmäyspiste, ja suuntana törmäyspinnan normaali, eli pinnasta ulospäin. Jokaisella silmukan iteraatiolla tarkistetaan, törmääkö uusi raycast ajoneuvoon, jos ei, siirretään lähtöpistettä 0.01 yksikköä ensimmäisen törmäyksen pinnan normaalista taaksepäin. Tämä johtaa lopulta siihen, että raycast törmää objektiin uudelleen, jolloin törmäyspiste on myös samalla poistumispiste. Tällöin silmukka päätetään, ja lasketaan pisteiden välinen etäisyys, tämä kertoo objektin syvyyden alkuperäisestä törmäyspisteestä poistumispisteeseen. Raycast ei oletuksena rekisteröi objektin sisältä tapahtuvaa törmäystä, jonka vuoksi tämä ratkaisu toimi, tämän ominaisuuden voi tarvittaessa kytkeä projektin asetuksista päälle. Funktio palauttaa projektiolle sijainnin, joka on määritetyn etäisyyden päässä törmäyspisteestä maailman koordinaatistossa.

Käytetyn tarrakomponentin skaalauksessa havaittiin, että 0.5 yksikön skaalaus vastasi 1 yksikön skaalausta pelimaailmassa, näin ollen syvyyden laskennassa tämä täytyi ottaa huomioon (ks. Kuvio 13), kuvion siniset nuolet ovat käsitteiden selkeyttämistä varten, ne eivät liity laskentaan. Etäisyydeksi laskettiin ensimmäisen törmäyksen ja toisen törmäyksen sijaintien välinen etäisyys jaettuna kahdella, jolloin saatiin objektin syvyys puoleen väliin. Piirtoaluetta ei haluttu poistumispisteen kohdalle, sillä se olisi saattanut aiheuttaa epätoivottuja projektioita tietyissä kulmissa. Määritelty piirtoetäisyys ja syvyys summattiin yhteen, ja skaalausero huomioiden näiden summa jaettiin kahdella, jolloin saatiin laskettua lopullinen syvyys, joka ulottuu määritellystä lähtöpisteestä törmäysobjektin syvyyden puoleen väliin. Syvyyteen lisättiin lopuksi pieni hienosäätö, jolla varmistettiin, että syvyys oli ohuessakin objektissa riittävä. Lopullinen syvyys ei vaikuttanut tarran lähtöpisteeseen, syvyydellä määriteltiin ainoastaan tarran piirtoalue, joka on kuviossa (ks. Kuvio 13) katkoviivasuorakulmio.



Kuvio 13. Syvyyden laskenta

6.3 Projektion skaalaus ja manuaalinen rotaatio

Tarran pyörittäminen ja skaalaus toteutettiin mobiililaitteille ominaisella tavalla, kahdella kosketuksella. DecalMouseController skriptin Update funktiossa tarkastettiin, jos käyttäjä koskettaa ruutua kahdella sormella, tällöin kutsuttiin pyörittämiseen ja skaalaukseen käytettyä funktiota (ks. Kuvio 14). Funktiossa tallennettiin alkuperäiset käyttäjän määrittämät arvot ja kosketusten lähtösijainnit sekä kytkettiin pyörittämisen ja skaalauksen totuusmuuttuja päälle, jolloin seuraavalla kutsukerralla laskettiin kosketusten kulma- ja etäisyysmuutokset alkuperäisiin vertaamalla.

Tämä jatkui niin pitkään, kunnes kosketuksia havaittiin yksi tai vähemmän, tällöin asetettiin totuusmuuttuja pois päältä.

Kosketuksilla oli ruudun X- ja Y-koordinaatit säilöttynä, jolloin nämä voitiin erottaa toisistaan. Atan2 funktiolla voitiin laskea X-akselin välinen kulma määriteltyyn vektoriin nähden ja Rad2Deg vakioarvolla voitiin muuntaa saatu tulos radiaaneista asteiksi. Tämä suoritettiin alkuperäisille kosketuksille ja uusille kosketuksille, ja erottamalla nämä toisistaan, saatiin uusi käyttäjän määrittämä kulma. Kahden kosketuksen päättyessä kuitenkin aiemmin määritelty kulma unohtuu, joten tätä varten erotettiin määritelty kulma aiemmin määritellyllä kulmalla, jolloin pyörittämisen välissä kulma ei unohtunut. Skaalaukseen hyödynnettiin samaa rakennetta, erona oli laskukaava, joka laski kosketusten väliset etäisyydet ja näiden eron. Lopullinen etäisyysero jaettiin tuhannella, jotta skaalaaminen oli luontevampaa.

```
private void RotateAndScaleDecal()
{
    Touch touch1 = Input.GetTouch(0);
    Touch touch2 = Input.GetTouch(1);
    if (!rotatingOrScaling)
    {
        origTouch1Pos = touch1.position;
        origTouch2Pos = touch2.position;
        origAngle = angle;
        origScale = new Vector3(scale.x, scale.y, scaleDepth);
        rotatingOrScaling = true;
    }
    else
    {
        Vector2 touch1Pos = touch1.position;
        Vector2 touch2Pos = touch2.position;

        if (touch1Pos == Vector2.zero || touch2Pos == Vector2.zero)
        {
            return;
        }

        float originalAngle = Mathf.Atan2(origTouch2Pos.y - origTouch1Pos.y, origTouch2Pos.x - origTouch1Pos.x) * Mathf.Rad2Deg;
        float newAngle = Mathf.Atan2(touch2Pos.y - touch1Pos.y, touch2Pos.x - touch1Pos.x) * Mathf.Rad2Deg;
        float angleDifference = originalAngle - newAngle;

        if (angleDifference != 0)
        {
            float finalAngle = origAngle - angleDifference;
            angle = finalAngle;
        }

        float origDistance = Vector2.Distance(origTouch1Pos, origTouch2Pos);
        float distance = Vector2.Distance(touch1Pos, touch2Pos);
        float distanceDifference = origDistance - distance;

        if (distanceDifference != 0)
        {
            distanceDifference = distanceDifference / 1000.0f;
            float xAxis = origScale.x - distanceDifference;
            float yAxis = origScale.y - distanceDifference;
            scale = new Vector2(xAxis, yAxis);
        }
    }
}
```

Kuvio 14. Rotaation ja skaalauksen laskeminen

6.4 Maailman koordinaatistosta paikalliseen muuntaminen

Tarran sijainnin tallentaminen ei onnistunut suoraan sellaisenaan. Lopullinen sijainninlaskentakaava oli kuvion mukainen (ks. Kuvio 15). Kaavan avulla liikuttiin törmäyspisteestä pinnan normaalivektorin suuntaan määritellyn etäisyyden verran. Tämä oli pakollinen tieto, sillä muuten sijainti olisi täsmälleen pinnassa kiinni, ja projektoitu tarra ei ”kääriytyisi” esimerkiksi kuperaan pintaan. Etäisyydeksi asetettiin 0.5, tällöin lopullinen sijainti oli törmäyspisteestä 0.5 yksikköä pinnasta ulospäin. Tämä oli kuitenkin maailman koordinaatistossa, sijainti ei sovellu useampaan skenaarioon, sillä ajoneuvot luodaan eri kohtiin maailmassa. Tätä varten Unity tarjosi InverseTransformPoint funktion Transform luokassa (ks. Kuvio 15). Funktiota kutsuttiin lasketulla sijainnilla, kutsu tapahtui halutun objektin Transform luokan kautta, jolloin maailman koordinaatistossa oleva sijainti muunnettiin paikallisen objektin koordinaatistoon. Saatu sijainti voitiin nyt tallentaa, ja se soveltui useampaan skenaarioon.

```
    position = hit.point + (hit.normal * decalOffsetDepth);

NormalizeScales();
Quaternion normalRotation = Quaternion.LookRotation(-hitNormal);
Quaternion rotation = normalRotation * Quaternion.Euler(0, 0, angle);
DecalManager.Instance.SetDecalTransform(
    kart.kartID, Kart.decalRoot.InverseTransformPoint(position),
    Quaternion.Inverse(Kart.decalRoot.rotation) * rotation,
    new Vector3(scale.x, scale.y, scaleDepth),
    mirrored);
RevertScalesAndRebuildDecals();
```

Kuvio 15. Sijainnin ja rotaation muuntaminen paikalliseen koordinaatiojärjestelmään

Rotaatiossa oli sama ongelma, laskettu rotaatio oli maailman koordinaatistossa. Quaternion luokka tarjosi tätä varten Inverse funktion (ks. Kuvio 15), joka palautti rotaation käänteisenä. Inverse funktioon syötettiin decalRoot objektin rotaatio ja saatu tulos kerrottiin käyttäjän määrittämällä rotaatiolla. Saatu lopputulos oli rotaatio, joka oli decalRoot objektin paikallisessa koordinaatistossa, ja tämä rotaatio soveltui useampaan skenaarioon.

Skaalauksessa ei ollut tarvetta muuntaa lokaaliin koordinaatistoon, mutta ennen sijainnin ja rotaation muunnoksia decalRoot ja decalRootin parent objektin skaalaukset muutettiin 1,1,1 muotoon, jotta tarrat ovat helppokäyttöisessä koordinaatiojärjestelmässä. Tarran tallentamisen jälkeen asetettiin muunnetut objektit takaisin alkuperäisiin skaalauksiin.

6.5 Käyttäjän syötteiden havaitseminen

DecalMouseController skriptissä tehtiin Update funktion sisään tarkistuslogiikka (ks. Kuvio 17). Käyttäjän painalluksia seurattiin tarkistamalla yhtä tai kahta kosketusta, editorissa hyödynnettiin hiiren painallusten tarkistusta. Käyttäjän syötteessä hyödynnettiin korutiineja (ks. Kuvio 16), sillä kahden kosketuksen kohdalla molemmat sormet eivät kosketa yhtäaikaisesti ruutua, jolloin koodi voi päätyä yhden kosketuksen logiikkaan; lopputulos olisi epätoivottu. Korutiinilla voidaan pysäyttää koodin suorittamista tietyksi ajaksi, joten yhden kosketuksen aktivointiin lisättiin korutiini, joka käänsi tarran liikuttamiseen käytetyn totuusmuuttujan päälle 25 millisekunnin yhtäjaksoisen painalluksen jälkeen.

```
1 reference
private IEnumerator ResetRotatingOrScaling()
{
    yield return new WaitForSeconds(0.1f);
    rotatingOrScaling = false;
}

private IEnumerator SetDecalMoveToTrue()
{
    yield return new WaitForSeconds(0.025f);
    allowDecalMove = true;
}
```

Kuvio 16. Käyttäjän syötteessä käytetyt korutiinit

Kahden kosketuksen syötteeseen ei erikseen vaadittu aktivointia, mutta kosketusten päättyessä estettiin tarran liikuttaminen 100 millisekunnin ajan. Tämän tarkoituksena oli välttää tarran liikahtamista, sillä käyttäjä saattoi koskettaa ajoneuvoa pyörittämisen ja skaalauksen päätyttyä. Koodiin lisättiin lopuksi totuusmuuttuja (ks. Kuvio 17) tallentamaan tarran muutokset vain, jos muutoksia tehtiin. Tarran sijaintia säilöttiin earlierPos muuttujassa, jotta tarraa pyörittäessä sijainti säilyi ennallaan, earlierPos määriteltiin MoveDecal funktion yhteydessä (ks. Kuvio 12).

```
#if UNITY_EDITOR
            if (Input.GetMouseButton(0))
#else
            if (Input.touchCount == 1)
#endif
            {
                decalMoveCoroutine = StartCoroutine(SetDecalMoveToTrue());
                if (!rotatingOrScaling && allowDecalMove)
                {
                    position = MoveDecal();
                    if (position == Vector3.zero)
                    {
                        position = earlierPos;
                    }
                    else
                    {
                        didChanges = true;
                    }
                }
            }
            else if (Input.touchCount == 2)
            {
                if (decalMoveCoroutine != null)
                {
                    StopCoroutine(decalMoveCoroutine);
                }
                allowDecalMove = false;
                position = earlierPos;
                RotateAndScaleDecal();
                didChanges = true;
            }
            else
            {
                if (rotatingOrScaling)
                {
                    decalRotateCoroutine = StartCoroutine(ResetRotatingOrScaling());
                }
                else
                {
                    if (decalRotateCoroutine != null)
                    {
                        StopCoroutine(decalRotateCoroutine);
                    }
                }
                if (decalMoveCoroutine != null)
                {
                    StopCoroutine(decalMoveCoroutine);
                }

                allowDecalMove = false;
            }
            if (didChanges)
            {
                NormalizeScales();
                Quaternion normalRotation = Quaternion.LookRotation(-hitNormal);
                Quaternion rotation = normalRotation * Quaternion.Euler(0, 0, angle);

                DecalManager.Instance.SetDecalTransform(
                    kart.kartID, Kart.decalRoot.InverseTransformPoint(position),
                    Quaternion.Inverse(Kart.decalRoot.rotation) * rotation,
                    new Vector3(scale.x, scale.y, scaleDepth),
                    mirrored);
                RevertScalesAndRebuildDecals();
            }
            didChanges = false;
        }
    }
```

Kuvio 17. Projektion kohdistaminen Update funktiossa

6.6 Tarran peilaaminen

Tarran peilaaminen oli varsin yksinkertaista toteuttaa. Peilaamisen tavoitteena oli peilata asetettu tarra myös ajoneuvon toiselle puolelle, jos käyttäjä näin valitsi. Tarraluokkaan lisättiin peilaukselle totuusmuuttuja, joka tarkistettiin tarrojen luomisen yhteydessä. Peilaamisen kytkentä hoidettiin DecalMouseController skriptin kautta, totuusarvo syötettiin tarran tallentamisen yhteydessä (ks. Kuvio 17). Ennen tarrojen luomista luotiin nonMirroredDecalRoot ja mirroredDecalRoot objektit, jotka kloonattiin decalRoot objektista. Tarrat lisättiin nonMirroredDecalRoot objektin alle, ja jos

tarran peilaaminen oli päällä, luotiin toinen tarra samalla iteraatiolla myös mirroredDecalRoot objektin alle. Tarrojen luomisen jälkeen mirroredDecalRoot objektin ja tarrakomponentin skaalauksen X-akseli käännettiin negatiiviseksi. Peilattu tarra projektoitiin vasta peilauksen jälkeen, sillä ajoneuvojen mallinnukset saattoivat olla epäsymmetrisiä. Tarrakomponentissa oli lisäksi offset muuttuja, jolla säädettiin tarraa projektoinnin jälkeen ulospäin pinnasta, peilatun tarran offset arvo säädettiin hiukan kauemmas, jotta päällekkäiset tarrat eivät vilku. Jos tarrat olisivat täsmälleen samalla tasolla, ne ikään kuin tappelisivat keskenään, kumpi piirtyy ensin. Lopuksi nonMirroredDecalRoot ja mirroredDecalRootin alta siirrettiin tarrat takaisin decalRoot objektin alle yhdistämistä varten, tästä tarkemmin kappaleessa 6.10.

6.7 JSON data

Tarrat täytyi säilöä JSON muodossa lokaalisti laitteeseen, ja palvelinpuoleen lähetettiin myös samaa dataa, karsien epäolennaiset tiedot pois. Jokaiselle ajoneuvolle tuli oma tarrojen asettelu, sillä jokainen tarra oli laskettu valitun ajoneuvon pintaan. Lokaalissa tallennuksessa voitiin hyödyntää sanakirjatyyppiä, jonka avaimena oli ajoneuvon tunniste ja arvona lista tarroja. Sanakirjan käyttö oli optimoitua, sillä tiedon hakeminen onnistui suoraan avaimella, hakuun ei tarvinnut esimerkiksi tehdä erillistä silmukkaa, joka käy listan pahimmassa tapauksessa kokonaan läpi.

Vektoreiden tarkkuudessa tuli vastaan ongelmia. JsonUtility teki liukuluvusta epätarkan, ja saattoi muuntaa esimerkiksi 0.19f arvon muotoon 0.18999f, desimaaleja tuli tarpeettoman paljon. Tiedon optimoimiseksi luotiin struktuuri (ks. Kuvio 18), joka säilöi vektorin arvot kokonaislukuna. Unityn Vector3 muunnettiin konstruktorissa TwoDecimalIntVector3-tyyppiseksi, ensin kertomalla vektori sadalla. Jokainen vektorin komponentti pyöristettiin tämän jälkeen kahden desimaalin tarkkuuteen, ja muutettiin kokonaislukutyyppiseksi. Näin ollen esimerkiksi 0.173412 liukulukuarvo muunnettiin 17 kokonaislukuarvoksi. Struktuuriin luotiin lisäksi staattinen TwoDecimalIntVector3ToVector3 funktio, joka muunsi TwoDecimalIntVector3-tyypin takaisin Vector3-tyypiksi, takaisin muuntaessa riitti sadalla jakaminen. Tämä funktio esti virhelaskentojen tekemisen, sillä jokainen takaisinmuunnos tehtiin samalla funktiolla. Funktio määriteltiin staattiseksi, jotta sitä voitiin kutsua luomatta TwoDecimalIntVector3-tyyppistä muuttujaa.

```
public struct TwoDecimalIntVector3
{
    public int x;
    public int y;
    public int z;
    3 references
    public TwoDecimalIntVector3(Vector3 input)
    {
        input *= 100.0f;

        x = Mathf.RoundToInt((float)Mathf.Round(input.x * 100.0f / 100.0f));
        y = Mathf.RoundToInt((float)Mathf.Round(input.y * 100.0f / 100.0f));
        z = Mathf.RoundToInt((float)Mathf.Round(input.z * 100.0f / 100.0f));
    }

    6 references
    public static Vector3 TwoDecimalIntVector3ToVector3(TwoDecimalIntVector3 input)
    {
        Vector3 vector3 = new Vector3
        {
            x = input.x / 100.0f,
            y = input.y / 100.0f,
            z = input.z / 100.0f
        };
        return vector3;
    }
}
```

Kuvio 18. JSON datan optimointiin käytetty struktuuri

JSON datan tallentamisessa ja lataamisessa voitiin hyödyntää JsonUtility luokkaa. Luokan hyödyntäminen käy yksinkertaisimmillaan FromJson ja ToJson funktioilla, nämä muuntavat JSON tuetun luokan JSON dataksi ja takaisin (ks. Kuvio 19). Tallentamisessa muunnettiin tarrasanakirja KartDecalSaveWrapper luokaksi, joka muunnettiin JSON muotoon JsonUtilityllä. Saatu merkkijonotyyppinen data voitiin tallentaa PlayerPrefs luokkaan, käytetty salausluokka on jätetty kuviosta pois ja korvattu PlayerPrefs luokalla. (ks. Kuvio 19). Tallennettu data voitiin hakea laitteen käynnistyksen yhteydessä latausfunktiolla, joka haki laitteelta merkkijonotyyppisen datan, muunsi sen JSON formaattiin ja loi sanakirjan takaisin. CheckKartSave funktio tarkisti, onko sanakirjassa tarvittavaa avainta, ja tarvittaessa loi sen ja LimitListSize funktio pilkkoi listan sallittuun kokoon, jos tarralistan koko on syystä tai toisesta korkeampi kuin sallittu.

```
1 reference
private void CreateJsonAndSave()
{
    if (DecalSaves.Count == 0) return;
    int index = 0;
    KartDecalSave[] kartDecalSaves = new KartDecalSave[DecalSaves.Count];
    KartDecalSaveWrapper kartDecalSaveWrapper = new KartDecalSaveWrapper();

    foreach (KeyValuePair<string, List<Decal>> kartDecalSave in DecalSaves)
    {
        Decal[] array = ConvertDecalListToArray(kartDecalSave.Value);
        kartDecalSaves[index] = new KartDecalSave() { KartId = kartDecalSave.Key, decals = array };
        index++;
    }
    kartDecalSaveWrapper.saves = kartDecalSaves;
    string jsonString = JsonUtility.ToJson(kartDecalSaveWrapper, false);
    if (jsonString != null)
    {
        PlayerPrefs.SetString("decals", jsonString);
    }
}

1 reference
private void LoadDecalFromDevice()
{
    string data = PlayerPrefs.GetString("decals", string.Empty);
    if (data == string.Empty) return;
    KartDecalSaveWrapper jsonKartDecalSaveArray = JsonUtility.FromJson<KartDecalSaveWrapper>(data);
    foreach (KartDecalSave save in jsonKartDecalSaveArray?.saves)
    {
        CheckKartSave(save.KartId);
        DecalSaves[save.KartId] = LimitListSize(new List<Decal>(save.decals), allowedDecalCount);
    }
}
```

Kuvio 19. JSON datan tallentaminen ja lataaminen

Palvelinpuolen kuormituksen vähentämiseksi hyödynnettiin eri luokkia sekä luokkien perintää (ks. Kuvio 20), mobiililaitteelle tallennettava data oli KartDecalSaveWrapper muodossa, joka sisälsi jokaisen ajoneuvon tallennetut tarrat. Palvelinpuolelle ei tarvinnut lähettää kuin DecalSaveWrapper muotoista dataa, olisi ollut täysin tarpeetonta lähettää pelaajan muidenkin ajoneuvojen tarrat, sillä ajoneuvoa ei voi kisan aikana vaihtaa. Myöskään ajoneuvon merkkijonotyyppistä tunnistetta ei lähetetty, sillä se ei vaikuttanut tarrojen luomiseen.

```
[System.Serializable]
4 references
public class KartDecalSaveWrapper
{
    public KartDecalSave[] saves;
}

[System.Serializable]
4 references
public class DecalSaveWrapper
{
    public Decal[] decals;
}

[System.Serializable]
8 references
public class KartDecalSave : DecalSaveWrapper
{
    public string KartId;
}

[System.Serializable]
30 references
public class Decal
{
    public string DecalId;
    public TwoDecimalIntVector3 Position;
    public TwoDecimalIntVector3 Rotation;
    public TwoDecimalIntVector3 Scale;
    public bool Mirrored;

    1 reference
    public Decal()
    { }

    1 reference
    public Decal(Decal decal)
    {
        DecalId = decal.DecalId;
        Position = decal.Position;
        Rotation = decal.Rotation;
        Scale = decal.Scale;
        Mirrored = decal.Mirrored;
    }
}
```

Kuvio 20. JSON dataan käytetyt luokat

6.8 JSON datan salaus

JSON datan täytyi olla salattua, sillä muuten pelaajat olisivat voineet halutessaan etsiä laitteestaan tallennustiedoston, ja manipuloida sen arvoja. Vektoreiden muutokset eivät olisi haitanneet, sillä niillä ei voi saada mitään varsinaista hyötyä. Tunnisteen muokkaaminen olisi kuitenkin ollut epätoivottua, sillä sitä muokkaamalla olisi voinut käyttää tarraa, jota ei omista.

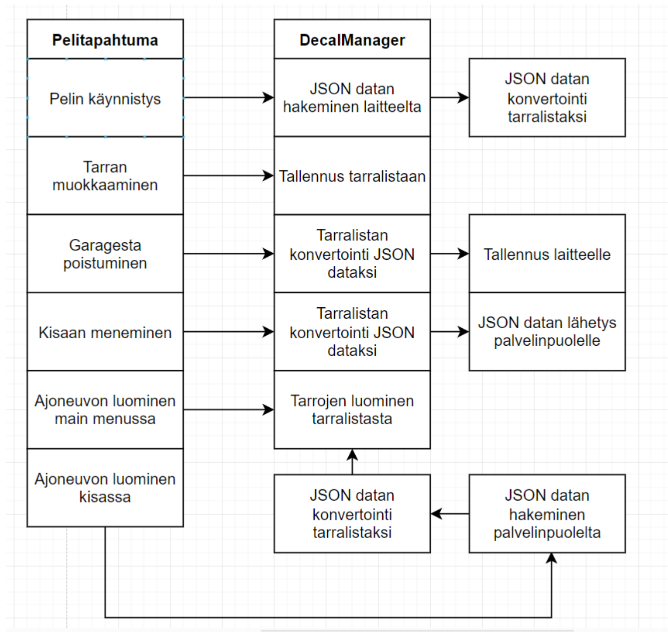
JSON datan salaamiseen hyödynnettiin ennestään käytettyä luokkaa, joka salasi syötetyt datat, vastaavassa toteutuksessa voitaisiin hyödyntää PlayerPrefs luokkaa. PlayerPrefs on Unityn tarjoama luokka, joka tallentaa tietoa laitteen rekisteriin suojaamattomana, joten sitä ei suositella käytettävän arkaluonteisen tiedon tallentamiseen (PlayerPrefs n.d). Kumpaankin luokkaan voitiin tallentaa muun muassa merkkijonotyyppistä dataa, JsonUtility tuotti JSON datan merkkijonona,

joten se voitiin tallentaa sellaisenaan. Näin ollen ei tarvittu erillisiä tiedostoja tai erillisiä tietokantoja, kun datan purkaminen tapahtui clientin puolella, palvelinpuoli vastaanotti ainoastaan merkkijonotyyppistä dataa. Tämä ratkaisu johtaa kuitenkin tietojen menetykseen, jos pelin poistaa, mutta sitä ei koettu tässä kohtaa ongelmaksi, sillä tarrojen omistukseen se ei vaikuttanut. Merkkijonotyyppisen tiedon salaamiseen olisi voinut myös käyttää AES-salausta, joka tekee JSON datasta luku- ja muokkaamiskelvotonta, ja tallentaa tämän datan PlayerPrefs luokkaan. AES-salauksesta Dan Cox on kirjoittanut hyvän esimerkin, joka soveltui JSON datan salaamiseen Unityssä, tätä ohjetta kokeiltiin salaamisessa, ennen kuin siirryttiin ennestään käytettyyn luokkaan (Cox 2021).

6.9 Singletonin toteuttaminen

Suunnitteluvaiheessa oli selvää, että tarrojen hallintaan liittyvää koodia tulee olemaan paljon. Lisäksi samoja funktioita täytyi kutsua päävalikossa sekä kisoissa, joten singletonin hyödyntäminen tuli tarpeeseen. Pelissä on objekti, jonka mukana singleton skriptit kulkevat, joten tähän objektiin luotiin DecalManager niminen skripti, joka asetettiin tyhjän objektin sisään. Skriptissä perittiin MonoSingleton luokka, jolloin luokka muunnettiin singletoniksi. Luotu DecalManager piti sisällään tallennetut tarrat sekä tarran prefabin. Tarran prefab oli käytännössä pohja tarralle, se sisälsi tärkeimmät komponentit tarraa varten, jolloin niiden arvoja voitiin muokata helposti tarran luomisen yhteydessä.

Skriptin tehtävänä oli ladata ja tallentaa tarrat listaan, sekä luoda tarrat annetun objektin sisään. Lisäksi skriptillä tallennettiin ja ladattiin JSON data, jolloin myös palvelinpuoleen lähetettävä tieto noudettiin tästä skriptistä (ks. Kuvio 21). JSON muotoisen datan käsittelyssä selkein ratkaisu oli muuntaa data listaksi ja takaisin, listan avulla tarrojen luominen oli yksinkertaista silmukan avulla, sekä listaa käytettäessä voitiin manipuloida listan elementtejä helposti.



Kuvio 21. DecalManager singletonin käyttö

6.10 Tarrojen yhdistäminen

Tarrat haluttiin yhdistää yhdeksi objektiksi, jotta suorituskyky paranisi, tässä työssä rajoituksena oli yksi tarra pelaajaa kohden, mutta peilattuna tarraobjekteja oli kuitenkin kaksi. Nämä voitiin yhdistää yhdeksi objektiksi, ja samalla jatkokehityksen kannalta yhdistäminen oli jo valmiina. Yhdistämisessä kokeiltiin ensimmäisenä hyödyntää Unityn tarjoamaa CombineMeshes funktiota, joka nimensä mukaisesti yhdistää objektien verkkorakenteet yhdeksi kokonaisuudeksi, mutta se ei tukenut eri materiaaleja. Tätä varten löydettiin Unityn keskustelufoorumilta ratkaisu, käyttäjät olivat luoneet ratkaisuja, jotka tukevat eri materiaaleja. Erityisesti BlackSnow_2 käyttäjän ratkaisu sopi tähän ongelmaan, funktio hyödynsi Unityn CombineMeshes funktiota ja lisäksi otti eri materiaalit huomioon (CombineMeshes with Different Materials 2021).

Yhdistämisessä tuli vastaan ongelmia, decalRoot objektin skaalaus oli 1,1,1, mutta kaikki parent objektit eivät olleet, pahimmillaan skaalaus oli epäyhtenäinen, esimerkiksi 1.1, 1.2, 1.4, joten skaalausta täytyi muuttaa ennen yhdistämistä. Unityn dokumentaatio kehoittaa välttämään epäyhtenäistä skaalausta, jos vain on mahdollista, sillä se voi luoda muun muassa suorituskykyongelmia (Transforms 2007). Ennen tarrojen luomista muutettiin ensin decalRoot objektin paikallinen skaalaus globaaliin skaalaukseen, tällä keinolla skaalaus ei ollut suhteessa parent objektiin. Globaaliin skaalaukseen muuntaminen oli vaatimuksena epäyhtenäisen skaalauksen toimivuuteen. Lisäksi

nollattiin decalRootin MeshFilter ja MeshRenderer komponentit, jottei toistuva yhdistäminen yhdistä samalla vanhaa verkkorakennetta (ks. Kuvio 22), nämä komponentit lisätään MeshCombine funktiossa. Näiden toimenpiteiden jälkeen luotiin tarrat aiempaan tapaan nonMirroredDecalRoot ja mirroredDecalRoot objektien alle, josta tarrat siirrettiin decalRoot objektin alle ennen yhdistämistä. Ennen yhdistämistä säilöttiin decalRootin parent objektin skaalaus sekä asetettiin tälle uusi skaalaus laskemalla skaalauskomponenttien keskiarvo, tämä ratkaisi epäyhtenäisten skaalausten tuomat ongelmat, joissa skaalausakselit ovat erikokoisia.

```
MeshFilter meshFilter = decalRoot.GetComponent<MeshFilter>();
if (meshFilter != null)
{
    decalRoot.GetComponent<MeshFilter>().mesh = null;
}
MeshRenderer meshRenderer = decalRoot.GetComponent<MeshRenderer>();
if (meshRenderer != null)
{
    decalRoot.GetComponent<MeshRenderer>().material = null;
}

decalRoot.transform.localScale = decalRoot.transform.lossyScale;
```

Kuvio 22. Visuaalisten komponenttien nollaus ja decalRootin skaalauksen muuntaminen

Tästä siirryttiin itse yhdistämiseen, kutsuttiin MeshCombine funktiota ja annettiin sille parametrinä decalRoot (ks. Kuvio 23). Funktion alussa asetetaan tuodun objektin Transform komponentin arvot oletusarvoihin, jotta yhdistäminen suoritetaan yksinkertaisimmassa koordinaatiojärjestelmässä. Seuraavaksi alustetaan materiaalilista sekä CombineInstance listat, sekä säilötään lapsiobjektien MeshFilter komponentit listaan. Jokainen MeshFilter käydään yksitellen läpi, otetaan materiaali talteen ja luodaan CombineInstance MeshRenderer komponentin perusteella. CombineInstance on struktuuri, jota käytetään CombineMeshes funktiossa, se pitää sisällään yhdistettävän datan. MeshFilter komponenttien läpikäynnin jälkeen yhdistetään verkkorakenteet CombineMeshes funktiota hyödyntäen, jonka jälkeen asetetaan oikeat materiaalit paikoilleen sekä poistetaan yhdistetyt objektit. Funktio palauttaa yhdistämisen jälkeen Transform komponentin arvot takaisin alkuperäiseen muotoonsa.

Lopuksi skaala täytyy vielä kerran muuttaa, yhtenäisesti skaalatuissa ajoneuvoissa lisämuutoksia ei vaadita, tarrat näkyvät nyt oikein, mutta epäyhtenäisissä tarroissa on heittoa, sillä tarrat yhdistettiin keskiarvoskaalatun parent objektin sisällä. Tätä varten lasketaan decalRoot parent objektin X-,

Y- ja Z-komponenttien ero verrattuna aiemmin laskettuun keskiarvoon. Lasketut komponentit summattiin decalRoot objektin skaalaan, jolloin sen alla olevat tarrat skaalautuvat oikein.

```
Vector3 originalParentScale = decalRoot.transform.parent.localScale;
float averageScale = (originalParentScale.x + originalParentScale.y + originalParentScale.z) / 3.0f;

decalRoot.transform.parent.localScale = new Vector3(averageScale, averageScale, averageScale);

decalRoot.transform.localScale = Vector3.one;
Utility.MaterialSafeMeshCombine.MeshCombine(decalRoot, true);

float x = originalParentScale.x - averageScale;
float y = originalParentScale.y - averageScale;
float z = originalParentScale.z - averageScale;
decalRoot.transform.localScale = new Vector3(
    decalRoot.transform.localScale.x + x,
    decalRoot.transform.localScale.y + y,
    decalRoot.transform.localScale.z + z);

decalRoot.transform.parent.localScale = originalParentScale;
```

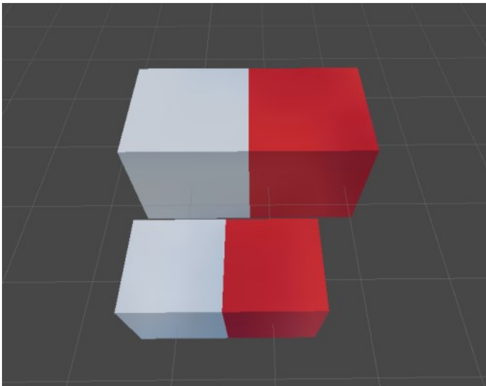
Kuvio 23. Tarrojen yhdistämisen skaalausten laskeminen

Yhdistämistä luodessa ilmeni myös muita skaalausongelmia, joka aiheutti tarran kokoon pientä heittoa. Nämä ongelmat ratkottiin muuttamalla tarvittavien objektien skaalaukset 1,1,1 muotoon ennen tarran tallentamista, ja palauttamalla skaalaukset tallentamisen jälkeen (ks. Kuvio 24). Kyseessä oli hyvin tapauskohtainen ratkaisu, mutta luultavasti toimisi samalla periaatteella muissakin vastaavissa tilanteissa. DecalRoot objektin skaalausta ei tässä palautettu, sillä siihen laskettiin yhdistämisen jälkeen skaalaus (ks. Kuvio 23).

```
        if (didChanges)
        {
            NormalizeScales();
            Quaternion normalRotation = Quaternion.LookRotation(-hitNormal);
            Quaternion rotation = normalRotation * Quaternion.Euler(0, 0, angle);
            DecalManager.Instance.SetDecalTransform(
                kart.kartID, Kart.decalRoot.InverseTransformPoint(position),
                Quaternion.Inverse(Kart.decalRoot.rotation) * rotation,
                new Vector3(scale.x, scale.y, scaleDepth),
                mirrored);
            RevertScalesAndRebuildDecals();
        }
        didChanges = false;
    }
}

4 references
private void NormalizeScales()
{
    Kart.decalRoot.localScale = Vector3.one;
    originalKartRootScale = Kart.transform.parent.localScale;
    Kart.transform.parent.localScale = Vector3.one;
}

4 references
private void RevertScalesAndRebuildDecals()
{
    RebuildDecals();
    Kart.transform.parent.localScale = originalKartRootScale;
}
```

Kuvio 24. Skaalausten normalisointi ja palauttaminen

Skaalausongelman voi konkreettisesti nähdä kuviosta (ks. Kuvio 25). Valkoiset kuutiot ovat maailman koordinaatistossa X-akselilla samassa kohdassa, ja objektien sisällä on punainen lapsiobjekti, jonka paikallinen koordinaatti on 1,0,0. Ylempi valkoinen kuutio on skaalattu 1.5, 1.5, 1.5 muotoon ja siirretty Z-akselilla 2 yksikköä kauemmas ja alempi 1,1,1 muotoon, lapsiobjektit ovat 1,1,1 muodossa. Alemman kuution lapsiobjektin maailmankoordinaatin X-akselin arvo on 1, ja ylemmällä vastaavasti 1,5.



Kuvio 25. Kaksi objektia eri skaaloilla

7 Tulokset

7.1 Projektiototeutuksen lopputulos

Projektion kohdistaminen halutulle paikalle onnistui hyvin ja JSON muotoinen tallennettava data oli optimoitua, palvelinpuolelle ei lähetetty mitään tarpeetonta. TwoDecimalIntVector3 struktuuria voi hyödyntää jatkossa muuallakin, jos on tarpeen optimoida JSON dataa ja kahden desimaalin tarkkuus riittää. Tarrojen asettelussa kahden desimaalin tarkkuus oli riittävä, tätä tarkempi asettelu olisi ollut tarpeetonta ja lähes huomaamatonta.

Projektion syvyyden laskeminen vaikutti olevan tarkka, mutta se ei ole täydellinen tapa asettaa tarralle syvyys. Syvyys toimi ovien kohdalla erinomaisesti, mutta esimerkiksi ajoneuvon takaosaan tarran asettaminen laski hyvin syvälle, sillä poistumispiste oli kuskin penkissä. Tämä johti kehnon näköiseen tarraan, sillä se sai piirtyä todella pitkälle. Tätä yritettiin rajoittaa asettamalla minimi ja maksimi tarran syvyys, joka toimi suhteellisen hyvin, mutta ei kuitenkaan täydellisesti. Syvyyslaskenta ei voi tietää mitä pelaaja haluaisi, jolloin vaihtoehtona olisi antaa pelaajalle mahdollisuus

asettaa syvyys, mutta tätä ei koettu kuitenkaan tarpeelliseksi. Tarrat kääriytyivät kuitenkin hyvin pinnanmuutosten mukana (ks. Kuvio 26).

Kuvio 26. Rakettitarra ajoneuvossa

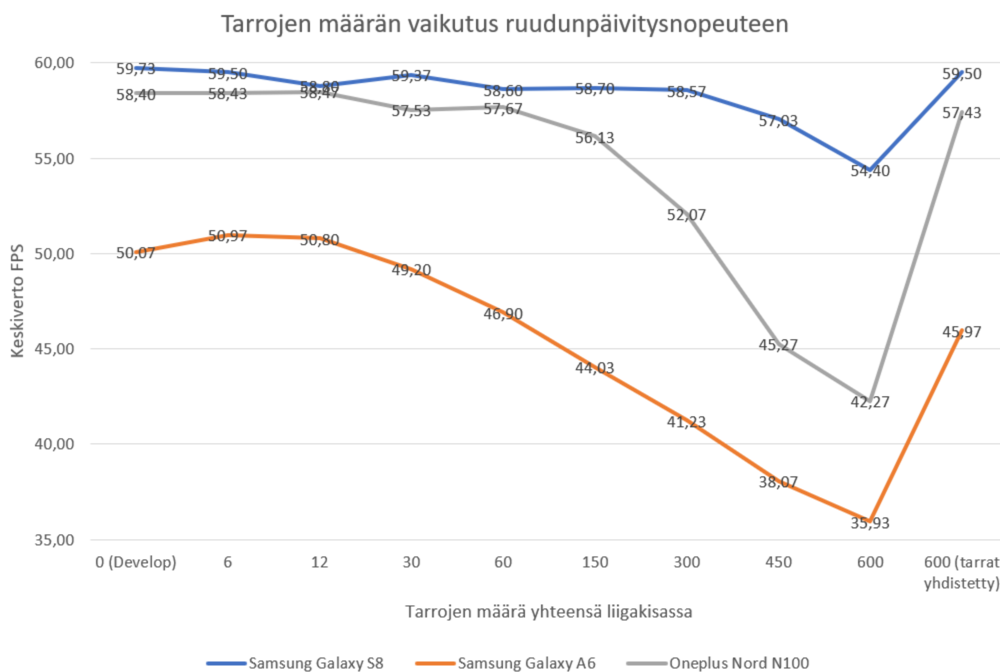
Tarrojen yhdistäminen toimi todella hyvin ja tuki eri skaalauksia, mutta koodin osalta ratkaisut olivat hyvin tapauskohtaisia, koodin hallinta menee sekavaksi hyvin äkkiä, jos samaa periaatetta jatketaan tulevaisuudessakin, parempi ratkaisu olisi korjata objektien skaalaukset. Tässä toteutuksessa tarrarajoitus oli yksi per pelaaja, joten yhdistämistä ei tapahdu kuin peilatun tarran kanssa. Yhdistämisen suorituskyky haluttiin kuitenkin mitata, joten tätä varten tehtiin ruudunpäivitysnopeustesti, jossa vertailtiin kolmen eri laitteen suorituskykyä ilman tarrojen yhdistämistä ja tarrojen yhdistämisen jälkeen, tästä tarkemmin kappaleessa 7.2.

7.2 Suorituskykytesti

Suorituskykytesti tehtiin vertaamalla develop- ja tarrasysteemihaaroja keskenään, tarrasysteemihaarassa testattiin lisäksi suurien tarramäärien vaikutusta ilman yhdistämistä ja yhdistämisen kanssa. Testeissä ei kuormitettu palvelinpuolta, vaan tarrat lisättiin silmukan avulla halutun määrän verran uudelleen. Testin tarkoituksena oli selvittää, onko tarrasysteemi haara vaikuttanut pelin suorituskykyyn negatiivisesti, ja saavutetaanko tarrojen yhdistämisellä merkittäviä eroja. Testiin

valittiin kolme android mobiililaitetta: Samsung Galaxy S8, Samsung Galaxy A6 sekä Oneplus Nord N100, näistä käytetään jatkossa lyhenteitä S8, A6 sekä N100.

A6 valittiin heikon suorituskyvyn vuoksi, pelin ruudunpäivitysnopeuden tavoite on 60, A6 kykeni tähän vaivoin, tämän arveltiin korostavan tarramäärien vaikutusta. S8 ja N100 pyörittivät peliä tavoitepäivitysnopeudessa ongelmitta, näiden laitteiden uskottiin toimivan tehokkaasti suurillakin tarramäärillä. Jokainen testi suoritettiin ajamalla liigakisoja normaalisti, pysymällä kuitenkin viimeisillä sijoilla, jotta tarroja on mahdollisimman paljon ruudulla samanaikaisesti, jokaisen ajon jälkeen käynnistettiin peli uudelleen, jotta lähtötilanne on sama. Ajon aikana laskettiin ruudunpäivitysnopeuksia viimeisen kolmen, kahden ja yhden minuutin ajalta, ajoja suoritettiin kymmenen jokaista tarramäärää kohden, ja näistä laskettiin lopullinen keskiarvo. Lopputuloksena saatiin määriteltyä tarramäärien vaikutukset ruudunpäivitysnopeuteen viivakaavion avulla (ks. Kuvio 27).



Kuvio 27. Ruudunpäivitystestit tarrojen yhdistämisen kanssa

Viivakaaviosta (ks. Kuvio 27) nähdään, että S8 ja N100 sietivät suuria tarramääriä hyvin, A6:sen suorituskyky lähti laskemaan tasaisesti jokaisen tarramäärän lisäyksen jälkeen. Develop haaraa ja kuuden tarran ajoja vertaamalla tarrasysteemin lisääminen ei vaikuttanut suorituskykyyn edes yhden ruudunpäivityksen vertaa, jolloin ero on mitätön. Jos tarkastellaan pelkästään S8 ja N100:n

suorituskykyä, käännekohta tulee 150–300 tarran välillä, siihen saakka suorituskyky on melko sama. N100:n ruudunpäivitysnopeus laskee kuitenkin 12–150 tarran välillä 2,34 yksikköä, mutta tämä on melko huomaamaton ero, merkittävä käännekohta alkaa 150 tarran jälkeen, tästä eteenpäin on vain suorituskyvyn laskua. A6:lla saatu data osoittautui korostavaksi, kuten arveltiinkin, mutta laite oli jo lähtökohtaisestikin heikkotehoinen. A6:sen data on näin ollen epäluotettavaa, mutta kertoo kuitenkin, että tarramäärillä oli suuri vaikutus.

Tarrojen yhdistämisen vaikutus oli merkittävä, tämä ilmenee viivakaavion viimeisessä testissä. Suorituskyvyt nousivat merkittävästi, lähes takaisin tavoitenopeuteen. A6:sen suorituskyvyssä oli edelleen ongelmia, alkuperäisestä nopeudesta jäätiin noin 5 ruudunpäivitystä alemmaksi. Yli 600 tarramäärien testejä ei koettu tarpeelliseksi, sillä Unity olisi rajoittanut verkkorakenteiden kokoa, tämä olisi vaatinut koodiin lisämuutoksia.

Tulosten tarkkuuteen vaikutti paljon kisojen erilaisuus, toisissa kisoissa meno saattoi olla varsin puhdasta ja toisessa ohjukset lensivät vähän väliä, lisäksi ajoradat olivat sattumanvaraisia liigakisoissa, jolloin lyhyempi rata saattoi olla laitteelle kevyempi. Ohjusten räjähdyspartikkelit söivät laskentatehoa, ja vääristivät keskiarvoja. Ajojen aikana oli havaittavissa ruudunpäivitysnopeuden laskevan noin 2–3 yksikköä, aina kun ohjus tai pommi räjähti. Tämä otettiin tarkastelussa huomioon, ja ruudunpäivitysnopeuksille arvioitiin 1–2 yksikön epätarkkuus.

7.3 Käyttöliittymävaihtoehtojen vertailu

Käyttöliittymän suunnittelussa tutustuttiin vastaaviin toteutuksiin. Forza Horizon 5:ssa tarran asettelussa käyttöliittymä oli hyvin selkeä, painikkeissa ei ollut tekstiä lainkaan. Painikkeiden symboleista tunnisti käyttötarkoituksen suoraan, tarraa voi liikutella, skaalata sekä kiertää. Peli tarjosi myös mahdollisuuden luoda tarraryhmiä, jolloin pelaaja voi tehdä ennalta määritellyillä muodoilla ryhmän, jonka voi asettaa yhtenä tarrana ajoneuvoon.

KartRider Drift+ mobiilipelissä käyttöliittymän painikkeet olivat myös symboleja, joiden käyttötarkoitus ei jäänyt epäselväksi. Tarrojen asettelu oli kuitenkin hiukan hämärää, välillä tarran asettelusuunta vaihtui ja täytyi liikuttaa esimerkiksi sormella ylöspäin, jotta tarra liikkui vasemmalle. Pelissä oli mahdollista skaalata tarroja erikseen X- ja Y-akselilla, kääntää tarran skaalaus negatiiviseksi sekä luoda tarraryhmiä Forza Horizon 5:sen tapaan.

Käyttöliittymää arvioitiin tiimin kesken monella tapaa. Eri painikkeita oli kolme: kameran liikuttaminen, tarran liikuttaminen sekä peilaus. Kameran ja tarran liikuttamiseen käytetyt painikkeet toimivat rinnakkain, vain toinen voi olla yhtäaikaisesti kytkettynä. Peilauspainike oli erillään tästä, jolloin päädyttiin ratkaisuun, että liikuttamiseen käytetyt painikkeet ja peilauspainike ovat selkeästi eroteltuna toisistaan. Tarraa voitiin pyörittää ja skaalata kahden sormen kosketuksella, tämä koettiin ennalta-arvattavaksi toiminnoksi, joka ei vaadi erillistä ohjeistusta.

Käyttöliittymän painikkeisiin lisättiin lisäsymboleja selkeyttämään painikkeen toimintoa (ks. Kuvio 28). Kameran ja tarran liikuttamiseen asetettiin nuolet, jotka kytketään näkyviin ainoastaan, jos painike on kytkettynä. Käyttöliittymään lisättiin näiden lisäksi myös ohjeistus, jos pelaaja ei ole valinnut mitään tarraa ja painaa liikuttamis- tai peilauspainiketta, pelaajalle näytetään näissä tapauksissa kahden sekunnin ajan infoikkuna ”Decal not selected”. Pelaajan ensimmäiseen kokemukseen kiinnitettiin myös huomiota, tätä varten luotiin oletussijainti tarralle. Kun pelaaja saa ensimmäisen tarransa ja klikkaa sitä, se asetetaan ajoneuvon kylkeen. Ilman tätä tarralla ei ole oletuksena laskettua sijaintia, ja pelaajan näkökulmasta tarrapainikkeet saattavat vaikuttaa rikkinäiseltä, tiimin jäsenten kokeiluilla tämä oli toistuva puheenaihe.

Kuvio 28. Käyttöliittymän ensimmäinen versio eri tilanteissa

Lopuksi arvioitiin symbolien ymmärrettävyyttä ensimmäisessä versiossa, kameran liikuttamiseen käytetty symboli vaikutti kuvankaappaussymbolilta. Lisäksi tarran symboli vaikutti osalle ensisilmäyksellä puhekuplalta. Näin ollen kuvakkeet päivitettiin selkeämmiksi (ks. Kuvio 29). Peilaamisen symbolia ei päivitetty, sillä sen merkitys oli selkeä. Ajoneuvokuvake nuolilla kertoi huomattavasti

paremmin painikkeen käyttötarkoituksen, sillä koodinkin osalta itse kameraa ei todellisuudessa liikutettu, vaan ajoneuvoa.

Kuvio 29. Käyttöliittymän viimeinen versio eri tilanteissa

8 Pohdinta

Muiden mobiilipelien vastaavista toteutuksista saatiin kattavasti tietoa, millaisia kustomisaatiovaihtoehtoja pelit tarjoavat. Toistuvana teemana oli ajoneuvon kattava kustomisointi, Boom Kartsissa hahmon kustomisaatio oli huomattavasti kattavampi. Kustomisaatiotarpeet kohdistuivat ajoneuvoon, ja valittu kohde oli tarrojen implementaatio. Alan vastaavien toteutusten kartoittaminen vaikutti olevan hyvä keino selvittää kehitettäviä kohteita.

Projektioteknologian valinnassa täytyy ensinnäkin tietää, millä renderöintiputkella peli on toteutettu. Built-in piirtoputkelle hyvä ratkaisu on Llockham-Industries yrityksen luoma Dynamic Decals paketti. Paketti on avointa lähdekoodia, joten sitä saa käyttää täysin vapaasti. Paketti tarjoaa metallisia ja heijastavia pintoja, jotka sopivat ajoneuvoon projektoitaviin tarroihin. Vaikka pakettia ei tämän työn toteutuksessa otettu käyttöön, se sopii todella hyvin vastaaviin toteutuksiin ja toteutetut projektion kohdistukset toimivat samalla periaatteella. Paketissa on saatavilla runsaasti ominaisuuksia, kuten eri tasoille aseteltavat tarrat. URP ja HDRP piirtoputkille on Unityn puolelta tuetut tarrakomponentit.

Suuri määrä erillisiä tarraobjekteja vaikutti mobiililaitteen suorituskykyyn yllättävän paljon, suorituskyvyn alenemista oli havaittavissa jo 150 tarralla. Tarrojen yhdistäminen ratkaisee tämän ongelman, mutta tuo samalla rajoituksia. Jos tarroja haluttaisiin liikuttaa luomisen jälkeen, tai muutoin animoida, niiden täytyisi olla omina objekteinaan, shaderin kautta animoitava materiaali toimii. Yhdistämiseen käytetty koodi ei myöskään tue valtavia määriä tarroja, sillä verkkorakenteen koko kasvaa liian isoksi ja Unity rajoittaa tämän. Tämä olisi ratkaistavissa pilkkomalla tarrat pienempiin kokonaisuuksiin, mutta tässä toteutuksessa sitä ei koettu tarpeelliseksi.

Tarrojen yhdistämisessä käytetty funktio soveltuu mihin tahansa staattisiin objekteihin, ja tässä työssä lisättiin eri skaalausten tuki, erityisesti epäyhtenäisten. Jos koko objektin hierarkia kuitenkin käyttää 1,1,1 skaalausta, tukea ei tarvita. Useille tarroille ei lisätty tukea eri tasoille, peilattu tarra menee peilaamattoman päälle pienellä hienosäädöllä. Tarrojen yhdistämiseen käytetty funktio soveltuu vain, jos tarraobjektit ovat verkkorakenteena, dynamic decals paketissa tarrat piirretään shaderin kautta, jolloin samaa yhdistämisfunktiota ei voi hyödyntää. Dynamic decals pakettia käytettäessä tulee kuitenkin välttää samaan tapaan suuria määriä erillisiä objekteja mobiiliympäristössä, ja säilöä tarrakomponentteja esimerkiksi listassa.

Jos tarramäärää nostetaan, täytyy myös tehdä eri tasot, ilman tätä tarrat ovat samalla tasolla ja vilkkuvat. Tarramääriä lisäämällä eri tasoja ei kannata toteuttaa muuttamalla offset muuttujaa, sillä sitä käyttämällä tarrat alkavat ennen pitkää olemaan näkyvästi irti ajoneuvon pinnasta. Tällainen toteutus täytyisi luultavimmin toteuttaa shaderin kautta, jossa renderöintijonoa voidaan muuttaa. Objektien skaalauksia ei myöskään työn perusteella kannata lähtökohtaisesti muuttaa, jos skaalaukset pidetään 1,1,1 muodossa, vältetään paljon ongelmia tulevaisuudessa. Mallinnetut objektit voi suoraan tehdä oikeankokoisiksi, jolloin skaalausta ei erikseen muuteta. Työssä ilmeni useaan kertaan eri skaalausten tuomat ongelmat, ja näitä ratkaistiin muuntamalla skaalaus 1,1,1 muotoiseksi ja laskentojen jälkeen muunnettiin skaalaus alkuperäiseen muotoon.

Tarratoteutuksen voi myös tehdä suoraan UV-mappia muokkaamalla ja lisäämällä tekstuurit siihen, tämä olisi pelinaikaisen suorituskyvyn kannalta tehokkain keino, tekstuuri voitaisiin luoda ennen kisan alkua. Tässä toteutuksessa se ei ollut ajallisesti järkevää toteuttaa, sillä osa ajoneuvojen

maalipinnoista on peilattu, jolloin lisätty tarrakin peilaantuisi ajoneuvon toiselle puolelle, tämä toiminto haluttiin pelaajan päätettäväksi. Maalipintoja on toteutettu niin paljon, että UV-mappien uudelleentekeminen olisi vaatinut paljon aikaa graafikoilta.

Tarrat piirtyvät tällä toteutuksella koko ajoneuvoon, sillä ajoneuvo on mallinnettu yhtenäiseksi. Jatkokehittämällä ajoneuvojen eri osat olisi mahdollista pilkkoa erilleen toisistaan, jos halutaan ettei tarrat piirry ikkunoihin, ajovaloihin tai kuskin penkkiin. Tämä vaatisi kuitenkin jokaisen ajoneuvon purkamista useisiin verkkorakenteisiin mukaan lukien uniikit mallinnukset, joten työmäärä olisi suuri. Mahdollinen toteutustapa voisi olla verkkorakenteen purkaminen niin sanotulle sallitulle piirtoalueelle ja sen asettaminen jokaiseen maalinpintakonfiguraatioon. Sallittua piirtoaluetta voisi tämän jälkeen hyödyntää tarran piirtämisessä, koko ajoneuvon mallinnuksen sijaan haetaan vain piirtoalue.

Lähteet

Amlin, J. 2022. Classes in C# using Unity. Level Up Coding 12.4.2022. Viitattu 9.5.2023. https://levelup.gitconnected.com/classes-in-c-using-unity-4325f2080353

AssetDatabase.CreateAsset. N.d. Artikkeli Unity asseteista Unityn dokumentaatiossa. Viitattu 9.5.2023. https://docs.unity3d.com/ScriptReference/AssetDatabase.CreateAsset.html

Bycer, J. 2017. Design differences: customization vs personalization. Artikkeli Game Wisdomin verkkosivuilla. Viitattu 10.5.2023. https://game-wisdom.com/critical/customization-vs-personalization.

Böffel, C., Würger, S., Müsseler, J. & Schlittmeier, S. 2022. Character Customization With Cosmetic Microtransactions in Games. Frontiers in psychology. Viitattu 27.4.2023. https://janet.finna.fi, PubMed

Chapple, C. 2020. Fortnite Mobile Revenue Hits $1 Billion in Two Years. Artikkeli SensorTowerin verkkosivustolla. Viitattu 10.5.2023. https://sensortower.com/blog/fortnite-mobile-revenue-1-billion

Colagrossi, M. 2021. How Microtransactions Impact the Economics of Gaming. Artikkeli Investopedian verkkosivuilla. Viitattu 10.5.2023. https://www.investopedia.com/articles/investing/022216/how-microtransactions-are-evolving-economics-gaming.asp

CombineMeshes with Different Materials. 2021. Unityn keskustelufoorumin keskustelu verkkorakenteiden yhdistämisestä. Viitattu 21.8.2023. https://discussions.unity.com/t/combinemeshes-with-different-materials/33290

Cox, D. 2021. Encrypting Game Data with Unity. Artikkeli AES salauksesta Unityssä. Viitattu 10.7.2023. https://videlais.com/2021/02/28/encrypting-game-data-with-unity/

Creating and Using Scripts. N.d. Artikkeli skriptien luomisesta ja käytöstä Unityn dokumentaatiossa. Viitattu 9.5.2023. https://docs.unity3d.com/Manual/CreatingAndUsingScripts.html

Decals and projectors. N.d. Artikkeli tarroista ja projektoreista Unityn dokumentaatiossa. Viitattu 9.5.2023. https://docs.unity3d.com/Manual/visual-effects-decals.html

Game development studio. 2023. Zaibatsu Interactive Oy:n verkkosivut. Viitattu 14.7.2023. https://zaibatsu.fi/

GameObject.GetComponent. N.d. Artikkeli GetComponent kutsusta Unityn dokumentaatiossa. Viitattu 9.5.2023. https://docs.unity3d.com/ScriptReference/GameObject.GetComponent.html

Interfaces. 2022. Artikkeli rajapinnasta Unityn opetussivustolla. Viitattu 9.5.2023. https://learn.unity.com/tutorial/interfaces

Introducing JSON. N.d. Artikkeli JSON formaatista JSON.org verkkosivuilla. Viitattu 22.6.2023. https://www.json.org/json-en.html

JsonUtility. N.d. Artikkeli JsonUtility luokasta Unityn dokumentaatiossa. Viitattu 22.6.2023. https://docs.unity3d.com/ScriptReference/JsonUtility.html

JsonUtility.FromJson. N.d. Artikkeli JsonUtility.FromJson funktiosta Unityn dokumentaatiossa. 22.6.2023. https://docs.unity3d.com/ScriptReference/JsonUtility.FromJson.html

Kordyaka, B. & Hribersek, S. 2019. Crafting Identity in League of Legends – Purchases as a Tool to Achieve Desired Impressions. Viitattu 27.4.2023. https://www.researchgate.net/publication/327871565_Crafting_Identity_in_League_of_Legends_-_Purchases_as_a_Tool_to_Achieve_Desired_Impressions

Liao, G., Cheng, T. & Teng, C. 2019. How do avatar attractiveness and customization impact online gamers' flow and loyalty? Internet research, 29, 2, 349-365. Viitattu 27.4.2023. https://janet.finna.fi, ProQuest

Luban, P. 2011. The Design of Free-To-Play Games: Part 1. Game Developer 22.11.2011. Viitattu 9.5.2023. https://www.gamedeveloper.com/design/the-design-of-free-to-play-games-part-1

Martin, E. 2022. Day 128 of Game Dev: MonoSingleton – Unity/C#. Artikkeli MonoSingleton luokasta. Viitattu 10.7.2023. https://blog.devgenius.io/day-128-of-game-dev-monosingleton-unity-c-83ecc8775072

Mesh Filter component. N.d. Artikkeli mesh filter komponentista Unityn dokumentaatiossa. Viitattu 9.5.2023. https://docs.unity3d.com/Manual/class-MeshFilter.html

Mesh Renderer component. N.d. Artikkeli mesh renderer komponentista Unityn dokumentaatiossa. Viitattu 9.5.2023. https://docs.unity3d.com/Manual/class-MeshRenderer.html

Naysmith, C. 2022. Gaming Skins Just Became A $50 Billion Industry. Artikkeli Yahoo Financen verkkosivuilla. Viitattu 10.5.2023. https://finance.yahoo.com/news/gaming-skins-just-became-50-143352555.html

Onur. A.E. 2021. Actions and Func In Unity. Artikkeli tapahtumista Unityssä. Viitattu 10.7.2023. https://aliemreonur.medium.com/actions-in-unity-cfe2f78bb05d

PlayerPrefs. N.d. Artikkeli PlayerPrefs luokasta Unityn dokumentaatiossa. Viitattu 14.7.2023. https://docs.unity3d.com/ScriptReference/PlayerPrefs.html

Prefabs. N.d. Artikkeli Prefabeista Unityn dokumentaatiossa. Viitattu 18.5.2023. https://docs.unity3d.com/Manual/Prefabs.html

Projector component. N.d. Artikkeli projector komponentista Unityn dokumentaatiossa. Viitattu 9.5.2023. https://docs.unity3d.com/Manual/class-Projector.html

Render pipelines introduction. Artikkeli piirtoputkista Unityn dokumentaatiossa. Viitattu 5.9.2023. https://docs.unity3d.com/Manual/render-pipelines-overview.html

Rovira, J. 2013. On Character Customization. Game Developer 9.6.2013. Viitattu 9.5.2023. https://www.gamedeveloper.com/programming/on-character-customization-part-0-

Schardon, L. 2023. What is Unity? – A Guide for One of the Top Game Engines. Artikkeli Unity pelimoottorista GameDev Academyn verkkosivuilla. Viitattu 9.5.2023. https://gamedevacademy.org/what-is-unity/

ScriptableObject. N.d. Artikkeli ScriptableObject luokasta Unityn dokumentaatiossa. Viitattu 9.5.2023. https://docs.unity3d.com/Manual/class-ScriptableObject.html

Scripting. N.d. Artikkeli Unityn skriptaamisesta Unityn dokumentaatiossa. Viitattu 9.5.2023. https://docs.unity3d.com/Manual/ScriptingSection.html

Script serialization. N.d. Artikkeli skriptien serialisaatiosta Unityn dokumentaatiossa. Viitattu 9.5.2023. https://docs.unity3d.com/Manual/script-Serialization.html

Thorn, A. 2014. Pro Unity Game Development with C#. Ensimmäinen painos. Berkeley: Apress. Viitattu 29.4.2023. https://janet.finna.fi, Skillsoft Books ITPro

Toikko, T. & Rantanen, T. 2009. Tutkimuksellinen kehittämistoiminta. Tampere: Tampere University Press. Viitattu 4.5.2023. https://trepo.tuni.fi/handle/10024/100802

Transforms. 2007. Artikkeli Transform komponentista Unityn dokumentaatiossa. Viitattu 14.7.2023. https://docs.unity3d.com/410/Documentation/Manual/Transforms.html

Welcome to Boom Karts. 2021. Uutisartikkeli Boom Kartsin julkaisusta Fingersoftin verkkosivuilla. Viitattu 9.5.2023. https://fingersoft.com/news/2021/05/06/welcome-boom-karts/

Welcome to Unity. N.d. Artikkeli Unitystä Unityn verkkosivuilla. Viitattu 9.5.2023. https://unity.com/our-company

Liitteet