OULUN AMMATTIKORKEAKOULU

Santtu Niskala

HYPERKASUAALIN MOBIILIPELIN TEKEMINEN

HYPERKASUAALIN MOBIILIPELIN TEKEMINEN

Santtu Niskala
Opinnäytetyö
Syksy 2019
Tietojenkäsittely
Oulun ammattikorkeakoulu

TIIVISTELMÄ

Oulun ammattikorkeakoulu
Tietojenkäsittely, Web-sovelluskehitys

Tekijä(t): Santtu Niskala
Opinnäytetyön nimi: Hyperkasuaalin mobiilipelin tekeminen
Työn ohjaaja: Matti Viitala
Työn valmistumislukukausi ja -vuosi: Syksy 2019 Sivumäärä: 35 + 2

Työn aiheena oli hyperkasuaalin mobiilipelin kehittäminen, jota voi pelata mobiililaitteella. Valitsin tämän aiheen koska olin työn aloitushetkellä töissä mobiilipeliyrityksessä, joka keskittyi hyperkasuaalien mobiilipelien kehitykseen. Työllä ei ollut toimeksiantajaa.

Tässä työssä kerrotaan hyperkasuaaleista mobiilipeleistä ja sellaisen kehittämisen eri alueista. Työssä kerrotaan pelin suunnittelussa tehdyistä valinnoista sekä myös kehityksen muutamasta eri osasta ja siinä vastaan tulleista haasteista ja niiden ratkaisemisesta. Pelin kehityksessä käytettiin apuna Unity-pelimoottoria ja muutamia sille tehtyjä ilmaisia ja maksullisia kehitystä helpottavia paketteja. Työn tietoperustana käytettiin omaa tietoani hyperkasuaaleista peleistä ja niiden kehityksestä ja myös netistä saatua ajantasaisempaa tietoa niiden markkinatilanteesta.

Työn tuloksena saatiin valmiiksi mobiililaitteilla pelattava peli, josta löytyy hyperkasuaaleille peleille tyypilliset mekaniikat ja ominaisuudet. Kehitettyyn peliin jäi vielä paljon jatkokehitettävää, koska pelin kehittämiseen käytettävä aika oli todella rajallinen.

Asiasanat: pelit, peliala, mobiilipelit, pelikehitys, ohjelmointi

ABSTRACT

Oulu University of Applied Sciences
Business Information Systems, Web Application Development

Author(s): Santtu Niskala
Title of thesis: Developing a hyper-casual mobile game
Supervisor(s): Matti Viitala
Term and year when the thesis was submitted: Autumn 2019 Number of pages: 35 + 2

The subject of this thesis is to develop a hyper-casual mobile game that is playable on a mobile device. I chose this subject because I was working for a mobile game company that mainly focused on developing hyper-casual games when I started my work on this thesis. There was no commissioner for this thesis.

This thesis examines hyper-casual mobile games and the different areas of developing a hyper-casual game. The thesis describes the choices made in the design of the game, as well as some aspects of the development and the challenges that I faced and how they were solved. The game was developed on the Unity game engine with the help of some free and paid assets.

The knowledge base for my work was my experience from working with hyper-casual games and their development, as well as more up-to-date information on their market situation from online sources.

The result of this thesis was a playable mobile game that has the mechanics and features typical of hyper-casual games. There was still a lot of room for further development of the game, as the time available for developing the game was very limited.

Keywords: games, game industry, mobile games, game development, programming

SISÄLLYS

1 JOHDANTO

Tämän opinnäytetyön aiheena on hyperkasuaalit mobiilipelit ja niiden kehitys. Valitsin aiheen, koska olin kirjoitushetkellä ohjelmoijana mobiilipeliyrityksessä ja koska hyperkasuaalit pelit olivat kovassa nousussa mobiilipeli markkinoilla.

Hyperkasuaalit pelit ovat todella yksinkertaisia ja lyhyitä pelikertoja hyödyntäviä pelejä. Ne sopivat hyvin mobiililaitteille, koska niillä pelatut pelisessiot ovat keskimäärin todella lyhyitä, tilastollisesti noin viisi minuuttia pitkiä (GameAnalytics 2018, viitattu 25.4.2019). Google Play -kaupan 10 maailmanlaajuisesti ladatuimman pelin joukossa oli vuonna 2018 kuusi eri hyperkasuaalia peliä, joista ensimmäisellä sijalla oli Helix Jump. (Gamesindustry.biz 2019, viitattu 25.4.2019.)

Opinnäytetyön tavoitteena oli kehittää mobiililaitteilla pelattava peli, josta löytyy hyperkasuaaleille peleille tavanomaisia mekaniikoita ja ominaisuuksia. Aika rajoitteiden takia pelistä ei ollut tarkoitus tehdä täysin viimeisteltyä, vaan siitä tuli löytyä vähintäänkin tärkeimmät ominaisuudet.

Kirjallisen osuuden ensimmäisessä osassa tutustutaan hyperkasuaaleihin peleihin yleisesti ja hieman syvemmin kolmeen eri hyperkasuaaliin peliin, jotka olivat kirjoitushetkellä todella suosittuja Google Play -palvelussa. Tässä osassa kerrotaan lyhyesti itse peleistä, niiden mekaniikoista ja monetisoinnista.

Seuraavassa osassa kerrotaan kehittämäni pelin konseptista, mekaniikoista, suunnittelusta ja kehityksestä. Siinä tutustutaan myös tarkemmin kehityksessä vastaan tulleisiin ongelmiin ja niiden ratkaisemiseen.

Raportin lopussa pohditaan, miten työ onnistui ja miten sitä voidaan vielä kehittää eteenpäin.

2 HYPERKASUAALIT MOBIILIPELIT

Hyperkasuaali on viime aikoina paljon suosiota kerännyt mobiilipelien alalaji. Niiden tarkoitus on olla mahdollisimman yksinkertaisia ja lyhyitä, mutta samalla todella koukuttavia ja uudelleenpelattavia. Tämä tekee niistä todella helposti aloitettavia ja pelattavia. Hyperkasuaalit pelit monetisoidaan yleensä pääsääntöisesti mainoksilla, joita voidaan näyttä esimerkiksi jokaisen pelikerran jälkeen, kun taas mobiilipelit tekevät tuottonsa mainoksilla ja pelin sisäisillä ostoilla (ironSource 2018, viitattu 26.4.2019). Yksinkertaisten mekaanikoiden ja visuaalisen tyylin ansiosta, hyperkasuaalien pelien suurin etu kehittäjälle on niiden lyhyt kehitysaika.

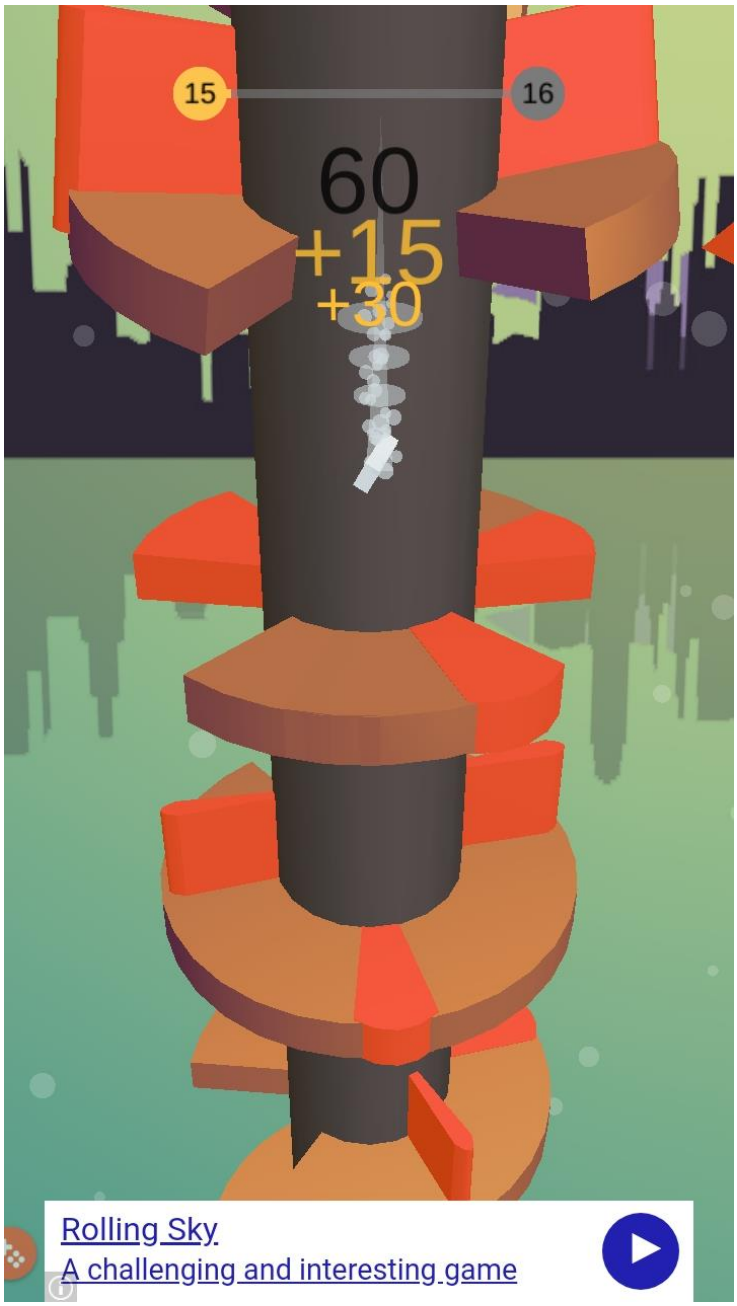
2.1 Helix Jump

Helix Jump on h8games:n kehittämä ja Voodoo:n julkaisema peli, jossa pelaaja yrittää saada pallon tippumaan tasanteiden läpi tason loppuun ilman, että pallo koskee tasanteiden punaisiin alueisiin. Pelaaja joutuu aloittamaan tason uudestaan alusta, jos pallo osuu punaiseen esteeseen tai tason alueeseen. Tasojen vaikeusaste kasvaa jokaisella tasoilla: tasojen pituus kasvaa, tasanteet alkavat liikkua ja tasoissa alkaa tulemaan vastaan enemmän punaisia alueita sisältäviä tasanteita. Jos pallon saa menemään kolmen tason läpi ilman, että se välissä pysähtyy tasanteelle, pallo tuhoaa seuraavan tasanteen sen koskiessa siihen. Pelissä on myös power-up, joka tuhoaa heti muutaman seuraavan tasanteen.

Pelissä ansaitaan pelin sisäistä valuuttaa läpäisemällä tasoja ja katsomalla mainoksia. Pelin sisäisellä valuutalla voidaan avata erimuotoisia pelaajan pallon korvaavia esineitä. Pelissä voidaan myös avata hieman erikoisempia esineitä katsomalla mainoksia ja avaamalla aarrearkkuja. Aarrearkkuja ansaitaan läpäisemällä viisi tasoa ja katsomalla sen jälkeen mainoksen.

Peli on kontrolleiltaan todella yksinkertainen, pelaaja pystyy ainoastaan pyörittämään tasoja liikuttamalla sormea ruudulla horisontaalisesti. Pelaaja ei pysty vaikuttamaan pallon liikkeeseen millään tavalla, vaan se pysyy aina ruudun keskellä ja tippuu suoraan alaspäin.

Peli on suurimmaksi osaksi monetisoitu mainoksilla. Ruudun alareunassa on bannerimainos ja pelaajan hävitessä pelin muutaman kerran, näytetään hänelle videomainos. Välillä myös tason läpäisemisen jälkeen näytetään videomainos. Hävitessä pelin, pelaajalla on myös mahdollista jatkaa samasta kohdasta ilman, että hän joutuu aloittamaan koko tason uudestaan katsomalla kokonaisen mainoksen. Pelaaja voi myös katsoa mainoksia ansaitakseen pelin sisäistä valuuttaa ja avatakseen aarrearkkuja. Pelistä on mahdollista ostaa premium-versio, jossa ei ole mainoksia.



KUVIO 1. Kuvankaappaus Helix Jump pelistä.

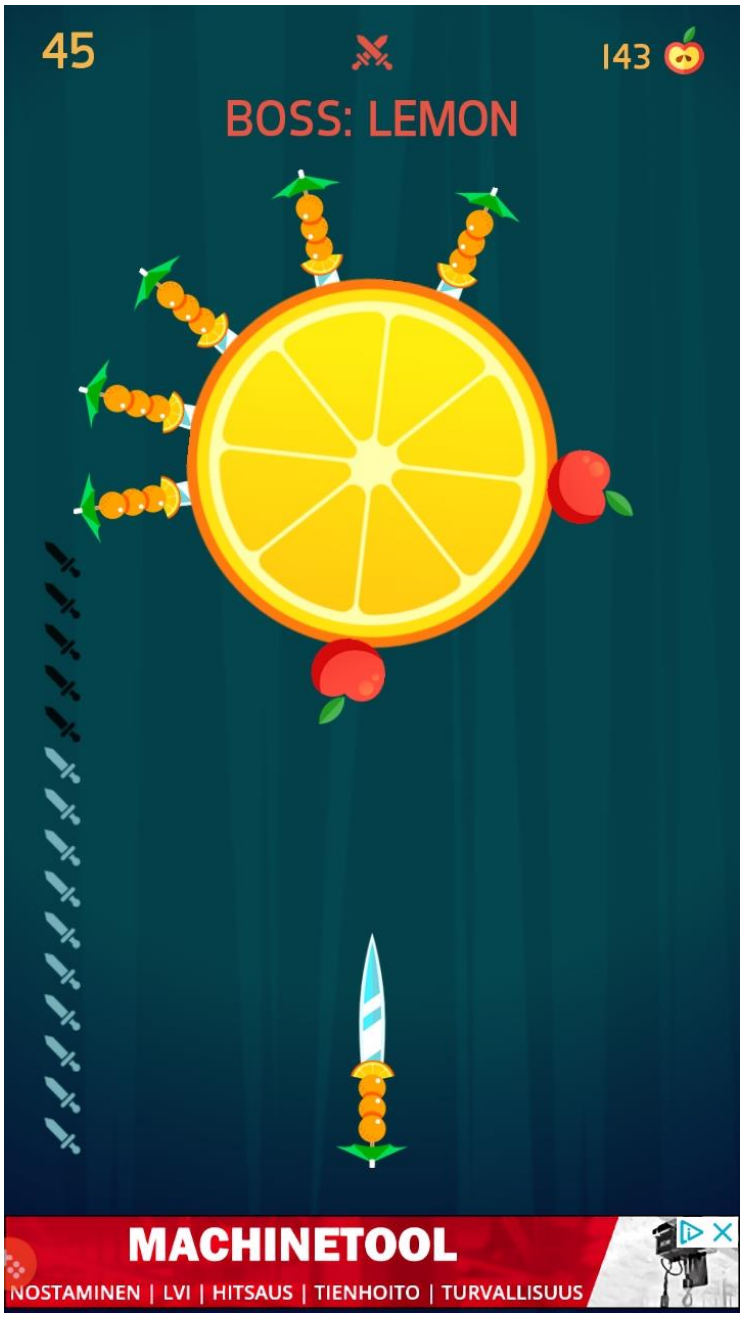
2.2 Knife Hit

Knife Hit on Estoty:n kehittämä ja Ketchapp:n julkaisema peli, jossa pelaajan tehtävänä on tuhota puun runkoja, sekä boss-tasoilla esiintyviä hedelmiä ja muita esineitä heittämällä niihin tietty määrä puukkoja ilman, että heitetty puukko osuu jo kohteessa kiinni olevaan puukkoon. Pelaajan täytyy aloittaa peli alusta, jos hän osuu puukkoihin liian monta kertaa. Tasojen vaikeusaste kasvaa pelissä jokaisella tasolla: heitettävien puukkojen määrä kasvaa, kohteessa jo kiinni olevien puukkojen määrä kasvaa, sekä myös kohteen pyörimisliikkeet muuttuvat arvaamattomimmiksi.

Pelissä voi ansaita ja avata useita erilaisia teräaseita, kuten veitsiä ja miekkoja. Niitä voi avata kuluttamalla pelissä valuuttana käytettyjä omenoita ja päihittämällä boss-tasoja. Pelissä on myös lyhyitä, mutta haastavia haastetasoja, joista voi ansaita kokemuspisteitä, jotka antavat pelaajalle erilaisia power-uppeja. Power-up voi esimerkiksi antaa pelaajalle 5% mahdollisuuden olla menettämättä heittoyrityksen puukon osuessa toiseen puukkoon.

Peli on kontrolleiltaan todella yksinkertainen, pelaaja pystyy ainoastaan painamaan sormellansa ruutua, kun hän haluaa heittää puukon. Pelaajan tulee siis ajoittaa puukon heitto oikein, koska hän ei pysty valitsemaan mihin suuntaan puukko heitetään, vaan puukko heitetään aina ylöspäin keskeltä ruudun alareunaa.

Peli on monetisoitu mainoksilla, maksullisella premium-versiolla, VIP-tilauksella ja pelin sisäisillä ostoilla. Pelissä on aloitusruudussa ruudun alareunassa bannerimainos. Pelaajan hävitessä, näytetään hänelle välillä videomainos ja jos hän pääsi tarpeeksi pitkälle, on hänellä mahdollista katsoa kokonainen mainos ja jatkaa samalta tasolta. Hävityn pelin jälkeen on myös mahdollista katsoa mainos, jolla ansaitaan omenoita. Pelistä on mahdollista ostaa premium-versio, jossa ei ole mainoksia tai tilata viikoittain maksettava VIP-versio, joka on mainokseton, antaa pelaajalle enemmän kokemuspisteitä ja antaa päivittäin enemmän pelissä valuuttana käytettyjä omenoita. Pelissä käytetään oikeaa rahaa erilaisten puukkopakettien ostamiseen.



KUVIO 2. Kuvankaappaus Knife Hit pelistä.

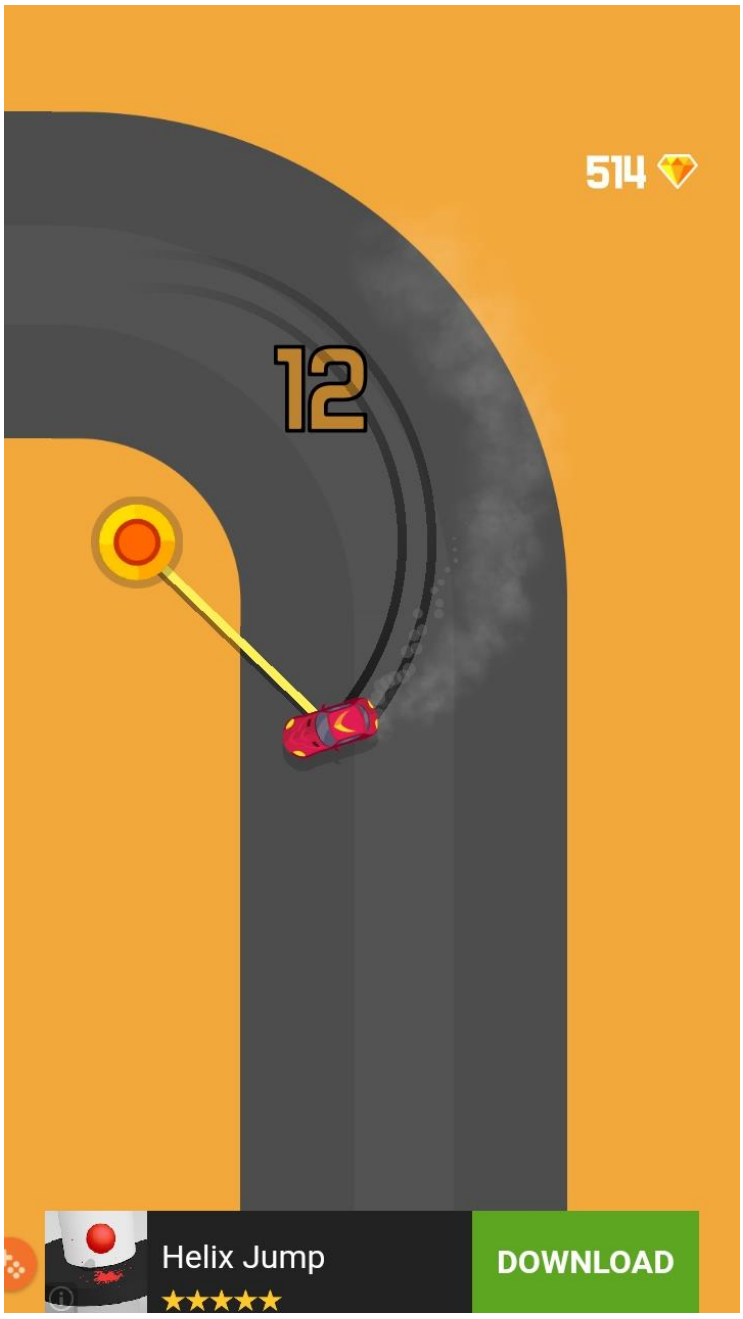
2.3 Sling Drift

Sling Drift on Ruby Games:n kehittämä ja Tastypill:n julkaisema peli, jossa pelaaja driftaa autollansa mutkia loputtomalla radalla. Peli päättyy, kun pelaaja osuu seinään. Vaikka rata on loputon, on se jaettu tietyn väliajoin vaihtuviin tasoihin, jolloin auton nopeus kasvaa, radasta tulee kapeampi ja vastaan alkaa tulla enemmän mutkia.

Pelissä käytetään valuuttana jalokiviä, joita voidaan ansaita läpäisemällä tasoja, pelaamalla peliä päivittäin, suorittamalla erilaisia haasteita ja katsomalla mainoksia. Jalokivillä voidaan ostaa uusia erinäköisiä autoja.

Peli on kontrolleiltaan todella yksinkertainen, pelaaja pystyy ainoastaan painamaan sormellansa ruutua, kun hän on tarpeeksi lähellä mutkaa ja haluaa aloittaa driftaamisen. Pelaaja päästää sormensa ylös, kun hän haluaa lopettaa driftauksen.

Peli on monetisoitu mainoksilla, maksullisella premium-versiolla, sekä myös viikoittain maksettavalla tilauksella. Pelissä on ruudun alareunassa bannerimainos ja pelaajan hävitessä pelin muutaman kerran, näytetään hänelle videomainos. Hävitessään pelin, pelaajalla on mahdollista jatkaa samasta kohdasta ilman, että hän joutuu aloittamaan täysin radan alusta katsomalla kokonaisen mainoksen. Pelistä on mahdollista ostaa premium-versio, jossa ei ole mainoksia tai tilata viikoittain maksettava VIP-versio, joka on mainokseton, sisältää uusia autoja ja antaa pelaajalle enemmän pelissä valuuttana käytettyjä jalokiviä.



KUVIO 3. Kuvankaappaus Sling Drift pelistä.

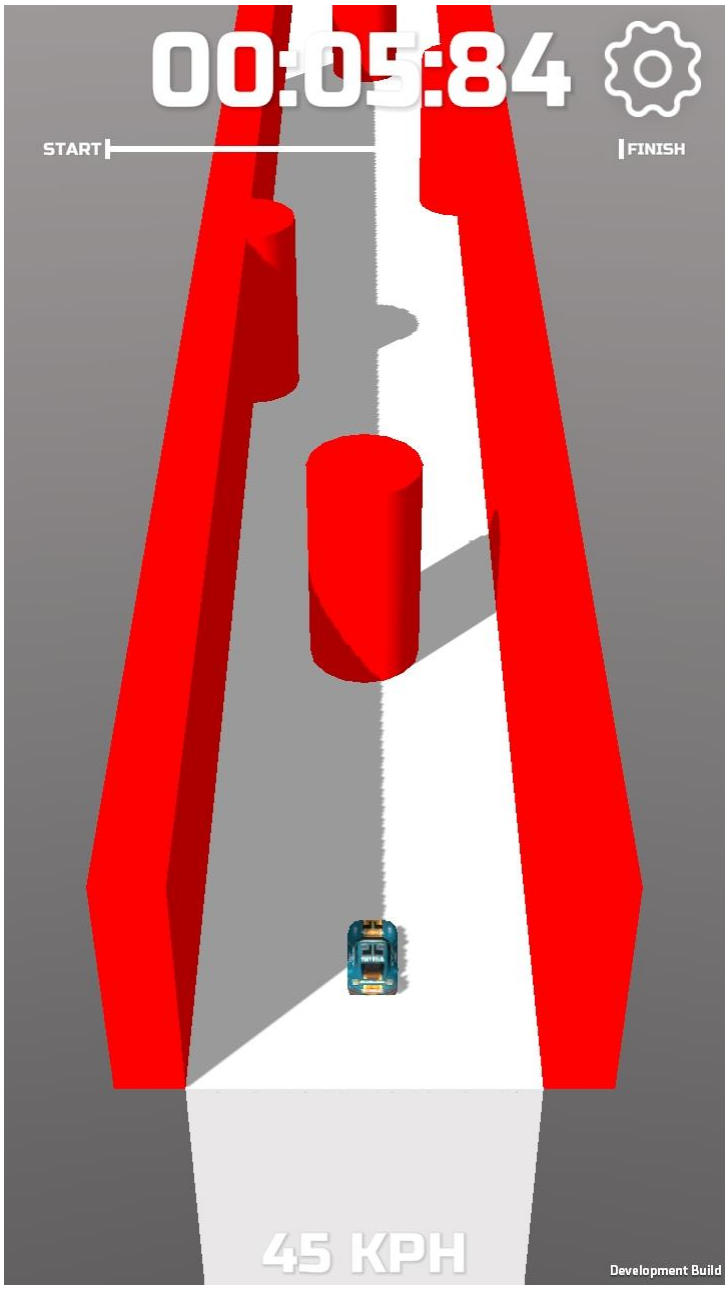
3 DRIVE HARD

Tässä luvussa käydään läpi kehittämäni pelin konsepti, suunnittelu ja itse kehityksen tärkeimmät kohdat. Suunnittelu osiossa kerrotaan, mitä valintoja tehtiin ja perustellaan miksi. Kehitys osiossa kerrotaan, miten peli kehitettiin ja miten kehityksessä esille tulleet ongelmat ratkaistiin.

3.1 Konsepti

Drive Hard on yhden pelaajan mobiililaitteella pelattava peli, jossa pelaajan tarkoituksena on ohjata radio-ohjattavaa autoa radalla, jossa on erilaisia esteitä ja selvitä radan loppuun. Pelaaja joutuu aloittamaan radan uudestaan, jos hän tippuu radalta ulos. Radalla esiintyviä esteitä ovat esimerkiksi kimmottavat seinät, rotkot, rampit ja liikkuvat tasot. Radan lopussa näytetään pelaajalle hänen saamansa pisteet, aika ja yrityksien määrä. Pelin inspiraationa toimivat runner-pelit ja Trackmania- ja Trials-pelisarjat.

Pelin on suunniteltu sisältävän uusia avattavia RC-autoja ja niille avattavia vaihtoehtoisia ulkoasuja. Uusia autoja ja ulkoasuja voidaan avata käyttämällä pelin sisäistä valuuttaa. Erikoisempia autoja voidaan avata suorittamalla pelissä olevia haasteita ja saavutuksia. Pelin sisäistä valuuttaa ansaitaan läpäisemällä tasoja.



KUVIO 4. Kuvankaappaus kehitetystä pelistä.

3.2 Suunnittelu

Pelisuunnittelun avuksi voidaan tehdä pelisuunnitteluasiakirja, eli GDD. Se on pelin kehitystiimin tekemä dokumentti, jossa kerrotaan selkeästi pelin myyntivaltit, kohdeyleisö, gameplay, visuaalinen tyyli ja niin edelleen. Sen tarkoitus on siis pitää pelin visio selkeänä koko kehitystiimin sisällä. (GameDesigning.org 2019, viitattu 4.12.2019.) Joskus myös julkaisijat voivat haluta nähdä julkaistavan pelin GDD:n.

Päätin jättää GDD:n tekemisen pois tästä projektista, koska tein projektin yksin ja halusin käyttää kaiken ylimääräisen ajan itse pelin kehitykseen. GDD:n tekeminen olisi tosin voinut selventää tehtyjä ja pitää pelisuunnittelun selkeänä.

Peli on kontrolleiltaan yksinkertainen: pelaaja pystyy ainoastaan ohjaamaan auton suuntaa painamalla sormella jompaankumpaan ruudun reunaan. Pelaaja ei pysty vaikuttamaan auton nopeuteen, vaan auto kulkee radalla automaattisesti eteenpäin pelaajan autolle määritetyllä nopeudella. Pelaaja voi halutessaan käyttää vaihtoehtoista ohjaustapaa, jossa pelaaja pystyy osoittamaan sormellansa suunnan, johon auto pyrkii kulkemaan.

Peli on alustavasti suunniteltu olemaan visuaaliselta tyyliltään yksinkertainen: radat on rakennettu abstrakteista muodoista ja ne käyttävät suurimmaksi osaksi ainoastaan perusvärejä tai yksinkertaisia tekstuureja. Pelaajan auto on low-poly 3D-malli, jossa käytetään myös yksinkertaista tekstuuria.

3.2.1 Tasot

Mobiilipeleissä voidaan käyttää joko tasoja, joissa on alku ja loppu tai loputonta tasoa, joka aloitetaan jokaisella pelikerralla alusta. Varsinkin hyperkasuaaleissa peleissä on tyypillistä, että tasot luodaan proseduraalisesti, eli ne generoidaan esimerkiksi osista noudattaen jonkinlaisia luomiselle asetettuja sääntöjä. Proseduraalista generointia käytetään, koska tasojen ei haluta loppuvan kesken ja niiden luominen käsin veisi todella paljon aikaa.

Halusin käyttää pelissä tasoja, joissa on alku ja loppu, koska ne sopivat mielestäni paremmin tälle pelille. Halusin myös, että ne luodaan proseduraalisesti osista, koska en halunnut, että tasot loppuvat kesken. Luotujen tasojen vaikeusaste kasvaa tason numeron mukaan: tasojen pituus kasvaa, kun niiden rakentamiseen käytetään enemmän osia ja rakentamisessa aletaan käyttää osia, joissa on vaikeampia esteitä. Esteitä voivat esimerkiksi olla ramppeja, osien sisällä tai kahden osan välillä olevia rotkoja tai seiniä, jotka kimmottavat auton johonkin toiseen suuntaan.

3.2.2 Käyttöliittymä

Hyperkasuaaleissa peleissä on tärkeää, että pelaaja pääsee aloittamaan pelaamisen mahdollisimman nopeaa. Tämän takia pelissä ei ole latausruutua päämenun ja itse pelin välillä, vaan se ainoastaan piilotetaan, kun pelaaja aloittaa pelin. Pelin ainoa latausruutu on ennen päämenua, jonka aikana ladataan pelaajan tallennukset ja asetukset tiedostosta, yhdistetään Google Play -palveluun ja initialisoidaan AdMob-mainokset. Latausruutu olisi pakollinen päämenun ja itse pelin välissä, jos pelissä olisi jotain isoa, joka täytyy ladata, ennen kuin peli voidaan aloittaa.

Päämenussa pelaaja voi aloittaa pelin, muuttaa pelin asetuksia, kustomoida RC-autoa ja tarkastella pelin uusimman päivityksen sisältämiä muutoksia. Päämenussa voidaan myös näyttää pelaajalle bannerimainos ruudun alareunassa.

Asetuksissa pelaaja voi muokata pelin:

- Yleisiä-asetuksia, kuten muuttaa auton ohjaustapaa, nollata kaikki asetukset ja poistaa pelaajan tallennukset.
- Graafiikka-asetuksia, kuten muuttaa resoluutiota, ottaa post-processing-efektit käyttöön, ottaa reunanpehmennyksen käyttöön ja rajoittaa ruudunpäivityksen 30 tai 60 FPS:ään.
- Ääni-asetuksia, kuten pelin master-, musiikin ja ääniefektien äänenvoimakkuutta.

Aloitettaessa pelin, pelaajalle näytetään aloituslaskentanäkymä, jonka jälkeen peli alkaa ja pelaaja voi alkaa ohjata autoa. Pelin aikana HUD:ssa näytetään pelaajan keräämät pisteet, kulunut aika ja auton nopeus. Peli voidaan tauottaa painamalla HUD:ssa olevaa hammasrataspainiketta. Tauko ruudusta pelaaja voi poistua takaisin päämenuun tai jatkaa pelaamista. Radan lopussa pelaajalle näytetään tason tulokset, kuten nykyisen tason numero, yritykset, kerätyt pisteet ja käytetty aika. Pelaaja pystyy jatkamaan tästä ruudusta seuraavaan tasoon tai palaamaan takaisin päämenuun.

3.2.3 Monetisointi

Mobiilipelejä voidaan monetisoida muutamalla eri tavalla: mainoksilla, premium-versiolla ja pelin sisäisillä ostoilla (ironSource 2018, viitattu 26.4.2019).

Yleisiä mainoksien sijainteja ovat main menussa oleva bannerimainos, pelin jälkeinen videomainos ja joskus myös pelin aikana näytettävä bannerimainos. Mainoksia käytettäessä on tärkeää suunnitella, ja myös testata, milloin niitä kannattaa näyttää ja kuinka usein, koska ne voivat haitata pelaajan pelikokemusta ja aiheuttaa ärsyyntymistä. Mainoksia voisi myös mahdollisesti sijoittaa pelin sisään esimerkiksi mainostaulun muodossa niin, että ne eivät vaikuta pelaajan kokemukseen liikaa (Hadley 2019, viitattu 4.12.2019).

Monista mobiilipeleistä on olemassa ilmainen mainoksellinen peli ja myös maksullinen premium-versio. Premium-version ostaminen antaa pelaajalle yleensä muutamia hyötyjä, kuten esimerkiksi mainoksia ei näytetä ja mahdollisesti myös pelin sisäistä valuuttaa.

Pelin sisäisillä ostoilla pelaaja voi ostaa pelissä käytettävää valuuttaa, jota pelaaja voi käyttää uusien pelattavien hahmojen avaamiseen tai muiden kustomointi valintojen avaamiseen. Pelistä riippuen on myös mahdollista, että pelaaja voi suoraan ostaa uusia kustomointi valintoja.

Tämä peli on alustavasti suunniteltu monetisoitavaksi ilmaisessa versiossa olevilla mainoksilla ja mainoksettomalla premium-versiolla. Pelissä näytetään bannerimainos pelin päämenussa ja videomainos, kun pelaaja läpäisee tason ja viime näyttö kerrasta on kulunut tarpeeksi kauan. Pelin aikana ei näytetä mainoksia, koska ne haittaavat mielestäni pelaajan kokemusta liikaa ja koska pelikerrat eivät ole kovinkaan pitkiä.

3.2.4 Analytiikka

Monissa mobiilipeleissä käytetään analytiikkaa pelaajan käyttökokemuksen parantelemisen apuna ja samalla käyttäjien käyttäytymisen seurantaan. Analytiikan avulla kehittäjä voi lisätä peliin erilaisia tapahtumia, joilla kerätään tietoa käyttäjien käyttäytymisestä pelin sisällä. Kerättyä tietoa voi olla esimerkiksi:

- Miten pitkään pelaajat pelaavat.
- Milloin he pelaavat.
- Kuinka usein he pelaavat.
- Miten he käyttävät rahaa pelin sisällä.
- Katsovatko he mainoksia.
- Suorittavatko pelaajat alku tutoriaalin kokonaan tai jäävätkö he jumiin johonkin tasoon.

- Toimiiko peli vakaasti.

Kerättyä analytiikkaa voidaan käyttää apuna pelin Day 1-retention parantamiseen, joka on ainakin monille julkaisijoille tärkeä metriikka. Osasyynä huonoon Day 1-retentioon voi olla esimerkiksi huonosti toteutettu tutoriaali, epävakaa peli tai huonosti toteutettu mainonta (Roseboom 2015, viitattu 4.12.2019). Jos pelissä on pelinsisäinen kauppa, saadaan sen avulla selville käyttävätkö pelaajat rahaa pelin sisällä ja mihin he sitä käyttävät. Tämän avulla kehittäjä voi parantaa kaupan tehokkuutta.

Unity:lle tarkoitettuja analytiikkapalveluita tarjoaa muutama eri tarjoaja, kuten esimerkiksi: Unity, Google ja Facebook. Yleensä palvelun käyttöönotto Unity-projektissa tapahtuu lisäämällä projektiin tarjoajan SDK-paketin, lisäämällä pelin palveluntarjoajan sivulla olevaan dashboardiin ja lisäämällä peliin tapahtumat, joita kehittäjä haluaa seurata.

Päätin jättää tässä projektissa analytiikan pois, koska aikaa kehittämiseen oli todella rajoitetusti ja analytiikan voi lisätä peliin helposti myöhemminkin.

3.3 Kehittäminen

Itse pelin kehittämiseen kuuluu monta eri osaa, mutta tässä projektissa niistä tärkeimmät osa-alueet ovat pelaajan auto, käyttöliittymä ja tietojen tallennus. Pelimoottoriksi valitsin Unity-pelimoottorin, koska minulla on sen käyttämisestä paljon kokemusta ja koska se sopii erinomaisesti nopeaan mobiilipelien kehittämiseen.

Käytin projektissa apuna muutamaa eri Unitylle tehtyä pelinkehitystä helpottavaa pakettia:

- Kinematic Character Controller. Se on maksullinen paketti, joka on luotu erityisesti ihmishahmojen liikuttamiseen 3D- ja 2.5D projekteissa. KCC:n tehtävä on liikuttaa hahmoa haluttuun suuntaan ja ratkaista mitä tapahtuu, jos hahmo osuu, esimerkiksi seinään. Se on luotu korvaamaan Unityn oma Character Controller-komponentti. (St-Amand 2019, viitattu 4.12.2019.)
- DoozyUI. Se on maksullinen käyttöliittymän toteutukseen erikoistunut paketti. Tämä paketti helpottaa eri näkymien hallintaa, animaatioiden ja ääniefektien lisäämistä käyttöliittymään sekä paljon muuta. (Doozy Entertainment 2019a, viitattu 4.12.2019.)

- Easy Mobile Pro. Se on maksullinen paketti, joka on luotu helpottamaan monia yleisimmin mobiilipeleissä käytettyjen ominaisuuksien implementointia. Sen avulla pystyy yksinkertaisesti ottamaan käyttöön pelissä mainokset, iOS- ja Google-palvelut, pelin sisäiset ostokset, ilmoitukset, jakamisen ja muita mobiililaitteiden ominaisuuksia. (SgLib Games 2019, viitattu 4.12.2019.)
- TouchKit. Se on ilmainen paketti, joka helpottaa mobiililaitteilla käytettyjen pelaajan kosketuksien ja eleiden tunnistamista (Prime31 2019, viitattu 4.12.2019).
- OdinSerializer. Se on ilmainen paketti, joka tekee monimutkaistenkin objektien muuntamisesta JSON- tai binäärimuotoon ja takaisin paljon yksinkertaisempaa (Sirenix 2019, viitattu 4.12.2019).
- Lunar Mobile Console. Se on ilmainen paketti, jonka avulla pystyy näkemään Unityn konsolin virheet, varoitukset ja logit, kun peliä testataan oikealla mobiililaitteella (SpaceMadness 2019, viitattu 4.12.2019).

Käytin Unity-projektin pohjana Lightweight Render Pipeline-pohjaa, koska se parantaa pelien suorituskykyä mobiililaitteilla, kun halutaan käyttää 3D-grafiikkaa (Cooper 2018a, viitattu 4.12.2019). Toisena syynä on ainoastaan HDRP- ja LWRP-pohjien kanssa toimiva Unityn Shader Graph-työkalu, joka mahdollistaa varjostimien tekemisen visuaalisien kaavioiden ja nodejen avulla ilman ohjelmointia (Cooper 2018b, viitattu 4.12.2019). Itse ohjelmoinnissa käytin JetBrains Rider-ohjelmointiympäristöä.

3.3.1 Käyttöliittymä

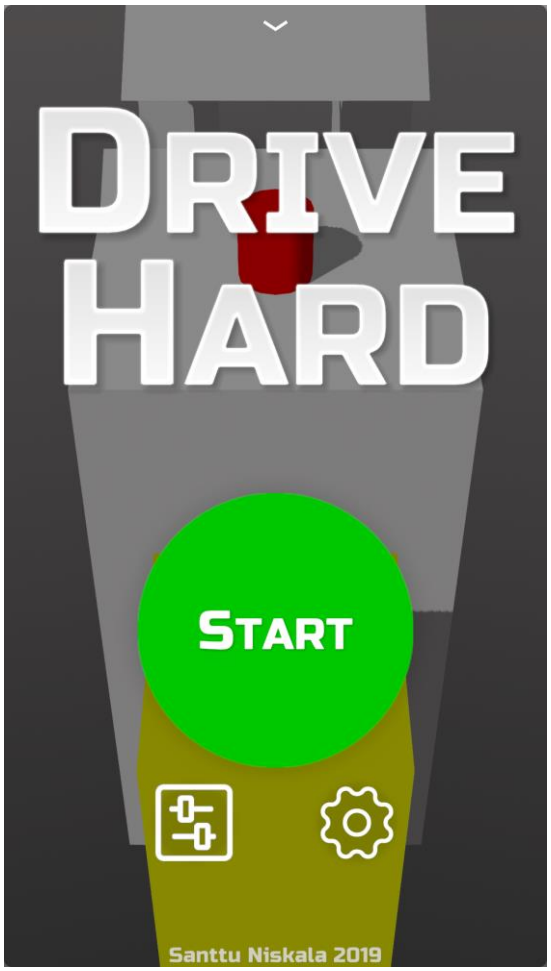
Käyttöliittymän toteutuksessa käytin apuna DoozyUI-pakettia, joka vähentää huomattavasti tarvittavan ohjelmoinnin määrää käyttöliittymää toteutettaessa. Sen avulla on helppo luoda monimutkaisiakin käyttöliittymiä ja lisätä niihin erilaisia animaatioita ja ääniefektejä.

DoozyUI:n mukana tulee muutamia erilaisia käyttöliittymän kanssa käytettäviä ääniefektejä ja musiikkia, animaatio-presettejä ja grafiikkaa. Sen mukana tulee myös korvaavat komponentit osalle Unityn UI-komponenteista, joiden avulla niihin saadaan muutamia lisäominaisuuksia (Doozy Entertainment 2019a, viitattu 4.12.2019). Sen mukana tulee myös kolme hyödyllistä työkalua:

- Nody-kaaviotyökalu, jolla tehdään näkymien hallintaan käytettyjä kaavioita (Doozy Entertainment 2019b, viitattu 4.12.2019).

- Soundy-äänimanageri, jota voidaan käyttää käyttöliittymän tai pelin ääniefektien ja musiikin soittamiseen (Doozy Entertainment 2019b, viitattu 4.12.2019).
- Touchy-kosketusmanageri, jota voidaan käyttää kosketuksien ja hiiren klikkauksien tunnistamiseen (Doozy Entertainment 2019b, viitattu 4.12.2019).

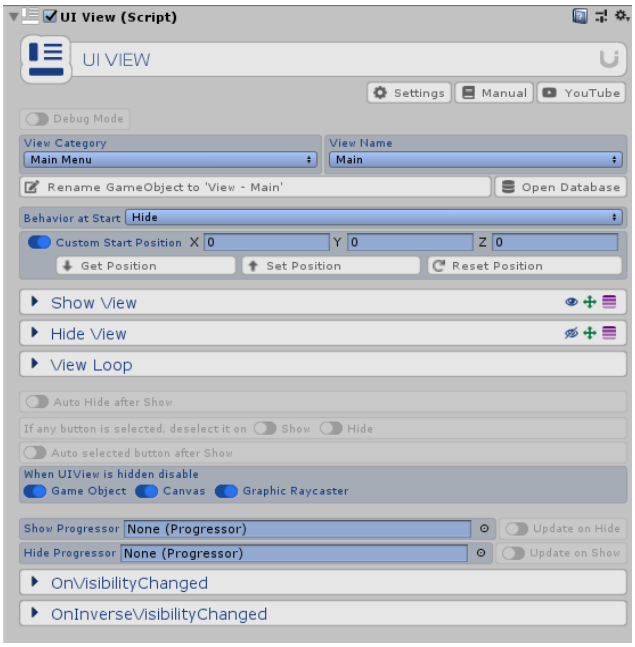
Käyttöliittymän tekemiseen tarvitaan vähintään DoozyUI:n UI Canvas, UI View ja Graph Controller-objektit. UI View-objektit, eli näkymät lisätään UI Canvas-objektin alle, joka näyttää näkymät ruudulla. Näkymien näyttämistä ja piilottamista ohjataan Graph Controllerille luodun kaavion avulla. Kaavio luodaan Nody-työkalulla modulaarisista Node-komponenteista.



KUVIO 5. Pelin päämenu.

Näkymän UI View-komponentissa voidaan hallita näkymän tärkeitä ominaisuuksia, kuten näytetäänkö se pelin alkaessa vai piilotetaanko se, hallita näyttämisessä ja piilottamisessa käytettyjä animaatioita ja lisätä animaatioiden alussa tai lopussa suoritettavia Unity-eventtejä (KUVIO 6). Nä-

kymän lapsiobjektiksi voidaan lisätä näkymässä käytetyt painikkeet, tekstit ja kuvat. Tässä projektissa tehtiin omat näkymät päämenulle, asetuksille, HUD:lle, taukomenulle sekä myös tason alulle, lopulle ja epäonnistumiselle.



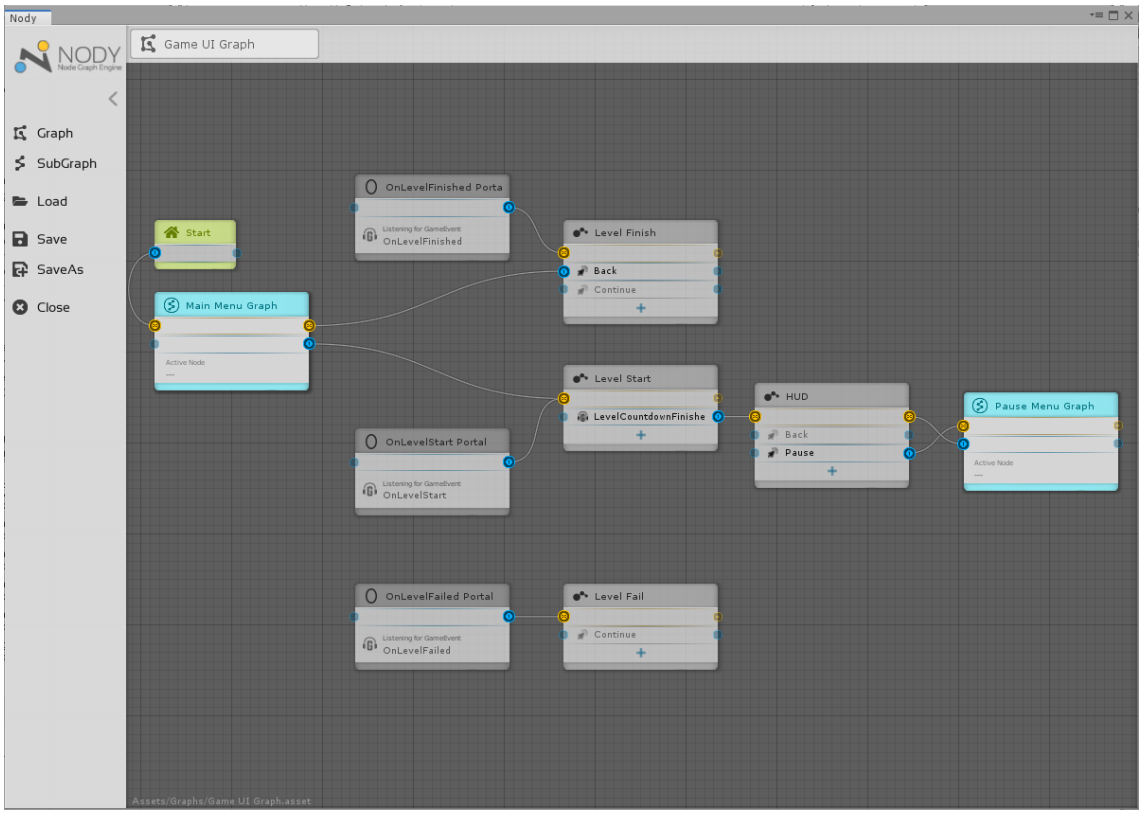
KUVIO 6. DoozyUI:n UI View-komponentti.

Käyttöliittymän näkymien ohjaamisen apuna käytetään abstraktista AbstractViewController-luokasta periytyviä luokkia, joiden tehtävä on ohjata näkymien toimintoja, kuten esimerkiksi niissä olevia painikkeita ja tekstejä. Painikkeiden toiminnot voitaisiin myös lisätä suoraan painikkeisiin inspectorissa, mutta itse lisään ne mieluummin koodin kautta. Abstraktilla ViewController-luokalla on ylikirjoitettavat metodit Setup, OnShowStart, OnShowFinish, OnHideStart ja OnHideFinish. Nämä voidaan ylikirjoittaa, jos halutaan suorittaa jotain esimerkiksi silloin, kun näkymää aletaan näyttää, kun näkymää aletaan piilottamaan tai kun näkymä on kokonaan näytetty tai, kun sen on kokonaan piilotettu (KUVIO 7).

```
public class AbstractViewController : MonoBehaviour
{
    [Header("View Controller")]
    public UIView UiView;  Unchanged

     Event function
    private void Awake()
    {
        Setup();
    }

     Event handler  No asset usages  9 usages  8 overrides
    public virtual void Setup()
    {
        UiView.ShowBehavior.OnStart.Event.AddListener(OnShowStart);
        UiView.ShowBehavior.OnFinished.Event.AddListener(OnShowFinish);


        UiView.HideBehavior.OnStart.Event.AddListener(OnHideStart);
        UiView.HideBehavior.OnFinished.Event.AddListener(OnHideFinish);
    }

     Event function  7 usages  7 overrides
    protected virtual void OnDestroy()
    {
        UiView.ShowBehavior.OnStart.Event.RemoveListener(OnShowStart);
        UiView.ShowBehavior.OnFinished.Event.RemoveListener(OnShowFinish);


        UiView.HideBehavior.OnStart.Event.RemoveListener(OnHideStart);
        UiView.HideBehavior.OnFinished.Event.RemoveListener(OnHideFinish);
    }

     4 usages  2 overrides
    protected virtual void OnShowFinish()
    {

    }

     9 usages  7 overrides
    protected virtual void OnShowStart()
    {

    }

     3 usages  1 override
    protected virtual void OnHideStart()
    {

    }

     3 usages  1 override
    protected virtual void OnHideFinish()
    {

    }
}
```

KUVIO 7. Abstrakti ViewController-luokka.

Näkymien näyttämistä ja piilottamista hallitaan Nody-työkalun avulla tehdyillä kaavioilla. Kaavioiden sisälle lisätään UI Nodeja, joihin voidaan määrittää mitä näkymiä piilotetaan ja mitä näytetään, kun nodeen siirrytään ja kun siitä poistutaan (KUVIO 8). UI Nodejen välille vedetään yhteyksiä, jotka määrittävät mihin nodeen siirrytään, kun esimerkiksi jotain näkymän painiketta painetaan. Nody-työkalussa on myös nodeja, joilla voidaan suorittaa kaavion sisällä muita toimintoja, kuten lähettää eventtejä, soittaa ääniä, ladata scenejä ja sulkea pelin. Kaavioiden sisällä voi myös olla alikaavioita, jotka helpottavat monimutkaisten kaavioiden hallintaa. Esimerkiksi pääkaavion alla voi olla oma alikaavio Main Menulle, Settings Menulle ja HUD:lle.



KUVIO 8. Nody-työkalulla tehty pelin käyttöliittymäkaavio.

3.3.2 Tallennukset

Pelissä on muutamia tärkeitä arvoja, jotka tulee voida palauttaa, kun pelaaja sulkee pelin ja avaa sen uudestaan seuraavan kerran. Tällaisia arvoja ovat esimerkiksi pelaajan viimeksi läpäisemän tason numero ja pelaajan asettamat asetukset. Pelin asetukset tallennetaan eriin tiedostoon, kuin pelaajan tiedot. Tallennettuja asetuksia ovat yleiset-, grafiikka- ja ääniasetukset.

Jotta tiedot voidaan tallentaa, on niille tehtävä ensin luokka, jossa kaikille tallennettaville arvoille on oma muuttuja. Tässä tapauksessa luokkia on kaksi: GamePlayerData ja GameSettingsData (KUVIO 9). Muuttujat voidaan laittaa suoraan luokkaan tai jos muuttujia on paljon, voidaan ne myös laittaa erillisiin strukteihin.

```
[Serializable]
7 usages  2 exposing APIs
public class GameSettingsData
{
    public GeneralSettings GeneralSettings;  Serializable
    public DisplaySettings DisplaySettings;  Serializable
    public AudioSettings AudioSettings;  Serializable

    1 usage
    public GameSettingsData()
    {
        GeneralSettings = new GeneralSettings();
        DisplaySettings = new DisplaySettings();
        AudioSettings = new AudioSettings( masterVolume: 1f, sfxVolume: 1f, uiVolume: 1f, musicVolume: 1f);
    }
}

[Serializable]
7 usages  2 exposing APIs
public class GamePlayerData
{
    public int RandomSeed;  Serializable
    public int CurrentLevel;  Serializable

    1 usage
    public GamePlayerData()
    {
        RandomSeed = 1337;
        CurrentLevel = 1;
    }
}
```

KUVIO 9. Pelaajan ja asetuksien Data-luokat.

Luokkien lukemiseen ja kirjoittamiseen käytin avointa OdinSerializer-pakettia, koska se tekee monimutkaistenkin objektien muuntamisesta JSON- tai binäärimuotoon paljon yksinkertaisempaa. Luokka tallennetaan tiedostoon geneerisellä SerializeData-metodilla, joka ottaa parametriksi tallennetavan datan luokan ja sen tallentamiseen asetetut asetukset. Metodissa tarkistetaan aluksi, että tallennuskansio on olemassa. Tallennettava luokka muutetaan lopuksi tavuryhmäksi OdinSerializerin avulla, jonka jälkeen se voidaan tallentaa tiedostona haluttuun kansioon (KUVIO 10).

```
//Save to file
4 usages
private void SerializeData<T>(T data, SaveDataSettings saveDataSettings)
{
    //
    var path = SaveConfig.GetSavePath();
    var fileName = saveDataSettings.FileName + SaveConfig.SavePathFormat;

    if (!Directory.Exists(path))
        Directory.CreateDirectory(path);

    var bytes = SerializationUtility.SerializeValue(data, SaveConfig.DataFormat);
    File.WriteAllBytes( path: path + fileName, bytes);
}
```

KUVIO 10. Tiedoston tallentaminen.

Kun luokka on onnistuneesti tallennettu tiedostona kansioon, voidaan se lukea, eli ladata OdinSerializerin avulla seuraavalla käynnistyskerralla ja palauttaa tallennetut arvot. Luokka ladataan tiedostosta geneerisellä DeserializeData-metodilla, joka ottaa parametriksi tyhjän tallennettavan datan luokan ja sen tallentamiseen käytetyt asetukset. Metodissa tarkistetaan aluksi, että tiedosto on olemassa ja palautetaan takaisin tyhjä data luokka, jos sitä ei ole olemassa. Jos tiedosto on olemassa, luetaan se aluksi tavuryhmäksi ja muunnetaan sitten takaisin luokaksi OdinSerializerin avulla (KUVIO 11).

```
//Load from file
2 usages
private T DeserializeData<T>(T data, SaveDataSettings saveDataSettings)
{
    var filePath = SaveConfig.GetFullSavePath(saveDataSettings);

    if (!File.Exists(filePath))
    {
        Debug.LogWarning( message: "Save Manager: Can't Load Save For '" + saveDataSettings.FileName + "', Because No Data Was Found!");
        return data;
    }

    byte[] bytes = File.ReadAllBytes(filePath);
    data = SerializationUtility.DeserializeValue<T>(bytes, SaveConfig.DataFormat);
    return data;
}
```

KUVIO 11. Tiedoston lataaminen.

Tästä pelistä puuttuu vielä yksi tärkeä ominaisuus: migraatio. Sen avulla varmistetaan, että tallennuksien lukemisessa ja kirjoittamisessa ei tule ongelmia, kun peliä päivitetään ja halutaan esimerkiksi lisätä tallennettavien tietojen määrää. Ilman migraatiota on mahdollista, että päivityksen jälkeen pelaajan tallennusta ei voida enää ladata onnistuneesti ja pelaaja joutuu aloittamaan koko pelin alusta.

3.3.3 Pelaajan Auto

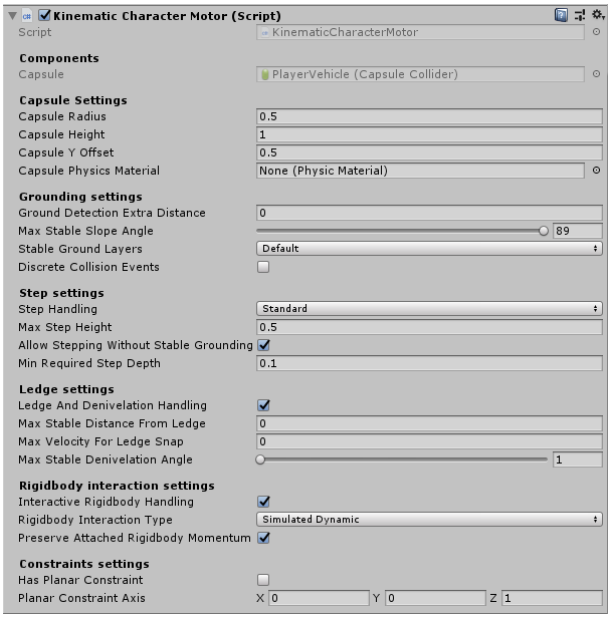
Auton fysiikat voi toteuttaa ainakin kahdella eri tavalla: käyttämällä apuna Unityn mukana tulevaa fysiikkamoottoria ja liikuttamalla autoa voimilla tai manuaalisesti liikuttamalla objektia esimerkiksi Transform.Translate-metodilla ja ratkaisemalla osumat itse. Kummallakin toteutustavalla on omat vahvuutensa ja huonot puolensa:

Rigidbody-fysiikoita käytettäessä autoa ohjattaisiin voimilla, joka tekee liikkeen tarkasta hallitsemisesta todella vaikeaa. Tämän takia myös ajettavuus on todella vaikea saada hyväksi. Hyvänä puolena oikeita fysiikoita käytettäessä on se, että auto voisi automaattisesti vuorovaikuttaa muiden rigidbody-objektien ja voimien kanssa ja auton collision-muotona voitaisiin käyttää melkein mitä tahansa muotoa.

Manuaalisesti liikuttaessa liikkeen hallitseminen on todella helppoa, mutta osumat pitää ratkaista itse. Osumien ratkaisemisella tarkoitetaan, sitä miten objekti reagoi, kun se osuu seinään tai toiseen liikkuvaan objektiin. Tästä voi tulla todella monimutkaista, varsinkin silloin, jos halutaan käyttää jotain erikoisempaa collision-muotoja, kuin esimerkiksi laatikkoa tai palloa. Myös interaktio rigidbody-objektien kanssa tulee hoitaa itse.

Koska fysiikkamoottorin käyttämiseen ja oman ratkaisun kehittämiseen olisi mennyt todella paljon aikaa, päätin käyttää apuna Kinematic Character Controller-pakettia. Kinematic Character Controller on matalan tason (engl. low-level) hahmojen hallitsemiseen käytetty paketti, jota voi käyttää minkälaisessa 3D- tai 2.5D-projektissa tahansa. Se ei käytä rigidbody-fysiikoita, vaan se käyttää ”collide and slide”-algoritmia, joka mahdollistaa todella sulavan, tarkan ja responsiivisen liikkeen. Tämän takia se ei pysty automaattisesti vuorovaikuttamaan rigidbody-objektien tai muiden voimien kanssa, vaan niiden väliset interaktiot täytyy ohjelmoida itse. (St-Amand 2019, viitattu 4.12.2019.)

KCC:n käyttämiseen tarvitaan vähintään kahta hahmo-objektiin lisättävää komponenttia: KinematicCharacterMotor-komponentti (KUVIO 12) ja hahmolle luodun ICharacterController-rajapintaluokan implementoivan controller-skriptin. Motor-komponentti hoitaa itse hahmon liikuttamisen ja osumien ratkaisemisen. Controller-skriptillä kerrotaan motor-komponentille mihin suuntaan ja kuinka nopeaa hahmoa halutaan liikuttaa tai pyörittää. Kontrolleri-skriptiin määritetään siis, miten hahmo liikkuu ja mahdollisesti myös mitä eri kykyjä hahmolla on. Tällaisia kykyjä voi olla esimerkiksi hyppääminen, uiminen, syöksyminen tai tikkaiden kiipeäminen.



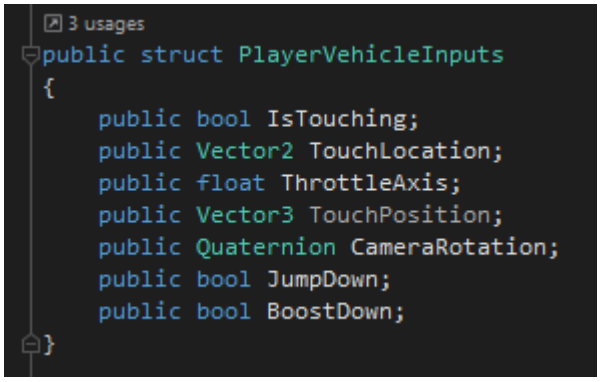
KUVIO 12. Auton motor-komponentti Unityn inspector-näkymässä.

Tässä projektissa käytettiin pelaajan auton ohjaamiseen KinematicCharacterMotor- ja VehicleCharacterController-skriptin lisäksi VehicleCharacterPlayer-skriptiä. Player-skriptin ainoa tehtävä on päivittää pelaajan syötteet ja lähettää ne kontrollerille. Pelaajan kosketuksen tunnistamiseen käytetään apuna ilmaista TouchKit-pakettia, joka helpottaa erilaisten mobiilipeleissä käytettyjen kosketuksien tunnistamista.

Pelaajan syötteet päivitetään jokaisella ruudunpäivityksellä Update-metodissa. HandlePlayerInput-metodissa tunnistetaan koskettaako pelaaja tällä hetkellä ruutua ja mihin kohtaan ruutua hän koskettaa (KUVIO 13). Kosketuksen sijainnin sisältävän Vector2-vektorin x ja y arvot muunnetaan luvuiksi, jotka ovat välillä 0.0f ja 1.0f, jolloin x:0.0f ja y:0.0f tarkoittaa, että pelaaja koskettaa ruudun vasenta alareunaa ja x:1.0f ja y:1.0f tarkoittaa, että pelaaja koskettaa ruudun oikeaa yläreunaa. Pelaajan syötteet kerätään PlayerVehicleInputs-struktiin (KUVIO 14), joka lähetetään lopuksi VehicleCharacterController-skriptiin sen SetInputs-metodia käyttäen.

```
Event function
private void Update()
{
    HandlePlayerInput();
}

Frequently called  1 usage
private void HandlePlayerInput()
{
    PlayerVehicleInputs vehicleInputs = new PlayerVehicleInputs();
    TouchKit.updateTouches();

    if (_allowInput)
    {
        var touchLocationNormalized = new Vector2(
            x: Mathf.Clamp01(_touchRecognizer.touchLocation().x / Screen.width),
            y: Mathf.Clamp01(_touchRecognizer.touchLocation().y / Screen.height)
            );

        vehicleInputs.IsTouching = _touchRecognizer.state == TKGestureRecognizerState.RecognizedAndStillRecognizing;
        vehicleInputs.TouchLocation = touchLocationNormalized;
        vehicleInputs.ThrottleAxis = 1f;

        //For alternative control option
        vehicleInputs.TouchPosition = GetTouchPoint(vehicleInputs.IsTouching, touchPosition: _touchRecognizer.touchLocation());
        //For moving vehicle relative to camera direction. Currently not used
        vehicleInputs.CameraRotation = Quaternion.identity;//CameraController.transform.rotation;
        //For testing
        vehicleInputs.JumpDown = Input.GetKeyDown(KeyCode.Space);
        vehicleInputs.BoostDown = Input.GetKeyDown(KeyCode.C);
    }
    // Apply inputs to character
    vehicleCharacter.SetInputs(ref vehicleInputs);
}
```

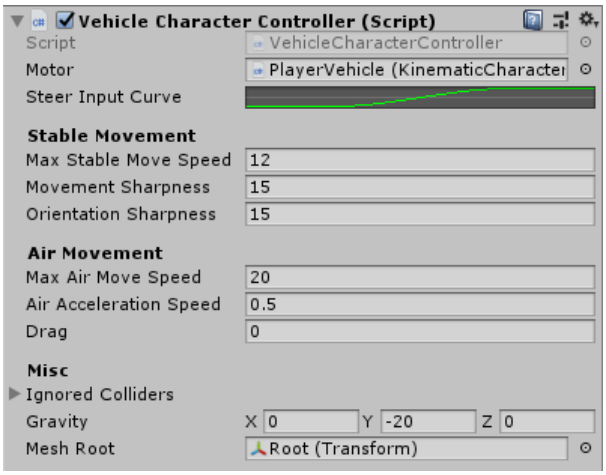
KUVIO 13. VehicleCharacterPlayer-luokan pelaajan syötteet käsittelevä metodi.



```
3 usages
public struct PlayerVehicleInputs
{
    public bool IsTouching;
    public Vector2 TouchLocation;
    public float ThrottleAxis;
    public Vector3 TouchPosition;
    public Quaternion CameraRotation;
    public bool JumpDown;
    public bool BoostDown;
}
```

KUVIO 14. Pelaajan syötteet sisältävä strukti.

Auton controller-skriptissä määritetään, kuinka auto liikkuu, kuinka nopeaa se liikkuu ja kuinka nopeaa se reagoi suunnan muutoksiin. Siinä käsitellään myös VehicleCharacterPlayer-luokan lähettämät pelaajan syötteet ja määritetään miten ne ohjaavat autoa.

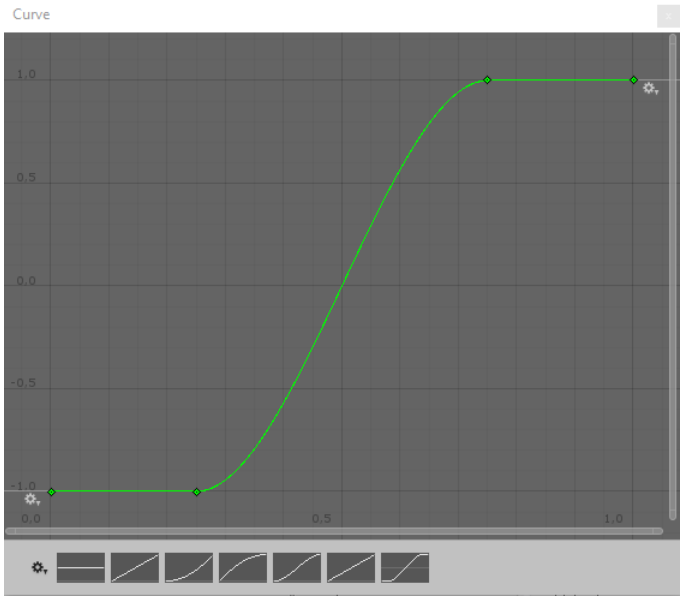


KUVIO 15. Auton controller-komponentti Unityn inspector-näkymässä.

Auton ohjaaminen alkaa SetInputs-metodista, joka käsittelee pelaajan syötteet ja muuntaa ne suunnaksi, johon auton halutaan liikkuvan (KUVIO 16). Ohjaamiseen käytetään ainoastaan koskettuksen sijainnin vektorin x arvoa, joka määrittää kuinka paljon autoa liikutetaan horisontaalisesti. Jotta pelaajan ei tarvitse painaa aivan ruudun reunaa, että auto liikkuu sivulle, käytetään apuna Unityn AnimationCurve-luokkaa. Niiden avulla voidaan mallintaa matemaattisia funktioita Unityn inspectorissa tehdyillä graafisilla käyrillä (KUVIO 17). Käyrän x-akseli kuvaa time-arvoa ja y-akseli kuvaa value-arvoa. AnimationCurve-luokan Evaluate-metodiin syötetään time-arvo, joka antaa ulos käyrällä olevan value-arvon. (Schöner 2018, viitattu 4.12.2019.)

```
Frequently called | 1 usage
public void SetInputs(ref PlayerVehicleInputs inputs)
{
    //Initialize move vector with throttle amount
    var moveInputVector = new Vector3( x: 0, y: 0, z: inputs.ThrottleAxis);

    //Add boost for testing
    if (inputs.BoostDown){...}

    if (inputs.IsTouching)
    {
        var time = Mathf.Lerp( a: -1, b: 1, t: inputs.TouchLocation.x);
        var axisRight = SteerInputCurve.Evaluate(time) * 0.5f;
        moveInputVector.x = axisRight;

        //Alternative control option
        /* var touchPosition = inputs.TouchPosition; ... */
    }

    // Calculate camera direction and rotation on the character plane
    Vector3 cameraPlanarDirection = Vector3.ProjectOnPlane( vector: inputs.CameraRotation * Vector3.forward, planeNormal: Motor.CharacterUp).normalized;
    if (cameraPlanarDirection.sqrMagnitude == 0f)
    {
        cameraPlanarDirection = Vector3.ProjectOnPlane( vector: inputs.CameraRotation * Vector3.up, planeNormal: Motor.CharacterUp).normalized;
    }
    Quaternion cameraPlanarRotation = Quaternion.LookRotation( forward: cameraPlanarDirection, upwards: Motor.CharacterUp);

    // Move and look inputs
    _moveInputVector = cameraPlanarRotation * moveInputVector;

    //Move towards player input when on ground
    if (Motor.GroundingStatus.IsStableOnGround)
    {
        _lookInputVector = _moveInputVector.normalized;
    }
    else
    {
        //Move more towards velocity direction when in air
        var velDir = Motor.Velocity.normalized;
        velDir.y = 0;
        _lookInputVector = Vector3.Lerp( a: velDir, b: _moveInputVector.normalized, t: 0.5f);
    }

    // Jumping input
    if (inputs.JumpDown)
    {
        Jump();
    }
}
```

KUVIO 16. Controller-luokan metodi, joka käsittelee pelaajan syötteet.



KUVIO 17. Pelaajan syötteisiin vaikuttava AnimationCurve-käyrä inspectorissa.

Kun pelaajan syötteet on saatu ja käsitelty, voidaan niitä käyttää hallitsemaan auton nopeutta (KUVIO 18) ja suuntaa (KUVIO 19) ICharacterController-rajapintaluokasta periytyvillä UpdateVelocity- ja UpdateRotation-metodeilla.

```
public void UpdateVelocity(ref Vector3 currentVelocity, float deltaTime)
{
    if (_resetRequested)
    {
        currentVelocity = Vector3.zero;
        Motor.SetPosition(Vector3.zero);
        return;
    }

    Vector3 targetMovementVelocity = Vector3.zero;

    // Add boost to move speed
    var boostMultiplier = BoostStatus.GetBoostMultiplier();
    var moveSpeed = MaxStableMoveSpeed * boostMultiplier;

    // Handle ground movement
    if (Motor.GroundingStatus.IsStableOnGround)
    {
        // Reorient velocity on slope
        currentVelocity = Motor.GetDirectionTangentToSurface( direction: currentVelocity, Motor.GroundingStatus.GroundNormal) * currentVelocity.magnitude;

        // Calculate target velocity
        Vector3 inputRight = Vector3.Cross( lhs: _moveInputVector, rhs: Motor.CharacterUp);
        Vector3 reorientedInput = Vector3.Cross( lhs: Motor.GroundingStatus.GroundNormal, rhs: inputRight).normalized * _moveInputVector.magnitude;
        targetMovementVelocity = reorientedInput * moveSpeed;

        // Smooth movement Velocity
        currentVelocity = Vector3.Lerp( a: currentVelocity, b: targetMovementVelocity, t: 1 - Mathf.Exp( power: -CurrentMovementSharpness * deltaTime));
    }
    else
    {
        // Handle air movement
        // Add move input
        if (_moveInputVector.sqrMagnitude > 0f)
        {
            targetMovementVelocity = _moveInputVector * moveSpeed * 2f;

            // Prevent climbing on un-stable slopes with air movement
            if (Motor.GroundingStatus.FoundAnyGround)
            {
                Vector3 perpenticularObstructionNormal = Vector3.Cross( lhs: Vector3.Cross( lhs: Motor.CharacterUp, rhs: Motor.GroundingStatus.GroundNormal), rhs: Motor.CharacterUp).normalized;
                targetMovementVelocity = Vector3.ProjectOnPlane( vector: targetMovementVelocity, perpenticularObstructionNormal);
            }

            Vector3 velocityDiff = Vector3.ProjectOnPlane( vector: targetMovementVelocity - currentVelocity, planeNormal: Gravity);
            currentVelocity += velocityDiff * AirAccelerationSpeed * deltaTime;
        }

        // Add Gravity
        currentVelocity += Gravity * deltaTime;

        // Add Drag
        currentVelocity *= (1f / (1f + (Drag * deltaTime)));
    }

    // Handle jumping
    /* { ... */

    // Take into account additive velocity
    if (_internalVelocityAdd.sqrMagnitude > 0f)
    {
        currentVelocity += _internalVelocityAdd;
        _internalVelocityAdd = Vector3.zero;
    }
}
```

KUVIO 18. UpdateVelocity-metodi, jolla muutetaan mihin suuntaan ja millä nopeudella autoa halutaan liikuttaa.

```
0+1 usages
public void UpdateRotation(ref Quaternion currentRotation, float deltaTime)
{
    if (_resetRequested)
    {
        currentRotation = Quaternion.identity;
        return;
    }

    if (_lookInputVector != Vector3.zero && OrientationSharpness > 0f)
    {
        // Smoothly interpolate from current to target look direction
        Vector3 smoothedLookInputDirection = Vector3.Slerp( a: Motor.CharacterForward, b: _lookInputVector, t: 1 - Mathf.Exp( power: -OrientationSharpness * deltaTime)).normalized;

        // Set the current rotation (which will be used by the KinematicCharacterMotor)
        currentRotation = Quaternion.LookRotation( forward: smoothedLookInputDirection, upwards: Motor.CharacterUp);
    }
}
```

KUVIO 19. UpdateRotation-metodi, jolla auton suuntaa muutetaan.

Yllä mainitut skriptit ohjaavat ainoastaan auton liikettä. Itse auton näkymää ohjataan VehicleViewController-skriptillä, joka mukauttaa auton 3D-mallin pinnan kallistuksen mukaan. Sitä voidaan myös käyttää auton renkaiden kääntämiseen ja pyörittämiseen vauhdin mukaan.

KUVIO 20. Auto mukautuu pinnan kallistukseen VehicleViewController-skriptin avulla.

Auton näkymä päivitetään jokaisella ruudunpäivityksellä viimeisenä Unityn LateUpdate-metodin avulla. Auton näkymän ylöspäin osoittava vektori osoittaa suoraan ylöspäin, jos auto ei koske maata, jos taas auto koskettaa maata, käytetään ylöspäin osoittavana vektorina KinematicCharacterMotor-komponentin tunnistamaa pinnan osumasta saatua normaalivektoria. Auton näkymä käännetään osoittamaan haluttuun suuntaan sulavasti Unityn Vector3.Slerp-metodin avulla (KUVIO 21).

```
Frequently called  1 usage
private void OrientToSurface()
{
    var currentRotation = Root.rotation;

    var upDir = Vector3.up;

    //Up direction is ground surface normal when touching the ground
    if (VehicleMotor.GroundingStatus.IsStableOnGround)
        upDir = VehicleMotor.GroundingStatus.GroundNormal;

    var smoothedNormal = Vector3.Slerp( a: Root.up,  b: upDir,  t: 1 - Mathf.Exp( power: -OrientSharpness * Time.fixedDeltaTime));
    currentRotation = Quaternion.FromToRotation((currentRotation * Vector3.up), smoothedNormal) * currentRotation;

    SetRotation(currentRotation);
}
```

KUVIO 21. VehicleViewController-skriptin metodi, joka mukauttaa auton maan pinnan kallistukseen.

Auton ääniefektejä ohjataan VehicleSoundController-skriptillä, joka soittaa auton käyttämiä eri ääniefektejä, kuten moottoria, renkaiden luisumista ja osumia. Koska pelissä ei simuloida oikeata moottoria, eikä renkaita millään tavalla, täytyy ääniefektien soittamiseen käytetyt arvot määrittää jollain muulla tavalla (KUVIO 22).

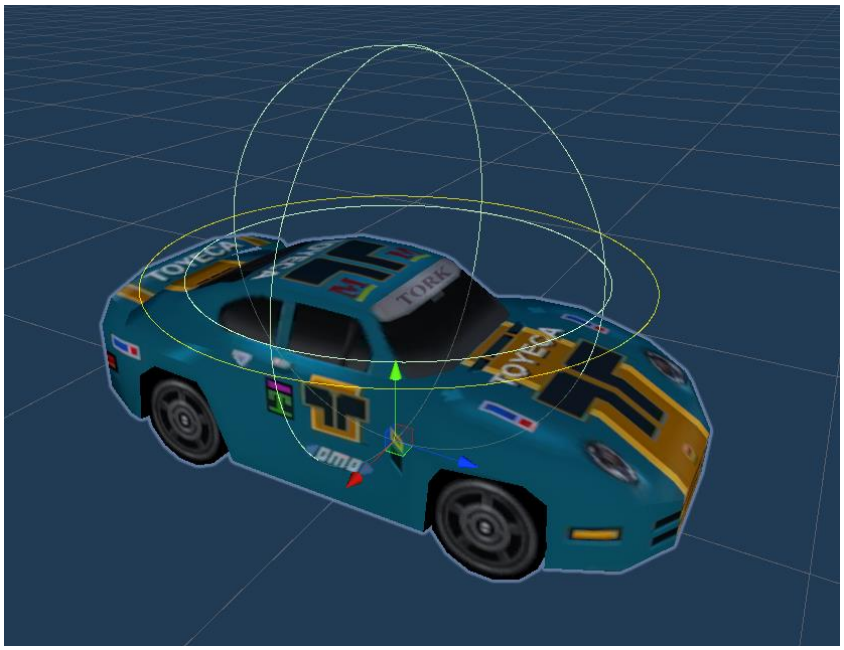
Moottori ääniefektin äänenvoimakkuus tulee nykyisestä nopeudesta. Sen sävelkorkeus tulee auton nykyisestä nopeudesta, onko se boost-tilassa ja siitä, että koskettaako se maata. Ääniefektin korkeuden vaihtelua tasoitetaan Mathf.Lerp-metodin avulla.

Renkaiden luisumisääniefektin äänenvoimakkuus ja sävelkorkeus saadaan selville ottamalla auton nykyisen suunnan ja auton oikealle osoittavan vektorin pistetulo Unityn Vector3.Dot-metodia käyttäen. Tällä metodilla saadaan ulos arvo, joka on 0.0f ja 1.0f välillä, jolloin 0.0f tarkoittaa, että auto kulkee täysin suoraan, eikä luisu yhtään sivuttaissuunnassa ja 1.0f tarkoittaa, että auto luisuu täysin sivuttaissuunnassa oikealle tai vasemmalle päin.

```
Frequently called  1 usage
private void HandleVehicleSound()
{
    var isGrounded = CharacterController.Motor.GroundingStatus.IsStableOnGround;
    var speedNormalized = Mathf.Abs( f: CharacterController.CurrentSpeed / _maxSpeed);

    //Engine volume
    EngineSource.volume = Mathf.Lerp( a: EngineSoundSettings.Volume.Min,  b: EngineSoundSettings.Volume.Max,  t: speedNormalized);

    //Engine pitch
    var pitchMultiplier = isGrounded ?
        EngineSoundSettings.PitchGroundedMultiplier.Min * CharacterController.BoostStatus.GetBoostMultiplier() :
        EngineSoundSettings.PitchGroundedMultiplier.Max * CharacterController.BoostStatus.GetBoostMultiplier();
    _targetEnginePitch = Mathf.Lerp( a: EngineSoundSettings.Pitch.Min,  b: EngineSoundSettings.Pitch.Max,  t: speedNormalized) * pitchMultiplier;
    EngineSource.pitch = Mathf.Lerp( a: EngineSource.pitch,  b: _targetEnginePitch,  t: EngineSoundSettings.PitchTime * Time.fixedDeltaTime);

    //Skid Sound
    var skidNormalized = isGrounded ?
        Mathf.Abs( f: Vector3.Dot( lhs: CharacterController.Motor.CharacterRight,  rhs: CharacterController.Motor.Velocity.normalized)) : 0f;
    SkidSource.volume = Mathf.Lerp( a: 0f,  b: 0.5f,  t: skidNormalized);
    SkidSource.pitch = Mathf.Lerp( a: 0.75f,  b: 1f,  t: skidNormalized);
}
```

KUVIO 22. VehicleSoundController-skriptin auton ääniefektejä hallitseva metodi.

Yksi KCC:n rajoitteista on se, että siinä pystytään ainoastaan käyttämään kapselin muotoista collision-muotoa. Lähimmäksi auton muotoa päästään käyttämällä kapselissa samaa pituutta ja leveyttä, jolloin muodosta tulee pallon muotoinen (KUVIO 23). Tämä ei vastaa aivan täysin auton muotoja, mutta tätä on vaikeaa huomata peliä pelatessa.



KUVIO 23. Auton collider Unityn scene-näkymässä.

4 POHDINTA

Aloin projektin alussa tekemään peliä, josta tuli lopulta aivan liian monimutkainen toteuttaa, suurimmaksi osaksi, koska pelissä oli tarkoitus olla Wi-Fin välityksellä pelattava moninpeli. Jouduin lopulta aloittamaan koko projektin alusta, pienentämään sen laajuutta ja keksimään uuden peli-idean. Vaikka ensimmäiseen peliin meni paljon aikaa hukkaan, oli se silti kokemus, joka opetti paljon moninpelin toteutuksesta, nettikoodista ja yleisesti projektin suunnittelusta.

Peliä kehitettäessä ei tullut vastaan mitään hirveän haastavaa. Käyttöliittymän toteutukseen tosin meni paljon aikaa, koska eri näkymiä on paljon ja DoozyUI-paketti oli minulle täysin uusi, joten minun piti käyttää paljon aikaa sen oppimiseen.

Ohjelmoinnin kannalta haluaisin päästä eroon Singleton-suunnittelumalliin turvautumisesta ja tutustua enemmän johonkin Inversion of Control-pakettiin kuten Zenject tai StrangeIoC. En vielä kumminkaan tähän projektiin halunnut ottaa niitä mukaan, koska tarkoituksena oli tehdä peli mahdollisimman nopeaa ja mutkattomasti ja itselleni aivan uuden suunnittelumallin oppimiseen olisi voinut kulua turhan paljon aikaa. Olen myös käyttänyt muutamassa peliprojektissa reaktiivisen ohjelmoinnin mahdollistavaa UniRx-pakettia, mutta en halunnut käyttää sitä tässä projektissa, koska halusin pitää sen kohtalaisen yksinkertaisena.

Peliä voisi vielä kehittää eteenpäin lisäämällä peliin erilaisia avattavia autoja, joille voi myös avata uniikkeja ulkoasuja. Toinen tärkeä lisättävä ominaisuus ovat Google Play -palvelun saavutukset, jotka lisäävät pelaajalle tekemistä. Peliin voisi myös lisätä tasoja, joissa on jokin eri tavoite, kuin vaan selviytyä tason loppuun. Esimerkiksi jonkin tietyn väliajoin voisi tulla vaikeampi taso, jossa pitää kerätä matkalla kolme esinettä ja selvitä tason loppuun ilman checkpointteja, jotta pelaaja pääsee siirtymään seuraavalle tasolle. Itse hyperkasuaalin pelin tekeminen ei välttämättä vie hirveän paljon aikaa, mutta pelin hiomiseen ja paranteluun voi aina käyttää melkein loputtomasti aikaa.

Visuaalista tyyliä voi myös parannella ja lisätä peliin enemmän tekstuuria. Pelissä käytetään muutamaa yksinkertaista partikkeliefektiä, mutta niitä voisi lisätä enemmän. Partikkeliefekteillä voi olla iso vaikutus pelaajan pelikokemukseen, joten niiden käyttäminen on tärkeää valmiissa pelissä.

Kaiken kaikkiaan olen tyytyväinen siihen, mitä sain aikaan ainoana kehittäjänä hieman alle kahdessa kuukaudessa. Tavoitteenani oli luoda hyperkasuaali mobiilipeli ja mielestäni peli saavutti nämä tavoitteet.

LÄHTEET

Cooper, T. 2018a. The Lightweight Render Pipeline: Optimizing Real Time Performance. Unity Blog. Viitattu 4.12.2019, https://blogs.unity3d.com/2018/02/21/the-lightweight-render-pipeline-optimizing-real-time-performance.

Cooper, T. 2018b. Introduction to Shader Graph: Build your shaders with a visual editor. Unity Blog. Viitattu 4.12.2019, https://blogs.unity3d.com/2018/02/27/introduction-to-shader-graph-build-your-shaders-with-a-visual-editor.

Doozy Entertainment 2019a. DoozyUI: Complete UI Management System. Unity Asset Store. Viitattu 4.12.2019, https://assetstore.unity.com/packages/tools/gui/doozyui-complete-ui-management-system-138361.

Doozy Entertainment 2019b. DoozyUI. Viitattu 4.12.2019, http://doozyui.com.

GameAnalytics 2018. A Global Analysis of Mobile Gaming Benchmarks. Viitattu 25.4.2019, https://public-production.gameanalytics.com/assets/GameAnalytics-Benchmarks-Report-2018.pdf.

GameDesigning.org 5.7.2019. Create Your First Game Design Document. Viitattu 4.12.2019, https://www.gamedesigning.org/learn/game-design-document.

Gamesindustry.biz 2019. App Annie: Mobile gaming was the fastest-growing gaming sector in 2018. Viitattu 25.4.2019, https://www.gamesindustry.biz/articles/2019-01-22-app-annie-mobile-gaming-fastest-growing-gaming-sector-2018.

Hadley, L. 22.10.2019. What Is In-Game Advertising? Bidstack. Viitattu 4.12.2019, https://www.bidstack.com/news-views/blog/what-is-in-game-advertising.

ironSource 2018. Hyper-casual games: What are they & how do you monetize them? Viitattu 26.4.2019, https://www.ironsrc.com/blog/what-are-hyper-casual-games-and-how-do-you-monetize-them.

Prime31 2019. TouchKit. GitHub. Viitattu 4.12.2019, https://github.com/prime31/TouchKit.

Roseboom, I. 2.6.2015. Why Players Leave: Understanding Day 1 Retention. deltaDNA. Viitattu 4.12.2019, https://deltadna.com/blog/analyzing-day-1-retention-ftue.

Schöner, S. 5.5.2018. Unity Animation Curves for Sampling. blog.s-schoener. Viitattu 4.12.2019, http://blog.s-schoener.com/2018-05-05-animation-curves.

SgLib Games 2019. Easy Mobile Pro. Unity Asset Store. Viitattu 4.12.2019, https://assetstore.unity.com/packages/tools/integration/easy-mobile-pro-75476.

Sirenix 2019. Odin Serializer. Odin Inspector. Viitattu 4.12.2019, https://odininspector.com/odin-serializer.

SpaceMadness 2019. Lunar Mobile Console - FREE. Unity Asset Store. Viitattu 4.12.2019, https://assetstore.unity.com/packages/tools/gui/lunar-mobile-console-free-82881.

St-Amand, P. 2019. Kinematic Character Controller. Unity Asset Store. Viitattu 4.12.2019, https://assetstore.unity.com/packages/tools/physics/kinematic-character-controller-99131.