Jussi Laitinen, Riku Havusalmi

3D-MOBIILIPELIN KEHITTÄMINEN UNITYLLÄ

Tietojenkäsittelyn koulutusohjelma

2016

samk

Satakunnan ammattikorkeakoulu
Satakunta University of Applied Sciences

3D-MOBIILIPELIN KEHITTÄMINEN UNITYLLÄ

Havusalmi Riku, Laitinen Jussi
Satakunnan ammattikorkeakoulu
Tietojenkäsittelyn koulutusohjelma
Maaliskuu 2016
Ohjaaja: Nieminen, Hans
Sivumäärä: 64
Liitteitä: 3

Asiasanat: Peliohjelmointi, Unity, Pelisuunnittelu, Pyöräily, Karhu

---

Tämän opinnäytetyön tavoitteena oli 3D-mobiilipelin kehittäminen käyttäen Unity3D-pelimoottoria. Pelin idea oli pyöräilypeli, missä pystyy heittämään pizzalaatikoita nälkäisille kodittomille sekä ampumaan ihmiskuntaa orjuuttavia karhuja.

Työn teoriaosuudessa selvitettiin tärkeimpiä tässä projektissa käytettyjä teknologioita. Perehdyttiin Unity-pelimoottoriin yleisesti ja sen tekniikkaan. Selvitettiin, miten Unityn komponenttimalli toimii. Tehtiin perinpohjainen selvitys Unityn MonoBehaviour-luokasta, josta itsekin opimme paljon. Näiden lisäksi esittelimme Unity-projektin rakennetta, Profiler-työkalua sekä kuinka monen käyttöjärjestelmän tuki toimii Unityssä.

Lisäksi teimme pikaisen katsauksen käyttämäämme C#-ohjelmointikieleen ja esittelimme suhteellisen suppeasti C#-ohjelmointikielen perusteet, automaattisen muistinhallinnan toimintaa sekä Unityn ja C#-kielen suhdetta. Selvitettiin myös käyttämämme SQLite-tietokannan sisäistä toimintaa, eli sen arkkitehtuuria, ominaisuuksia, suorituskykyä ja myös sen teknisiä rajoituksia.

Tämän jälkeen selvitettiin lyhyesti ketterää ohjelmistokehitystä ja erityisesti siihen perustuvaa Extreme Programming -ohjelmistokehitysmetodologiaa. Siihen liittyvä pariohjelmointi oli tässä työssä erityisen läsnä. Kävimme myös läpi hyviä ohjelmointikäytäntöjä, ja mikä niiden laiminlyömisen vaikutus voi olla.

Käytännön osuudessa esittelimme kehitetyn 3D-mobiilipelin tärkeimpiä osa-alueita, niiden mahdollisesti tuomia haasteita ja niiden ratkaisuja. Aiheisiin kuului muun muassa kentän ikuinen generointi, pyörän liikkuminen ja sen ohjaaminen, ampuminen, vihollisten toimintalogiikka ja tekoäly, SQLite-tietokannan implementointi, polkupyörien rakentaminen, saavutusjärjestelmä ja erinäiset tärkeät optimointitekniikat.

3D MOBILE GAME DEVELOPMENT WITH UNITY

Havusalmi Riku, Laitinen Jussi
Satakunnan ammattikorkeakoulu, Satakunta University of Applied Sciences
Degree Programme in Business Information Technology
March 2016
Supervisor: Nieminen, Hans
Number of pages: 64
Appendices: 3

Keywords: Game Development, Unity, Game Design, Cycling, Bear

---

The goal of this thesis was to develop a 3D mobile game using the Unity3D game engine. The idea was to make a cycling game, in which the player can throw pizza boxes for the homeless hungry bums and shoot bears that are enslaving humanity.

The theoretical part of this thesis focuses on the main technologies used in this project. We familiarized ourselves with the Unity game engine and its techniques. We investigated how the component model of Unity works. We thoroughly examined the MonoBehaviour class of Unity, from which we learned a lot. We also introduced the structure of the Unity project, the Profiler tool, and also the way Unity supports multiple platforms.

We also made a quick overview of the C# programming language. We briefly introduced the basics of it, its automatic memory management, and also its relation to the Unity game engine. We examined how SQLite works internally, specifically its architecture, features, performance and its technical limitations.

Next we familiarized ourselves briefly with agile software development of which Extreme Programming in particular. Pair programming associated with Extreme Programming was especially present in this project. We also went through good programming practices and what the impact of neglecting them could be.

The practical part of this thesis focuses on the most important aspects of the developed game and their possible challenges and solutions. Topics include the infinite generation of the game world, controlling the bicycle, shooting, the enemies' artificial intelligence, implementation of the SQLite database, bicycle customization, achievement system and various important optimization techniques.

SISÄLLYS

1 ALKUSANAT ... 5
2 KÄYTETYT TEKNIIKAT ... 6
2.1 Unity-pelimoottori ... 6
2.1.1 Unityn tekniikka ... 7
2.1.2 Unityn komponenttimalli ... 9
2.1.3 MonoBehaviour-luokka ... 10
2.1.4 Unity-projektin rakenne ... 12
2.1.5 Profiler ... 13
2.1.6 Monen alustan tuki ... 13
2.2 C#-ohjelmointikieli ... 14
2.2.1 Tyypit ... 15
2.2.2 C# kielioppi ja näkyvyysalue ... 18
2.2.3 Muistinhallinta ... 19
2.2.4 Unity ja C# ... 19
2.3 SQLite-tietokanta ... 20
2.3.1 Arkkitehtuuri ... 21
2.3.2 Ominaisuudet ja filosofia ... 23
2.3.3 Suorituskyky ja rajat ... 24
3 OHJELMISTON KETTERÄ KEHITTÄMINEN ... 25
3.1 Extreme Programming ... 26
3.2 Hyviä ohjelmointikäytäntöjä ... 28
3.3 Versionhallinta ... 29
3.3.1 Git-versionhallintaohjelmisto ... 30
4 CASE: HUNGRY BUMS ... 32
4.1 Game Manager ... 33
4.2 Kentän generointi ... 34
4.3 Kentän valinta ... 35
4.4 Pyörän liikkuminen ja ohjaus ... 37
4.5 Ampuminen ... 39
4.5.1 Bullet time ... 39
4.5.2 Asejärjestelmä ... 41
4.6 Viholliset ... 42
4.6.1 Raketinheittimellä varustettu karhu ... 43
4.6.2 Mellakkakarhut ... 44
4.7 Tiellä kulkevien vihollisten tekoäly ... 46
4.8 SQLite-tietokannan implementointi ... 48
4.9 Polkupyörien rakentaminen ... 49
4.10 Pelin kauppa ... 52
4.11 Saavutukset ... 53
4.12 Hahmojen puheen hallinta ... 54
4.13 Optimointi ... 55
4.13.1 Grafiikkojen optimointi ... 56
4.13.2 Koodin optimointi ... 58
5 LOPPUSANAT ... 59
LÄHTEET ... 62

1 ALKUSANAT

Peliala on ollut viihdeteollisuuden nopeimmin kasvava ala koko 2000-luvun ajan. Myös Suomessa peliala on hurjassa kasvussa – pelkästään vuosien 2011 ja 2014 välisenä aikana Suomeen on syntynyt 179 uutta pelialan yritystä. Samassa ajassa pelialalla työskentelevien määrä Suomessa on noussut 1264 työntekijästä 2500 työntekijään, melkein tuplaten Suomessa pelialalla työskentelevien määrän. Tällä hetkellä Suomessa on noin 260 aktiivisesti toimivaa pelialan yritystä. (Neogames 2015, Lappalainen 2015.) Selvä kasvu pelialalla sekä oma mielenkiintomme pelien kehittämistä kohtaan saivat meidät valitsemaan opinnäytetyön aiheeksi oman peliprojektimme.

Opinnäytetyömme aiheena onkin 3D-mobiilipelin kehitys Unityllä, ja työn keskiössä on oma peliprojektimme, Hungry Bums. Olemme kehittäneet peliä harrastusprojektinamme alkusyksystä 2015. Näiden lukuisten kuukausien aikana olemme kohdanneet monia sudenkuoppia, ongelmia ja yllättyneet yleisesti pelinkehityksen haastavuudesta, etenkin verratessa perinteisempiin sovellusprojekteihin. Meidän onneksemme meillä oli juuri oikeat ainekset projektin aloittamiseen Satakunnan ammattikorkeakoulusta: erinomaiset C#-taidot ja hyvä yleiskäsitys relaatiotietokantojen toiminnasta. Huomasimme, että oma henkilökohtainen, itsenäinen lisäopiskelu näiden kolmen ja puolen vuoden aikana on ollut kuitenkin korvaamatonta projektin kestävään läpiviemiseen tähän asti.

Tämä ei ole suinkaan ensimmäinen yhteinen ohjelmistoprojektimme. Meillä on takana useita yhteisiä projekteja, suurimmaksi osaksi pieniä prototyyppimaisia peliprojekteja ja olemme oppineet olemaan hyvinkin rehellisiä toistemme koodin laadusta. Tämä on johtanut koodin laadun jatkuvaan kehitykseen. Rehellisyys onkin paikallaan, sillä käytämme hyvin usein Extreme Programming -ohjelmistokehitysmetodologiasta tuttua parityöskentelyä. Tämä on osoittautunut tehokkaaksi tavaksi tuottaa erittäin laadukasta ja jopa kaunista (luettavaa) koodia. Jos olet lipsumassa hölmöilyyn, toinen iskee sinut takaisin oikeille raiteille. Toinen kriittinen silmäpari on myös ollut hyvä tapa välttää omalle työlleen sokeutumista, mikä tarkoittaa kyvyttömyyttä havaita tekemiään virheitä.

Tässä työssä käymme läpi monia kehittyneempiä pelinkehityksen ideoita ja tekniikoita, yleisesti tarvittavia teknologioita sekä erilaisia kohtaamiamme ongelmia ja niiden ratkaisuja tässä kyseisessä peliprojektissa. Toivomme, että tästä työstä jäisi pelinkehityksestä kiinnostuneille hyvä yleisluontoinen käsitys, miten Unity-pelimoottorin avulla rakennetaan pelejä. Selonteko opinnäytetyödokumentin työnjaosta löytyy liitteestä kaksi.

Suurella kunnialla kirjoittaen,
Jussi Laitinen & Riku Havusalmi.

2 KÄYTETYT TEKNIIKAT

2.1 Unity-pelimoottori

Videopeli on yksinkertaisimmillaan interaktiivinen ohjelma, joka saattaa piirtää grafiikkaa, toistaa ääntä, lukea käyttäjän syöttölaitteita sekä joskus myös mallintaa fysiikkaa tai siirtää dataa lähiverkon tai Internetin välityksellä. Pelimoottori on järjestelmä, joka sitoo pelinkehityksessä tarvittavat teknologiat toimivaksi paketiksi, jolla pelinkehityksestä tulee helpompaa ja tehokkaampaa. (Enger 2013.)

Unity on järjestelmä, joka mahdollistaa videopelien kehityksen vaatimatta syvempää tietotaitoa kaikista pelinkehityksessä tarvittavista teknologioista. Koska tarvittavaa teknologiaa ei tarvitse integroida tai ohjelmoida itse, voi pelinkehittäjä keskittyä vain itse pelin luomiseen, näin ollen kiihdyttäen kehitysprosessia. Unity tarjoaa korkean tason järjestelmän, mikä tarkoittaa sitä, että kehittäjän ei esimerkiksi tarvitse tietää, miten pelimoottori piirtää pelin grafiikat tai miten peli kommunikoi näytönohjaimen kanssa. Windowsin lisäksi myös muille käyttöjärjestelmille pelin kääntäminen onnistuu yleensä vain nappia painamalla, koska Unity-pelimoottori osaa kääntää projektin lähestulkoon kaikille merkittäville käyttöjärjestelmille. Toisinaan voi kuitenkin olla

tarpeellista ottaa huomioon kohteena oleva käyttöjärjestelmä ohjelmakoodin suorituksessa ja kirjoittaa osa koodista eri tavalla. (Felicia 2015, 15; Thorn 2015, 186.)

Unity sai alkunsa, kun David Helgason, Joachim Ante ja Nicholas Francis ryhtyivät 2000-luvun alussa koodaamaan työkalua videopelien tekemiseen. Alussa Unity tuki vain Mac-tietokoneita, joka olikin Helgasonin sanojen mukaan ”bisneksen näkökulmasta huonoin mahdollinen päätös”, sillä tuohon aikaan Mac-pelien osuus peliteollisuudessa oli suoraan sanottuna mitätön. Tuki Windows-koneille ja selaimille tuli kuitenkin myöhemmin. (Brodkin 2013.)

Ensimmäinen versio Unitystä julkaistiin vuonna 2005. Vuoteen 2008 mennessä Unity oli jo kehittynyt huomattavasti ja Helgasonin vuonna 2004 perustama Unity Technologies työllisti jo noin tusinan työntekijöitä. Yrityksen käännekohta tuli vuoden 2008 puolessa välissä, kun Apple avasi iPhone App Storen. Unityä kehitettiin nopeasti tukemaan iPhonea ollen vuoden 2008 lopussa ensimmäinen iPhonea tukeva pelimoottori. (Brodkin 2013.)

Tänä päivänä yli 300 ihmistä työllistävä Unity-pelimoottori on pelikehittäjien keskuudessa käytetyin pelimoottori. Unityllä on nykyisellään jopa 45 prosenttia alan markkinoista suurimman kilpailijan osuuden ollessa 17 prosenttia ja muiden kilpailijoiden osuuksien ollessa yhteensä 38 prosenttia. Tähän päivään mennessä Unity on siis kasvanut markkinoiden suurimmaksi tekijäksi. (Unity Technologies A.)

Unityllä on tällä hetkellä yli neljä miljoonaa rekisteröitynyttä käyttäjää ja sillä tehdyillä peleillä on jopa 600 miljoonaa pelaajaa. Amerikan mobiilipelimarkkinoilla jopa 50 prosenttia peleistä Amerikassa on tehty Unityllä, kun taas Kiinan mobiilipelimarkkinoilla Unity-pelien osuus on jopa 75 prosenttia. (Unity Technologies A.)

2.1.1 Unityn tekniikka

Grafiikan piirtämiseen Unity käyttää eri käyttöjärjestelmillä eri grafiikkaohjelmointirajapintoja: Windowsilla Direct3D, Mac- ja Linux-koneilla OpenGL ja mobiililaitteilla OpenGL ES. HTML5-sovellukseksi käännetty peli käyttää WebGL-rajapintaa.

Unity tarjoaa mahdollisuuden sekä kaksi- että kolmiulotteisen grafiikan piirtämiseen, joten kaikenlaisten pelien kehittäminen on mahdollista. Käytössä olevien grafiikka-ohjelmointirajapintojen ansiosta Unity-pelit kääntyvät useimmiten identtisesti usealle käyttöjärjestelmälle. Tämän ansiosta pelinkehittäjän ei useimmiten tarvitse nähdä vaivaa ohjelmakoodin muuttamiseen kääntäessä eri alustoille. (Amazonas 2014; Unity Technologies B.)

Äänipuolen Unity-pelimoottorissa hoitaa FMOD-ohjelmisto, joka on yksi käytetyimpiä ääniteknologian väliohjelmistoja. Unityssä voidaan toistaa sekä 2D- että 3D-ääniä tarvittaessa erilaisilla efekteillä höystettynä. Tärkeimmät äänentoiston komponentit Unityssä ovat Audio Source- ja Audio Listener -komponentit, joista ensimmäinen toistaa ääntä ja jälkimmäinen kuulee ääntä (katso kuva 1). Unityssä on mahdollista tuoda projektiin aiff-, wav-, mp3- ja ogg-tiedostomuotoisia äänitiedostoja. Tämän lisäksi Unity tukee xm-, mod-, it- ja s3m-muotoista tracker-musiikkia. Tracker-musiikki on monikanavaiseen ääninäytteiden toistoon perustuvaa tietokonemusiikkia. (Unity Technologies C; Firelight Technologies.)

Kuva 1. Audio Source ja Audio Listener kuvastettuna (Unity Technologies C)

3D-fysiikkamallinnuksen Unity-pelimoottorissa hoitaa Nvidian kehittämä usean alustan PhysX-fysiikkamoottori, joka mahdollistaa realistisen fysiikan mallintamisen Unityllä tehtävissä 3D-peleissä. PhysX:n avulla 3D-peliobjekteja voidaan liikutella ja mahdollisesti myös tuhota kappaleiksi todentuntuisesti. Myös kangaspintojen mallintaminen on mahdollista PhysX:n ansiosta. Tärkeimmät 3D-fysiikkaan sidotut komponentit Unityssä ovat Collider-komponentit sekä Rigidbody-komponentti. Collider-komponentit mallintavat objektin 3D-pinnan yksinkertaisempana versiona, jota Rigidbody-komponentti käyttää fysiikkalaskuissa. Collideria käytetään törmäyksentunnistuksessa ja Rigidbody vastaa objektin liikuttamisesta PhysX:n lakien mukaisesti. (Unity Technologies D; NVIDIA.)

2D-pelien fysiikat mahdollistavat Erin Catton kehittämä Box2D-fysiikkamoottori, joka on toinen Unityn käyttämistä fysiikkamoottoreista. Box2D tarjoaa 2D-peleihin dynamiikkaa tehokkaalla 2D-fysiikkamallinnuksella, joka mahdollistaa monipuoliset fysiikkalaskennat. 3D-fysiikkojen tapaan, 2D-fysiikkojen aikaansaamiseksi Unity-pelimoottorissa käytetään peliobjekteissa Collider2D- ja Rigidbody2D -komponentteja. (Catton; Unity Technologies E.)

2.1.2 Unityn komponenttimalli

Unityllä pelit rakentuvat niin sanotuista peliobjekteista (game object). Peliobjekti on ikään kuin tyhjä purkki, minkä sisälle voidaan laittaa komponentteja. Se tarkoittaa sitä, että Unityn peliobjektit ovat tyhjiä objekteja, joihin lisätään funktionaalisuus ja grafiikka komponenttien avulla. Peliobjektissa voi olla esimerkiksi seuraavanlaisia komponentteja:

- Mesh renderer-komponentti, joka piirtää 3D-mallin.
- Collider-komponentti, joka tarjoaa fysiikkamoottorille yksinkertaistetun 3D-muodon fysiikkamallinnusta varten.
- Rigidbody-komponentti, joka Collider-komponentin avulla lisää fysiikkamallinnuksen peliobjektille. (Unity Technologies F; Porter 2013)

Pelit ohjelmoidaan komentosarjojen eli skriptien avulla, jotka myös lisätään komponentteina Unityn peliobjekteihin. Skriptikomponentit voivat keskustella muiden peliobjektin komponenttien kanssa. Unity mahdollistaa skriptien kirjoittamisen kahdella eri kielellä; C# ja UnityScript. (Unity Technologies G.)

Komponenttien etu on se, että kun komponentti on ohjelmoitu, sitä voidaan potentiaalisesti uusiokäyttää monissa erityyppisissä peliobjekteissa. Tämä nopeuttaa pelin kehitystä, kun samaa funktionaalisuutta ei tarvitse moneen kertaan ohjelmoida uudelleen. Aikaisemmin luotu komponentti voidaan yksinkertaisesti liittää haluttuun peliobjektiin. (Porter 2013.)

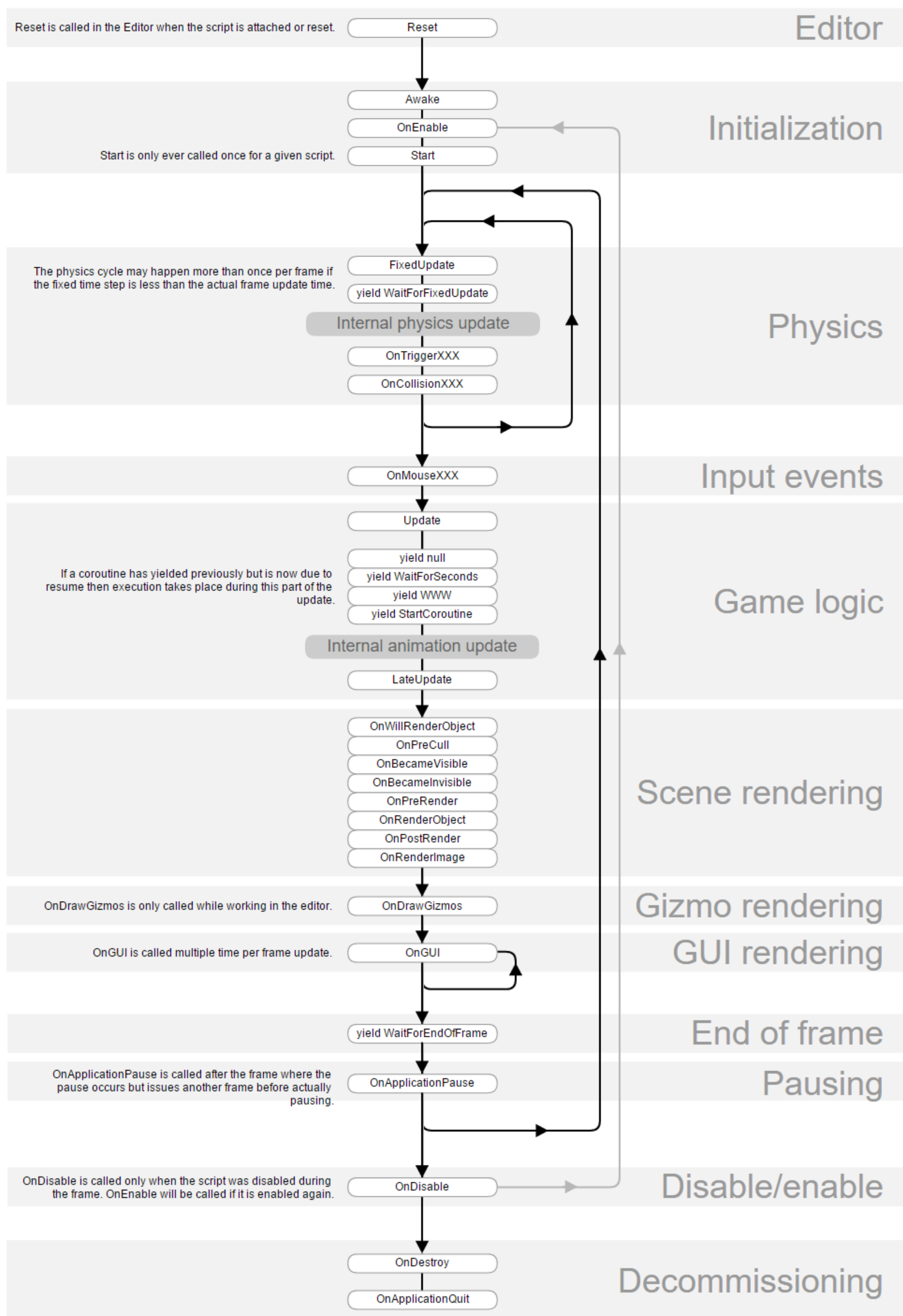
2.1.3 MonoBehaviour-luokka

Kaikki luokat (skriptit), jotka halutaan liittää komponentteina Unityn peliobjekteihin, täytyy periyttää MonoBehaviour-luokasta. MonoBehaviour-luokkaa voidaan ajatella eräänlaisena komponenttimallin pohjapiirustuksena, jonka päälle voidaan rakentaa omaa toiminnallisuutta. MonoBehaviourista periytyminen tuo tullessaan useita tapahtumia, mitä Unity kutsuu oman pelisilmukkansa sisällä, jos luokalla on implementaatio kyseiselle tapahtumalle. Yleisimpiä näistä on Awake, Start -ja Update-tapahtumat. (Hocking 2015, 15; Unity Technologies G). Tapahtumalla ei kuitenkaan tarkoiteta olioparadigman yhteydessä käytettyä käsitettä event. Kuvassa 2 näkyy Start -ja Update -tapahtumien implementaatiot.

```
using UnityEngine;
using System.Collections;

public class HelloWorld : MonoBehaviour {

    void Start() {
        // do something once
    }

    void Update() {
        // do something every frame
    }
}
```


Kuva 2. Start- ja Update -tapahtumat (Hocking 2015, 16)

Miten Unity kutsuu silmukassaan metodeja kuten Start ja Update vain periyttämällä MonoBehaviourista, kuitenkaan ilman metodien ylikirjoittamista (override)? Itse asiassa, MonoBehaviour-luokalla ei ole näitä metodeja eikä niitä edes periydy sille mistään. Sen sijaan Unity käyttää hyväkseen reflektiota (reflection) kaikkien näiden metodien kutsumiseen ajon aikana. Samalla Unityn ei tarvitse käydä "turhaan" kaikkia tapahtumia läpi jokaisessa luokassa, vaan ainoastaan ne, mitkä on implementoitu. (Hocking 2015, 15; Unity Technologies H; Unity Answers.) Kuvassa 3 näkyy kaikki implementoitavissa olevat tapahtumat ja niiden kutsumisjärjestys.

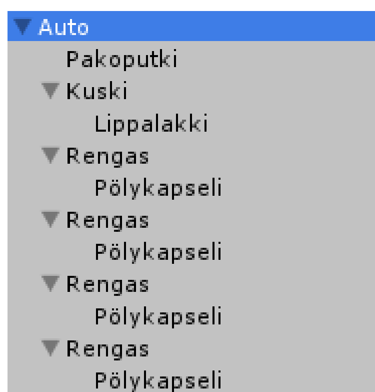


Kuva 3. MonoBehaviourin tapahtumien kutsumisjärjestys (Unity Technologies H)

2.1.4 Unity-projektin rakenne

Kuten aiemmin on mainittu, Unityssä pelit rakentuvat peliobjekteista (game object), joita luodaan yksittäisiin kenttiin (scene). Oletuksena peliobjekti ei sisällä mitään muuta kuin transform-komponentin, joka on pakollinen komponentti kaikissa peliobjekteissa. Transform-komponentti kertoo ja säilöö peliobjektin sijainnin, kulman ja skaalauksen. Vaikka peliobjekti olisikin täysin näkymätön, eikä sen sijainnilla ole merkitystä, transform-komponentti on silti välttämätön, jotta vanhempi-lapsi -suhteet ovat käytössä. (Thorn 2014, 25; Unity Technologies I.)

Peliobjektien vanhempi-lapsi -suhteet ovat yksi tärkeimmistä asioista Unity-projektin rakenteessa. Kun peliobjekti on toisen peliobjektin lapsi, tämä liikkuu, kääntyy ja skaalautuu vanhempansa mukana. Kaikilla peliobjekteilla voi olla monta lasta, mutta vain yksi vanhempi. (Thorn 2014, 25; Unity Technologies J.) Esimerkkinä kuva 4, missä on Auto-peliobjekti, jolla on lapsina 4 rengasta, pakoputki sekä kuski. Renkailla on lapsina pölykapselit ja kuskilla lippalakki.

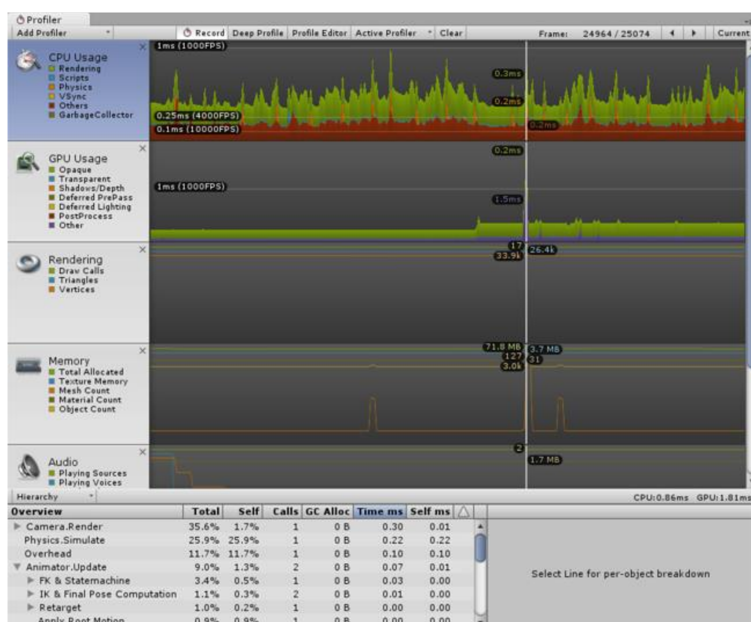


Kuva 4. Esimerkki peliobjektien suhteista, jossa vanhemmalla on lapsia ja lapsilla omia lapsia

Peliobjekteista voidaan tallentaa prefabeja. Prefab on objekti, joka säilöö peliobjektin, sen lapsipeliobjektit sekä näiden komponentit ja komponenttien asetukset. Prefab-objektin voi luomisen jälkeen tuoda kenttään, ja jos prefabia muutetaan, kaikki kentässä olevat samat prefab-objektit muuttuvat sen mukaan. Esimerkkinä, jos kuvan 4 autosta luotaisiin prefab, voitaisiin kenttään tuoda monia samanlaisia autoja. Jos auton prefabista poistetaan renkaista pölykapselit, kaikista kentän autoista poistuu pölykapselit. (Unity Technologies K.)

2.1.5 Profiler

Profiler on Unityn työkalu, jonka avulla voidaan analysoida Unity-pelin suorituskykyä. Profiler tallentaa jokaisen nauhoitetun ruudunpäivityksen aikajanaksi (Kuva 5), jonka kautta voidaan tutkia jokaista yksittäistä ruudunpäivitystä. Yksittäisen ruudunpäivityksen tietoihin tallentuu, kuinka kauan menee minkäkin koodin suorittamiseen tai grafiikan piirtämiseen. Profiler näyttää muun muassa, kuinka monta prosenttia kokonaislaskenta-ajasta mikäkin yksittäinen suoritus kattaa. Profiler on elintärkeä työkalu Unity-pelin optimointiin, sillä sen avulla löydetään helposti ongelmapisteet, mitkä kaipaavat korjaamista tai optimointia. Profiler-työkalua voidaan käyttää myös etänä, esimerkiksi älypuhelimella. Kun Profiler yhdistetään älypuhelimella käynnissä olevaan Unity-peliin, Profiler tallentaa aikajanalle puhelimella suoritetut laskennat. (Unity Technologies L; Yorick 2012.)



Kuva 5. Profiler-työkalun aikajana (Unity Technologies L)

2.1.6 Monen alustan tuki

Unity-pelimoottorilla kehitettyjä pelejä voidaan kääntää lähes kaikille mahdollisille alustoille, joista tärkeimpiin lukeutuu: iOS, Android, Windows Phone, Windows, Mac OS X, Linux, PS3, PS4, Xbox 360, Xbox One sekä Wii U. Pelejä voi kääntää myös selaimella pelattavaan muotoon. Tähän vaaditaan joko Unity Web Player, joka

vaatii selaimelle asennettavan lisäosan, tai sitten sen tulevaisuudessa korvaava WebGL-grafiikkarajapintaa tukeva selain. (Amazonas 2014; Unity Technologies B.)

Monen alustan tuki on mahdollista, koska Unityn tukemat ohjelmistorajapinnat ja väliohjelmistot ovat laajalti tuettuja eri alustoilla. Grafiikan piirtämisen puolella monen alustan tuki tulee usean grafiikkarajapinnan ansiosta – Unity tukee OpenGL, OpenGL ES, WebGL, Metal ja DirectX-rajapintoja, joista jokainen toimii jollain Unityn tukemalla alustalla. Fysiikat mahdollistavat PhysX sekä Box2D toimivat jokaisessa Unityn tukemassa alustassa. (Amazonas 2014; Unity Technologies B.)

Mono, avoimen lähdekoodin implementaatio ECMA-standardiin perustuvasta .NET-ohjelmistokehyksestä, on keskiössä monen alustan tukemisessa. Unityn ydintoiminnallisuus on ohjelmoitu natiivina C- tai C++-koodina, ja ”kaikki muu” toimii Monon ajoympäristön virtuaalikoneen kautta, mikä on paketoitu alustalle natiivin sovelluksen mukaan. (Amazonas 2014.) Tästä kerrotaan enemmän luvussa 2.2.4.

2.2 C#-ohjelmointikieli

C# on yleiskäyttöinen, vahvasti tyypitetty olio-ohjelmointikieli. Kielen kehitti Anders Hejlsberg, joka oli aiemmin työskennellyt Turbo Pascalin sekä Delphin parissa. C# tasapainoilee helppokäyttöisyyden, ilmaisuvoiman ja tehokkuuden välillä. Kieli on alustavapaa, joskin se on kirjoitettu Microsoftin .NET-sovelluskehystä silmällä pitäen. (J. Albahari & B. Albahari 2012, 1.)

Riippumattomuus ympäristöstä on asetettu jo erittäin aikaisessa vaiheessa .NET-kehitysympäristön luonnissa. Sen sijaan, että koodi käännettäisiin konekielelle prosessorin ymmärtämään muotoon, se käännetään Microsoftin Intermediate Language (MSIL) -kielelle. Tätä koodia taas suorittaa C#:n koodin ajamiseen vaadittava ajoympäristössä (Common Language Runtime, CLR) toimiva virtuaalitietokone. Ajoympäristö takaa ohjelmakoodin hallitun suorituksen hyödyntäen prosessorin ja käyttöjärjestelmän kaikkia ominaisuuksia. Se myös takaa hallitun muistinkäytön. (Nakov & Kolev 2013, 80-81.)

Microsoft antoi vuonna 2000 C#-kielen määritelmät ECMA:lle, joka standardoi kielen. Teoriassa siis kuka tahansa voisi luoda oman C#-kääntäjän sekä ajoympäristön ja ajaa sitä minkä käyttöjärjestelmän päällä tahansa. Näin on itse asiassa käynytkin: Mono on Ximianin (aiemmin Novellin) vetämä, alustavapaa implementaatio Microsoftin .NET ja C# -teknologioista. (Davis & Sphar 2005, 1; Icaza 2011.)

2.2.1 Tyypit

C# on pääasiassa vahvasti tyypitetty ohjelmointikieli. Tämä tarkoittaa, että jokaisella muuttujalla on oma tyyppinsä, ja ne voivat saada arvokseen vain oman tyyppinsä mukaisia arvoja. C# estää ohjelmoijaa käyttämästä merkkijonoa kuin se olisi kokonaisluku. C#-kääntäjä estää suuren määrän erilaisia virhetilanteita tarkistamalla vahvan tyypityksen jo kääntämisvaiheessa, jolloin ohjelmaa ei edes ehditä ajaa virheiden havaitsemiseksi. Tämän ansiosta suuret ohjelmat ovat helpommin hallittavissa, ennustettavampia toiminnaltaan ja vankempia perusteiltaan. Vahva tyypitys auttaa ohjelmoijaa myös ohjelmointiympäristössä (IDE). Esimerkiksi Microsoftin Visual Studio sisältää Intellisense-avustajan, joka tietää aina minkä tyyppinen muuttuja on ja osaa tarjota sille kuuluvia elementtejä. (J. Albahari & B. Albahari 2012, 2.)

C#-kielessä on mukana aika montakin erilaista perustietotyyppiä, niin sanottuja primitiivisiä tietotyyppejä. Eniten erilaisia tyyppejä on numeerisille arvoille. Näiden lisäksi on vielä tyyppi totuusarvolle, merkille, merkkijonolle sekä oliolle. Taulukko 1 esittelee kaikki C#:n primitiiviset tietotyypit. (Nakov & Kolev 2013, 112.)

Taulukko 1. C#-kielen tietotyypit (Nakov & Kolev 2013, 112)

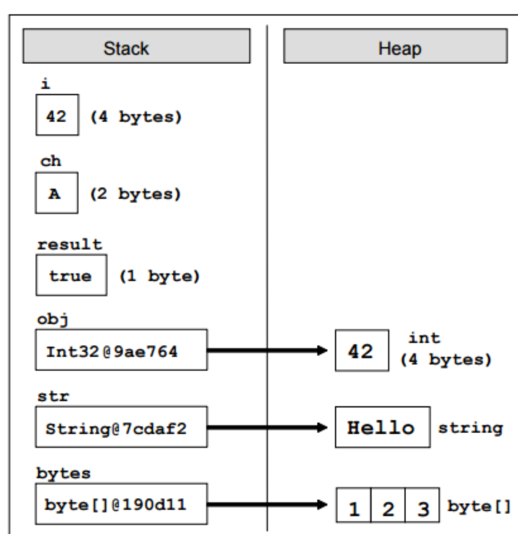
Tietotyyppi	Minimiarvo	Maksimiarvo
sbyte	-128	127
byte	0	255
short	-32768	32767
ushort	0	65535
int	-2147483648	2147483647
uint	0	4294967295
long	-9223372036854775808	9223372036854775807
ulong	0	18446744073709551615
float	$\pm 1.5 \times 10^{-45}$	$\pm 3.4 \times 10^{38}$
double	$\pm 5.0 \times 10^{-324}$	$\pm 1.7 \times 10^{308}$
decimal	$\pm 1.0 \times 10^{-28}$	$\pm 7.9 \times 10^{28}$
bool	Kaksi mahdollista arvoa: true tai false.	
char	'\u0000'	'\uffff'
object	-	-
string	-	-

Tietotyypit jaetaan arvotyyppeihin (value types) sekä viittaustyyppeihin (reference types). Näiden kahden yleisimmän lisäksi on vielä geneeriset tyypit (generic type parameters) sekä osoitintyypit (pointer types). Arvotyyppeihin lukeutuvat suurin osa perustietotyypeistä: jokainen numeerinen tyyppi sekä char että bool -tietotyypit. Viittaustyyppeihin kuuluvat kaikki luokka-, taulukko-, delegaatti- ja rajapintatyypit sekä String. String on viittaustyyppinä muuttumaton (immutable). Tämä tarkoittaa, että kun string-tyyppisen muuttujan arvoa muutetaan, arvo ei muutu vaan muuttuja viittaa uuteen paikkaan kasamuistissa. Perustavanlaatuinen ero arvo- ja viittaustyypeillä on, miten niitä käsitellään muistissa. (J. Albahari & B. Albahari 2012, 19.)

Arvotyyppisen muuttujan sisältö on kaikessa yksinkertaisuudessaan pelkkä arvo. Esimerkiksi perustietotyypin int sisältö on aina 32-bittinen. Tämä tarkoittaa, että arvotyyppinen tietotyyppi vie aina yhtä paljon muistia. Ohjelmoija voi luoda omia arvotyyppejä käyttämällä avainsanaa struct. (J. Albahari & B. Albahari 2012, 19; Nakov & Kolev 2013, 128.)

Viittaustyyppi on arvotyyppiä monimutkaisempi. Viittaustyypissä on kaksi osaa: olio sekä viittaus tähän olioon. Viittaustyyppinen muuttuja sisältää ainoastaan viittauksen muistiin, mistä arvo löytyy. Asettamalla muuttujaan viittaustyypin, kopioidaan vain viittaus, eikä itse arvoa (olio). Tämä eroaa arvotyypeistä, missä asettamalla arvon toiseen muuttujaan, arvo kopioituu. Viittaustyypit eivät ole kooltaan aina yhtä suuria, vaan niille varataan - ja niistä vapautetaan - dynaamisesti muistia. Käyttämättömäksi jääneet viittaustyyppiset muuttujat poistuvat muistista itsestään, kun roskienkerääjä (garbage collector) huomaa, että arvoon ei enää viitata mistään. (J. Albahari & B. Albahari 2012, 20-21; Nakov & Kolev 2013, 128.)

Muistin kannalta olennainen ero arvotyypin ja viittaustyypin välillä on, että arvotyyppi tallentuu ohjelman pinomuistiin (stack), ja viittaustyyppi dynaamiseen kasamuistiin (heap). Arvotyyppiin päästään siis käsiksi suoraan, mutta viittaustyypin arvo täytyy hakea viittauksen avulla kasamuistista. Jos esimerkiksi string-tyyppisen muuttujan merkkijonoa muutetaan, jää vanha merkkijono kasamuistiin odottamaan roskienkerääjää, koska siihen ei enää viittaa yksikään muuttuja. (Nakov & Kolev 2013, 129-131.) Jos metodille annetaan arvotyyppinen parametrimuuttuja, joka on merkitty avainsanalla out tai ref, sitä käsitellään viittaustyyppisesti. Tällöin metodissa viitataan suoraan metodille parametrina annettuun muuttujaan. (Trivedi 2014.)



Kuva 6. Pinomuisti ja kasamuisti (Nakov & Kolev 2013, 129)

2.2.2 C# kielioppi ja näkyvyysalue

C#-kielen kielioppi on saanut inspiraatiota perinteisistä C- ja C++ -kielistä. C#:n kielioppi muistuttaa näitä kahta kieltä ja niiden lisäksi muita kieliä, mitkä ovat ottaneet vaikutteensa niistä, kuten Java. Näiden kielien – ja monien muiden – tapaan C# on tasoherkkä: erikokoisin kirjaimin kirjoitetut muuttujat ovat eri muuttujia. (J. Albahari & B. Albahari 2012, 20-21; Wikibooks 2015.)

Nimiavaruudet merkitään käyttämällä avainsanaa using. Luokka merkitään avainsanalla class. Muuttujat merkitään kirjoittamalla ensin tietotyyppi ja sen jälkeen nimi. Muuttujanimet tyypillisesti kirjoitetaan pienellä alkukirjaimella. Luokalle voidaan lisätä toiminnallisuutta metodien avulla, mitkä merkitään tavallisesti isolla alkukirjaimella CamelCase-periaatteella. Muuttujalle ja metodille voidaan lisätä näkyvyysmääre (access modifier), jolla voidaan hallita kyseisen muuttujan tai metodin näkyvyyttä toisissa luokissa. (J. Albahari & B. Albahari 2012, 12.)

Luokkien ja metodien sisältö laitetaan koodilohkojen sisään. Koodilohkon alku ja loppu merkitään aaltosulkeilla. Koodilohkot merkitsevät myös uuden näkyvyysalueen syntyä (scope). Näkyvyysalueella tarkoitetaan, kuinka esimerkiksi siellä merkityt muuttujat näkyvät muualla ohjelmakoodissa. Alemman tason näkyvyysalue saa käyttöönsä kaikki ylemmän tason muuttujat, mutta ylemmän tason näkyvyysalue ei näe alemmassa tasossa merkittyjä muuttujia. (Carr 2007.)

```
1  public class EnsimmainenNakyvyysAlue {
2      private int a = 1;
3
4      private void ToinenNakyvyysAlue {
5          a = a+1; // Toimii
6          int b = 2;
7          {
8              //Kolmas Näkyvyysalue
9              int c = 3;
10             b = b*2; // Toimii
11         }
12         c = 0; // Ei toimi!
13     }
14
15     b = 1; // Ei toimi!
16     a = a; // Toimii.
17 }
```

Kuva 7. Koodilohkot ja niiden näkyvyysalueet

2.2.3 Muistinhallinta

Yksi .NET-ajoympäristön parhaimmista ominaisuuksista on sen sisään rakennettu automaattinen muistinhallinta. Se jättää ohjelmoijan harteilta vaikean ja monimutkaisen tehtävän varata ja vapauttaa muistia oikealla hetkellä. Automaattinen muistinhallinta nostaa huomattavasti tuottavuutta sekä yleisesti .NET-kielellä kirjoitettujen ohjelmien laatua. (Nakov & Kolev 2013, 80.)

.NET-ajoympäristössä on erityislaatuinen komponentti, joka katsoo muistinhallinnan perään: roskienkerääjä (garbage collector), automaattinen muistin siivousjärjestelmä. Roskienkerääjällä on selvä tehtävä. Se tarkistaa, onko muuttujalle varattu muisti enää käytössä, ja jos näin ei ole, vapautetaan se muistista. On kuitenkin huomattava, että ei ole täysin selvää millä hetkellä muistia siivotaan käyttämättömistä olioista. C#:n kielen määrittelyn mukaan se tapahtuu jonkin ajan päästä siitä, kun muuttuja katoaa näkyvyysalueelta (scope). Ei ole kuitenkaan täsmennetty, tapahtuuko se heti, jonkin ajan päästä vai vasta kun muisti alkaa olla poikkeuksellisen täynnä. (Nakov & Kolev 2013, 80.)

2.2.4 Unity ja C#

Vaikka Unitylla kaikki ohjelmointityö tehdäänkin C#-kielellä (tai vaihtoehtoisesti UnityScriptillä), Unity on C++-pohjainen pelimoottori. Virtuaalikoneessa suoritettava hallittu koodi (managed code) on huomattavasti hitaampaa suorittaa kuin natiivi koodi, ja peleissä on tärkeää saada irti viimeisetkin mehut tietokoneesta korkean ruudunpäivitysnopeuden takaamiseksi. (Tuliper 2014.)

Suuri osa Unityn sisäisestä toiminnasta on kirjoitettu C -tai C++ -kielellä suorituskyvyn maksimoimiseksi. Tästä voi herätä kysymys, miten näitä metodeja voidaan käyttää C#-koodista käsin. Vastaus tähän kysymykseen on sovitin (wrapper). Sovittimen avulla on mahdollista kutsua natiivia C/C++ -koodia suoraan C#-koodista käsin. Tällöin voidaan hyödyntää C#:n helppokäyttöisyys ja kuitenkin ottaa hyöty irti C -ja C++ -kielien tehokkuudesta. (Amazonas 2014.)

Oma koodi kirjoitetaan kuitenkin jollain Unityn hallituista kielistä, kuten C#. Tätä omaa koodia ajetaan pelin suorituksen aikana alustasta riippuen joko Monossa tai Microsoftin .NET-ajoympäristössä. Kaikilla alustoilla paitsi Applen iOS-käyttöjärjestelmässä oma koodi on JIT-käännettyä (Just-in-Time). Sen sijaan iOS-käyttöjärjestelmä ei salli JIT-koodia, ja täten iOS-käyttöjärjestelmällä Mono kääntää koodin natiivikoodiksi AOT-tekniikalla (Ahead-of-Time). (Tuliper 2014; Amazonas 2014.)

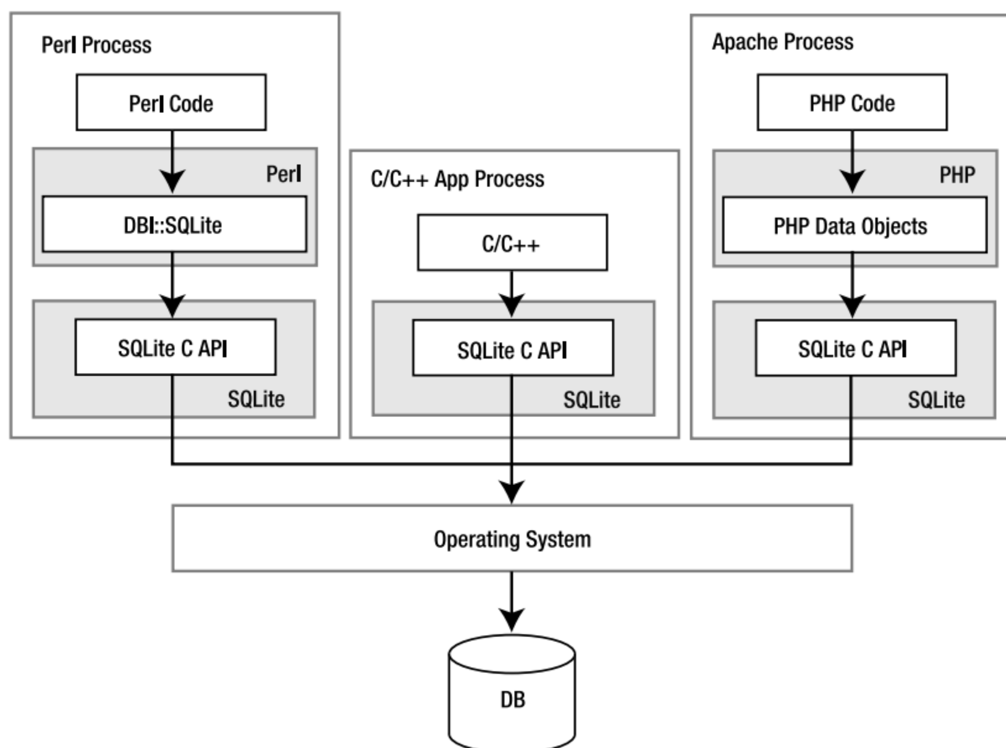
Just-in-Time -kääntäminen tarkoittaa lyhyesti, että kaikkea ajoympäristön virtuaalikoneen ymmärtämää MSIL-kieltä ei käännetä heti natiivikoodiksi. Sen sijaan kieltä käännetään "juuri ajoissa" natiivikoodiksi sitä mukaan, kun sitä kutsutaan. Tämä käännetty lohko säilötään muistiin ja sitä ei tarvitse kääntää enää toista kertaa. (Tomar, 2011.) Ahead-of-Time on samankaltainen prosessi, mutta kaikki koodi käännetään ennalta ennen sovelluksen käynnistystä eikä ajon aikana "juuri ajoissa". (Mittag 2015.)

Mielenkiintoisesti, kuten aiemmin todettu, C# vaatii oman ajoympäristön suoritukseensa. Windowsilla tyypillisesti tämä on .NET, mutta muilla alustoilla voidaan käyttää Monoa. Ajoympäristö on kuitenkin asennettava käyttöjärjestelmään, jotta ohjelmaa voidaan ajaa. Nokkela kuitenkin ymmärtää, että esimerkiksi Androidilla ei taida mitään Monoa ollakaan; Android-sovellukset kirjoitetaan tyypillisesti Javalla. Mono-kääntäjä liittää käännösvaiheessa ohjelman mukaan tarpeellisen osan .NET-luokkakirjastosta, eli ne luokat, mitä tarvitaan ohjelman suoritukseen, sekä tietysti ajoympäristön. (Amazonas 2014.)

2.3 SQLite-tietokanta

SQLite on vuonna 2000 D. Richard Hippin aloittama avoimen lähdekoodin, "upotettava" relaatiotietokantajärjestelmä. Upotettavalla tarkoitetaan tässä, että SQLite-tietokantaa ei ajeta omassa prosessissaan, vaan se ajetaan sitä hyödyntävän ohjelman omassa prosessissa – se on siis uppoutunut, tai kietoutunut, isäntäohjelmaansa (kuva 8). Loppukäyttäjä ei voi tietää, että kyseinen järjestelmä hyödyntää SQLite-

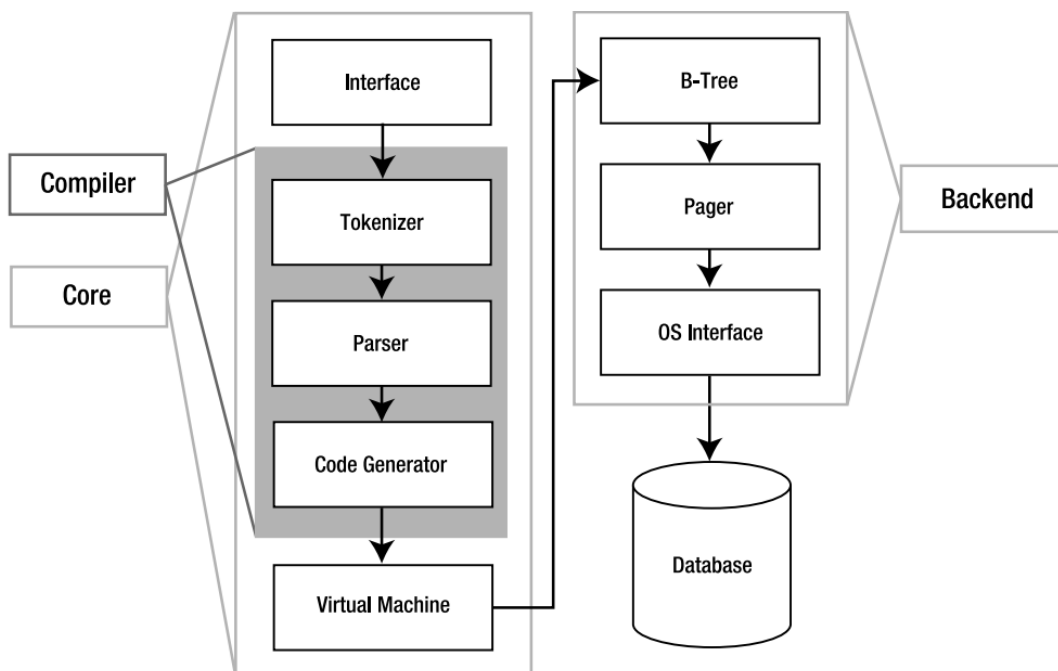
tietokantaa. Tästä, ja minimalistisen luonteensa ansiosta teknologiaa käytetään usein pienlaitteissa, kuten MP3-soitin. (Owens 2006, 1-4.)



Kuva 8. SQLite kietoutuu isäntäohjelmaansa mukaan, eikä suoritu omassa prosessissaan. (Owens 2006, 2)

2.3.1 Arkkitehtuuri

SQLitessä on modulaarinen arkkitehtuuri, missä on melko omalaatuisia lähestymistapoja relationaalisen tietokantajärjestelmän hallintaan. Siinä on kahdeksan erillistä moduulia jaoteltuna kolmeen alajärjestelmään (katso kuva 9). Nämä moduulit jakavat kyselyn prosessoinnin irrallisiksi tehtäviksi, vähän kuin tehtaan kokoonpanolinjalla. (Owens 2006, 6.)



Kuva 9. SQLite-tietokantajärjestelmän arkkitehtuuri (Owens 2006, 5)

Moduulipinon päällimmäisenä on rajapinta-moduuli, mikä pitää sisällään SQLiten C-rajapinnan. Kyselyn kääntäminen tapahtuu moduulikokonaisuudessa kääntäjä. Kääntäjä-vaiheen kaksi ensimmäistä moduulia, tokenizer ja parser (jäsennin), työskentelevät keskenään saadakseen tekstimuotoisesta SQL-kyselystä käännettyä tietorakenteen, jota alemman tason moduulit voivat hyödyntää. Koodigeneraattori (code generator) kääntää jäsennetyn puurakenteen SQLiten omalle assembly-tyyliselle kielelle. Tässä assembly-kielessä on 128 erilaista tietokantakeskeistä konekielistä käskyä, minkä avulla voidaan suorittaa mikä tahansa SQL-kysely. Koodigeneraattori luovuttaa koodin pinon keskellä olevalle virtuaalikoneelle. (Owens 2006, 6.)

Pinon keskellä olevaa virtuaalikonetta kutsutaan nimellä ”virtual database engine” (VDBE). VDBE keskeisin tehtävä on ottaa koodigeneraattorin luoma tavukoodi ja suorittaa se. VDBE:n tavukoodissa on 128 erilaista konekielistä käskyä, mistä jokainen joko suorittaa tietokantaoperaation tai valmistelee pinoa jollain tapaa tietokantaoperaatiota varten. Oikeassa järjestyksessä käskykannalla voidaan suorittaa jokainen validi SQLite-kysely. Jokainen tietokantakysely siis käännetään VDBE:n ymmärtämälle tavukoodille, mikä toimi itsenäisenä ohjelmana kertoen mitä pitää suorittaa. Voikin ajatella, että VDBE on koko SQLiten sydän: kaikki moduulit sen yläpuolella luovat VDBE-ohjelman, ja kaikki sen alla olevat moduulit ovat olemassa sen suoritusta varten. (Owens 2006, 6-7.)

Backend-kokonaisuus koostuu moduuleista B-puu (B-tree), sivuvälimuisti (page cache) sekä käyttöjärjestelmärajapinta (OS interface). B-puu ja sivuvälimuisti järjestelevät sekä siirtelevät tietokantasivuja välittämättä mitä niiden sisällä on. Tietokantasivu on yhtäläisen kokoinen pala dataa, mikä on tehty liikuteltavaksi. B-puun tehtävä on järjestely: se ylläpitää monimutkaisia yhteyksiä eri sivujen välillä ja pitää sivut yhteydessä ja helposti löydettävissä. Nimensä mukaisesti B-puu järjestää sivut puumallisiin rakenteisiin, mitkä on optimoitu hakuja varten. Sivuvälimuisti palvelee B-puuta: sen päätehtävä on syöttää B-puulle sivuja. Sivuvälimuisti hakee ja vie sivuja kiintolevyn ja keskusmuistin välillä. Koska kiintolevyllä tehtävät operaatiot ovat hitainta mitä tietokoneella voi tehdä, sivuvälimuisti yrittää nopeuttaa toimintaa pitämällä useimmiten käytettyjä sivuja keskusmuistissa. Se yrittää myös arvata mitä sivuja B-puu tarvitsee tulevaisuudessa ja täten yrittää pitää toiminnan mahdollisimman sujuvana kaiken aikaa. (Owens 2006, 7.)

Moni asia täytyy tehdä eri tavalla eri käyttöjärjestelmissä – kuten tiedostojen lukitseminen. Käyttöjärjestelmärajapinta tarjoaa abstraktin kerroksen mikä piilottaa nämä eroavaisuudet muilta SQLiten moduuleilta. Tämä tarkoittaa käytännössä siis sitä, että muut moduulit näkevät vain yhden rajapinnan käyttöjärjestelmän kanssa kommunikointiin – käyttöjärjestelmärajapinta ratkaisee sitten, miten asia hoidetaan ajettavassa käyttöjärjestelmässä. Moduuli voi esimerkiksi käskeä käyttöjärjestelmärajapintaa lukitsemaan jonkin tiedoston – sama käsky toimii sekä Windows -että Unix -käyttöjärjestelmissä. Käyttöjärjestelmärajapinta pitää koodin selkeämpänä ja siistinä, ja samaan aikaan helpottaa SQLiten sovittamista uusiin käyttöjärjestelmiin. (Owens 2006, 7.)

2.3.2 Ominaisuudet ja filosofia

SQLite tarjoaa yllättävänkin suuren määrän ominaisuuksia ja kyvykkyyttä verrattain sen hyvin pieneen kokoon. Se tukee hyvin suurta osaa ANSI SQL92 -standardista ja monia muita ominaisuuksia, mitä on tyypillisesti löydettävissä relaatiotietokannoista, kuten laukaisimet (triggers) ja indeksit. Muihin ominaisuuksiin lukeutuvat muun muassa muistissa olevat tietokannat (in-memory databases) ja konfliktin ratkaisija (conflict resolution). (Owens 2006, 8.)

SQLite on alusta alkaen suunniteltu niin, että sen hallinnointiin ei tarvita erillistä tietokannan ylläpitäjää (DBA). SQLiten konfigurointi sekä ylläpito on melko yksinkertaista. SQLiten ominaisuuslista on sen verran suppea, että se on ohjelmoijan helppo omaksua. Samaan tyyliin se myös vaatii hyvin vähän sekä kiintolevytilaa että RAM-muistia. (Owens 2006, 8.)

SQLite suunniteltiin erityisesti siirreltäväksi (portable) eli käytettäväksi useissa eri ympäristöissä. SQLite kääntyy yleisimpien Windowsin, Mac OS X:n ja Linuxin lisäksi erittäin monelle muullekin käyttöjärjestelmälle. Tietokantatiedostot ovat yhtä siirreltäviä kuin itse koodikin: tietokantoja pystyy avaamaan kaikilla käyttöjärjestelmillä yhtä lailla, riippumatta missä ne on luotu. (Owens 2006, 8.)

Nimensä mukaisesti SQLite on suunniteltu erittäin kevyeksi ja kompaktiksi kokonaisuudeksi. Huomioitavana asiana se ei käytä erillistä tietokantapalvelinta vaan kaikki – asiakasohjelma, palvelin ja virtuaalikone – on pakattuna noin megatavun neljännekseen. SQLitestä on mahdollista karsia ominaisuuksia ja pudottaa tiedostokokoa entisestään, jopa alle 170 kilotavun. SQLitestä on myös patentoitu versio, mikä on kooltaan vain 69 kilotavua, jolloin sitä voidaan käyttää jopa sirukorteissa. (Owens 2006, 9.)

2.3.3 Suorituskyky ja rajat

SQLiteä voisi kuvailla sanoilla ”nopea” tai ”vilkas”. Termit ovat kuitenkin hyvin subjektiivisia, eivätkä kerro todellisuudesta oikeastaan mitään. Yksinkertaisissa kyselyissä SQLite on yhtä nopea, ellei nopeampi kuin muut relaatiotietokantajärjestelmät. Tämä johtuu siitä, että sillä on vähemmän yleiskulua (overhead) transaktion aloittamisessa tai kyselysuunnitelman (query plan) generoimisessa. Sen yksinkertaisuus siis tekee siitä nopean. Kun kyselyistä alkaa tulla monimutkaisia ja suuria, yleensä suuremmat tietokantajärjestelmät alkavat näyttämään hampaitaan. Tällaisissa kyselyissä voittaja on järjestelmä, missä on paras optimoija. SQLitessä ei ole erityisen hienostunutta optimoijaa tai kyselyn suunnittelijaa, ja täten se ei nopeudessa voita, kun kyse on monimutkaisista kyselyistä. (Owens 2006, 11.)

Kaiken kaikkiaan SQLitessä on kolme merkitsevintä rajoitusta. Ensinnäkin samanaikaisuus. SQLite sallii monta lukijaa, mutta vain yhden kirjoittajan kerrallaan. Kirjoittaja lukitsee koko tietokannan eikä kenelläkään muulla ole pääsyä tietokantaan tämän aikana. SQLite yrittää minimoida lukitun ajan, ja lukitus on käytössä joitakin millisekunteja kerrallaan. Jos tietokantaan kirjoitetaan kuitenkin erittäin tiheään tahtiin, voi tästä muodostua ongelma. (Owens 2006, 11.)

SQLiten tietokannat voivat skaalautua kahteen teratavuun, mutta huomattavampana rajoituksena sen RAM-vaatimukset nousevat tietokantakoon mukana. Aloittaessaan tapahtuman SQLite luo bittikartan (bitmap) niin sanottujen likaisten sivujen (dirty pages) seuraamiseen, mikä auttaa hallitsemaan mahdollisesti peruutettavia tapahtumia. Likainen sivu on sellainen, joka sisältää muutettua dataa, jota ei ole vielä talletettu pysyväismuistiin. Tähän SQLite tarvitsee 256 tavua RAM-muistia per tietokannan megatavu. Jos tietokannasta tulee huomattavan suuri, bittikartan koosta per transaktio voi tulla huomattavan suuri. Esimerkkinä 100 gigatavun tietokannassa jokainen transaktio vaatisi 25 megatavua RAM-muistia ennen suoritusta. Eli vaikka teoreettinen maksimikoko on muutama teratavu, RAM-rajoitus tulee mahdollisesti vastaan aiemmin. (Owens 2006, 11.)

Viimeisenä, vaikka SQLiten tietokannat voi jakaa verkkotiedostojärjestelmissä, sen mukana tuoma viive aiheuttaa suorituskyvyn heikentymistä. Virheet verkkotiedostojärjestelmissä voivat tehdä SQLitestä myös virhealttiin. Jos tiedostojärjestelmän lukitus ei toimi kunnolla, edessä voi olla jopa korruptoitunut tietokanta. (Owens 2006, 12.)

3 OHJELMISTON KETTERÄ KEHITTÄMINEN

Ketterä ohjelmistokehitys (agile software development) on projektin hallintamenetelmä. Ketteriä menetelmiä on useita, mutta yleisesti niissä kaikissa painotetaan nopeaa liiketoiminnallista arvoa, jatkuvaa projektin ja prosessien kehitystä, projektin laajuuden joustavuutta, koko tiimin syötettä sekä hyvin testattujen ja asiakkaiden tar-

peet huomioivien tuotteiden toimittamista asiakkaille. Jostain syystä, vaikka teknologia on edennyt huimaa vauhtia, ohjelmistokehityksen prosessit ovat jääneet jälkeen. Osa ohjelmistokehittäjistä käyttää yhä 1950-luvulla kehitettyjä projektinhallintamenetelmiä. Viimeisen parin vuosikymmenen ajan projektien kanssa työskentelevät ihmiset ovat huomanneet kasvavan ongelman perinteisten projektinhallintamenetelmien kanssa. Tämän seurauksena on kehittynyt ketterä projektinhallinta. (Layton 2012, 6-10.)

Ketterät menetelmät perustuvat empiiriseen ohjaustapaan (empirical control method), prosessiin missä päätöksenteko perustuu todellisiin havaintoihin projektin osalta. Empiirinen ohjaus vaatii kolmea asiaa:

- Läpinäkyvyyttä: jokainen projektissa mukana oleva tietää mitä tapahtuu ja miten projekti etenee.
- Tiheää tarkastelua: projektissa mukana olevien tulisi tarkastella ja arvioida tuotetta ja prosessia säännöllisin väliajoin.
- Mukautumiskykyä: tehdään oikaisuja nopeasti ja minimoidaan ongelmia. Jos tarkastelu näyttää, että jotain tulisi muuttaa, niin muutetaan se nopeasti. (Layton 2012, 12.)

Tiheän tarkastelun ja mukautumiskyvyn sovittamiseksi ketterät ohjelmistoprojektit tehdään pienemmissä palasissa. Ketterissä projekteissa tehdään saman tyyppistä työtä kuin perinteisessä vesiputousmallissakin. Täytyy luoda määritykset ja suunnittelut, kehittää ohjelmisto ja mahdollisesti integroida tuote toisiin tuotteisiin. Projektia testataan, ongelmat korjataan ja lopulta se otetaan käyttöön. Näitä vaiheita ei tosin tehdä kaikkea kerralla, vaan projekti jaetaan pieniin palasiin, mitä kutsutaan myös sprinteiksi. (Layton 2012, 12.) Joitakin suosituimpia ketteriä ohjelmistokehitysmetodologioita ovat Feature Driven Development (FDD), Lean Software Development, Crystal, Scrum sekä Extreme Programming. (Coffin & Lane 2006.)

3.1 Extreme Programming

Yksi tällainen suosittu ketterä ohjelmistokehitysmenetelmä on extreme programming (lyh. XP). Sen on kehittänyt Kent Beck 1996 Ward Cunninghamin ja Ron Jeffriesin

avulla. Extreme Programmingin keskipisteenä on asiakastyytyväisyys. Tiimi luo ominaisuuksia tuotteeseen silloin, kun asiakas tarvitsee niitä. Jos ongelmia ilmenee, tiimi kokoontuu yhteen ja ratkaisee ongelman niin tehokkaasti kuin mahdollista. (Layton 2012, 58.)

Extreme Programmingin lähestymistavat perustuvat ketterän ohjelmistokehityksen periaatteisiin. Koodaus on ydinaktiviteetti. Ratkaisun lisäksi koodausta voidaan käyttää tutkimaan ongelmia. Esimerkiksi ohjelmoija voi havainnollistaa ongelmaa koodaamalla. Extreme Programmingissa tiimit tekevät paljon testausta. Itseasiassa kehittäjät eivät aloita koodaamista ollenkaan, ennen kuin onnistumisen kriteerit ovat selvät ja yksikkötestaukset on suunniteltu. Kommunikointi asiakkaan ja ohjelmoijan välillä on suoraa; ohjelmoijan täytyy ymmärtää asiakkaan tarve ja suunnitella sen pohjalta ratkaisu. Laajoissa projekteissa jonkin asteinen suurpiirteinen suunnittelu on välttämätöntä. (Layton 2012, 59.)

Testivetoinen kehitys (test-driven development) on yksi extreme programming -menetelmän toimintatavoista. Ohjelmistokehittäjä aloittaa luomalla testin ominaisuudelle, minkä hän haluaa luoda. Kehittäjä ajaa testin, minkä pitäisi epäonnistua sillä kyseistä ominaisuutta ei ole vielä tehty. Hän kehittää ominaisuutta, kunnes testi menee läpi. Tämän jälkeen hän aloittaa refaktoroinnin, eli parantaa koodin sisäistä rakennetta, eli luettavuutta. (Layton 2012, 220; Jeffries 2011.)

Pariohjelmointi on tyypillinen toimintatapa extreme programming -menetelmässä. Kehittäjät työskentelevät kahden hengen ryhmissä. Molemmat kehittäjät istuvat saman tietokoneen ääressä ja työskentelevät yhdessä luodakseen yhden ominaisuuden tuotteeseen. Kehittäjät vaihtavat välillä, kumpi on tietokoneesta hallinnassa. Kun toinen kehittäjä tarkkailee kehitystä vierestä, virheet tulee huomattua nopeasti. Ongelmien ratkominen on myös nopeampaa kahden pään avulla. Pariohjelmointi lisää ohjelmistokoodin laatua. (Layton 2012, 220; Jeffries 2011.)

Extreme programming -menetelmässä kaikki omistavat koko koodipohjan. Jokainen kehitystiimissä oleva voi luoda, muuttaa tai korjata mitä tahansa osaa koodista projektissa. Yhteinen koodin omistajuus voi nopeuttaa kehitystä, rohkaista innovointia ja nopeuttaa virheiden löytämistä. (Layton 2012, 221; Jeffries 2011.)

Extreme Programmingissa tiimi seuraa yhteistä koodausstandardia, jotta kaikki koodit näyttäisivät siltä kuin ne olisi kirjoittanut yksittäinen – erittäin pätevä – henkilö. Tärkeää ei ole kuitenkaan se, millaista standardia noudattaa. Pääasia on, että kaikki koodi näyttää samalta, jolloin se tukee yhteisen koodipohjan ideaa. (Jeffries 2011.)

3.2 Hyviä ohjelmointikäytäntöjä

Hyvien ohjelmointikäytäntöjen noudattaminen ei ehkä tule itsestään selvyytenä. On aivan liian helppoa kirjoittaa nopeasti ja huonosti, pahimmillaan mitä sattuu ilman minkäänlaista kuria. Myöhemmin voi vaipua epätoivoon, kun joutuu selvittämään omaa sotkuaan. Ikävä kyllä myös sivulliset saattavat joutua huonon koodin uhriksi. Tämän takia on erityisen tärkeää ylläpitää edes jonkinlaista koodausstandardia.

Nimet ovat arkipäivää ohjelmoinnissa. Nimeämme luokkamme, muuttujamme, metodimme ja niin edelleen. Koska nimeämme asioita niin usein, meidän on parempi nimetä ne kunnolla. Hyvän nimen keksiminen vie aikaa, mutta säästää aikaa senkin edestä. Paremman nimen ilmetessä, on parempi muuttaa vanha nimi. Kaikki koodia lukevat ilahtuvat siitä. Muuttujan, metodin tai muun sellaisen nimen pitäisi kertoa selvästi tarkoituksensa. Jos nimi vaatii kommentointia, nimi ei kerro tarkoitustaan. (Martin 2009, 18.)

Luokkien pitäisi olla suhteellisen lyhyitä. Luokan kokoa voidaan mitata sillä, kuinka monesta asiasta se on vastuussa. Luokan nimen pitäisi kertoa, minkä vastuun se kantaa. Mitä epämääräisempi tai tulkinnanvaraisempi luokan nimi on, sitä todennäköisemmin se kantaa aivan liikaa vastuuta. (Martin 2009, 136-138.) Metodien tulisi olla mahdollisimman lyhyitä ja tehdä yksi asia, ja tehdä se kunnolla. Koodia tulisi pystyä lukemaan ylhäältä alaspäin, ikään kuin kertomusta. Toisiinsa liittyvät metodit tulisi olla lähekkäin toisiaan. (Martin 2009, 37.)

3.3 Versionhallinta

Uutta projektia ei kannata lähteä tekemään ilman varasuunnitelmaa. Data on yllättävänkin helposti katoavaista ja ohimenevää, sillä koodia muutetaan jatkuvasti. Onkin järkevää ylläpitää jatkuvaa arkistointia omasta työstään. Mahdollisesti rikkoessaan koko järjestelmän koodimuutoksilla, voi aina halutessaan palata siihen mistä lähti liikkeelle. Oma työ kannattaa myös tallentaa pilveen, sillä tunnetusti kiintolevyt eivät kestä ikuisesti. Rikkinäiseltä massamuistilta voi olla jopa mahdotonta saada dataansa takaisin. (Loeliger & McCullough 2012, 1.)

Teksti- ja koodausprojekteille tällainen varasuunnitelma sisältää tyypillisesti versionhallinnan. Jokainen kehittäjä voi tehdä useita muutoksia päivässä, ja jatkuvasti kasvava koodikokoelma palvelee samanaikaisesti talletuspaikkana, projektin kuvauksena, kommunikointivälineenä sekä tiimin- että projektinhallintatyökaluna. Versionhallinta on tehokkaimmillaan, kun se on räätälöity projektitiimin tapoja ja tavoitteita ajatellen. Työkalua, joka hallinnoi ja seuraa muutoksia ohjelmistossa tai muussa sisällössä kutsutaan tyypillisesti versionhallintajärjestelmäksi (version control system, VCS), lähdekoodimanageriksi (source code manager, SCM), muutoksenhallintajärjestelmäksi (revision control system, RCS) sekä useilla muilla nimillä, missä esiintyy sanoja kuten versio, koodi, sisältö, hallinta ja muita sen sellaisia. (Loeliger & McCullough 2012, 1.)

The Source Code Control System (SCCS) oli yksi ensimmäisiä järjestelmiä Unixille. Sen kehitti M. J. Rochkind 1970-luvun alussa. SCCS tarjosi niin sanotun repositoryn eli talletuspaikan, ja tämä käsite on säilynyt tähän päivään asti. SCCS-järjestelmää seurasi seuraavaksi 1980-luvun alussa Walter F. Tichyn esittelemä Revision Control System (RCS). Molemmissa järjestelmissä oli yksinkertainen lukitusjärjestelmä. Jos kehittäjä tarvitsi tiedostoja suorittaakseen ohjelman, hän ottaisi ne lukitsemattomana. Muokatakseen tiedostoa kehittäjän täytyisi kuitenkin ottaa tiedosto lukittuna. Kun työ tulee valmiiksi, kehittäjä syöttää tiedoston takaisin talletuspaikkaan (repository) ja avaa lukon. (Loeliger & McCullough 2012, 4.)

Samalla vuosikymmenellä tuli The Concurrent Version System (CVS), jonka alun perin suunnitteli ja toteutti Dick Grune vuonna 1986. Noin neljä vuotta myöhemmin

Brian Berliner ja kumppanit laajensivat järjestelmää suurella menestyksellä. CVS tarjosi useita parannuksia verrattuna RCS-järjestelmään, kuten hajautetun kehityksen. Siinä missä aiemmat järjestelmät olivat vaatineet tiedoston lukitsemista muutoksien ajaksi, CVS antoi jokaiselle kehittäjälle oman toimivan kopion projektista. Kehittäjä muutti omaa versiotaan ohjelmistosta. Eri kehittäjien muutokset liitettiin yhteen automaattisesti CVS:n toimesta. Jos kaksi kehittäjää oli yrittänyt muokata samaa kohtaa tiedostosta, ristiriita (conflict) täytyi ratkaista manuaalisesti kehittäjien toimesta. Lukitusmekanismin poisto salli kehittäjien kirjoittaa rinnakkain samoja tiedostoja. (Loeliger & McCullough 2012, 5.)

Kuten usein käy, CVS-järjestelmässäkin huomattiin omat heikkoutensa ja täten luotiin jälleen uusi versionhallintaohjelmisto. Myöhemmin tuli Subversion (SVN), BitKeeper, Mercurial sekä Git. BitKeeper ja Mercurial poikkesivat huomattavasti edellä mainituista järjestelmistä, sillä kummassakaan ei ollut enää tarvetta keskitetylle talletuspaikalle. Sen sijaan talletuspaikka oli täysin hajautettu, antaen jokaiselle kehittäjälle oman, jaettavan kopion. Myöhemmin luotu Git-versionhallinta on johdettu tästä vertaisverkko-ajattelusta (peer-to-peer). (Loeliger & McCullough 2012, 6; Santacroce 2015, 1.)

3.3.1 Git-versionhallintaohjelmisto

Linux-ytimen (kernel) kehittäminen tapahtui alun perin kaupallisella BitKeeper-versionhallintaohjelmistolla, joka oli aikanaan soveliaampi projektille kuin ilmaiset vastineet. BitKeeper oli ilmainen avoimen lähdekoodin ohjelmistoille, kunnes se asetti rajoituksia ilmaisversioon keväällä 2005. Tällöin Linux-yhteisö totesi, että BitKeeper ei ole enää soveltuva ratkaisu projektille. Loppujen lopuksi Linus Torvalds sekä moni muu kehittäjä tekivät yhteistyössä uuden, vapaan lähdekoodin versionhallintaohjelmiston. Ohjelmisto sai nimekseen ”Git”, mikä tarkoittaa isobritannialaisessa slangissa ääliötä. Nimi on Linus Torvaldsin antama, ja hän kommentoi nimeä seuraavasti: ”I’m an egotistical bastard, and I name all my projects after myself. First Linux, now git.” (olen itsekeskeinen paskiainen ja nimeän kaikki projektini itseni mukaan. Ensin Linux, sitten git). (Loeliger & McCullough 2012, 2-7; Santacroce 2015, 1.)

Git jakaa paljon samaa muiden versionhallintajärjestelmien kanssa. Useimmat termit joita käytetään muissa versionhallintaohjelmistoissa löytyvät myös gitistä. Tällaisia ovat esimerkiksi repository, commit sekä changelog. (Loeliger & McCullough 2012, 19.) Jos on aiemmin käyttänyt jotakin muuta versionhallintaohjelmistoa kuten SVN tai CVS, useimmat komennot voivat näyttää hyvinkin tutuilta. (Loeliger & McCullough 2012, 31.)

Gitin yksi oleellisimpia ominaisuuksia on jo ensimmäisistä versionhallintaohjelmistoista tuttu repository, tietovarasto. Repository on yksinkertaistettuna eräänlainen tietokanta, mikä säilyttää kaiken tiedon mitä tarvitaan säilyttämään ja hallitsemaan projektin muutoksia. Gitissä monien muiden versionhallintaohjelmistojen tapaan repository pitää sisällään kopion koko projektista koko sen elinkaaren ajan. Toisin kuin useimmat versionhallintaohjelmistot, Git pitää projektin lisäksi kopion myös itse repositorysta minkä kautta työskennellään. Repository pitää sisällään muun muassa myös konfigurointiasetuksia, esimerkiksi kyseisen repositoryn omistajan nimen sekä sähköpostiosoitteen. Jos repository kopioidaan (clone), nämä tiedot eivät kopioidu mukaan. (Loeliger & McCullough 2012, 31; Santacroce 2015, 13.)

Gitin repositoryn keskeisimpänä ominaisuutena on objektivarasto (object store). Varasto sisältää kaikki alkuperäiset tiedostot ja lokimerkinnät. Varaston avulla on mahdollista uudelleen rakentaa mikä tahansa versio projektista. Git laittaa neljää erilaista objektia varastoon: blob, tree, commit ja tag. Jokaista tiedoston eri versiota edustaa oma blob. Blob tulee sanoista ”binary large object”, millä tarkoitetaan tiedostoa mikä voi sisältää minkä kaltaista dataa tahansa eikä Gitin tarvitse välittää sen sisällöstä. Tree eli puu edustaa yhden hakemiston informaatiota. Se tallentaa blobien tunnisteet ja tiedostopolut. Tämän lisäksi se ylläpitää hakemistosta hieman metadataa. Commit-objekti säilöö metadataa siitä, mitä muutoksia repositoryyn on tehty. Jokainen commit osoittaa tree-objektiin mikä säilöö repositoryn tilan sillä hetkellä, kun commit suoritetaan. Lopulta tag-objekti asettaa vapaavalintaisen, ihmiselle helpommin luettavan nimen jollekin objektille. Useimmiten tälläinen objekti on commit-objekti. Commit-objektiin voidaan viitata ilman tagia sen SHA1-tiivisteen avulla, mutta se on hankala kirjoittaa, muistaa tai lukea. (Loeliger & McCullough 2012, 32; Santacroce 2015, 25.)

Git ei kuitenkaan suinkaan säilö oikeasti jokaisen tiedoston eri versiota kokonaisuudessaan. Jos esimerkiksi tiedostosta muutetaan yksi rivi, niin Git ei säilö kahta lähes identtistä tiedostoa. Git käyttää tehokasta säilytysmekanismia, mitä kutsutaan pack fileksi (tiedostopakkaus). Luodakseen paketin, Git paikantaa tiedostot jotka ovat hyvin samankaltaisia keskenään ja säilöö kokonaisuudessaan niistä yhden. Sen jälkeen se käy läpi tiedostojen eroavaisuudet ja tallentaa ne. Palaten aiempaan esimerkkiin, jos muutettaisiin yhtä riviä tiedostosta, Git säilöisi oletettavasti kokonaisuudessaan uudemman version ja tallentaisi merkinnän, mikä rivi muuttui tiedostosta. Tämä ei ole Gitille omalaatuinen ratkaisu, vaan samankaltaista ratkaisua on käyttäneet jo vuosikymmeniä muut versionhallintaohjelmistot. (Loeliger & McCullough 2012, 36.)

Kun halutaan tehdä muutoksia repositoryyn, tehdään niin sanottu commit-operaatio. Commit tallentaa repositoryyn kaikki muutokset, mitä on tehty edellisen commit-operaation jälkeen. Gitissä on kuitenkin eräänlainen oma tasonsa työskentelyhakemiston ja repositoryn välillä, mitä kutsutaan indeksiksi (index). Gitin indeksi ei ylläpidä mitään tietoa tiedostojen sisällöstä. Se ainoastaan seuraa muutoksia työhakemistossa. Kun lopulta suoritetaan commit-operaatio, Git tarkistaa indeksistä muutokset sen sijaan että se katsoisi työskentelyhakemistosta. Gitin korkean tason komennot on suunniteltu piilottamaan indeksi ja tehden näin käytöstä yksinkertaisempaa. (Loeliger & McCullough 2012, 48; Santacroce 2015, 11.)

4 CASE: HUNGRY BUMS

Hauskaa, mistä kaikesta peli-idea voikin syntyä. Saimme idean Hungry Bums-peliin, kun ohitsemme pyöräili poika, joka piteli pyörän tavaratelineellä pizzalaatikkoa. Koska yritämme kääriä inspiraatiota kaikkialta ympäröivästä maailmastamme, huomasimme nopeasti, että tässähän on täysin uudenlaisen pelin ainekset. Lyhyen, välittömän keskustelun jälkeen päätimme, että jatkamme saamamme idean jalostamista myöhemmin. Liitteessä yksi on pelin tarina.

Lopulliseksi pelin ideaksi muodostui kokonaisuus, missä pyöräillään kaduilla jakaen pizzaa karhujen runtelemille ja nälkäisille kodittomille sekä yritetään vapauttaa kaupunki karhujen vallasta eliminoimalla nämä erilaisten aseiden avulla. Täten tärkeimmäksi toteutettavaksi tuli itse pyöräily, kentän ikuinen generointi, pizzojen jakaminen, ampuminen sekä vihollisten tekoäly. Ajattelimme myös, että peliin saisi lisäarvoa, jos pelaaja voisi rakentaa pyörän itse erilaisista osista. Päätimme käyttää projektissa meille jo tuttua ja turvallista Unity3D-pelimoottoria kehityksen nopeuttamiseksi.

Pelin alustavalinnaksi muodostui ensisijaisesti Android, mutta tarkoituksena olisi hyödyntää Unityn kattavaa alustatukea ja mahdollisesti julkaista peli kaikille merkittäville mobiilikäyttöjärjestelmille. Androidin lisäksi tämä siis tarkoittaa iOS- sekä Windows Phone -käyttöjärjestelmiä. Testaamisen helpottamiseksi pelissä on tosin hyvät valmiudet myös tietokoneella pelaamiseen. Selonteko ohjelmoinnin työnjaosta löytyy liitteestä kolme.

4.1 Game Manager

Erityisesti pelin kehittämisen nopeuttamiseksi kehitimme komponentin, mikä sisältää referenssit muihin tärkeisiin komponentteihin, mihin tarvitaan usein viite muista skripteistä. Tällaisia ovat muun muassa BikeController, Player, ObjectPoolManager, EffectPoolManager, RoadGenerator, MobileUI, AchievementManager ja monia muita ohjelmoimiamme komponentteja. Tällöin voidaan hakea helposti ja teknisesti nopeasti viite komponenttiin (kuva 10).

```
protected override void OnDeath()
{
    base.OnDeath();
    animator.Play("death");
    GameManager.Instance.EffectPoolManager.RequestEffectAtPosition("CoolBloodFX", hitBox.transform.position);
    GameManager.Instance.EffectPoolManager.RequestEffectAtPosition("EpicExplosion", hitBox.transform.position,0f,2.5f,4f);
    Invoke("Despawn", 4f);
    this.hitBox.enabled = false;
}
```

Kuva 10. GameManagerin käyttöä vihollisen kuolemismetodissa

GameManagerilla on staattinen, julkinen oman tyyppinsä muuttuja nimeltä Instance. Staattisuuden ansiosta viitteisiin päästään käsiksi suoraan kaikkialta Instance-muuttujan kautta, ilman että aina tarvitsisi hakea erikseen GameManager-

komponenttia. GameManager-komponenttia saa olla vain yksi instanssi kerrallaan. Tätä kutsutaan singleton-suunnittelumalliksi. Tämän varmistamiseksi tuhoamme Awake-metodissa uuden instanssin, jos staattinen Instance-muuttuja ei ole arvoltaan null (kuva 11). Kun kentästä poistutaan, Unity kutsuu komponentin OnDestroy-metodia, missä määritetään Instance taas null-arvoiseksi.

```
public static GameManager Instance { get; private set; }

Get/Setters

public void Awake()
{
    if (Instance != null)
    {
        Destroy(this);
        return;
    }
    Instance = this;
```

Kuva 11. GameManager-komponentista voi olla vain yksi instanssi

4.2 Kentän generointi

Kentän generointi on yksi projektin keskeisimmistä toiminnoista, ja halusimme tehdä siitä niin hyvän, että teoriassa kenttä pystyisi jatkumaan loputtomiin. Kentän generoinnista vastaa niin kutsuttu ”RoadGenerator”-luokka, joka on suurimmaksi osaksi Jussin käsialaa. RoadGenerator vastaa yksinomaan kentän ensisijaisesta luomisesta ja tämän jälkeen sen jatkamisesta pelaajan edetessä, sekä vanhojen palojen poistamisesta. Kuva 12 näyttää luodun kentän ylhäältä päin kuvattuna.

Aluksi RoadGenerator loi ilmentymiä massamuistissa olleista kenttäpalasien prefab-tiedostoista ja sekoitti niiden sisältöä, jolloin samasta tiepalasesta oli monta erilaista variaatiota. Tämä oli kuitenkin liian hidasta, sillä se aiheutti huomattavaa nykimistä palasta ladatessa jopa tehokkaamman älypuhelimen kanssa. Tämän johdosta loimme ”pooling”-järjestelmän tiepalasille, mitä voi ajatella eräänlaisena varastona. Kentän alussa luodaan sopiva määrä variaatioita jokaisesta tiepalasesta ja instansioidaan ne valmiiksi kenttään epäaktiivisina. Kun järjestelmä haluaa tuoda pelaajalle uuden tiepalasen, se hakee epäaktiivisen tiepalasen kentältä, muuttaa sen koordinaatteja 3D-

avaruudessa x- ja z-akselilla sekä aktivoi sen. Palasen poistuessa kentältä se muutetaan taas epäaktiiviseksi ja täten tavallaan merkataan käytettäväksi uudelleen.

Kuva 12. RoadGeneratorin luoma (normaalia pidempi) kenttä juuri pelin alettua

Vaikka pelissä on useita pelattavia kenttiä, pelissä on vain yksi varsinainen kenttätiedosto (scene). RoadGeneratorilla on muuttuja tileSet, jonka perusteella RoadGenerator luo kentän halutuista tiepalasista. TileSet saa arvonsa Unityn PlayerPrefs-luokasta, johon on tallennettu pelaajan kenttävalinta avaimella ”level” (kuva 13). Tällaisella ratkaisulla ei taritse ylläpitää useampaa kenttätiedostoa, säästäen arvokasta aikaa, vaivaa ja tallennustilaa.

```
// Which tileset to load? If 'level' is undefined, tileSet will be given the default value, Pori
tileSet = PlayerPrefs.GetString("level", "Pori");

// Load all roadblock prefabs from given tileSet
roadblocks = Resources.LoadAll("Roads/" + tileSet + "/");

// Create an array for the roadblocks
roads = new GameObject[roadAmount];
```

Kuva 13. Ote RoadGeneratorin Awake-metodista

4.3 Kentän valinta

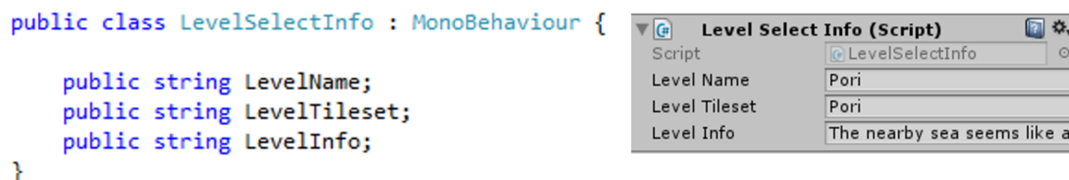
Kenttävalintaa varten rakennettiin kuvan 14 mukainen kenttävalikko. Valikossa näkyvät avonaiset lukot esittävät pelattavia kenttiä, ja suljetut lukot esittävät myöhemmin avautuvia kenttiä. Kenttävalintaa varten ohjelmoitiin LevelSelectManager-komponentti, jonka vastuulla on tallentaa PlayerPrefs-muistiin pelaajan valitsema kenttä sekä pelikentän käynnistäminen. Avonaiset lukko-kuvakkeet toimivat painik-

keina, joita napsauttaessa kutsutaan LevelSelectManagerin metodia LevelSelect_OnClick, joka päivittää valikossa näkyvät tekstit napsautetun kentän tietojen perusteella sekä tallentaa valitun kentän avaimen PlayerPrefs-muistiin (kuva 15).

Kuva 14. Kenttävalikko

Kentän tiedot LevelSelectManager saa lukkopainikkeessa kiinni olevasta LevelSelectInfo-komponentista. LevelSelectInfo on yksinkertainen luokka sisältäen kolme muuttujaa: LevelName, LevelTileSet ja LevelInfo. LevelSelectInfon muuttujat määritellään Unityn editorissa kuvan 16 mukaisesti. Valikon alakulmassa on painike, jota napsauttaessa kutsutaan LevelSelectManagerin PlayButton_OnClick-metodia, joka lataa pelikentän (Kuva 17). Varsinainen kenttä generoituu RoadGeneratorin puolesta, joka luo kentän LevelSelectManagerin tallentaman kenttävalinnan perusteella.

```
public void LevelSelect_OnClick(Button button)
{
    LevelSelectInfo levelInfo = button.transform.GetComponent<LevelSelectInfo>();

    //Set selected level to memory
    PlayerPrefs.SetString("level", levelInfo.LevelTileset);
    PlayerPrefs.Save();

    //Change UI Texts
    levelSelectText.text = levelInfo.LevelName;
    levelSelectInfo.text = levelInfo.LevelInfo;

    Debug.LogWarning(PlayerPrefs.GetString("level"));
}
```
Kuva 15. LevelSelectManagerin LevelSelect_OnClick-metodi



Kuva 16. LevelSelectInfo-komponentin koodi kokonaisuudessaan

```
public void PlayButton_OnClick()
{
    GameObject.Find("Fade").GetComponent<FadeInOut>().EndScene("gamelevel");
}
```

Kuva 17. LevelSelectManagerin PlayButton_OnClick-metodi

4.4 Pyörän liikkuminen ja ohjaus

Polkupyörän liikkuminen ja ohjaus ovat ehdottomasti pelin keskeisin osa, joka osoittautui myös huomattavasti vaikeammaksi osaksi toteuttaa, kuin osasimme odottaa. Halusimme polkupyörän ohjauksen tuntuvan luonnolliselta, mutta pyörä ei saisi kuitenkaan kaatua. Se vaati paljon fysiikkamoottorin kanssa taistelemista, mutta lopulta keksimme meille toimivan ratkaisun.

Päädyimme käyttämään Unityn fysiikkamoottorin tarjoamaa WheelCollider-komponenttia polkupyörän renkaiden simulointiin. WheelCollider tarjoaa esimerkiksi renkaan koon, kääntymiskulman, pyörimisvoiman sekä jarrutusvoiman hallinnan. Suurin ongelma oli estää polkupyörän kaatuminen, kuitenkin niin että pyörällä kääntyessä pyörä kallistuisi kääntösuuntaan realistisella tavalla.

Ongelmaan keksimme nokkelan ratkaisun – pidämme polkupyörän fyysisen objektin sekä polkupyörän graafisen objektin erillään toisistaan. Fyysisen objektin pidämme aina, myös kääntyessä, pystyssä. Käytännössä ohjelmoimamme koodi korjaa fyysistä polkupyörää jatkuvasti pystyasentoon. Graafinen objekti sen sijaan kallistuu kääntyessä, näin ollen vaikuttamatta fysiikkoihin. Kuvassa 18 on koodi, joka korjaa polkupyörän fyysistä objektia pystyasentoon sekä kallistaa polkupyörän graafista objektia, kun pyörä kääntyy vauhdissa.

```
// Fix the bike's rigidbody rotation upright
bikeRigidBody.MoveRotation(bikeRigidBody.rotation * bikeRigidBodyRotation);

// If steering and accelerating, tilt the bike
if((steering != 0f || ((steering > 0.33f) || (steering < -0.33f))) && acceleration > 0)
{
    Vector3 localrot = tiltPivot.transform.localEulerAngles;
    localrot.z = Mathf.LerpAngle(localrot.z, bikeModelRotation, Time.deltaTime * 8f);
    tiltPivot.transform.localEulerAngles = localrot;
}
// Otherwise, if not steering
else
{
    Vector3 localrot = tiltPivot.transform.localEulerAngles;
    localrot.z = Mathf.LerpAngle(localrot.z, 0, Time.deltaTime * 4f);
    tiltPivot.transform.localEulerAngles = localrot;
}
```

Kuva 18. Ote BicycleController-komponentin FixedUpdate-metodista

Pyörän ohjauksessa päädyimme ratkaisuun, missä pyörää ohjataan puhelinta kallistamalla. Polkeminen tapahtuu kallistamalla puhelinta eteenpäin, ja kääntyminen tapahtuu kallistamalla puhelinta sivuttain. Kuvassa 19 on koodi, joka lukee puhelimen kallistumisen sekä asettaa polkupyörän steering ja acceleration-arvot, jotka saavat polkupyörän liikkumaan ja kääntymään.

```
void Update () {
    #if MOBILE_INPUT
        bikeControl.steering = Mathf.Clamp(Input.acceleration.x * 1.5f, -1.0f, 1.0f);
        bikeControl.acceleration = Mathf.Clamp(Input.acceleration.z * -2f, 0.0f, 1.0f);
    #endif
}
```

Kuva 19. Ote MobileControls-komponentista

Pyörä saatiin liikkeelle säätämällä WheelCollider-komponentin muuttujaa motorTorque. Takarenkaan motorTorque lasketaan acceleration-muuttujan ja polkupyörän BikePower-muuttujan perusteella. Eturenkaan WheelColliderin steerAngle-muuttuja määrittelee kääntymiskulman, joka lasketaan niin, että mitä nopeammin polkupyörä liikkuu, sitä vähemmän polkupyörä voi kääntyä. Molemmista WheelCollider-komponenteista otetaan talteen niiden kulmat X-akselilla, jonka jälkeen käännetään graafiset renkaat X-akselilla oikeaan kulmaan. Näin saatiin graafiset renkaat pyörimään realistisesti fyysisten renkaiden mukaisesti. Kuvassa 20 näkyy liikkumisen ja kääntymisen toteutus.

```
//Power up rear wheel
rearWheelCollider.motorTorque = BikePower * acceleration;

//Rotate rear wheel
rearWheelCollider.GetWorldPose(out position, out rotation);
eulerRotation.x = rotation.eulerAngles.x;
rearAxle.localEulerAngles = eulerRotation;

//Set steering angle
float SteerAngleMulti = 1f-((speed / MaxSpeed)*0.5f);
frontWheelCollider.steerAngle = steering * (MaxSteerAngle * SteerAngleMulti);

//Rotate front wheel
frontWheelCollider.GetWorldPose(out position, out rotation);
eulerRotation.x = rotation.eulerAngles.x;
frontAxle.localEulerAngles = eulerRotation;
```

Kuva 20. Ote BicycleController-komponentin FixedUpdate-metodista

4.5 Ampuminen

Ampuminen on myös yksi erittäin keskeinen osa tätä projektia. Koska peli on hyvin nopeatempoinen, päätimme että ampumisen tulisi tapahtua ”bullet time”- tilassa, missä peli hidastuu, jotta tähtääminen ja ampuminen olisivat mahdollista. Monipuolisuuden vuoksi halusimme, että pelaaja voi itse valita, mitä aseita pyörään asennetaan. Polkupyörään voidaan asentaa kerrallaan kolme asetta, joten pelaajalla on lukuisia mahdollisuuksia erilaisiin aseyhdistelmiin.

4.5.1 Bullet time

Peliin tehtiin Matrix-elokuvasta ja Max Payne -pelistä tuttu Bullet time -ominaisuus, eli ajan hidastus. Bullet time oli suhteellisen helppo toteuttaa, sillä Unityn Time-luokka mahdollistaa ajan nopeuden (time scale) muuttamisen. Unityn Time-luokan muuttuja timeScale on kerroin, mikä vaikuttaa Unityn ajan laskemiseen. TimeScale-muuttujan arvon ollessa 1,0 aika kuluu oletusnopeudella, kun taas arvolla 0,5, aika kuluu puolta hitaammin. Fysiikat hidastuvat automaattisesti pelkästään timeScalea muuttamalla, mutta itse lasketut liikkeet täytyy kertoa Time-luokasta löytyvällä deltaTime-arvolla, jolloin kaikki liikkeet hidastuvat tai nopeutuvat ajan nopeuden mukaan. Äänien hidastumista varten ohjelmoitiin AudioSpeed-komponentti, joka säätää äänilähteiden pitch-arvoa ajan nopeuden mukaisesti.

```
using UnityEngine;
using System.Collections;

public class AudioSpeed : MonoBehaviour {

    private AudioSource source;

    void Start () {
        source = GetComponent<AudioSource>();
    }

    void Update () {
        source.pitch = Time.timeScale;
    }
}
```

Kuva 21. AudioSpeed-komponentti kokonaisuudessaan

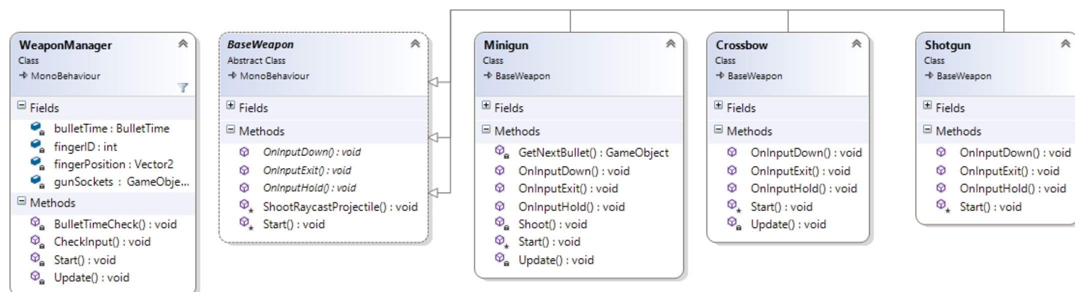
Bullet time -tilaa varten Riku ohjelmoi BulletTime-komponentin. Komponentti mahdollistaa Bullet time -tilan aktivoimisen, jolloin aika hidastuu, ruudun värikylläisyys vähenee ja kamera muuttaa sijaintiaan ensimmäisen persoonan kuvakulmaan. Kun bullet time -tilasta lähdetään, kamera palautuu kolmannen persoonan kuvakulmaan, ajan nopeus palautuu normaaliksi ja värikylläisyys palautuu normaaliksi. Kuva 22 esittää koodin, joka muuttaa kameran arvoja sen mukaan, ollaanko siirtymässä tai poistumassa bullet time -tilasta.

```
if (!doneSwitcing)
{
    float step = 8f * Time.deltaTime;
    if (inBulletTime)
    {
        // Disable third person view
        smoothFollow.enabled = false;

        // Get current camera info
        Vector3 cameraPos = mainCam.transform.position;
        Vector3 cameraRot = mainCam.transform.localEulerAngles;
        float cameraFov = mainCam.fieldOfView;

        // Smoothly move camera to first person view
        mainCam.transform.position = cameraPos = Vector3.Lerp(cameraPos, switchCam.transform.position, step);
        mainCam.transform.localEulerAngles = cameraRot = Vector3.Lerp(cameraRot, switchCam.transform.localEulerAngles, step);
        mainCam.fieldOfView = cameraFov = Mathf.Lerp(cameraFov, switchCam.fieldOfView, step);

        // Smoothly decrease timeScale
        Time.timeScale = Mathf.Lerp(Time.timeScale, (inBulletTime) ? bulletTimeScale : originalTimeScale, step);

        // Is the camera in it's final position?
        if (cameraPos == switchCam.transform.position && cameraRot == switchCam.transform.localEulerAngles)
        {
            doneSwitcing = true;
        }
    }
    else
    {
        // Smoothly change timeScale back to normal
        Time.timeScale = Mathf.Lerp(Time.timeScale, (inBulletTime) ? bulletTimeScale : originalTimeScale, step);

        // Done moving?
        if (smoothFollow.distance == originalDistance)
        {
            doneSwitcing = true;
        }
    }
```

Kuva 22. Ote BulletTime-komponentin Update-metodista

4.5.2 Asejärjestelmä

Kaikki pelin aseet periytyvät abstraktista BaseWeapon-luokasta (katso kuva 23). Tällä on abstraktit metodit OnInputDown, OnInputExit ja OnInputHold. Periytyvien luokkien on pakko siis toteuttaa nämä kolme metodia. WeaponManager-luokka ajaa Update-metodissaan CheckInput-metodia, jos bullet time -tila on aktiivinen. Tällöin WeaponManager kutsuu aseen tai aseiden aiemmin mainittuja metodeja tilanteissa, kun sormella on painettu, sitä pidetään pohjassa ja kun sormi poistuu näytöltä. Se, mitä näissä metodeissa tapahtuu, on täysin aseluokan päätettävissä. Esimerkiksi Crossbow-luokan implementaatio OnInputHold-metodista ei tee yhtään mitään, kun taas minigun ampuu niin kauan, kun sormea pidetään näytöllä.



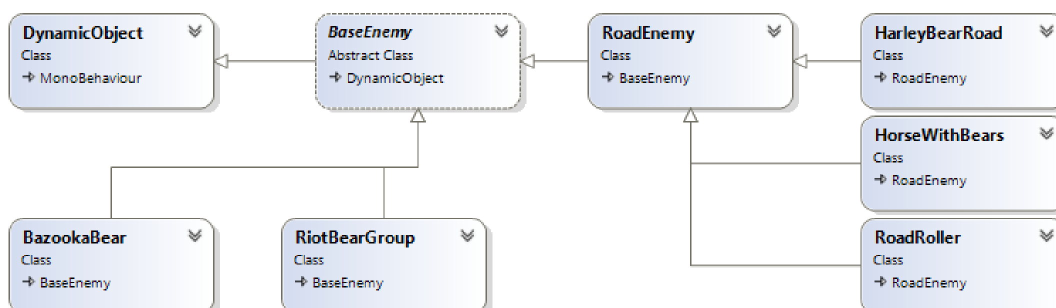
Kuva 23. Aseisiin liittyvä luokkakaavio

WeaponManager-luokassa on taulukkomuuttuja gunSockets. Taulukossa säilötään viittaukset maksimissaan kolmeen peliobjektiin, minkä alle aseet instansioidaan (katso kuva 24). Pelissä voi nimittäin varustaa pyöränsä jopa kolmella eri aseella: molempiin sarviin ja ohjaustangon keskelle. WeaponManager hoitaa myös aseiden kohdistamisen sormen osoittamaan kohtaan, kun ollaan bullet time -tilassa. Muutoin aseet osoittavat suoraan eteenpäin.

Kuva 24. Violetit kuutiot näyttävät, missä asepaikat ovat. Oikeassa yläkulmassa Unity Editorissa näkyvät peliobjektit

4.6 Viholliset

Pelissä on tällä hetkellä kahden tyyppisiä vihollisia: tietä pitkin kulkevat sekä vapaasti liikkuvat tai liikkumattomat viholliset. Tietä pitkin kulkevat viholliset osaavat mennä tietä pitkin ja jatkaa jopa seuraavalle tiepalaselle älykkäästi, käyttäen tätä varten ohjelmoitua koordinaatistojärjestelmää (katso luku 4.7). Suoraan BaseEnemy-luokasta periytyvälle viholliselle ohjelmoidaan täysin oma logiikkansa liikkumisen osalta – jos liikkumiselle on edes tarvetta. Kuvassa 25 näkyvät kirjoitushetken viholliset ja niiden periytyminen.



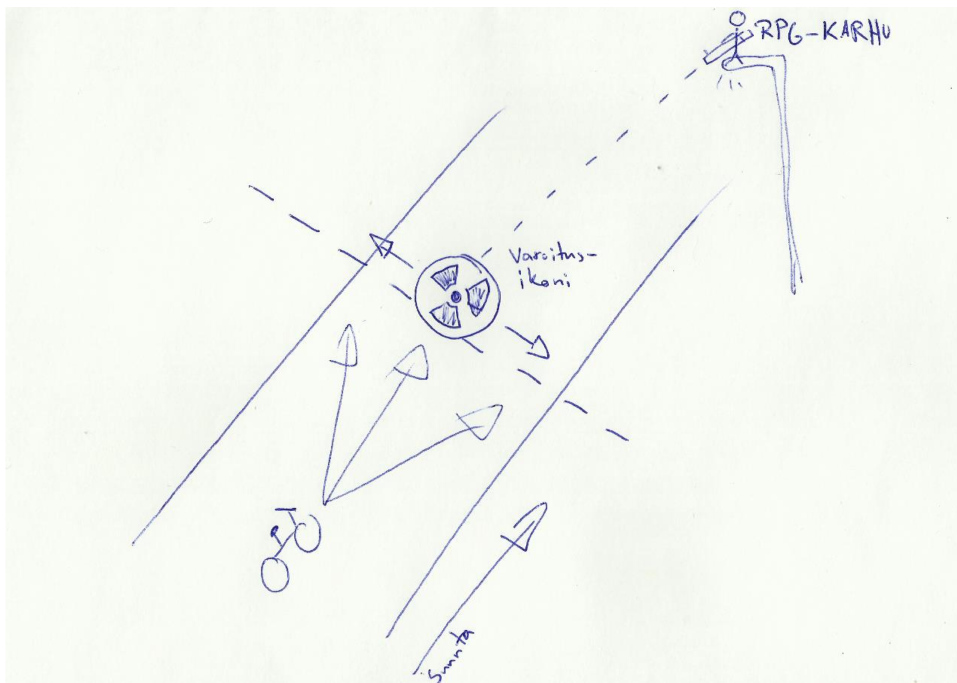
Kuva 25. Vihollisten luokkakaavio, mistä näkyy periytyminen

4.6.1 Raketinheittimellä varustettu karhu

Yksi ideoitu vihollistyyppi oli raketinheittimellä varustettu karhu. Ideana oli, että raketinheitinkarhu väijyy katulampun päällä kuten kuvassa 26 näkyy. Pelaajan lähestyessä peli näyttää varoituskuvakkeen, mikä liikkuu hitaasti sivusuunnassa tien mukaisesti pelaajan sijainnin mukaan (katso kuva 27). Raketinheitinkarhu ampuu raketin kohti varoituskuvaketta, kun pelaaja on tarpeeksi lähellä. Ampumishetkeen vaikuttaa myös pelaajan nopeus, jotta räjähdys tapahtuisi oikealla hetkellä. Raketinheitinkarhun toteutuksesta vastasi Jussi.

Kuva 26. Raketinheitinkarhu vaanii saalistaan katulampun päällä

Haastavinta oli saada varoituskuvake liikkumaan oikeassa suunnassa, sillä tiepalasten käännökset vaikuttavat siihen, pitääkö kuvakkeen liikkua 3D-avaruudessa x- vaiko z-akselilla. Käytännössä kuvake liikkuu kohti pelaajan sijaintia 3D-avaruudessa, mutta sen x- tai z-akseli korvataan kuvakkeen omalla kyseisellä arvolla. Koska pelaaja ei voi kääntyä pelissä ympäri monistakin syistä, ainakin tällä hetkellä akseli katsotaan pelaajan rotaation itseisarvosta. Kuten monet muutkin vihollistyypit, myös raketinheitinkarhun voi ampua ennen kuin se ehtii laukaisemaan ohjuksen.



Kuva 27. Raketinheitinkarhun logiikan suunnittelua paperilla

4.6.2 Mellakkakarhut

Mellakkakarhut ovat vihollisia, jotka kävelevät tietä pitkin, kunnes ne havaitsevat lähellä pyöräilevän pelaajan (katso kuva 28). Pelaajan nähtyään ne syöksyvät häntä kohti, tarkoituksenaan törmätä pelaajaan kuolettavalla voimalla. Mellakkakarhuilla on luodinkestävät kilvet, joten ne voidaan ampua kuoliaaksi vain, kun ne hypätessään paljastavat selkänsä pelaajalle. Mellakkakarhujen toteutuksesta vastasi Riku.

Kuva 28. Mellakkakarhut ovat valmiina syöksymään Taavin kimppuun

Mellakkakarhut olivat kohtuullisen helppo toteuttaa. Ne vain kävelevät hitaasti eteenpäin, kunnes ne ovat tarpeeksi lähellä pelaajaa (kuva 29). Karhujen ja pelaajan välinen etäisyys tarkistetaan Unityn Vector3-luokan Distance-metodilla, jolle annetaan parametriksi kaksi Vector3-oliota, jotka esittävät kahta eri pistettä 3D-koordinaatistossa. Pelaajan ollessa tarpeeksi lähellä, määritellään mellakkakarhu pelaajan nähneeksi ja lasketaan paikka mihin karhun tulisi hypätä (kuva 30).

Pelaajan törmätessä mellakkakarhuihin pelihahmo lentää pois pyörän selästä ja vastaanottaa suuren määrän vahinkopisteitä. Samaa törmäyslogiikkaa käytetään kaikissa törmäyksissä pelissä, eli tämä ei ole varsinaisesti vain mellakkakarhuihin liitettävä ominaisuus.

```
if (_hasNoticedPlayer)
{
    // Jump towards the player
    Vector3 bearGroupPos = this.transform.position;
    bearGroupPos = Vector3.Lerp(this.transform.position, _jumpSpot, Time.deltaTime * 2f);
    this.transform.position = bearGroupPos;
    if(Vector3.Distance(bearGroupPos,_jumpSpot) <= 0.1f)
    {
        hitBox.enabled = false;
    }
}
else
{
    // Move the riot bear forward
    Vector3 move = this.transform.position;
    move += this.transform.forward * Time.deltaTime * 5f;
    this.transform.position = move;
}
```

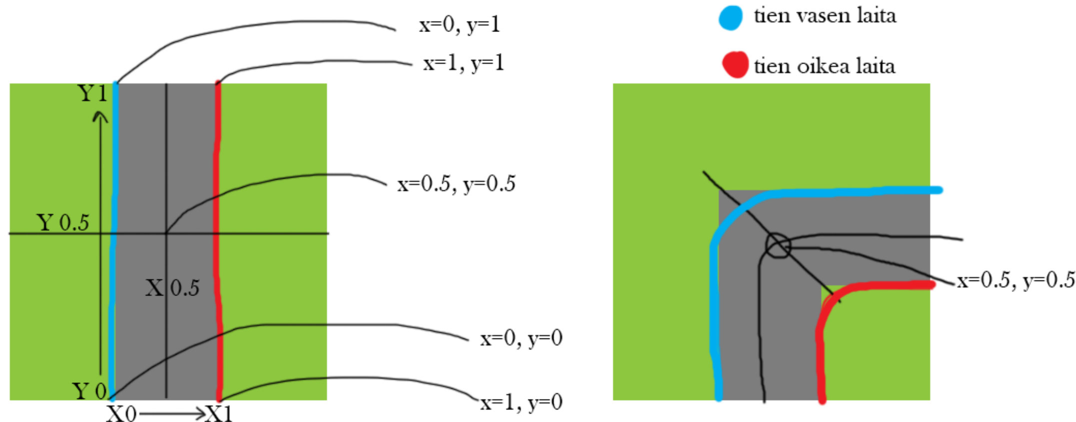
Kuva 29. Mellakkakarhun liikuttaminen

```
//Check if group is near enough to the player
if (!_hasNoticedPlayer && Vector3.Distance(_player.transform.position, this.transform.position) <= 40f )
{
    _hasNoticedPlayer = true;

    // Set the position where the bears will jump to
    _jumpSpot = _player.transform.position;
    _jumpSpot.x += _player.transform.forward.x * 20f;
    _jumpSpot.y = this.transform.position.y;
    _jumpSpot.z += _player.transform.forward.z * 20f;

    // Start jump animations
    foreach (Animator child in childrenAll)
    {
        child.SetBool("Jump", true);
        hitBox.enabled = true;
    }
}
```

Kuva 30. Mellakkakarhun ja pelaajan välisen etäisyyden mittaaminen

4.7 Tiellä kulkevien vihollisten tekoäly

Yksi tekoälyn ongelmista oli saada pelin viholliset kulkemaan fiksusti tietä pitkin. Ensimmäisessä tekoälyn versiossa viholliset osasivat kulkea vain suoraa tietä. Suoralta tieltä käännöspalaselle tultaessa, vihollinen ei kääntynyt mutkassa vaan jatkoi samaa rataa kulkien seinien läpi ulos pelikartalta.



Kuva 31. Suunnitelma koordinaatistolle

Ongelman ratkaisuksi Riku suunnitteli katupalasille oman koordinaattisysteemin (kuva 31), jonka avulla viholliset voisivat kulkea pitkin katua, oli se sitten miten mutkitteleva tahansa. Järjestelmän toteuttamisessa käytettiin apuna Unity Asset Storesta ladattua BezierCurve-lisäosaa, joka mahdollistaa kolmiulotteisten linjojen rakentamisen. BezierCurven ominaisuuksiin lukeutuu mahdollisuus palauttaa 3D-koordinaatti tietystä linjan pisteestä metodilla GetPointAt(float t), missä t on prosentuaalinen piste linjan alusta loppuun (0,0 – 1,0).

Koordinaattisysteemin kulmakivinä toimivat katupalasen laidat määrittelevät linjat. Linjat sitovat yhteen katupalasia varten ohjelmoitu RoadCoordinates-komponentti, jonka kautta koordinaatistoa käytetään. Komponentilla on metodi GetPosition(float x, float y), joka palauttaa 3D-koordinaatit parametreina annettujen tiekoordinaattien perusteella (kuva 32). RoadCoordinates myös säilöö viittauksen seuraavan tiepalasen koordinaatistoon, jotta viholliset osaavat kulkea tiepalaselta toiselle saumattomasti.

```
public Vector3 GetPosition(float x, float y)
{
    Vector3 position = Vector3.Lerp(leftBound.GetPointAt(y), rightBound.GetPointAt(y), x);
    return position;
}
```

Kuva 32. RoadCoordinates-komponentin GetPosition-metodi

Tietä kulkevat viholliset periytetään RoadEnemy-luokasta, jolla on viite vihollisen käyttämään tiekoordinaatistoon. Luokka sisältää tällöin metodin Move, joka hoitaa vihollisen liikuttamisen tietä pitkin.

```
// Set the new road Y coordinate
coord.y -= Time.deltaTime * speed * speedMulti;

// If Y goes below 0, switch the currend road coordinate system to the next coordinate system
if (coord.y < 0f)
{
    roadCoord = roadCoord.nextRoadCoordinates;
    coord.y = 1f + coord.y;
}
```

Kuva 33. Ote RoadEnemyn Move-metodista

Viholliselle määritellään tiellä liikkumisen nopeus. Move-metodi muuttaa vihollisen y-koordinaattia nopeuden mukaan. Jos y menee alle nollan, vaihdetaan vihollisen tiekoordinaatisto seuraavan tiepalasen tiekoordinaatistoksi (kuva 33). Näin vihollinen osaa kulkea tieltä toiselle saumattomasti. Mikäli y menee alle nollan, eikä seuraavaa tiekoordinaatistoa ole olemassa, kutsutaan metodia Despawn, joka siirtää vihollisen takaisin vihollisten pooliin odottamaan uudelleenkäyttöä.

```
if (roadCoord != null)
{
    Vector3 newPos = Vector3.Lerp(this.transform.position, roadCoord.GetPosition(coord.x, Mathf.Clamp(coord.y, 0f, 1f)), 1f);
    Vector3 lookat = this.transform.position +  (newPos - this.transform.position).normalized*2f;

    if(coord.y >= 0.1f && coord.y <= 0.9f)
    {
        this.transform.LookAt(newPos);
    }

    this.transform.Translate((newPos - this.transform.position).normalized * (newPos - this.transform.position).magnitude,Space.World);
}
```

Kuva 34. Toinen ote RoadEnemyn Move-metodista

Kun Move-metodi on laskenut vihollisen uuden tiekoordinaatin, vihollinen käännetään kohti uutta 3D-koordinaattia, joka saadaan tiekoordinaatiston GetPosition-metodilla. Kääntämisen jälkeen vihollinen siirretään uuteen 3D-koordinaattiin. (Kuva 34.)

4.8 SQLite-tietokannan implementointi

Projektin edetessä, ja etenkin kun alkoi olla puheita muun muassa ostettavista pyöräpalasista, oli selvää, että jonkinlainen tietojen säilytys täytyi implementoida. Ajattelimme, että SQLite voisi olla juurikin omiaan tähän projektiin. Löysimme Roberto Huertasin tekemän ”SQLite4Unity3d”-kirjaston, mikä tuo sqlite-net -kirjaston helposti Unitylla käytettäväksi. Mikä parasta, sqlite-net mahdollistaa C#:n LINQ-ominaisuuden käytön kyselyiden tekemisessä. Kuvassa 35 esimerkki LINQ-ominaisuudesta.

```
68    public static int GetPartCategoryID(string partname)
69    {
70        return Conn.Table<BikePartCategories>().Where(x => x.Name == partname).FirstOrDefault().Id;
71    }
```

Kuva 35. Kysely käyttäen C#:n LINQ-ominaisuutta

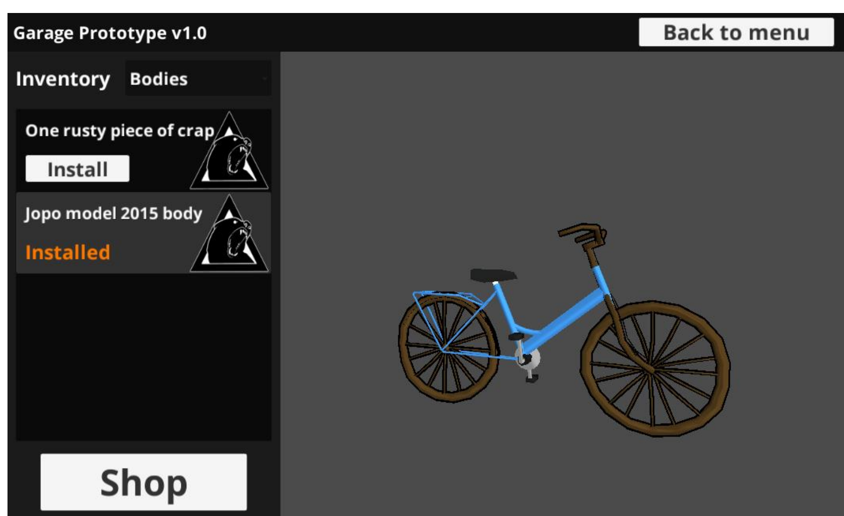
Jussi kirjoitti peliä varten staattisen DatabaseManager-luokan. Luokalla on loppujen lopuksi aika vähän tehtävää; se vastaa tietokantatiedoston luomisesta tai lukemisesta sekä taulujen luomisen ensimmäisellä suorituskerralla (kun tauluja ei ole). Kuva 36 havainnollistaa taulun luomisen ja siihen rivien lisäämisen. Tietokantataulut määritellään normaaleina C#-luokkina, minkä takia niitä pystyykin käsittelemään erittäin kätevästi LINQ:n avulla.

```
127         // If there is no such table as Weapons
128         if (!_conn.GetTableInfo("Weapons").Any())
129         {
130             // Create table Weapons
131             _conn.CreateTable<Weapons>();
132 
133             // Insert default data.
134             _conn.InsertAll(new[]
135             {
136                 new Weapons
137                 {
138                     Name = "Minigun"
139                 },
140                 new Weapons
141                 {
142                     Name = "Crossbow"
143                 }
144             });
145         }
```

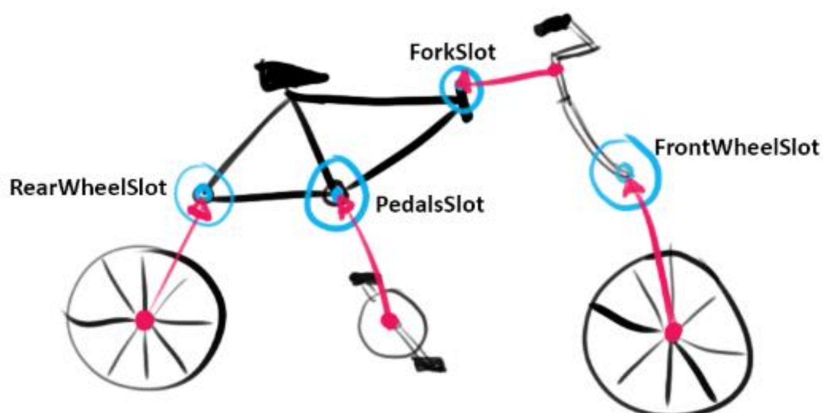
Kuva 36. Tietokantataulun luominen ja oletusdatan tuominen

4.9 Polkupyörien rakentaminen

Koukuttavuuden ja mielenkiintoisuuden edistämiseksi halusimme antaa pelaajalle mahdollisuuden rakennella omia polkupyöriä. Polkupyörän osia voi ostaa pelissä kerätyllä valuutalla, jonka jälkeen omiin polkupyöriin voidaan vaihtaa omistuksessa olevia osia. Kuvassa 37 näkyy pelin ”autotalli”, missä pyörien rakentelu ja osien ostaminen tapahtuvat. Osilla on omat bonuksensa esimerkiksi nopeuteen, kiihtyvyyteen ja hallittavuuteen. Bonukset motivoivat pelaajaa keräämään valuuttaa ja rakentamaan mahdollisimman hyvän polkupyörän.



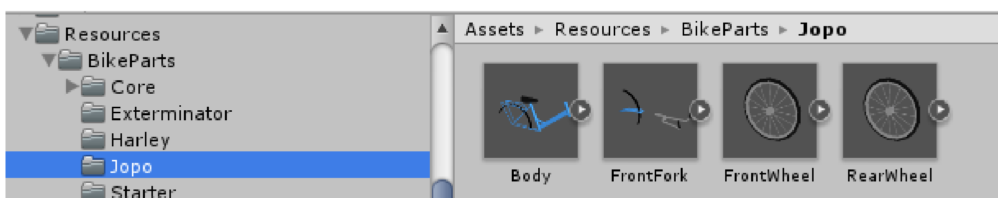
Kuva 37. Pelin autotalli



Kuva 38. Suunnitelma polkupyörän hierarkiasta

Kuvan 38 mukaisesti polkupyörien runkoihin on määritelty pisteet, mihin mikäkin osa liitetään. Runkojen lisäksi myös etuhaarukoissa on etuakseleille määritellyt pisteet. Näin mihin tahansa pisteeseen voidaan liittää mikä tahansa sen tyyppiseen pis-

teeseen kuuluva osa. Kaikki yksittäiset pyörän osat kuuluvat johonkin pyöräperheeseen, mutta yhdessä pyöräperheessä ei voi olla kahta saman tyyppistä pyörän osaa, esimerkiksi kahta eturengasta. Pyörien osat on tallennettu prefabeiksi omiin kansioihin seuraavan kaavan mukaan: Resources/BikeParts/<pyöräperhe>/<osan tyyppi>.prefab (kuva 39). Resources-kansion alla olevia prefabeja voidaan instantioida pelin ajon aikana. Pyörien osien täytyy löytyä oikeasta osoitteesta, jotta ohjelmoimamme järjestelmä osaa hakea halutun osan halutusta pyöräperheestä.



Kuva 39. Polkupyörän osia prefabeina

Rikun ohjelmoima komponentti BikeBuilder hoitaa pyörän kokoamisen metodilla BuildBike, joka hakee tietokannasta nykyiseen pyörään kuuluvat osat ja rakentaa polkupyörän kyselyn vastauksen perusteella (kuva 40).

```
private void BuildBike()
{
    // Get currently selected bike parts.
    DatabaseManager.BikePartsCollection bikeParts = DatabaseManager.ReturnInstalledBikeParts();

    // Spawn the bike body
    bike = InstPart("Body", bikeParts.bikeBody, this.transform);
    bike.tag = "BikeBody";

    // Spawn the fork
    frontFork = InstPart("FrontFork", bikeParts.bikeFrontFork, FindChild("FrontForkSlot").transform);

    // Spawn the rear wheel
    rearAxle = FindChild("RearAxle").transform;
    InstPart("RearWheel", bikeParts.bikeRearWheel, rearAxle);

    // Spawn the front wheel
    frontAxle = FindChild("FrontAxle").transform;
    InstPart("FrontWheel", bikeParts.bikeFrontWheel, frontAxle);

    // And finally the pedals
    Transform pedals = InstPart("Pedals", BikeFamilies.Core, FindChild("PedalsSlot").transform);

    CalculateBicycleStats();
}
```

Kuva 40. BikeBuilder-komponentin BuildBike-metodi

BuildBike-metodi käyttää hyväkseen toista BikeBuilder-komponentin metodia InstPart (kuva 41), jolle annetaan parametrina osan tyyppi, osan pyöräperhe ja määritelty piste, mihin osa liitetään. Osan tyypin ja pyöräperheen perusteella ohjelma lataa oi-

kean palasen Resources/BikeParts-kansion alta ja sitten liittää uuden palan sille kuuluvalle paikalle.

```
private Transform InstPart(string part, BikeFamilies family, Transform alignTo = null)
{
    // Load the wanted bike part and store it in temp
    GameObject temp = Resources.Load<GameObject>("BikeParts/" + family.ToString() + "/" + part);
    // Instantiate the part to the game world
    Transform obj = GameObject.Instantiate<GameObject>(temp).transform;
    // Set the name back to the original name
    obj.name = temp.name;

    // If we want to align the part to a slot
    if(alignTo != null)
    {
        obj.parent = alignTo;
        obj.localRotation = Quaternion.identity;
        obj.localPosition = Vector3.zero;
    }
    return obj;
}
```

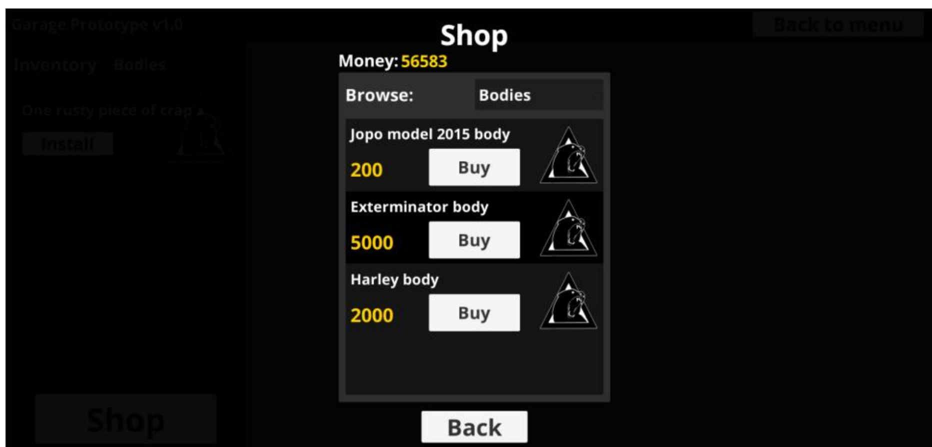
Kuva 41. InstPart-metodi BikeBuilder-komponentissa

Järjestelmä mahdollistaa mielenkiintoisten ja jopa humorististen polkupyörien kokoamisen, näin nostaen pelikokemuksen hauskuutta. Koemmekin, että tämä ominaisuus on yksi pelimme valttikorteista. Kuvassa 42 yksi järjestelmällä rakennettu pyörä.

Kuva 42. Esimerkki BuildBike-metodilla luodusta polkupyörästä

4.10 Pelin kauppa

Kun pelaaja on saanut kerättyä tarpeeksi rahaa, pelaaja voi ostaa uusia polkupyörän osia pelin autotallista löytyvästä kaupasta. Ostetut pyörän osat tulevat pelaajan autotalliin, missä ostettuja osia voi asentaa polkupyörään. Kuvassa 43 näkyy kauppanäkymä.



Kuva 43. Kuvakaappaus pelin kaupasta

Kaupassa voi selata ostettavia osia neljästä eri kategoriasta: rungot, etuhaarukat, eturenkaat ja takarenkaat – tulevaisuudessa myös polkimet ja ohjaustangot. Osien listausta varten Riku kehitti DatabaseManageriin uuden metodin GetBikeParts, joka palauttaa tietokannasta listan pyörän osista, parametreina annettujen ehtojen perusteella. Kuvassa 44 GetBikeParts-metodin toteutus. Kaupan osien listaus tapahtuu Shop-komponentin metodissa UpdateShopPartsList, joka päivittää kaupan listan valitun kategorian perusteella, GetBikeParts-metodia hyödyntäen.

```
public static BikeParts[] GetBikeParts(bool unlocked = false, int category = -1, int family = -1)
{
    BikeParts[] parts = Conn.Table<BikeParts>().Where(x => x.Unlocked == unlocked).ToArray();

    if(category != -1)
    {
        parts = parts.Where(x => x.PartCategory == category).ToArray();
    }

    if(family != -1)
    {
        parts = parts.Where(x => x.PartFamily == family).ToArray();
    }

    return parts;
}
```

Kuva 44. DatabaseManagerin GetBikeParts-metodi

4.11 Saavutukset

Nykyään lähes kaikissa peleissä on saavutukset. Tästäpä syystä kyseinen toiminto piti saada myös tähän projektiin. Tähän käyttötarkoitukseen Jussi teki AchievementManager-luokan sekä tietokantaan oman Achievements-taulun (katso kuva 45). Tietokannassa pidetään kaikki tieto liittyen saavutuksiin, jotka on yksilöity id-numerolla. Itse koodissa tietyn saavutuksen näyttäminen on tehty vaivattomaksi. Koodin kirjoittaja kutsuu AchievementManagerin metodia AppendToQueue (katso kuva 46), mikä tarkistaa heti ensikättelyssä, onko kyseinen saavutus jo saatu tai löytyykö kyseistä saavutusta ollenkaan. Kummassakin tapauksessa suoritus keskeytyy siihen paikkaan ja tulostetaan vain varoitusviesti Unityn debug-ikkunaan. Muutoin saavutus lisätään jonoon.

Jonojärjestelmä tehtiin siltä varalta, että pelaaja mahdollisesti onnistuisi saamaan samaan aikaan tai lyhyen aikavälin sisällä kaksi saavutusta. Tällöin toinen ja kaikki siitä eteenpäin saapuvat saavutukset menevät jonoon. Saavutukset näkyvät pelin oikeassa yläkulmassa saapumisjärjestyksessä vuoron perään, eli päällekkäin ne eivät mene missään tapauksessa.

```
public class Achievements
{
    //Default Values
    private bool _unlocked = false;

    [PrimaryKey, NotNull]
    public int Id { get; set; }
    [NotNull]
    public string Name { get; set; }
    [NotNull]
    public string Description { get; set; }
    [NotNull]
    public string IconFilename { get; set; }
    [NotNull]
    public bool Unlocked {
        get { return _unlocked; }
        set { _unlocked = value; }
    }
}
```

Kuva 45. Achievements-taulun rakenne kuvattuna C#-luokkana

```
// Check if the player has enough beers collected for achievement.
if (beerCount == 1)
{
    GameManager.Instance.AchievementManager.AppendToQueue(1);
}
        void AchievementManager.AppendToQueue(int achievementId)
```

Kuva 46. Saavutuksen näyttäminen pelissä

4.12 Hahmojen puheen hallinta

Jo projektin alkuvaiheista oli selvää, että Taavi (pelaajan ohjaama hahmo) tulisi heittämään humoristisia puherepliikkejä eri tilanteissa. Jussi kehitti tätä varten komponentin, jonka kautta pystyi helposti aktivoimaan satunnaisen äänileikkeen halutusta ääniryhmästä ja näyttämään sille kuuluvan tekstityksen, kuten kuva 47 esittää. Myöhemmin tuli myös idea, että Suomen fiktiivinen tasavallan presidentti välillä soittaa Taaville kannustaakseen. Lopulta myös Ukko ylijumalan supervoimaa käytettäessä Ukko heittää satunnaisen kommentin ennen iskua. Tässä kohtaa komponenttia täytyi uudistaa – Taavi ei ollut enää ainoa puhuja pelissä. Nyt komponentin tulisi osata antaa eri henkilöille oma painoarvonsa, jotta päällekkäin puhumista ei tapahtuisi.

Kuva 47. Tekstitys ilmestyy kuvakkeiden yläpuolelle, kun Taavi puhuu

Käytännössä jokaisella ääniryhmällä on oma Voice-tyypin List-olio. Voice on C#-kielen rakenne (struct), jolla on ominaisuudet tekstitys, äänileike ja äänen prioriteetti. Nämä listat on merkitty attribuutilla SerializeField, jolloin ne voidaan alustaa Unity Editorissa. Kuvassa 50 näkyy editorissa alustettuja arvoja. Tällöin on mahdollista käyttää Unityn graafista editoria arvojen asettamiseen. Komponentin Awake-metodissa yhdistetään kaikki listat yhden Dictionary-olion alle, josta niitä voi hakea helposti ryhmän omalla avaimella (katso kuva 48 ja 49).

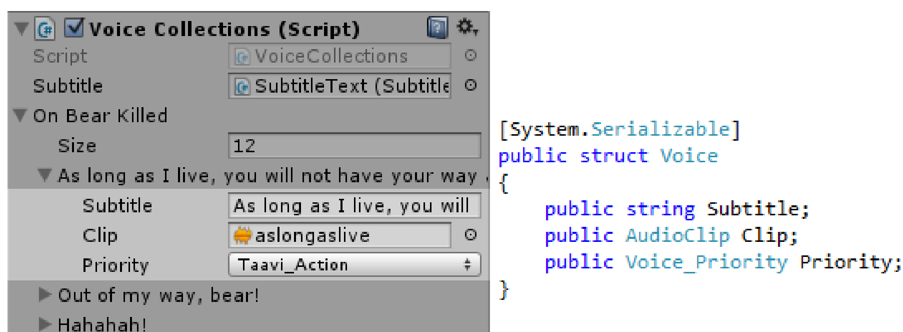
```
void Awake()
{
    allVoices = new Dictionary<Voice_Types, List<TaaviShouts>>();

    //Add voice lists to dictionary
    allVoices.Add(Voice_Types.On_Bear_Killed, onBearKilled);
    allVoices.Add(Voice_Types.On_Pizza_Delivery, onPizzaDelivery);
    allVoices.Add(Voice_Types.Idle_Chat, idleChat);
    allVoices.Add(Voice_Types.To_Bullet_Time, toBulletTime);
    allVoices.Add(Voice_Types.On_Taavi_Crash, onTaaviCrash);
    allVoices.Add(Voice_Types.President_Cheer, presidentCheer);
    allVoices.Add(Voice_Types.Ukko_SuperPower, ukkoSuperPower);
}
```

Kuva 48. Äänilistat laitetaan yhden Dictionary-olion alle Awake-metodissa

```
public void ActivateWrathOfUkko()
{
    // Check that the player has enough charge to use this ability
    if (superPowerCharge < superPowerChargeMax)
    {
        return;
    }

    // Play random voice by Ukko
    GameManager.Instance.VoiceCollections.PlayRandomVoice(Voice_Types.Ukko_SuperPower);
```

Kuva 49. Äänen toistaminen muualta käyttäen hyödyksi GameManager-luokkaa

Halusimme, että vain yksi ääni kuuluu kerralla. Lisäksi mielestämme jotkin äänet ovat muita tärkeämpiä. Näistä syistä kehitimme prioriteettijärjestelmän. Esimerkiksi Taavin törmätessä ja lentäessä pyörän selästä, mikä tahansa Taavin repliikki loppuu kesken ja Taavi aloittaa kärsimyshuutonsa.



Kuva 50. Äänien alustaminen Unity Editorissa

4.13 Optimointi

Kääntäessämme ensimmäisiä versioita pelistä Android-laitteelle olimme hyvin pettyneitä pelin suorituskykyyn. Unityn Profiler-työkalun avulla selvisi, että muun muassa grafiikkojen piirtämiseen ja tiepalasten instantiointiin ajon aikana meni aivan liikaa

aikaa murskaten suorituskyvyn ja mahdollisuuden pelaamiseen puhelimella. Tämä tuli yllätyksenä, sillä tietokoneella pelatessa ei suorituskykyongelmia ollut, mutta onhan tietokoneissa huomattavasti enemmän tehoja kuin älypuhelimissa.

4.13.1 Grafiikkojen optimointi

Luulimme ensin, että 3D-malleissamme on liikaa polygoneja ja uskoimme suorituskyvyn heikkouden johtuvan siitä. Mutta huomasimme kuitenkin Profiler-työkalun avulla, että grafiikan piirtokutsujen lukumäärä (drawcall) oli liikaa mobiililaitteille. Piirtokutsu syntyy, kun ohjelma lähettää 3D-mallin ja sen materiaalin näytönohjaimelle piirrettäväksi.

Ratkaisuksi Riku ohjelmoi komponentin, joka yhdistää peliobjektin lapsista löytyvät 3D-mallit ja tarvittaessa myös niiden tekstuurit yhdeksi suureksi meshiksi ja atlastekstuuriksi. Mesh on kokoelma pisteitä (vertex) ja niistä muodostuvia kolmioita (triangles) joista 3D-mallin pinta muodostuu. Tekstuuri on kuva mikä piirretään 3D-mallin pinnalle. Tekstuureja voidaan yhdistää yhteen suurempaan atlastekstuuriin, jolloin näytönohjaimen muistiin tarvitsee ladata monen yksittäisen tekstuurin sijasta vain yksi tekstuuri. Yhdeksi meshiksi yhdistetyt peliobjektit, mitkä käyttävät yhtä atlastekstuuria, kuluttavat yhteensä vain yhden piirtokutsun. Tämä parantaa ruudunpäivitysnopeutta, joskin suuremman muistin kulutuksen hinnalla. Mobiililaitteilla muistin menetys näytti kuitenkin olevan pieni hinta huomattavasta suorituskyvyn paranemisesta. Ratkaisu toimii kuitenkin vain staattisissa, liikkumattomissa peliobjekteissa. Yhdistäjäkomponenttia käytetään tiepalasten sisältämien talojen, puiden ja muiden koristeiden yhdistämiseksi. Ongelman ansiosta opimme lisää näytönohjaimen toiminnasta ja siitä, miten 3D-peli saadaan suoriutumaan tehokkaammin. Tekniikasta on varmasti hyötyä myös PC-puolella, jos pelimaailmassa on runsain määrin staattisia objekteja. Tämä tullaan pitämään muistissa tulevaisuuden projekteja ajatellen.

Kuvassa 51 on pätkä MeshCombiner-komponentin koodista, joka valmistautuu 3D-mallien yhdistämiseen sekä luo yhdistetyn tekstuurin yhdistettävien 3D-mallien tekstuureista. Kaikki peliobjektit, jotka halutaan yhdistää, on MeshCombiner-

komponentin sisältävän peliobjektin lapsia. Aluksi komponentti hakee kaikki lapsista löytyvät MeshFilter-komponentit ja tekstuurit ja sitten yhdistää tekstuurit. Lopuksi MeshCombiner luo CombineInstance-taulukon, johon syötetään yhdistettävät 3D-mallit (kuva 52). CombineInstance-taulukkoa käytetään Mesh-luokan CombineMeshes-metodin parametrina, jonka lopputuloksena on yhdistetty 3D-malli. Vanhat 3D-mallit, joista yhdistetty malli on luotu, tuhotaan muistin vapauttamiseksi.

```
// Find all the children's meshes
MeshFilter[] filters = GetComponentsInChildren<MeshFilter>();

// Create an array for the textures
Texture2D[] textures = new Texture2D[filters.Length];

// Create a new material for the combined mesh
Material newMaterial = new Material(this.gameObject.GetComponent<Renderer>().material);

// Store the textures in the texture array
if (!onlyMesh)
{
for (int i = 0; i < filters.Length; i++)
{
    if (filters[i].mesh != null)
    {
        textures[i] = (Texture2D)filters[i].gameObject.GetComponent<Renderer>().material.mainTexture;
    }
}

// Pack the textures to an atlas texture
packedTexture = new Texture2D(atlasTextureSize, atlasTextureSize);
Rect[] uvs = packedTexture.PackTextures(textures, 0, atlasTextureSize, false);
packedTexture.name = this.transform.root.name;

// Assign the new atlas texture to the new material
newMaterial.mainTexture = packedTexture;
```

Kuva 51. Ote MeshCombiner-komponentista: alustus ja tekstuurien yhdistys

```
// Create a CombineInstance array to use with the CombineMeshes-method
CombineInstance[] combine = new CombineInstance[filters.Length];
int f = 0;
while (f < filters.Length)
{
    combine[f].mesh = filters[f].sharedMesh;
    combine[f].transform = transform.worldToLocalMatrix * filters[f].transform.localToWorldMatrix;
    filters[f].transform.GetComponent<MeshRenderer>().enabled = false;
    Destroy(filters[f].transform.GetComponent<MeshRenderer>());
    Destroy(filters[f].transform.GetComponent<MeshFilter>());
    f++;
}

// Add a mesh filter to this gameobject
transform.gameObject.AddComponent<MeshFilter>();

// Create a new empty mesh for the mesh filter and combine the meshes with CombineMeshes
transform.GetComponent<MeshFilter>().mesh = new Mesh();
transform.GetComponent<MeshFilter>().mesh.name = "static_combined_mesh_"+id;
transform.GetComponent<MeshFilter>().mesh.CombineMeshes(combine);
transform.GetComponent<MeshFilter>().mesh.RecalculateNormals();
transform.GetComponent<MeshRenderer>().enabled = true;

// Assign the new combined texture and material to the mesh renderer
if (!onlyMesh)
{
    transform.GetComponent<MeshRenderer>().material = newMaterial;
}
```

Kuva 52. Ote MeshCombiner-komponentista: meshien yhdistys

4.13.2 Koodin optimointi

Kehityksen alkuajoilla pelatessamme peliä älypuhelimella huomasimme, että uutta tiepalasta generoidessa pelissä tapahtui pieni pelaamista häiritsevä nykäisy. Profiler-työkalua tarkkaillessa huomasimme, että tiepala-prefabien instantiointi ajon aikana oli syy tähän häiritsevään nykäisyyn. Koska prefabien instantiointi on liian raskasta, päädyimme käyttämään object pooling -menetelmää. Object pooling tarkoittaa sitä, että objekteja instantioidaan pelin alussa tarvittava määrä, joita sitten käytetään ja uusiokäytetään pelin aikana. Object pooling -järjestelmää käytettäessä säästytään instantioinnin aiheuttamilta hidastuksilta. Tämän hinta kuitenkin on suurempi muistin kulutus, sillä objekteja on yhtäaikaisesti ladattuna muistiin huomattavasti enemmän. Muistin kulutus on kuitenkin tälläkin kertaa sen arvoista.

Otimme tästä opiksi ja implementoimme object pooling -järjestelmän moniin muihinkin osa-alueisiin. Esimerkiksi pelin viholliset, partikkeliefektit sekä luodit ladataan omiin pooleihinsa, joista niitä otetaan käyttöön pelin aikana. Tehtävänsä tehneet peliobjektit palautetaan takaisin pooliin, jolloin niitä voidaan taas käyttää uudelleen. Esimerkiksi viholliset palautuvat pooliinsa pian kuolemansa jälkeen ja partikkeliefektit palautuvat pooliinsa, kun koko efekti on toistanut itsensä loppuun. Pooling-järjestelmien käyttöönotto johti entistä sulavampaan pelikokemukseen. Tekniikka osoittautui niin tehokkaaksi, että sitä tulemme varmasti käyttämään tulevaisuuden projekteissakin.

Pooling-järjestelmää varten Jussi ohjelmoi ObjectPoolManager-komponentin, joka pelin alussa instantioi poolattavat objektit sekä ylläpitää masterPool-listaa, joka sisältää kaikki poolatut objektit ja niiden objektityypit. ObjectPoolManagerilla on metodi RequestFromPool, jonka parametriksi annetaan objektityyppi, jota halutaan käyttää. Metodi palauttaa halutun objektityypin mukaan viitteen peliobjektiin, jonka avulla muut komponentit voivat ottaa peliobjektin käyttöön (kuva 53).

```
public GameObject RequestFromPool(ObjectPoolTypes type)
{
    GameObject requestedObject = masterPool[type]
        .Where(z => !z.activeSelf)
        .OrderBy(n => Random.value)
        .FirstOrDefault();

    return requestedObject;
}
```

Kuva 53. ObjectPoolManagerin RequestFromPool-metodi

5 LOPPUSANAT

3D-mobiilipelin kehittämisen haastavuus yllätti meidät oikein olan takaa. Jo projektin alkuvaiheilla huomasimme, kuinka suuri kuilu on työpöytäkoneen ja mobiililaitteen suorituskyvyn välillä. Aloimme tehdä projektia niin, että kuvittelimme sen olevan hyvinkin kevyt, mutta käytännön testitulokset osoittivatkin jotain ihan muuta. Sysirumaksi tekemämme peli olikin umpiraskas suorittaa eikä kummoisiin ruudunpäivitysnopeuksiin todellakaan päässy. Tästä emme lannistuneet, vaan aloimme aktiivisesti tutkia asiaa. Projektin edetessä opimmekin huomattavan hyviä, parempia käytäntöjä monissa asioissa, etenkin 3D-mobiilipelikehityksen kannalta. Riemunpurskahdukset pääsivätkin siinä vaiheessa, kun onnistuimme saamaan pelin pyörimään siedettävällä ruudunpäivitysnopeudella (yli 30 ruutua sekunnissa) jopa ikivanhalla Samsung Galaxy S -älypuhelimella.

Tämän opinnäytetyön valmistumiseen mennessä peliprojektin kaikki ydintoiminnot ovat hyvässä kunnossa. Tämän kokoisessa pelissä löytyy tietysti aina kohtia, joita voisi parantaa. Harrastamme aktiivista refaktorointia muun muassa muuttujanimien vaihtamisella parempaan sitä mukaan, kun sellaisia sattuu vastaan. Seuraava suuri askel pelin julkaisuvalmiutta varten onkin pelin sisällön luominen uusien kenttien, pyöräpalasten, aseiden ja muun sisällön muodossa. Tarkoituksena olisi myös tutkia pelin sisäisten ostosten mahdollisuutta ja sen käytännön toteutuksen haasteellisuutta.

Meillä oli ja on hyvin samankaltaiset lähtötasot, mitä tulee C#-ohjelmointiin. Rikulla on hieman pitempiaikainen kokemus Unity-pelimoottorista ja Jussilla taas relaatiotietokannoista. Koska olemme ohjelmoineet paljon ja usein parityöskentelynä niin sanotuissa ”all nighter”-iltamissa, olemme oppineet erittäin paljon toisiltamme ja täydentäneet omia tietojamme ja paikanneet heikkouksiamme. Meillä ei ole muuta kuin hyvää sanottavaa parityöskentelystä ohjelmoinnin parissa hyvän ystävän kanssa ja voimme sitä suositella varauksetta ihan kaikille.

Tärkeimpinä oppimiimme asioihin kuuluu muun muassa SQLiten implementointi C#-projektiin ja sen käyttäminen, mobiililaitteilla tärkeät optimointitekniikat sekä yleisesti ottaen paremmat ja tehokkaammat ohjelmointikäytännöt. C#-ohjelmointikielestä erityisesti Linq on tullut tässä projektissa tutummaksi ja sitä tulemme varmasti käyttämään jatkossakin C#-projekteissa. Olemme myös oppineet, että kurinalaisen, hyvän koodin kirjoittaminen on myös erittäin tärkeää ja sillä säästää loppujen lopuksi silkkaa aikaa, vaivaa ja hermoja. Unitylla on myös yllättävän helppo kirjoittaa suorituskyvyllisesti erittäin huonoa koodia, ja tämä on vaatinut hieman omanlaisensa ajattelumallin kehittämistä Unity-kehityksessä. Lisäksi olemme oppineet paljon yleisesti pelien kehittämisestä. Unityn tärkeimmät luokat ja editorin käyttöliittymä on projektin aikana tullut hyvinkin tutuksi ja kehityksestä onkin tullut enemmän suorittamista kuin miettimistä ja tiedon etsimistä Internetistä.

Päätimme olla myös fiksuja, ja tällä kertaa laittaa projekti alun alkaen Git-versionhallintaan. Olemmekin oppineet Git-versionhallinnasta aika paljon ja huomanneet sen tehokkuuden. Rehellisesti sanottuna olimme aika aloittelijoita Gitin kanssa aluksi, mutta erinäisten konfliktitilanteiden ja muiden ongelmien ratkaisemisen jälkeen olemme huomattavasti itsevarmempia Gitin käyttäjiä.

On välillä hyvä pysähtyä miettimään, mitä olisi voinut tehdä paremmin. Ainakin tämän projektin osalta suunnittelu olisi voinut olla tarkempaa alkuvaiheessa – meillä on vähän paha tapa lähteä toteuttamaan projekteja ilman sen suurempia suunnitelmia. Alkuperäinen, hyvin suurpiirteinen suunnitelma on kuitenkin mielestämme toteutunut melko hyvin. On kuitenkin tullut vaiheita, jossa kunnollisen suunnitelman puuttumisen huomasi selvästi.

On ollut rohkaisevaa huomata, että esimerkiksi Satakunnan pelialan kokoontumisissa sekä omissa lähipiireissämme peliprojektimme on otettu hyvin ja innolla vastaan. Se onkin tuonut tuulta purjeisiin ja välillä peli onkin edennyt hyvinkin nopeaa vauhtia, uhraten suuren osan vapaa-ajastamme projektille. Meillä on ollut aika huono tapa aloittaa uusia projekteja ennen kuin nykyinen on päässyt edes alkumetrejä pidemmälle, joten tämän projektin kanssa teimme lupauksen, että mitään uutta projektia ei aloiteta ennen kuin Hungry Bums on valmis. Tästähän me aiomme myös pitää kiinni. Vaikka opinnäytetyö aiheesta saatiin valmiiksi, niin pelin kehittäminen saa jatkua. Toivomme, että peli on jo lähitulevaisuudessa ainakin Googlen Play Store -sovelluskaupassa ladattavissa. Emme ole asettaneet mitään suuria tavoitteita pelaajamäärien tai rahallisen tulon kannalta. Olemme tehneet projektia lähinnä harrastusmielessä ja siksi emme osaa yhtään arvioida tulevaisuuden pelaajamääriä. Mielenkiinnolla odotamme, että millaisen vastaanoton peli saa.

Valmistumisen jälkeen Rikulla on varma visio tulevaisuudestaan: työllistyminen pelialalle. Riku muuttaakin Tampereelle pelialan työpaikkojen perässä. Varmaa on, että Rikulla pelien kehittäminen jatkuu myös harrastustoimintana jatkossakin. Jussi sen sijaan aikoo pysyä ainakin toistaiseksi Porissa ja jatkaa uran rakentamista työpaikallaan. Jussi ei pääse kirjoittamaan olio-ohjelmointia työssään, joten tämäkin projekti oli hyvää kertausta ja lisäharjoitusta sen suhteen. Koska työpaikalla ei saa tehdä ihan mitä itse haluaa, ohjelmointi tulee olemaan Jussillakin varmasti läsnä myös harrastusaktiviteettina, etenkin peliprojektien parissa. Ikuista oppimista on aina painotettu Satakunnan ammattikorkeakoulussa ja siihen me uskomme myös vankasti itse. Tässä työssä käsitelty peliprojekti tulee olemaan mielestämme erittäin hieno lisäys henkilökohtaisiin portfolioihin työllistymisen tueksi.

LÄHTEET

Albahari, B & Albahari, J. 2012. C# 5.0 in a Nutshell, Fifth Edition. O'Reilly Media.

Amazonas, M. 2014. How does Unity export to so many platforms? Viitattu 12.01.2016. Saatavissa: https://sometimesicode.wordpress.com/2014/02/19/how-does-unity-export-to-so-many-platforms/

Amazonas, M. 2014. How does Unity3D Scripting work under the hood? Viitattu 12.01.2016. Saatavissa: https://sometimesicode.wordpress.com/2014/12/22/how-does-unity-work-under-the-hood/

Brodkin, J. 2013. How Unity3D Became a Game-Development Beast. Viitattu 11.01.2016. Saatavissa: http://insights.dice.com/2013/06/03/how-unity3d-become-a-game-development-beast/

Carr, R. 2007. C# Variable Scopes. Viitattu 21.02.2016. Saatavilla: http://www.blackwasp.co.uk/CSharpVariableScopes.aspx

Catton, E. About. Viitattu 12.01.2016. Saatavissa: http://box2d.org/about/

Davis, S & Sphar, C. 2005. C# 2005 For Dummies. Wiley Publishing, Inc.

Enger, M. 2013. Game Engines: How do they work? Viitattu 12.01.2016. Saatavissa: http://www.giantbomb.com/profile/michaelenger/blog/game-engines-how-do-they-work/101529/

Felicia, P. 2015. Unity 5 From Zero to Proficiency. Patrick Felicia.

Firelight Technologies. About Firelight Technologies. Viitattu 12.01.2016. Saatavissa: http://www.fmod.org/about-us/

Hocking, J. 2015. Unity in Action: Multiplatform Game Development in C#. Manning Publications Co.

Icaza, M. 2011. Mono in 2011. Viitattu 20.02.2016. Saatavilla: http://tirania.org/blog/archive/2011/Dec-21.html

Jeffries, R. 2011. What is Extreme Programming? Viitattu 27.01.2016. Saatavilla: http://ronjeffries.com/xprog/what-is-extreme-programming/

Kolev, V & Nakov, S. 2013. Fundamentals of Computer Programming with C#. Faber, Veliko Tarnovo.

Lappalainen, E. 2015. Pelialan uudet luvut julki: Neljässä vuodessa Suomeen syntyi 179 uutta pelifirmaa. Talouselämä. Viitattu 16.03.2016. Saatavissa: http://www.talouselama.fi/kasvuyritykset/pelialan-uudet-luvut-julki-neljassa-vuodessa-suomeen-syntyi-179-uutta-pelifirmaa-3472213

Layton, M. 2012. Agile Project Management For Dummies. John Wiley & Sons, Inc.

Loeliger, J & McCullough, M. 2012. Version Control with Git, Second Edition. O'Reilly Media.

Martin, R. 2009. Clean Code. Prentice Hall.

Mittag, J. 2015. Understanding the differences: traditional interpreter, JIT compiler, JIT interpreter and AOT compiler. Viitattu: 20.02.2016. Saatavilla: http://programmers.stackexchange.com/questions/246094/understanding-the-differences-traditional-interpreter-jit-compiler-jit-interp/269878#269878

Neogames. 2016. Tietoa toimialasta. Viitattu 16.03.2016. Saatavissa: http://www.neogames.fi/tietoa-toimialasta/

Nvidia Gameworks. GameWorks PhysX Overview. Viitattu 12.01.2016. Saatavissa: https://developer.nvidia.com/gameworks-physx-overview

Owens, M. 2006. The Definitive Guide to SQLite. Apress.

Porter. 2013. Unity: Now You're Thinking With Components. Viitattu 13.01.2016. Saatavissa: http://gamedevelopment.tutsplus.com/articles/unity-now-youre-thinking-with-components--gamedev-12492

Saints, J. 2011. What is a bare git repository? Viitattu 10.03.2016. Saatavissa: http://www.saintsjd.com/2011/01/what-is-a-bare-git-repository/

Thorn, A. 2014. Pro Unity Game Development With C#. Apress.

Thorn, A. 2015. How to Cheat in Unity 5. CRC Press.

Tomar, N. 2011. JIT (Just-In-Time) Compiler. Viitattu 21.02.2016. Saatavilla: http://www.c-sharpcorner.com/UploadFile/nipuntomar/jit-just-in-time-compiler/

Trivedi, J. 2014. Ref Vs Out Keywords in C#. Viitattu 09.03.2016. Saatavissa: http://www.c-sharpcorner.com/UploadFile/ff2f08/ref-vs-out-keywords-in-C-Sharp/

Tuliper, A. 2014. Unity : Developing Your First Game with Unity and C#. Viitattu 21.02.2016. Saatavilla: https://msdn.microsoft.com/en-us/magazine/dn759441.aspx

Unity Answers. 2011. How does Unity's built in event system work behind the scenes, as opposed to C# events? Viitattu 18.02.2016. Saatavissa: http://answers.unity3d.com/questions/177245/how-does-unitys-built-in-event-system-work-behind.html#answer-177292

Unity Technologies A. THE LEADING GLOBAL GAME INDUSTRY SOFTWARE. Viitattu 11.01.2016. Saatavissa: https://unity3d.com/public-relations

Unity Technologies B. Platform Specific. Viitattu 12.01.2016. Saatavissa: http://docs.unity3d.com/Manual/PlatformSpecific.html

Unity Technologies C. Audio Overview. Viitattu 12.01.2016. Saatavissa: http://docs.unity3d.com/Manual/AudioOverview.html

Unity Technologies D. 3D Physics Reference. Viitattu 12.01.2016. Saatavissa: http://docs.unity3d.com/Manual/Physics3DReference.html

Unity Technologies E. 2D Physics Reference. Viitattu 12.01.2016. Saatavissa: http://docs.unity3d.com/Manual/Physics2DReference.html

Unity Technologies F. Using Components. Viitattu 13.01.2016. Saatavissa: http://docs.unity3d.com/Manual/UsingComponents.html

Unity Technologies G. Creating and Using Scripts. Viitattu 13.01.2016. Saatavissa: http://docs.unity3d.com/Manual/CreatingAndUsingScripts.html

Unity Technologies H. Execution Order of Event Functions. Viitattu 18.02.2016. Saatavissa: http://docs.unity3d.com/Manual/ExecutionOrder.html

Unity Technologies I. GameObjects. Viitattu 15.01.2016. Saatavissa: http://docs.unity3d.com/Manual/GameObjects.html

Unity Technologies J. Transfroms. Viitattu 18.02.2016. Saatavissa: http://docs.unity3d.com/Manual/Transforms.html

Unity Technologies K. Prefabs. Viitattu 15.01.2016. Saatavissa: http://docs.unity3d.com/Manual/Prefabs.html

Unity Technologies L. Profiler. Viitattu 15.01.2016. Saatavissa: http://docs.unity3d.com/Manual/Profiler.html

Wikibooks. 2015. C# Programming/Syntax. Viitattu 06.02.2016. Saatavilla: https://en.wikibooks.org/wiki/C_Sharp_Programming/Syntax

Yorick. 2012. 4 Ways To Increase Performance of your Unity Game. Viitattu 15.01.2016. Saatavissa: http://www.paladinstudios.com/2012/07/30/4-ways-to-increase-performance-of-your-unity-game/

LIITE 1

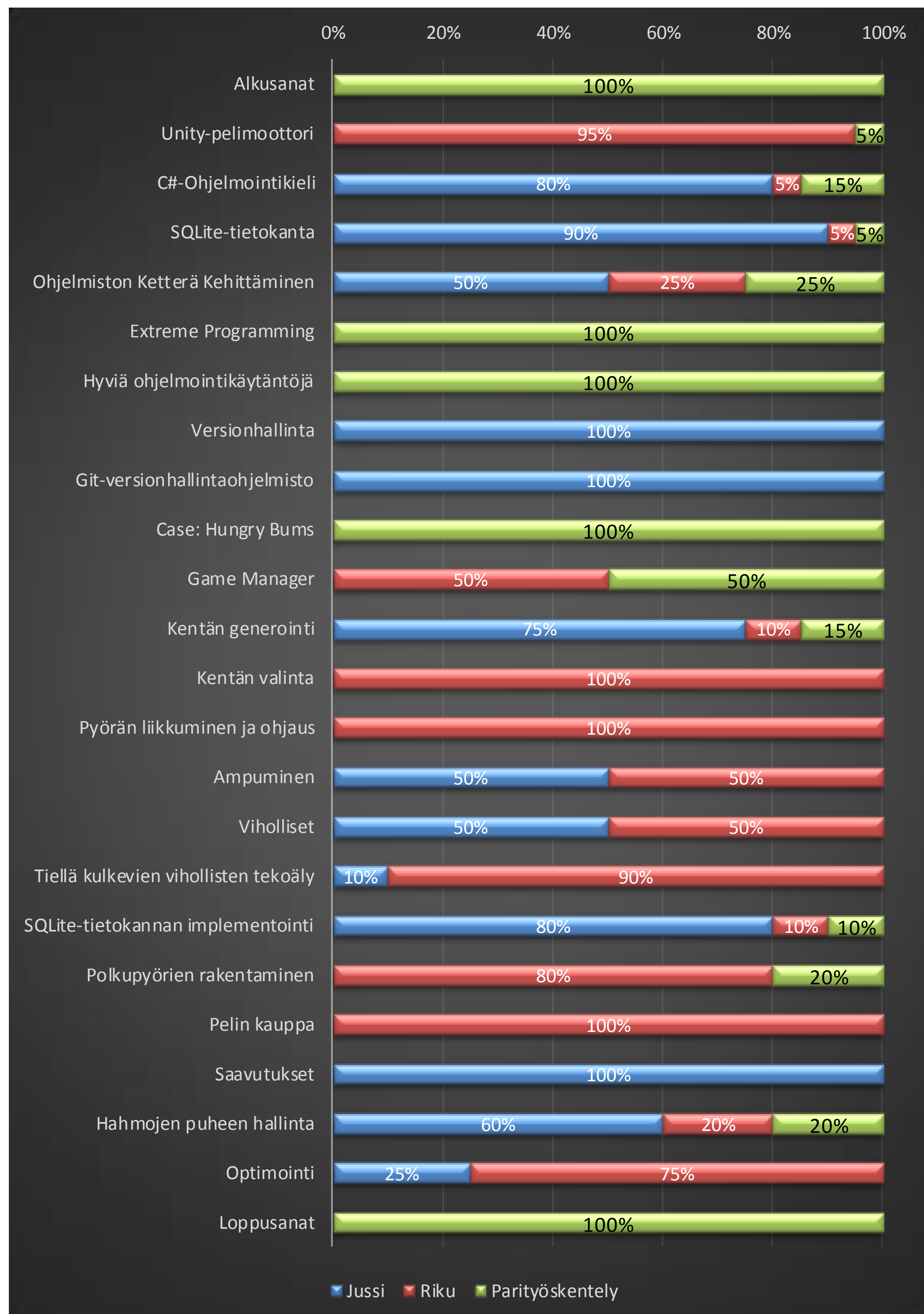
PELIN TARINA

Pelin sankarina toimii Taavi, sysisuomalainen vanhahko mies. Taavi asuu erakoituneena yksin Karhuvuoren metsän siimeksessä. Taavilla ja Karhuvuoressa lymyilevillä karhuilla on ollut jo pitkä ja verinen yhteinen historia. Taavi on lapsena saanut Ukko ylijumalan siunauksen, minkä takia voisi sanoa, että hänellä on yli-inhimillisiä voimia. Vaikka Taavi onkin täysi erakko, joutuu hän hakemaan oluensa kaupungista.

Peli sijoittuukin juuri aikaan, jolloin Taavi joutuu hakemaan ruosteisella pyörällään täydennystä olutvarastoonsa kaupungista. Kaupungissa olikin oluen sijaan pieni yllätys; karhut olivat vallanneet kaupungin ja oluesta saa vain haaveilla, ennen kuin karhut on hoideltu. Taavi tietää, että niin kauan, kun karhut ovat vallassa, oluttuotanto on pysähdyksissä. Karhut ovat takavarikoineet ihmisten ruoat, joten Taavin täytyy myös alkaa ruokkia karhujen pahiten runtelemia katupummeja.

TYÖNJAKO, OPINNÄYTETYÖN KIRJOITTAMINEN

Alla oleva kaavio kuvaa suuntaa antavasti opinnäytetyön kirjoittamisen työnjakoa.



TYÖNJAKO, OHJELMOINTI

Alla oleva kaavio kuvaa suuntaa antavasti ohjelmointityön työnjakoa.