Teppo Kavander

MVC-Arkkitehtuuri peliohjelmoinnissa

Tietotekniikan pro gradu -tutkielma

10. kesäkuuta 2019

Jyväskylän yliopisto

Informaatioteknologian tiedekunta

Tekijä: Teppo Kavander

Yhteystiedot: t.kavander@gmail.com

Ohjaaja: Antti-Jussi Lakanen

Työn nimi: MVC-Arkkitehtuuri peliohjelmoinnissa

Title in English: MVC architecture in game programming

Työ: Pro gradu -tutkielma

Opintosuunta: Ohjelmisto- ja tietoliikennetekniikka

Sivumäärä: 39

Tiivistelmä: Peliohjelmointi muuttuu monimutkaisemmaksi ja projektit entistä laajemmiksi. Tavallisessa ohjelmistokehityksessä on käytössä erilaisia suunnittelumalleja auttamaan kehityksessä sekä erilaisia määritelmiä laadun varmistukseen. Näitä ominaisuuksia ei ole yhtä vahvasti peliohjelmoinnin puolella. Pro gradu -tutkielmassa tutkitaan MVC-arkkitehtuurin sovittamista peliohjelmointiin aiempiin tutkimuksiin ja omiin kokeiluihin perustuen. Aiemmissa tutkimuksissa ei ole pelimoottori ja sen hyödyntäminen ollut esillä. Havaintojen perusteella, MVC-arkkitehtuuri sopii joihinkin pelimoottoreihin hyvin.

Avainsanat: MVC, peliohjelmointi, suunnittelumallit, pelimoottorit

Abstract: Game programming is changing more difficult and projects are growing in size. In normal software development there are different design patterns to help developing and to ensure quality. These features are not that strong in field of game programming. In this master thesis it will be tested if MVC architecture could be used in game programming. Results are based on prior research and own observations. In prior research usefulness of game engine has not been noted well. Based on my observations MVC architecture could offer benefits with some game engines.

Keywords: MVC, game programming, design patterns, game engines

Termiluettelo

C#	Ohjelmointikieli.
Java	Ohjelmointikieli.
Käsittelijä	Tunnetaan myös nimeltä ohjain. Käyttöliittymän toimintojen perusteella ohjelmaa ohjaavaa olioluokka.
Libgdx	Ohjelmistokehys, jota käytetään mm. javalla.
Malli	Datan ja toiminnallisuuden sisältävä olioluokka.
Monogame	Peliohjelmointiin liittyvä ohjelmistokehys.
MVC	Model-View-Controller. Suunnittelumalli, jossa lähdekoodi on jaettu malliin, näkymään ja käsittelijään.
Näkymä	Olioluokka, joka renderöi ohjelman käyttäjälle, esim graafisen käyttöliittymän.
Pygame	Pelikirjasto, jota käytetään pythonilla.
Python	Ohjelmointikieli.
Scene2D	Libgdx-ohjelmistokehyksen graafisen käyttöliittymän rakentamiseen liittyvä kirjasto.
Unity	Pelimoottori, jota käytetään c#-ohjelmointikielellä.
XNA	XNA’s Not Acronymed. Peliohjelmointiin liittyvä ohjelmistokehys.

Sisältö

1 Johdanto

Peliprojektien koko kasvaa entistä suuremmiksi. Peliohjelmoinnista on löydettävissä samat ongelmat ja haasteet kuin tavallisessa ohjelmistokehityksessä, kuten muuttuvat suunnitelmat ja suurten monimutkaisten projektien hallinta. (Qu, Song ja Wei 2016)

Ohjelmistokehityksessä projektien hallintaan ja ongelmien ratkaisuun on kehitetty erilaisia abstrakteja suunnittelumalleja, kuten MVC-arkkitehtuuri. MVC-arkkitehtuuria käytetään suosituissa web-pohjaisissa ohjelmistokehyksissä, kuten Ruby on Rails, Django, ASP.net ja Spring. Sen käyttö pelimoottoreissa on kuitenkin vähäistä. Tässä työssä arvioidaan MVC-arkkitehtuurin soveltuvuutta peliohjelmointiin. Arvioinnissa käytetään apuna aiempia tutkimuksia ja kolmea toisistaan eroavaa pelimoottoria. Aiemmissa tutkimuksissa ei ole huomioitu pelimoottoreiden keskinäisiä eroja. Niissä pelimoottori on ollut yleinen abstrakti käsite, jota ei ole tarkemmin määritelty tai erikseen mainittu, mistä pelimoottorista on kyse. Pelimoottorit ovat tärkeä osa pelinkehitystä nykypäivänä (Anderson E, Engel S, McLoughlin L, Comninos P. 2008). Pelimoottorit eroavat monin tavoin toisistaan aina tuetusta ohjelmointikielestä, oletuksena tarjoamistaan suunnittelumalleista sekä siitä, miten pelikehittäjä luo omaa lähdekoodia ja miten sidoksissa se on pelimoottoriin.

Luvussa 2 esitellään MVC-arkkitehtuuri ja sen eri osa-alueet. Luvussa 3 käydään läpi pelimoottoreiden eri tyyppejä ja kolmea valittua pelimoottoria, joita käytetään esimerkkipelissä sekä tutustutaan aiempiin tutkimustuloksiin MVC-arkkitehtuurin käytöstä peliohjelmoinnissa. Luvussa 4 suunnitellaan esimerkkipeli ja luvussa 5 esimerkkipeli toteutetaan ja toteutus arvioidaan. Luvussa 6 pohditaan ilmenneitä tuloksia ja tehdään yhteenveto MVC-arkkitehtuurin soveltuvuudesta peliohjelmointiin.

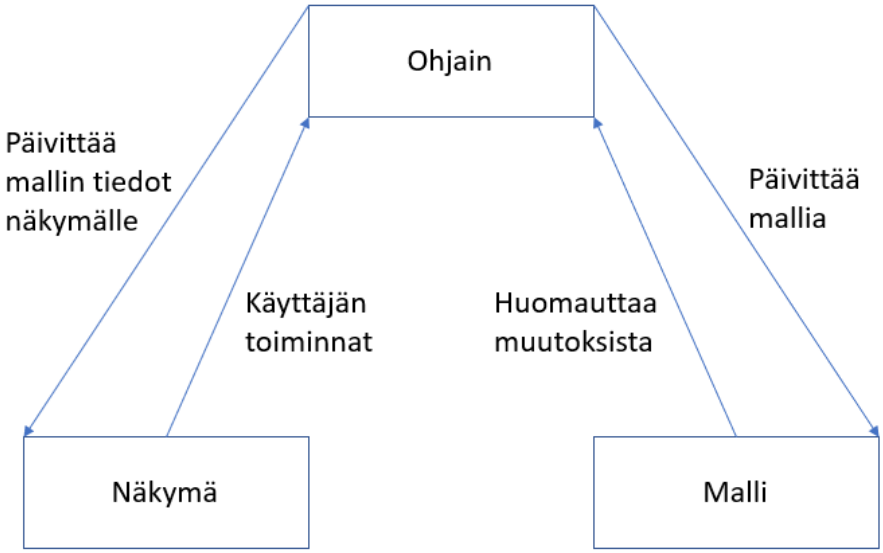
2 MVC-arkkitehtuuri

Tässä luvussa esitellään mikä on MVC-arkkitehtuuri, mistä se koostuu, missä sitä käytetään ja mihin sitä on käytetty. Luvussa käydään myös läpi mitä hyötyjä ja ongelmia MVC-arkkitehtuurin käytössä on huomattu.

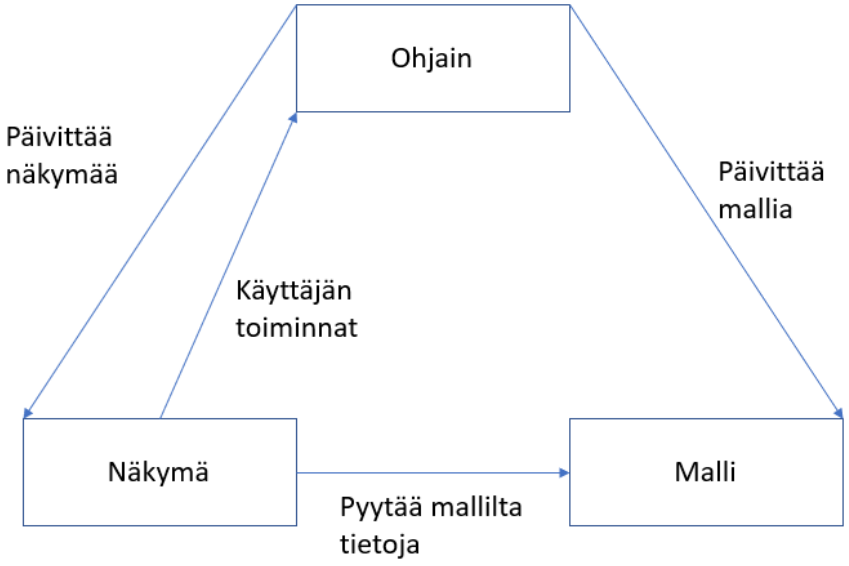
2.1 Rakenne ja toiminnallisuus

MVC-arkkitehtuuri (engl. Model-View-Controller architecture) on yksi tunnetuimmista suunnittelumalleista (Anzures-Garcia A, Sánchez-Gálvez L, Hornos M, Paderewski-Rodriguez P. 2016 ja Caltagirone S, Schlief B, Keys M, Willshire M. 2002). Se koostuu kolmesta eri osasta: mallista, näkymästä ja käsittelijästä. MVC-arkkitehtuurissa lähdekoodin eri osa-alueet pyritään jaottelemaan näihin kolmeen osaan niiden toiminnallisuuksien mukaan. Suunnittelumallina MVC-arkkitehtuuri pyrkii tarjoamaan joustavuutta ja uudelleenkäytettävyyttä (Gamma, Helm, Johnson ja Vlissides 2009 ja Ampatzoglou A, Chatzigeorgiou A. 2007). Kuten muissakin suunnittelumalleissa, hyvän suunnittelumallin pitäisi selkeyttää ja tehostaa projektin kehitystä. Mikäli suunnittelumalli tai arkkitehtuuri aiheuttaa ylimääräistä työtä tai hankaloittaa projektia, niin silloin sitä tulisi välttää (Zavcer G, Mayr S, Petta P. 2014).

MVC-arkkitehtuurin toteutustapa vaihtelee, mutta perusidea, eli asioiden pilkkominen osiin toiminnallisuuksien mukaan, on kuitenkin yhteneväinen (Gamma 2009). Kuvassa 1 ja kuvassa 2 näytetään kaksi hieman toisistaan poikkeavaa tapaa hahmotella MVC-arkkitehtuuria.



Kuva 1. MVC-arkkitehtuurin rakenne, jossa näkymä ja malli eivät tunne toisiaan. (mukaillen Ahex 2018)



Kuva 2. MVC-arkkitehtuurin rakenne, jossa näkymä tuntee mallin. (mukaillen Tutorialsteacher 2018)

Seuraavissa kappaleissa esitellään MVC-arkkitehtuurin osat ja näytetään esimerkit esimerkkiohjelmasta. Esimerkkiohjelma on tehty pythonilla ja sen tarkoitus on havainnollistaa MVC-arkkitehtuuria. Ohjelman idea on komentorivin kautta kysellä käyttäjältä numeroa niin kauan, kunnes käyttäjä antaa tyhjän arvon. Esimerkki on tehty kuvan 2 mukaisella arkkitehtuurilla, eli näkymä voi kysellä mallilta tietoja.

2.1.1 Malli

Malli (engl. model) koostuu objektin datasta, toiminnoista ja logiikasta. Malli voi palauttaa vastauksen käsittelijältä saatuun komentoon (Buckert 2009). Mallin tila ja toiminnot eivät saa olla riippuvaisia näkymästä tai käsittelijältä. Mallin vastuulla on tiedon säilytys ja muokkaaminen (Burbeck S. 1992). Mallia voidaan pitää datan ja logiikan alueena.

Kuvassa 3 on esimerkki mallista. Malli säilyttää tiedon numerosta. Mallilla on kaksi metodia. Ensimmäinen metodi setNumber antaa mahdollisuuden muuttaa mallin numeroa. Jos annettu arvo on numero, se muuttaa numeron ja palauttaa tosi. Jos se on jokin muu kuin numero, esimerkiksi teksti, se ei muuta numeroa ja palauttaa epätoden. Mallilta voi myös kysyä nykyisen numeron arvon getNumber metodilla.

```
class Model:
    def __init__(self):
        self.number = 0

    def setNumber(self, number):
        try:
            number = int(number)
        except:
            #not number, return false
            return False
        self.number = number
        return True

    def getNumber(self):
        return self.number
```

Kuva 3. Esimerkki mallista.

2.1.2 Näkymä

Näkymän (engl. view) tehtäviä ovat mallin havainnollistaminen käyttäjälle ja tarjota käyttäjälle tapa käsitellä ohjelmaa ja sitä kautta mallia (Burbeck S. 1992). Näkymä voi tietää mallista tai sitten ei. Tämä riippuu miten MVC-arkkitehtuuri on toteutettu. Jos näkymä tietää mallin, on se MVC-arkkitehtuurissa oikeutettu vain kysymään mallin nykyistä tilaa. Näkymänä voidaan pitää graafista käyttöliittymää tai jotain muuta tapaa, jolla mallin tiedot saadaan esitettyä käyttäjälle.

Kuvassa 4 on esimerkki näkymästä. Näkymällä on yhteys malliin mutta se saa ainoastaan pyytää siltä tiedon nykyisestä numerosta. Näkymällä on erilaisia metodeja, joilla antaa käyttäjälle tietoja. Näkymä myös pyytää käyttäjältä uutta numeroa ja palauttaa sen.

```
class View:
    def __init__(self, model):
        self.model = model
        print("MVC example")
        self.printNumber()

    def getInput(self):
        text = input("Give new integer: ")
        return text

    def printError(self):
        print("Something wrong with number...")

    def printNumber(self):
        print("Current number is: " + str(self.model.getNumber()))
```

Kuva 4. Esimerkki näkymästä.

2.1.3 Käsittelijä

Käsittelijä (engl. controller) toimii näkymän ja mallin välissä. Käsittelijä vastuulla on käyttäjän syötteen käsittely ja sen perusteella toimiminen. Esimerkiksi kun käyttäjä painaa graafisessa käyttöliittymässä painiketta, niin siitä lähtee tieto näkymältä käsittelijälle ja käsittelijä pyytää sen perusteella mallilta jonkun tiedon ja välittää sen takaisin näkymälle (Burbeck S. 1992). Riippuen MVC-arkkitehtuurin toteutuksesta, voi myös näkymä tietää mallista, jolloin käsittelijä ei välttämättä palauta mitään näkymälle, vaan käsittelijä muokkaa mallia ja näkymä kysyy mallilta suoraan tiedot.

Kuvassa 5 nähdään esimerkki käsittelijästä. Käsittelijällä on yhteys sekä malliin että näkymään. Käsittelijässä on ohjelman pääsilmukka ja se saa näkymältä käyttäjän antaman numeron. Se vie numeron eteenpäin mallille ja mallin vastauksen perusteella päivittää näkymää. Käsittelijä myös lopettaa koko ohjelman, jos näkymältä tulee tyhjä arvo.

```
class Controller:
    def __init__(self):
        self.model = Model()
        self.view = View(self.model)

    def update(self):
        while True:
            text = self.view.getInput()
            if (text == ""):
                break
            if not self.model.setNumber(text):
                self.view.printError()
            self.view.printNumber()
```

Kuva 5. Esimerkki käsittelijästä.

2.2 Hyödyt ja ongelmat

MVC-arkkitehtuurin suurimmat hyödyt tulevat uudelleenkäytettävyydestä ja projektin hallinnan helpottamisesta. Kun eri toiminnot ovat toisistaan selkeästi eroteltuina, niin niiden muokkaaminen tai korvaaminen helpottuu. (Gamma, Helm, Johnson ja Vlissides 2009) Tällöin esimerkiksi voidaan näkymä rakentaa ilman malliin koskemista ja mallia voidaan testata ilman näkymää. Tästä seuraa ongelmana se, että tämä jako voi olla vaikea tehdä. Riippuen toteutettavasta sovelluksesta, ei välttämättä ole täysin selvää, pitääkö jokin toiminto sitoa näkymään, malliin tai käsittelijään.

Tästä syystä MVC-arkkitehtuurilla on monta variaatiota. Näissä kaikissa on sama perusidea, eli toimintojen jako näkymään, malliin ja käsittelijään. Näissä eri variaatioissa on pieniä eroja keskenään, kuten esimerkiksi kuvissa 1 ja 2 nähtiin toisistaan erilaiset toteutustavat. Variaatioiden tarkempi esittely ja erojen läpikäyminen ei kuitenkaan ole tämän gradun kannalta olennaista eikä niitä siksi käydä läpi, sillä gradussa keskitytään siihen, onko MVC-arkkitehtuuri hyödyllinen peliohjelmoinnissa. Lyhyesti eroavaisuudet

toisistaan ovat käytännössä siinä, miten mallin ja näkymän väliltä löytyy ja miten iso ja minkälainen toteutus näillä eri osilla on.

Joskus myös voidaan MVC-arkkitehtuurin näkymä ja käsittelijä yhdistää. Tämänlainen muokkaus voi olla perusteltua, jos on kyse pienestä ohjelmasta tai keskenään toisiinsa sidotuista toiminnoista, kuten valikko (Reenskaug T. 2003).

Ongelmana voidaan myös pitää jaon takia tapahtunutta luokkien määrän kasvua ja niiden keskinäisten suhteiden hallintaa. Suunnittelumalli ei ole hyvä, jos se hankaloittaa projektin kehitystä. (Zavcer G, Mayr S, Petta P. 2014) Tapauksissa, jossa MVC-arkkitehtuurin toteutus ei ole täysin selvillä tai jaottelussa on ongelmia, voi kehityksessä ilmetä ongelmia. Ongelmia voivat olla esimerkiksi se, että toiminnallisuudet, jotka kuuluisivat mallille, ovatkin käsittelijässä tai näkymässä (Olsson T, Toll D, Wingkvist A. 2016). Tämän tapaiset ongelmat rikkovat MVC-arkkitehtuurin idean ja projektin hallinta hankaloituu. Toisaalta ominaisuuksien jako pienempiin kokonaisuuksiin on olio-ohjelmoinnin perusteita. Toimintojen erottelu parantaa mm. ohjelmiston testausta. MVC-arkkitehtuuri hyvin toteutettuna tarjoaa mallin testaamisen helpottamista, kun siihen ei ole sidottu graafisia elementtejä. (Olsson, Toll ja Wingkvist 2015)

2.3 Käyttökohteet

Vaikka MVC-arkkitehtuurilla on ongelmansa, on se kuitenkin hyväksi koettu suunnittelumalli monelle web-kehitysympäristöille. Monet web-kehitysympäristöt, kuten Django (Django project 2018), Ruby on Rails (Ruby on Rails 2018), ASP.net (Asp.net 2018), Backbone.js (Backbone.js 2018) ja moni muu ohjelmistokehys toteuttavat MVC-arkkitehtuurin periaatteita muodossa tai toisessa. Tavallisessa ohjelmistokehityksessä hyväksi koetut toimintamallit pätevät yleensä myös pelikehityksessä. (Kanode C. ja Haddad H. 2009) Pelimoottoreissa ei kuitenkaan ole MVC-arkkitehtuuria oletuksena missään käytössä ja sen takia gradun aiheena onkin tutkia, sopiiko MVC-arkkitehtuuri peliohjelmointiin.

2.4 Aikaisemmat tutkimukset MVC-arkkitehtuurista peliohjelmoinnissa

MVC-arkkitehtuurista on olemassa aiempia tutkimuksia. Tässä kappaleessa esitellään muutama tutkimus ja niiden päätelmät. Myöhemmässä vaiheessa toteutettua esimerkkipeliä peilataan näihin tutkimuksiin.

2.4.1 Mallin ja logiikan erottaminen pelimoottorista

Sheffieldin yliopistossa on huomattu, että kehitettävät pelit ovat usein raskaasti sidottuja valittuun pelimoottoriin. Yliopistossa on mietitty MVC-arkkitehtuurin sovellelmaa, jossa pelimoottori toimisi osana, joka olisi vaihdettavissa. Hyödyiksi tässä ajattelussa on todettu, että se nopeuttaisi pelikehitystä, koska pelimoottorin valinta ei olisi enää niin suuri osa projektia, vaan käytettävän pelimoottorin voisi valita myöhemmässäkin vaiheessa projektia. Myös jos pelimoottori jostain syystä lakkaa olemasta, niin se ei olisi niin suuri takaisku pelikehittäjille (BinSuBaih, Maddock ja Romano 2006).

MVC-arkkitehtuurin sovellelmassa on mietitty, että malli pitäisi sisällään pelin tilat, datan ja logiikan. Eri pelimoottorit olisivat näkymiä. Käsittelijä toimisi näiden kahden välillä adapterina. Käsittelijä luotaisiin aina jokaiselle pelimoottorille erikseen. Ideana olisi pitää pelin logiikka, tilat, data yms yhdessä muuttumattomassa paketissa ja pelimoottoreita varten luodut adapterit muokkaisivat kutsut ja toiminnot pelimoottoreille. Pelimoottorissa luotaisiin samat objektit kuin mallissa, mutta malli ei muuttuisi ja logiikka pysyisi mallissa. Pelimoottori kutsuisi päivityksiä adapterin kautta mallille, jossa pelilogiikka tapahtuisi, ja joka sitten lähettäisi vastaukset adapterin kautta takaisin. (BinSuBaih ja Maddock 2007).

2.4.2 MVC-Arkkitehtuurin toteutus ja arviointi

Linnaeusin yliopistolla on tutkittu ja arvioitu MVC-arkkitehtuurin käyttöä peleissä. Yliopistossa on kiinnitetty huomiota arkkitehtuurin toteutukseen ja mitkä tähän vaikuttavat. Tutkimuksissa on myös huomioitu, mitä ongelmia ja hyötyjä on havaittu.

MVC-arkkitehtuurin on todettu olevan selkeä ja hyvin toimiva interaktiivisissa järjestelmissä. Näissä MVC-arkkitehtuurin jako osiin on helpompi toteuttaa ja ohjeita helpompi noudattaa. Peliohjelmoinnissa ei MVC-arkkitehtuuri ole kuitenkaan yhtä selkeä jakoinen (Olsson, Toll ja Wingkvist 2015).

Sääntöjen ja sovittujen toimintatapojen pitäisi olla selvät, kun lähdetään MVC-arkkitehtuuria toteuttamaan. On havaittu, että liian monimutkaiset liitokset luokkien kesken vaikeuttavat arkkitehtuurin toteuttamista ja ymmärtämistä ja se johtaa virheisiin. On myös havaittu, että peliprojektin pitkäikäisyys on riskitekijä arkkitehtuurin vääränlaiselle soveltamiselle. Esimerkiksi jos projektin alkupuolella ei noudateta ohjeita tarkasti ja tehdään arkkitehtuuria rikkova virhe, jota ei korjata, niin myöhemmin tämä sama virhe voidaan toistaa, kun eri kehittäjät luulevat virheen olevankin oikea tapa toteuttaa arkkitehtuuria. Tämä voi myös luoda lumipalloefektin, jossa tämä virheen toisto luo uusia, toisenlaisia virheitä. Pitkäikäisissä projekteissa voivat myös tekijät vaihdella ja on myös huomattu, että samojen tekijöiden tapa toteuttaa arkkitehtuuria voi muuttua ajan kanssa tai eri projektilla MVC-arkkitehtuuria toteutetaan eri tavoin. (Olsson, Toll ja Wingkvist 2016)

MVC-arkkitehtuurin on kuitenkin huomattu tuovan paljon hyötyjä peliohjelmointiin. Kun näkymät, mallit ja käsittelijät onnistutaan erottamaan toisistamaan ja toteuttamaan laadittujen sääntöjen mukaan, saadaan pelikehitykseen paljon apukeinoja. Kun näkymät ovat erillisinä käsittelijöistä ja malleista, voidaan luoda monta erilaista näkymää, kuten erilliset testausnäkymät, debuggausnäkymät, editointinäkymät jne. Myöskin eri alustoille, kuten tietokone ja puhelin, voidaan luoda erilaiset näkymät ilman että käsittelijöihin ja malleihin kosketaan. Tällöin muutetaan vain pelin näkymää ja toiminnallisuus toimii edelleen täysin samoin (Olsson, Toll ja Wingkvist 2016).

Linnaeusin yliopiston tutkimuksissa todetaan pelimoottorin käytön hyödyt mutta myös sen rajoitettavuus. Tälle ei kuitenkaan esitetä sen tarkempaa ratkaisua vaan pyritään pitämään MVC-arkkitehtuuri mahdollisimman abstraktina sekä pitäytymään projektin alussa luotujen sääntöjen puitteissa. MVC-arkkitehtuurin rakenteen määrittely ja siinä pitäytyminen on erittäin hankalaa. (Olsson, Toll ja Wingkvist 2016).

Linnaeusin tutkimuksissa on nähty useampaa eri variaatiota arkkitehtuurista ja osa niistä on toisia parempia (Olsson, Toll ja Wingkvist 2016). Joissain tapauksissa malli halutaan kokea pelkkänä datamallina, joka ei osaa suorittaa simulaatiota tai itsensä päivitystä ja tällöin vastuu jää käsittelijälle. Joissakin tapauksissa taas käsittelijä on pelkästään kommunikaatiokanava näkymän kautta tulleille käyttäjän komennoille, joilla ohjataan pelin kulkua. Yhteisenä tekijänä on kuitenkin löydettävissä MVC-arkkitehtuurin perusidea: pilkkoa näkymä erikseen datasta ja jollain tapaa kuitenkin yhdistää nämä keskenään.

3 Pelimoottorit

Tässä luvussa tutustutaan kolmeen erilaiseen pelimoottoriin. Esitellyt pelimoottorit on valittu kahdesta syystä. Ensimmäinen syy on, että ne ovat tyypeiltään ja ominaisuuksiltaan erilaisia ja toinen syy on se, että ne ovat ennestään tutkielman kirjoittajalle tuttuja. Erilaisilla pelimoottoreilla halutaan testata, miten MVC-arkkitehtuuri niihin on sovellettavissa.

3.1 Pelimoottorien lajityypit

Pelimoottori terminä on yleisnimitys, jolla viitataan peliohjelmoinnissa apuna toimiviin työkaluihin, joiden avulla kehitetään pelejä (Game career guide 2008). Pelimoottori –termi voidaankin jaotella tarkempiin kuvauksiin, kuten kirjasto, ohjelmistokehys ja pelimoottori. Erot näiden kolmen kesken voivat vaihdella eri tulkintojen mukaan ja tässä tutkielmassa on pyritty käyttämään perusteltua lajittelua sekä myös ottamaan huomioon, miten viitattu pelimoottori itse itsensä määrittelee.

Pelimoottoreilla on jokaisella omat käytäntönsä ja toteutustapansa, joten MVC-arkkitehtuurin sovittaminen pelimoottorin valitsemiin tekniikoihin voi olla hankalaakin. Tämän takia on perustelua testata MVC-arkkitehtuuria erityyppisten pelimoottorien kanssa, jolloin saadaan laajempi näkökulma aiheeseen. Pelimoottorin valinta on tärkeää, koska se voi määrätä hyvinkin paljon, millainen pelin lähdekoodin arkkitehtuurista tulee (Peker A, Can A. 2011).

3.1.1 Kirjasto

Kirjasto on pienin pelimoottoriksi luokiteltava osa. Kirjaston piirteinä on se, että se keskittyy yhteen yksittäiseen osa-alueeseen ja on kokoelma ohjelmakoodia ja dataa, joka liittyy tähän yksittäiseen asiaan. Yksi piirre on myös sen uudelleenkäytettävyys (Game from scratch 2015). Esimerkkeinä kirjastosta voisi olla vaikka grafiikkakirjasto, jonka keskittyy pelkästään grafiikan piirtämiseen ruudulle ja sitä koskeviin toimintoihin.

3.1.2 Ohjelmistokehys

Ohjelmistokehys on kokoelma kirjastoja ja työkaluja, jotka on koottu yhteen toimimaan keskenään (Game from scratch 2015). Näitä kirjastoja kutsutaan avoimien ohjelmointirajapintojen kautta. Ohjelmistokehys eroaa myös kokonaisesta pelimoottorista siinä, että ohjelmoija itse päättää mitä kirjastoja kutsuu missäkin välissä, kun pelimoottorin voi kuvitella ajautuvan yhtenä kokonaisena palana, joka kyselee ohjelmoijan tekemää ohjelmakoodia. Ohjelmistokehys taas tarjoaa ohjelmoijalle suuremman vapauden, missä ja milloin kutsutaan mitäkin ohjelmakoodia. (Hoey 2015, s.10). Ohjelmistokehyksen voidaan kuvitella tarjoavan abstraktit rungot pelille, jotka ohjelmoijan on katsomallaan tavalla täytettävä.

3.1.3 Pelimoottori

Pelimoottori on viimeisin ja se kattaa kaiken edellä mainitun. Pelimoottoriksi luokittelemiseen tarvitaan myös se, että pelimoottori tarjoaa graafisen käyttöliittymän tai tavan toteuttaa ohjelmakoodia graafisesti (engl. Scene graph) (Game from scratch 2015). Pelimoottori koetaan myös suljetummaksi ympäristöksi, jossa ohjelmoija ei näe ihan yhtä vapaasti koko ohjelman toimintaa kuten vaikka kirjaston tai tuotekehyksen kohdalla on. Pelimoottorin voidaan myös nähdä kutsuvan ohjelmoijan luomaa ohjelmakoodia, kun kirjastossa ja ohjelmistokehyksessä ohjelmoija kutsuu niiden ohjelmakoodia (Hoey 2015).

3.2 Pelimoottorit

Tässä luvussa esitellään jokaisesta yllä mainittua pelimoottorilajia vastaava pelimoottori ja esitellään ne lyhyesti. Esiteltävät pelimoottorit eroavat keskenään myös tukemiensa ohjelmointikielten kohdalta, mutta koska kaikki ovat olio-ohjelmointia tukevia kieliä, ei tämä seikka ole merkittävä.

3.2.1 Pygame

Pygame on luokitellaan kirjastoksi, vaikka se myös sisältää kokoelman muita, pienempiä kirjastoja (Pygame, Pygame Docs). Pygame myös luokittelee itse itsensä kirjastoksi (Pygame, Pygame wiki). Pygame ei pakota käyttämään mitään tiettyä suunnittelumallia tai toteutustapaa, vaan jättää vapaat kädet ohjelmoijalle. Pygame on ilmainen ja avointa lähdekoodia.

Pygamea käytetään python –ohjelmointikielellä. Kuvassa 6 on pieni esimerkki, miten pygamea käytetään. Yksinkertaisimmillaan ohjelmassa ladataan moduuli pygame, jolle annetaan ikonit, otsikot ja ikkunan koot ja luodaan pääsilmukka.

```
import pygame
pygame.init()
try:
     screen = pygame.display.set_mode((200, 200))
     pygame.display.flip()
     while True:
          event = pygame.event.wait()
          if event.type == pygame.QUIT:
               break
finally:
     pygame.quit()
```

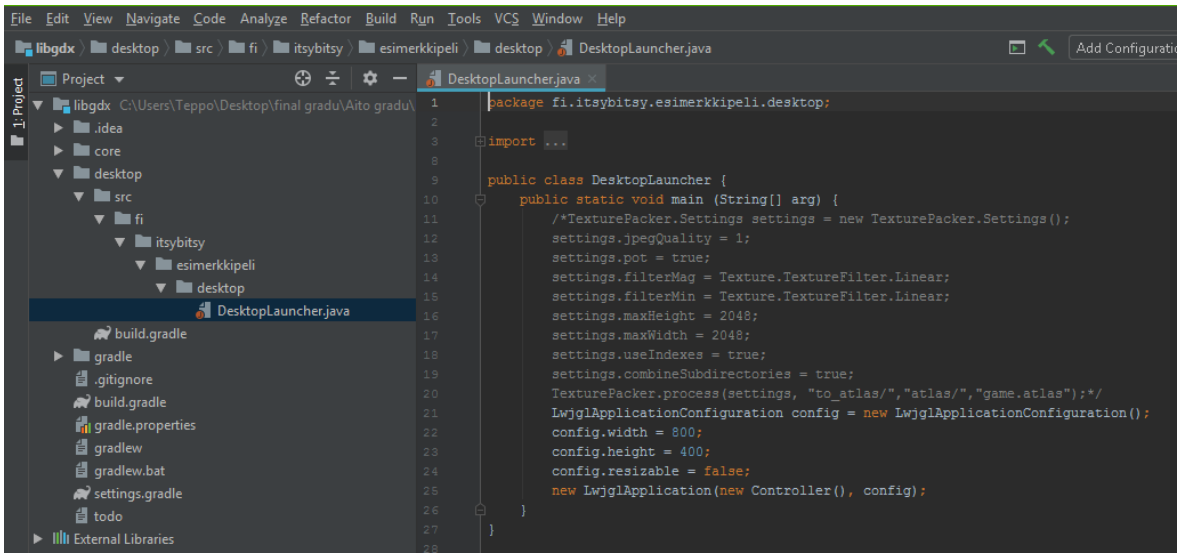
Kuva 6. Esimerkki pygamen rungosta. (mukaillen Pygame, Pygame Docs)

3.2.2 Libgdx

Libgdx on ilmainen avoimen lähdekoodin ohjelmistokehys, joka tukee useita JVM-pohjaisia ohjelmointikieliä, kuten javaa. Libgdx-projektissa on paljon ominaisuuksia ja pääsilmukka on luotuna valmiina. Peliohjelmoijan vastuulle jää täyttää nämä pääsilmukan tyhjät kohdat omalla lähdekoodillaan.

Libgdx:stä puuttuu graafinen käyttöliittymä eikä siinä ole skene-editoria, joten sitä ei voida käytetyllä määritelmällä laskea kokonaiseksi pelimoottoriksi. Libgdx ei tarjoa suoraan mitään suunnittelumallia, joskin siitä löytyy oma vapaaehtoinen entiteettijärjestelmä Ashley, joka perustuu rekursiokoosteeseen (engl. composition).

Libgdx:ssä on jokaiselle tukemalleen alustalle (Windows/Linux/MacOS, android, iphone, html) omat moduulinsa ja ns. käynnistysluokkansa. Näiden lisäksi on yhteinen core-moduuli, jossa käytännössä koko pelin lähdekoodi sijaitsee. Käynnistysluokan tehtävä on määritellä alustakohtaiset asetukset, kuten resoluutio ja työpöytäpelissä ikkunan koko ja voiko ikkunan kokoa muuttaa yms asetuksia. Kuvassa 7 nähdään rakennetta ja työpöytäversion käynnistysluokka.



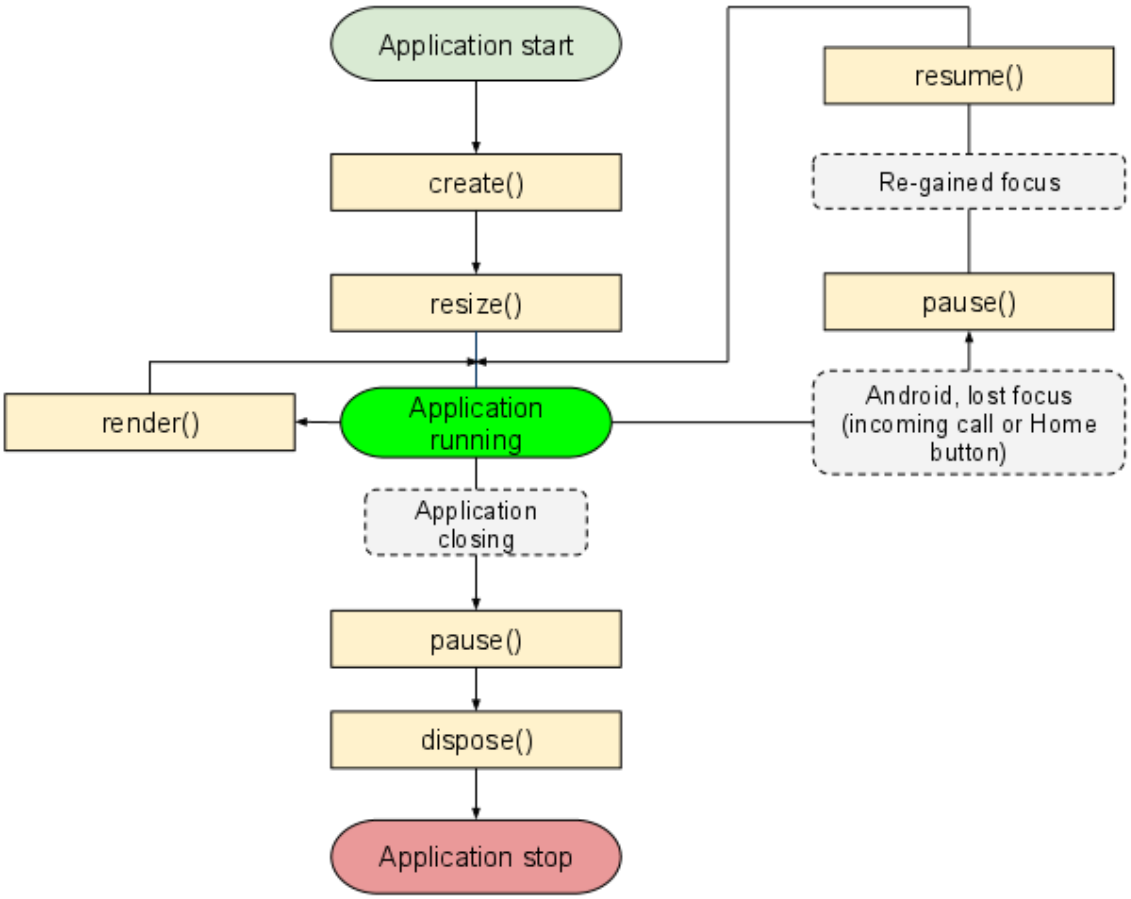
Kuva 7. Libgdx projektin rakenne ja DesktopLauncher –käynnistysluokka.

Core-moduulissa pelin pääsilmukka pyörii ApplicationListener-rajapinnan toteuttavan Game-luokan, tai sen perivän luokan, render-metodissa. Tämä runko näkyy kuvassa 8.

```
package fi.itsybitsy.esimerkkipeli;

import com.badlogic.gdx.Game;
import com.badlogic.gdx.Gdx;
import com.badlogic.gdx.assets.AssetManager;
import com.badlogic.gdx.graphics.Texture;
import com.badlogic.gdx.graphics.g2d.TextureAtlas;
import com.badlogic.gdx.scenes.scene2d.ui.Skin;
import fi.itsybitsy.esimerkkipeli.models.WorldModel;
import fi.itsybitsy.esimerkkipeli.views.GameView;

public class Controller extends Game {
   private WorldModel worldModel;
   private AssetManager assetManager;

   @Override
   public void create() {
   }

   @Override
   public void render() {
   }
```

Kuva 8. Libgdx pääsilmukka.

Render-metodi ei nimensä mukaisesti ole piirtämistä varten, vaan se voidaan kuvitella olevan yleinen päivitysmetodi. Kuvassa 9 on havainnollistettu libgdx-projektin kulkua alusta loppuun.



Kuva 9. Libgdx:n sovelluksen kierto. (Libgdx lifecycle)

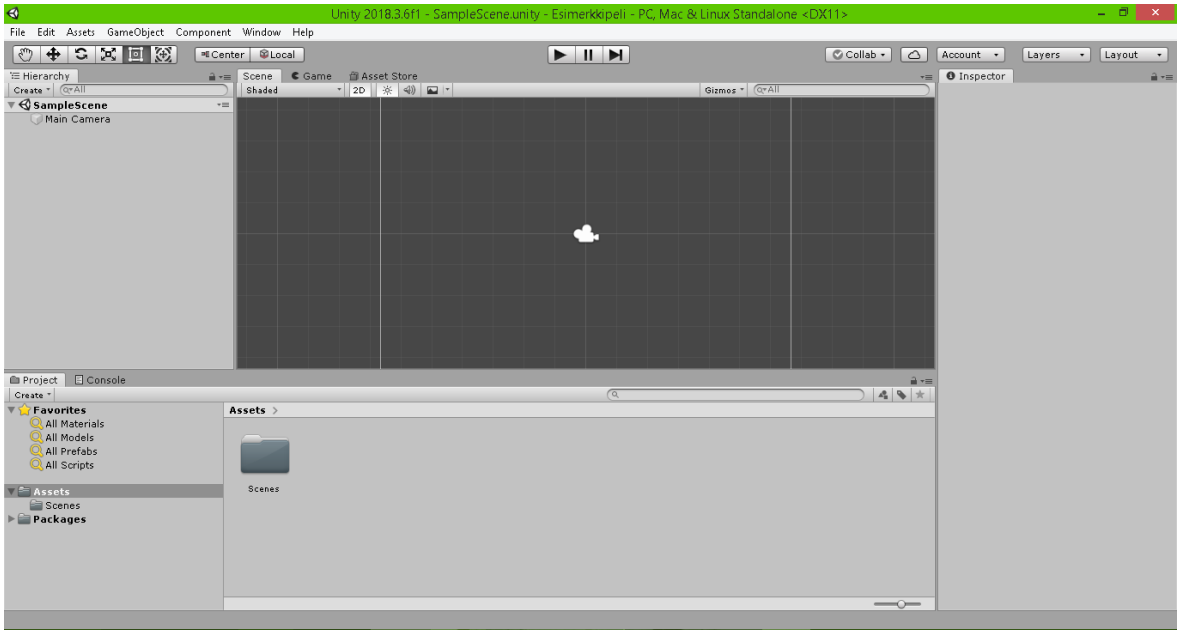
3.2.3 Unity

Unity on näistä kolmesta pelimoottorista laajin ja se luokitellaan pelimoottoriksi. Siitä löytyy runsaasti valmiita toimintoja ja skene-editori ja graafinen käyttöliittymä. Unity tukee C#-ohjelmointikieltä. Toisin kuin pygame ja libgdx, jotka eivät suoraan pakota ohjelmoijaa käyttämään tiettyä mallia, unityssä on käytössä rekursiokooste. Tämä luo tiettyjä haasteita MVC-arkkitehtuurin sovittamiseen unity ympäristöön.

Unity on kolmesta esitellystä pelimoottorista kaikkein suosituin. Unitystä löytyy sekä ilmainen että maksullinen versio. Maksullinen versio tarjoaa mm. suoraa tukea unityn

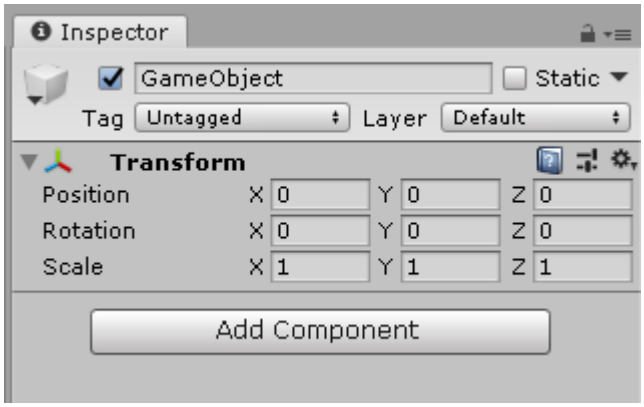
kehittäjiltä, analytiikkaa sekä muita lisäominaisuuksia. Pygamesta ja libgdx:stä poiketen, unityn lähdekoodi ei ole avointa eikä saatavilla pelikehittäjille.

Kuvassa 10 on näkymä unityn kenttäeditorista. Unityssä on mahdollista luoda uusia objekteja pelimaailmaan valikosta vetämällä. Käyttöliittymä tarjoaa helpot tavat muokata ja muuttaa pelimaailmaa hiirtä käyttäen.



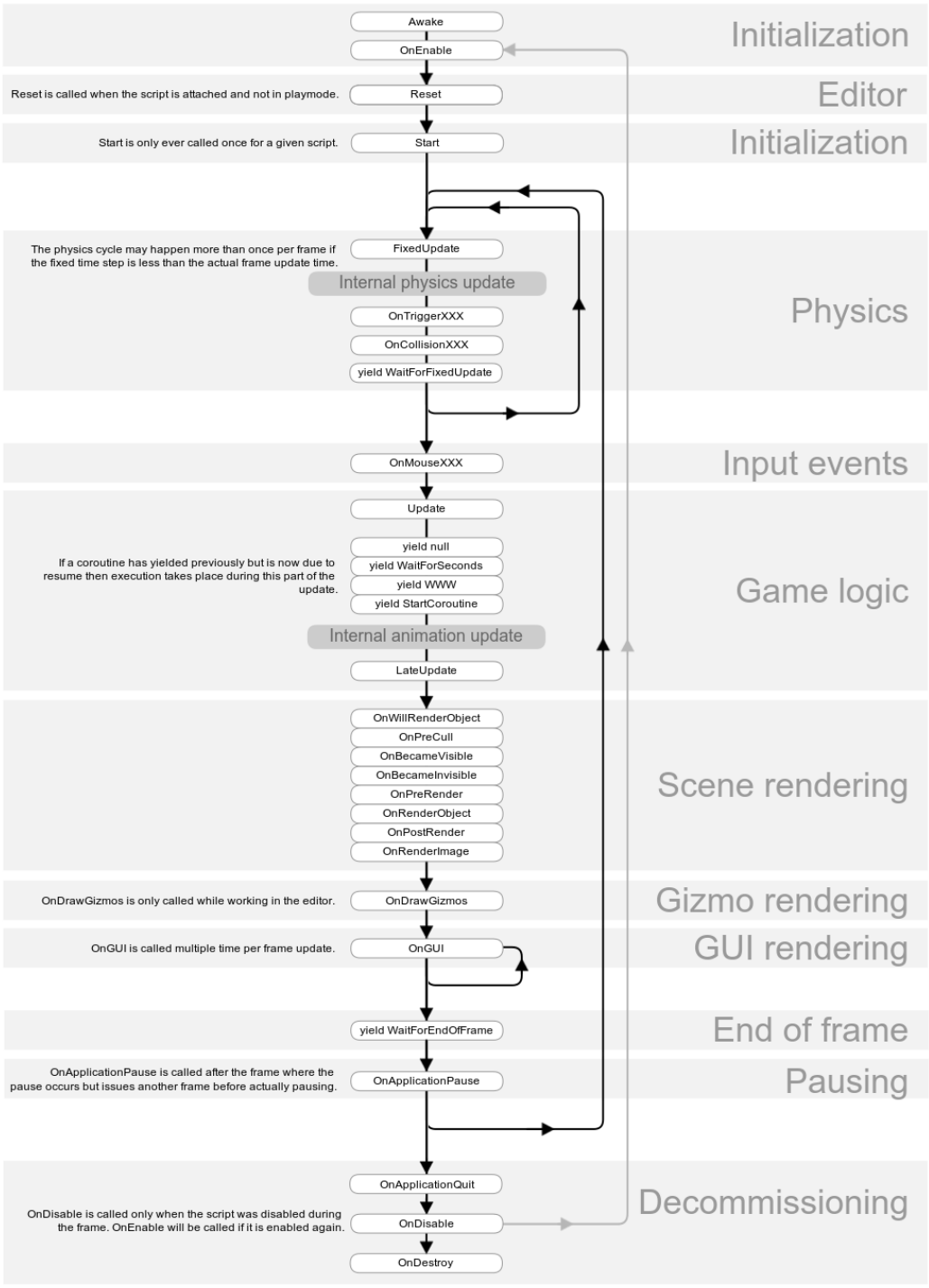
Kuva 10. Unityn editori.

Unityssä kaikki objektit perityvät GameObject-luokasta. Kuvassa 11 on esitetty yksinkertaisin mahdollinen objekti. Jokaisella GameObject-luokan perivällä objektilla on myös Transform-luokan ilmeentymä, joka pitää sisällään mm. sijaintivektorin. Tämän takia täysin täydellistä MVC-arkkitehtuuria ei unityllä saa aikaan, sillä näkymillä tulee aina olemaan sijaintivektori eikä MVC-arkkitehtuurissa näkymän kuuluisi pitää sisällään mallille kuuluvia osasia. Tämä ei kuitenkaan tarkoita, etteikö MVC-arkkitehtuuria voisi soveltaa unityn kanssa.



Kuva 11. Tyhjä peliobjekti.

Unity on suljettua lähdekoodia. Unityssä on hyvin paljon toiminnallisuuksia ja siten myös sillä toteutetun pelin elämänkaari on monimutkaisempi, kuin esimerkiksi libgdx:ssä. Kuvassa 12 on esitelty unityn elämänkaarta alusta loppuun. Eroten pygameen ja libgdx:ään, unityssä ei pääsilmukkaa päästä muokkaamaan tai päättämään missä järjestyksessä asiat toteutetaan. Unityssä noudatetaan kuvan 12 mukaista järjestystä ja pelimoottori itse kutsuu kaikkien skriptien päivitysmetodeita automaattisesti.



Kuva 12. Unityn kiertokulku. (Unity execution order)

4 Tutkimusasetelma

Tässä kappaleessa suunnitellaan, millainen esimerkkipeli luodaan kolmella pelimoottorilla ja miten rakenne suunnitellaan. Kappaleessa käydään läpi millä tavoin MVC-arkkitehtuurin soveltuvuutta peliohjelmointiin arvioidaan.

Peliohjelmoinnin ongelmana on, että peliohjelmoinnin laatua mitataan usein peliobjektin toiminnan toteutumisessa eikä niinkään lähdekoodin tasokkuudella, kuten uudelleenkäytettävyydellä ja joustavuudella. Sen takia pelin kehitysvaiheessa saatetaan olemassa olevaa lähdekoodia joutua muokkaamaan paljon tai jopa kirjoittamaan kokonaan uudelleen, kun tulee uudenlaisia muutospyyntöjä. MVC-arkkitehtuuri voi tuoda peliohjelmointiin apua, koska MVC-arkkitehtuuri tarjoaa abstraktit ratkaisut lähdekoodin rakenteelle.

4.1 Menetelmäkuvaus

Arviointia MVC-arkkitehtuurin soveltuvuudesta peliohjelmointiin toteutetaan kehittämällä peli kolmella eri pelimoottorilla. Peli on samanlainen toiminnaltaan kaikilla pelimoottoreilla. Peliä kehitettäessä käytetään MVC-arkkitehtuuria. MVC-arkkitehtuuria ei pyritä eristämään pelimoottorista vaan rakentamaan pelimoottoria apuna käyttäen. Esimerkiksi malliin voidaan käyttää pelimoottoriin sidottuja luokkia tai objektit voivat periytyä pelimoottoriin sidoksissa olevista luokista.

Arviointiperusteena keskitytään siihen, miten hyvin MVC-arkkitehtuuri sopii ja pystyy hyödyntämään pelimoottorin tarjoamia ominaisuuksia ja luokkia. Arvioinnin kannalta hyvässä pelimoottorissa arkkitehtuuri on sovellettavissa ilman suurempaa lisätyötä ja samalla luontevasti pelimoottoriin sopien, esimerkiksi pelimoottorin luokkia perien. Hyvä pelimoottori arvioinnin kannalta ei myöskään tee MVC-arkkitehtuurin soveltamisesta vaikeaa. Jos on vaikea erotella malli, näkymä ja käsittelijä toisistaan, niin se on ongelmallista. MVC-arkkitehtuurin yhtenä piirteenä on helpottaa projektin hallintaa, joten arkkitehtuurin käyttö pelimoottorissa ei saisi hankaloittaa pelikehitystä.

Ensimmäinen peli toteutetaan libgdx:ää käyttämällä. Se on itselleni tutuin pelimoottori. Kun sen avulla on saatu yksi esimerkkipeli valmiiksi, teen seuraavan pygamella ja viimeisenä unityllä. Käytän apuna ja mallina libgdx:llä tehtyä esimerkkipeliä, kun toteutan pygame ja unity versiot esimerkkipelistä. Peliä kehittäessä otan huomioon miten toteutukset eroavat toisistaan ja mitä oppimista tai havaintoja löysin.

4.2 Esimerkkipeli: tornipuolustus

Peliä suunnitellessa pyritään pitämään peli yksinkertaisena, mutta sen verran monipuolisena että siinä on nähtävissä MVC-arkkitehtuurin elementtejä sekä että niistä voidaan tehdä johtopäätöksiä. Peli pidetään suppeana ja sen visuaalisuuteen ei panosteta, koska tyylikkäät animaatiot ja visuaalinen ilme ovat merkityksettömiä vertailua tehdessä. Myös se, että peli toteutetaan kolme kertaa eri pelimoottorilla, on yksi syy pitää peli suppeana.

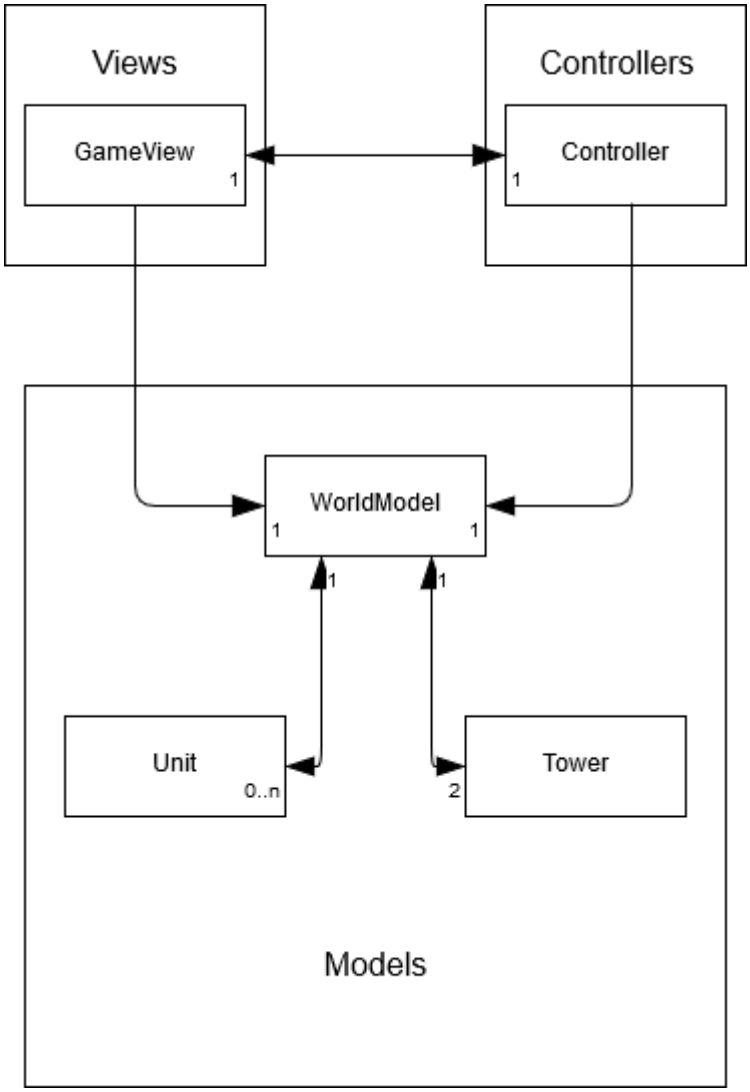
Vertailua varten luodaan yksinkertainen 2D torninpuolustuspeli, jossa kentän molemmilla laidoilla on kahden eri pelaajan tornit. Tornista voidaan luoda yksikkö, joka vaeltaa kohti vihollisen tornia. Törmätessään vihollisen yksikköön tai torniin, ne alkavat taistelemaan. Pelin voittaa se, joka ensin tuhoaa vihollisen tornin.

Ideana on tuottaa kolmella pelimoottorilla johonkin pisteeseen asti runko tämän tapaiselle pelille ja nähdä, miten MVC-arkkitehtuuri sopii kyseisiin pelimoottoreihin.

4.3 MVC-arkkitehtuuri esimerkkipelissä

MVC-arkkitehtuuri koitetaan pitää esimerkkipeleissä yksinkertaisena. Peliä varten luodaan vain yksi näkymä, yksi kontrolleri ja yksi isompi malli. Isompi malli toki koostuu pienemmistä objekteista, jotka ovat itsessään malleja, mutta tässä pelissä niille ei luoda erillisiä omia näkymiään tai käsittelijöitään. Esimerkiksi pelaajan tornille sekä pelaajan yksikölle voisi olla olemassa omat yksittäiset näkymä- ja käsittelijäobjektinsa, mutta tässä tapauksessa mennään edellämainitun tapaan yksillä yhteisillä näkymillä, käsittelijöillä ja mallilla. Kuvassa 13 on esiteltynä suunnitteltua kaaviota arkkitehtuurista esimerkkipelille.

Arkkitehtuuri saattaa muuttua hieman riippuen pelimoottorista, mutta pääpiirteittäin se on kuin kuvassa.



Kuva 13. Arkkitehtuurin osat esimerkkipelissä.

Mallin tehtävä esimerkkipelissä on pitää sisällänsä dataa ja pelilogiikkaa. Näkymä piirtää koko pelimaailman ja sen yksiköt. Näkymä tarjoaa myös painikkeen, jolla pelaaja voi kutsua uusia yksikköjä taistelemaan puolestaan. Käsittelijä vastaanottaa näkymän tapahtumat (esimerkiksi nappulan painalluksen) ja kysyy mallilta, onko mahdollista luoda uusi yksikkö pelaajalle ja jos on, käskee mallia luomaan sen. Käsittelijä myös kontrolloi

yleisesti peliä, kuten tarkistaa onko voittaja selvinnyt ja jos on, lopettaa mallin update-metodin kutsumisen. Näkymä pyytää mallilta sijainnit ja mitä sen pitää piirtää.

Kuvan 13 arkkitehtuurissa voidaan pitää kriittisenä ja potentiaalisena ongelmana sitä, että näkymällä on suora yhteys malliin. Tämän kautta on siis mahdollista, että näkymä käskyttäisi suoraan mallia. Turvallisempana ratkaisuna voisi pitää sitä, että mallin tiedot tulisi pelkästään käsittelijän kautta, mutta tämä lisää arkkitehtuurin monimutkaisuutta.

Riippuen miten MVC-arkkitehtuurin haluaa soveltaa, niin pelilogiikka voi olla myös kokonaan käsittelijällä, jolloin mallin rooli on sisältää pelkkää dataa ja käsittelijän rooli on muokata ja ohjata mallia ja sen dataa. Tässä esimerkkipelissä käsittelijälle annetaan vastuuksi käskyttää mallia näkymän graafisen käyttöliittymän käytön perusteella sekä hallita pelin yleistä tilaa, kuten kun jompikumpi pelaaja häviää, niin käsittelijä ohjaa pelin uusiin tiloihin. Graafinen käyttöliittymänä pelissä toimii painike, josta pelaaja voi luoda yksiköitä. Painikkeita kontrolloidaan hiirellä. Kuten on aiemmin jo mainittu, MVC-arkkitehtuuri on ohjeistus ja siitä on erilaisia variaatioita. Kunkin variaation ja arkkitehtuurin toteutustapa ovat sidoksissa minkälaista projektia tehdään ja mikä lähestymistapa koetaan parhaimmaksi.

Mikäli peli olisi hyvin suuri ja laaja, voitaisiin pilkkoa toiminnallisuuksia vielä pienemmiksi osiksi kuin nyt esimerkkipelissä tehdään. Esimerkiksi jokaisella mallilla voisi olla omat käsittelijänsä ja näkymänsä, toisin kuin nyt näitä on niputettu yhteen, jolloin näkymä hoitaa kaikkien objektien piirtämisen ruudulle, ja jokainen malli sijaitsee pelimaailmamallin sisällä ja käsittelijä toimii näiden välikätenä ja yleisenä pelin tilojen hallitsijana.

5 Esimerkkipelin toteutuksen arviointi

Tässä kappaleessa tutustutaan esimerkkipelin toteutukseen ja siinä esille nousseisiin havaintoihin ja näytetään joitain tärkeiksi koettuja lähdekoodin katkelmia. Tärkeinä lähdekoodin katkelmina pidetään niitä kohtia lähdekoodista, jossa voidaan näyttää MVC-arkkitehtuurin osia tai niiden käyttöä pelimoottorissa. Käydään myös pelimoottorikohtaisesti arviointia läpi. Arvioinnissa keskitytään siihen, miten helpoksi MVC-arkkitehtuurin sovitus koettiin kunkin pelimoottorin kohdalla luvun 4 arviointikriteerien perusteella. Tässä kappaleessa keskitytään omiin havaintoihin.

5.1 Yleinen arviointi tehdystä esimerkkipelistä

Esimerkkipelissä pidettiin hyvin yksinkertaisena MVC-arkkitehtuurin toteutus ja sen myötä käsittelijöitä ja näkymiä oli molempia tasan yksi kaikissa esimerkkipelin versioissa. Näkymiä olisi voinut olla useampia lisää, kuten aloitus ja asetukset-valikko jne. Näille kaikille näkymille olisi voinut olla sitten oma käsittelijänsä tai pitää edelleen se yksi iso käsittelijäluokka, jonka sisälle voisi sujuvasti vaihtaa aina kuhunkin näkymään tai tilanteeseen liittyvän oman käsittelijän tai peräti useamman. Esimerkiksi jos peli olisi tasoloikkapeli, olisi voinut pelaajalla olla ihan oma käsittelijäluokkansa, jonka ilmentymä olisi isomman käsittelijäluokan sisällä. MVC-arkkitehtuuri ja sen sovellelmat eivät ole kiveen hakattuja vaan niitä voidaan pitää ohjenuorana. Tärkeimpänä on asioiden, kuten piirtämisen ja datan, erottelu toisistaan.

5.2 Pygame

Pygamessa MVC-arkkitehtuuri oli helppo sovittaa pelimoottoriin. Pygame-kirjaston omia toimintoja kutsuttiin vain vähäisissä määrin. Näkymässä luotiin ikkunat ja käsittelijässä käytettiin Clock-luokan objektia rajoittamaan pelin päivitystiheys 60 kertaan sekunnissa ja laskemaan delta-aikaa, kuten kuvassa 14 nähdään. Malleissa ei pygame ominaisuuksia käytetty vaan ne ovat täysin puhdasta python-ohjelmointikieltä ilman sidonnaisuuksia pygame-kirjastoon.

```
class MyGame:
    def __init__(self):
        self._running = True
        self.clock = pygame.time.Clock()

   def on_loop(self):
       dt = self.clock.tick(60)
       dt = 1/dt
       self.update(dt)
       self.view.update()
```

Kuva 14. Pygamen käyttö käsittelijässä.

Näkymässä käytettiin pygame-kirjastoa alustamaan ikkuna, lataamaan grafiikat sekä piirtämään grafiikat ruudulle. Kuvassa 13 nähdään näkymän rakentaja- ja piirtometodit ja pygamen käyttö. Kuvassa 15 on tilan takia kommentoitu loput grafiikkojen lataukset piiloon, mutta käytännössä menevät kuten tornien latauskin.

```
class WorldView:
    def __init__(self, controller):
        pygame.init()
        self.size = self.width, self.height = 800, 400
        self._display_surf = pygame.display.set_mode(self.size)
        self.background = pygame.image.load("assets/bg.png").convert_alpha()
        self.tower1 = pygame.image.load("assets/tower_gre.png").convert_alpha()
        self.tower2 = pygame.image.load("assets/tower_red.png").convert_alpha()
        #... loppujen grafiikkojen lataus
        self.background = pygame.transform.scale(self.background, (800, 400))
        self.model = controller.getModel()
        self.controller = controller
        self.button = pygame.Rect(100, 100, 50, 50)
        self.screen = pygame.display.set_mode(self.size)

   def draw(self):
      self.drawBackground()
      self.drawPlayersAndUnits()
      self.drawGUI()
      pygame.display.flip()
```

Kuva 15. Pygamen käyttö näkymässä.

Pygamen kanssa MVC-arkkitehtuuri on helppo toteuttaa. Pygame-kirjasto tosin on kokoelma pienempiä moduuleita, kuten käsittelijässä käytetty time- tai esimerkiksi music-moduuli. Sen kautta on näkymällä mahdollisuus ladata ja toistaa musiikkitiedostoja, vaikka ne eivät näkymälle kuuluisikaan. Tällä ei kuitenkaan ole niin väliä, jos ohjelmoija tekee selkeän rajan missä kohdassa tehdään mitäkin. Käytännössä pygame voi olla hyvin

minimaalisesti mukana ja suurimmaksi osaksi peliä voidaan kehittää pythonilla ja turvautua pygameen vain tarvittaessa, kuten ikkunan luonnissa ja grafiikan piirtämisessä ruudulle. Pygamen kohdalla ei onnistu luontevasti periytyttää näkymiä, malleja tai käsittelijöitä pygamen tarjoamista luokista.

5.3 Libgdx

Libgdx:llä pystyi hyvin määrittämään MVC-arkkitehtuurin rakenteen ohjelmistokehyksen omia luokkia hyväksikäyttäen. Libgdx:n Screen-rajapinnan toteutuksen kautta saatiin luotua näkymä ja Game-luokan perivän luokan kautta käsittelijä. Libgdx:ssä on pygamea monimutkaisempi rakenne ja sitä lienee hyvä avata tarkemmin.

Libgdx:n esittelyssä luvussa 3 puhuttiin, että jokaisella alustalla (Windows/Linux/Mac, android, ios, html5) on oma käynnistysluokkansa. Kuvassa 16 nähdään esimerkkipelin käynnistysluokka pelin työpöytäversiolle.

```
package fi.itsybitsy.esimerkkipeli.desktop;

import com.badlogic.gdx.backends.lwjgl.LwjglApplication;
import com.badlogic.gdx.backends.lwjgl.LwjglApplicationConfiguration;
import com.badlogic.gdx.graphics.Texture;
import com.badlogic.gdx.tools.texturepacker.TexturePacker;
import fi.itsybitsy.esimerkkipeli.Controller;

public class DesktopLauncher {
   public static void main (String[] arg) {
      LwjglApplicationConfiguration config = new LwjglApplicationConfiguration();
      config.width = 800;
      config.height = 400;
      config.resizable = false;
      new LwjglApplication(new Controller(), config);
   }
}
```

Kuva 16. Käynnistysluokka työpöytäversiolle esimerkkipelistä.

Kuvan 16 main-metodin lopussa luotava LwjglApplication-olio ja sille parametrina annettu Controller-luokan olio periytyy libgdx:n Game-luokasta. Game-luokan render-metodin voidaan kuvitella olevan pelin pääsilmukka. Kuvassa 17 nähdään Controller-luokan periytyminen sekä create-metodi, jossa ladataan grafiikka, alustetaan malli sekä liitetään näkymä.

```
public class Controller extends Game {
   private WorldModel worldModel;
   private AssetManager assetManager;

   @Override
   public void create() {
      worldModel = new WorldModel();
      assetManager = new AssetManager();
      assetManager.load("atlas/game.atlas", TextureAtlas.class);
      assetManager.load("background1.png", Texture.class);
      assetManager.load("gui/uiskin.json", Skin.class);
      assetManager.finishLoading();
      setScreen(new GameView(worldModel,this));
   }
```

Kuva 17. Controller-luokan, eli käsittelijän, alustus.

Game-olioluokka pitää sisällään Screen-rajapinnan toteuttavan olion, jota se kutsuu omassa render-metodissaan. Kuvassa 18 nähdään Controller-luokan render-metodi, jossa päivitetään mallia sekä kutsutaan Game-luokan omaa render-metodin toteutusta, jonka kautta saadaan näkymä päivittymään.

```
@Override
public void render() {
   if (worldModel.getPlayerOne().getHealth() > 0 && worldModel.getPlayerTwo().getHealth()
> 0){
      //players are alive, keep updating game
      worldModel.update(Gdx.graphics.getDeltaTime());
   }
   super.render(); //calls screen render() method with deltatime
}
```

Kuva 18. Controller-luokan render-metodi.

Kuten edellä olevista kuvista on nähty, on libgdx-ohjelmistokehyksen ohjelmakoodia enemmän käytössä sekä käsittelijässä että näkymässä. Libgdx:n käyttö rajaa hieman, miten ohjelmakoodi päätetään toteuttaa mutta tarjoaa silti hyvin vapaat kädet toteutukselle. MVC-arkkitehtuuri myös nivoutuu hyvin yhteen libgdx:n luokkien kanssa.

Toisin kuin pygamen kohdalla, myös mallissa on käytössä libgdx:n luokkia, eli malli on sidottu pelimoottoriin. Mallissa käytetään mm. Vector2-olioluokkaa pitämään yllä sijaintia. Muita käytettäviä luokkia ovat libgdx:n tarjoamat matematiikkakirjastot sekä libgdx:n oma taulukko-olioluokka, joka on optimoidumpi pelejä ajatellen kuin javan omat taulukkorakenteet. Kuvassa 19 on esitelty Unit-luokan alkupäätä.

```
package fi.itsybitsy.esimerkkipeli.models;

import com.badlogic.gdx.math.Rectangle;
import com.badlogic.gdx.math.Vector2;
import com.badlogic.gdx.utils.Pool;

public class Unit implements Pool.Poolable {
    private Vector2 position;
    private Rectangle hitbox;
    private float attackRange;
    private int team; //1 or 2
    private float statetime;
    private int state; //1 run, 2 att, 3 deadAnim, 4 deadComp
    private int type; //1 ninja, 2 robot
    private int health;
    private float speed = 1.5f;
    private float direction; //-1 or 1
    private WorldModel worldModel;
```

Kuva 19. Libgdx:n kirjastoja mallissa.

Libgdx:n kohdalla näkymä on hyvin samankaltainen kuin pygamessa. Se piirtää asiat ruudulle ja hakee sijainti- ja tilatiedot mallilta. Kuvassa 20 nähdään Screen-luokan toteuttavan näkymän render-metodi, jossa piirtäminen tapahtuu.

```
@Override
public void render(float delta) {
    Gdx.gl20.glClear(GL20.GL_COLOR_BUFFER_BIT);
    stage.act(delta);
    batch.setProjectionMatrix(viewport.getCamera().combined);
    batch.begin();
    batch.disableBlending();
    batch.draw(background,0,0,16,8);
    batch.enableBlending();
    drawPlayers();
    drawUnits();
    batch.end();
    stage.draw();
}
```

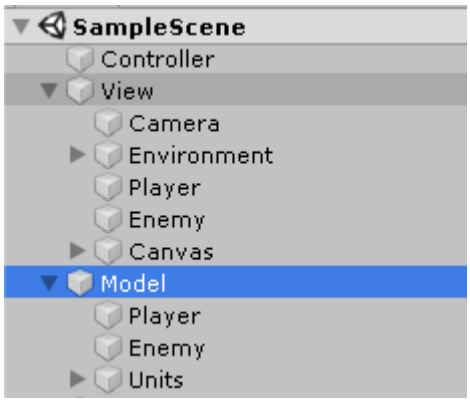
Kuva 20. Screen-luokan toteuttavan näkymän render-metodi.

Kuvassa 18 nähdään olio stage. Se on osa libgdx:n scene2D-kirjastoa, jonka avulla piirretään peleihin graafinen käyttöliittymä. MVC-arkkitehtuuria on paljon käytetty mm. websovellusten kanssa ja MVC-arkkitehtuuri onkin yleisesti osa graafisia käyttöliittymiä. Scene2D-kirjastossa on toteutettu toiminnallisuus, kontrolli ja piirto kaikki yhdessä. Sitä ei siis saa pilkottua MVC-arkkitehtuuriin. Kuvassa 20 nähdään stage-olion act- ja draw-metodit.

5.4 Unity

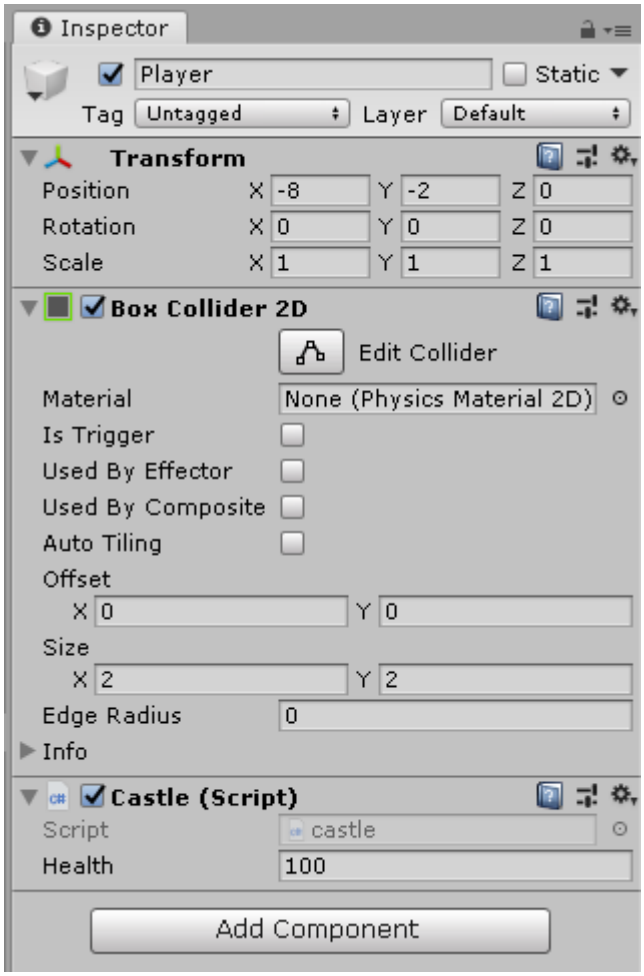
Unityn kohdalla on MVC-arkkitehtuurin kanssa suurimmat haasteet. Kuten luvussa 3 todettiin, jokainen objekti unityssä periytyy Gameobject-luokasta. Unity toteuttaa valittuna suunnittelumallinaan rekursiokoostetta ja saa myös aina Transform-olion mukanaan. Täten meillä on jo lähtökohtaisesti näkymällä ja mallilla sekä käsittelijällä kaikilla omat Transform-olionsa.

Unityn kohdalla MVC-arkkitehtuuri päätettiin toteuttaa kuvan 21 tapaisella ratkaisulla. Tehtiin näkymästä, mallista ja käsittelijästä oma tyhjä Gameobject-olio ja lisättiin niiden alle hierarkisesti niihin kuuluvat palaset. Kuvassa 21 näkyvien view ja model objekteilla ei ole muuta toiminnallisuutta, kuin toimia hierarkian ensimmäisinä kohteina. Kaikki toiminnallisuus on niiden alapuolella olevilla objekteilla.



Kuva 21. Unityn MVC-arkkitehtuurin runko.

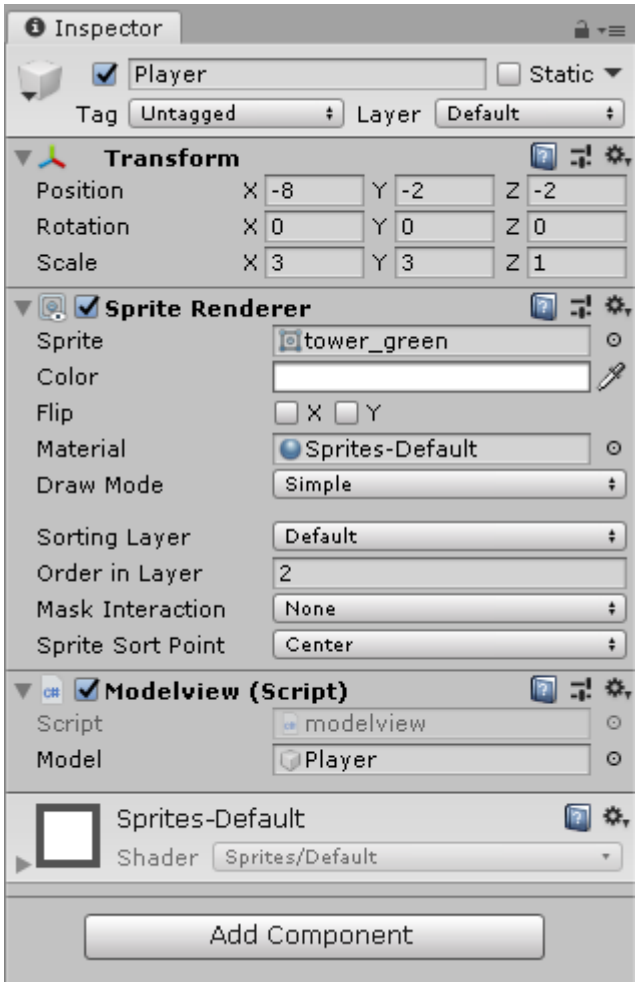
Unityssä on kätevä graafinen käyttöliittymä, jonka kautta pystyy tekemään paljon asioita pelkällä hiirellä. Sen kautta on myös helppo nähdä ja toteuttaa rekursiokoostetta. Esimerkiksi kuvassa 22 on pelaajan tornin malli ja sen kaikki osat.



Kuva 22. Pelaajan tornin malli.

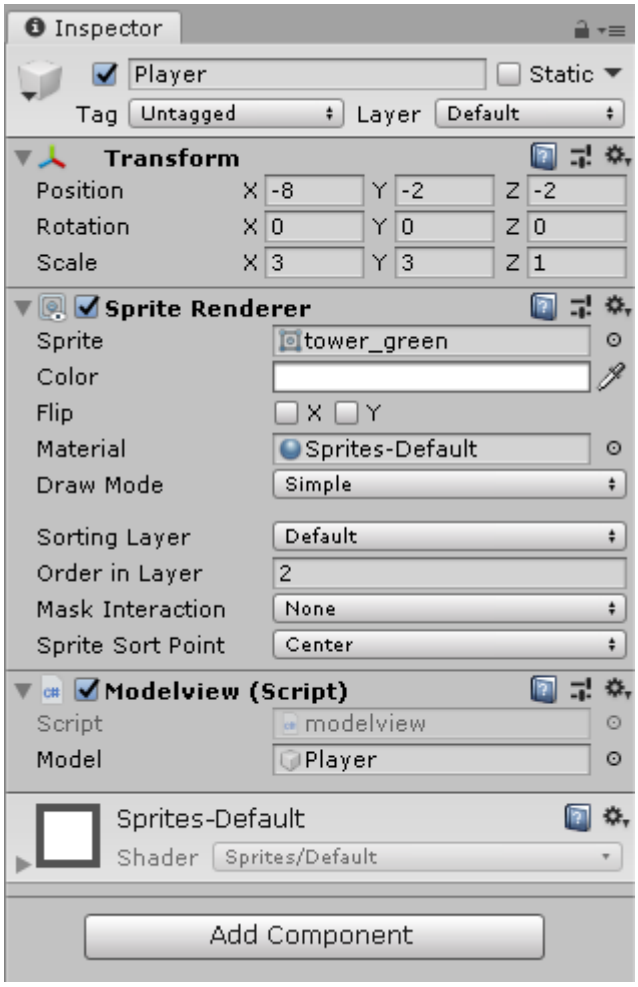
Kuvan 22 mallilla on kaikki sen tarvitsemat palaset. Siltä löytyy sijaintitiedot, törmäyksen testaamisiin tarvitsemat osumalaatikot sekä skriptit.

Kuvassa 23 nähdään pelaajan tornin näkymä. Näkymällä on sijaintitiedot, graafinen ilmeentymä, jonka se piirtää ruudulle sekä skripti.



Kuva 23. Pelaajan tornin näkymä.

Unityssä on hyvänä piirteenä se, että skriptin muuttujia voi päästä muokkaamaan graafisen käyttöliittymän kautta. Kuvan 24 esimerkissä skriptillä modelview muuttujana model ja sen arvona player.



Kuva 24. Pelaajan tornin näkymä.

Tämä on käytännössä tehty niin, että kuvassa 22 näkyvä malli Player on siirretty skriptiin, jolloin skripti on saanut muuttujan arvonsa graafisen käyttöliittymän kautta. Skripti itsessään ei tee muuta, kuin hakee sen muuttujaan annetun olion, eli mallin, sijaintitiedot ja päivittää ne näkymälle, kuten kuvassa 25 nähdään.

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

public class modelview : MonoBehaviour
{
    public GameObject model;
    private Transform pos;

    // Start is called before the first frame update
    void Start()
    {
        pos = model.GetComponent<Transform>();
    }

    // Update is called once per frame
    void Update()
    {
        transform.position = pos.position;
    }
}
```

Kuva 25. Modelview-skripti päivittää näkymän sijainnin samaksi, kuin mallin sijainti.

MVC-arkkitehtuuri on sovellettavissa myös unityn kanssa, mutta siinä menee aikaa enemmän konfigurointiin kuin pygamella tai libgdx:llä. Unityssä ei esimerkkipelin yhteydessä löytynyt hyväksikäytettäviä luokkia kuten libgdx:n kohdalla löytyi, jos ei lasketa mukaan sitä, että unityssä kaikki objektit periytyvät GameObject-luokasta. Kuten libgdx:n kohdalla, myös unityssä on malli sidottu pelimoottoriin vahvasti, kuten myös näkymä ja käsittelijäkin. Myös skriptit ovat sidottu pelimoottoriin oletuksena, sillä ne periytyvät Monobehavior-luokasta.

6 Yhteenveto

MVC-arkkitehtuurin käytöllä voidaan havaita hyötyjä peliohjelmoinnissa. Sen avulla lähdekoodia saadaan jaettua eri toimintojen perusteella osiin ja luotua selkeämpää ja helpommin hallittavampaa lähdekoodia. Varsinkin isoimmissa peliprojekteissa tästä on hyötyä.

MVC-arkkitehtuurin sovittaminen voi olla hankalaa. Joissakin pelimoottoreissa sovittaminen on helpompaa kuin toisissa. Esimerkiksi libgdx:n kohdalla saatiin MVC-arkkitehtuuri sovitettua libgdx:n tarjoamiin luokkiin aika helposti, vaikka libgdx:ää ei ole suunniteltu MVC-arkkitehtuurilla käytettäväksi.

Yksi MVC-arkkitehtuurin ongelmista on se, että abstraktina ratkaisuna sille ei ole yhtä ainoaa oikeaa tapaa toteuttaa. Tämän vuoksi on hyvin tärkeää, että MVC-arkkitehtuuri on suunniteltu alusta alkaen tarkkaan, miten sitä aiotaan käyttää. On hyvä luoda projektin alussa tarkka suunnitelma ja säännöt ja projektin edetessä valvoa, että näistä pidetään kiinni.

Variaatiosta riippuen myös osien yhteys toisiinsa on erilainen. Esimerkkipelin tapauksessa näkymällä oli yhteys malliin mutta joissain variaatioissa on niin, ettei näkymällä ole mitään kosketuspintaa malliin muuta kuin käsittelijän kautta, jolloin käsittelijä päivittää näkymän tiedot mallin muutoksilla. Esimerkkipeli onkin siinä hyvä esimerkki, että vaikka näkymällä olisi mahdollisuus käsitellä mallia, on se projektin alussa määritetty niin, että näkymä ei saa tehdä muuta kuin hakea mallilta tietoja sen nykyisestä tilasta ja datasta. Mikäli tätä sovittua sääntöä noudatetaan, pystytään MVC-arkkitehtuuri pitämään kasassa mutta säännöistä poikkeaminen rikkoo nopeasti koko arkkitehtuurin.

Aiemmissa tutkimuksissa on nostettu esille uudelleenkäytettävyyden yhtenä piirteenä sitä, että mm. malli voidaan siirtää pelimoottorista toiseen. Tässä on se ongelma, että vaikka pygamessa ja libgdx:n tapauksessa näin voitaisiin tehdä, menetettäisiin pelimoottorin tarjoamia hyötyjä kuten libgdx:n natiiviin javaan verrattuna optimoidummat kokoelma-luokat. Mielestäni MVC-arkkitehtuurin tulisi pystyä hyödyntämään kaikilta osin käytettävää pelimoottoria.

Gradun esimerkkipelit olivat hyvin pieniä, joten niistä tehdyt havainnot eivät välttämättä päde aina. Jatkoa ajatellen voisi olla perusteltua keskittyä MVC-arkkitehtuurin sovittamiseen vain yhtä pelimoottoria käyttäen ja silloinkin luoda joku monipuolisempi ja laajempi peli. Myöskin lähdekoodin ja pelin muutokset, ja miten helppo muutoksia on tehdä, voisi olla tarpeellista testata tarkemmin. Toisena mielenkiintoisena tutkimusaiheena voisi olla erilaisten MVC-arkkitehtuurin toteutuksien vertailu keskenään samaa pelimoottoria käyttäen.

Tieteelliset lähteet

Ampatzoglou A, Chatzigeorgiou A. Evaluation of object-oriented design patterns in game development. Information and Software Technology 49.5. 2007.

Anderson E, Engel S, McLoughlin L, Comninos P. *The Case for Research in Game Engine Architecture*. Bournemouth University, UK. 2008.

Anzures-Garcia A, Sánchez-Gálvez L, Hornos M, Paderewski-Rodriguez P. *MVC Design Pattern Based-Development of Groupware.* 4th International Conference in Software Engineering Research and Innovation (CONISOFT). 2016

BinSuBaih A, Maddock S. *G-factor Portability in Game Development Using Game Engines*. Proceedings of the 3rd International Conference on Games Research and Development. 2007.

BinSuBaih A, Maddock S, Romano D. *An Architecture for Portable Serious Games*. Doctoral Symposium hosted at the 20th European Conference on Object-Oriented Programming ECOOP. 2006.

Burbeck S. *Applications Programming in Smalltalk-80™: How to use Model-View-Controller (MVC).* Smalltalk-80 v2 5. 1992.

Caltagirone S, Schlief B, Keys M, Willshire M. *Architecture for a Massively Multiplayer Online Role Playing Game Engine*. Journal of Computing Sciences in Colleges 18.2. 2002.

Kanode C, Haddad H. *Software Engineering Challenges in Game Development*. Sixth International Conference on Information Technology: New Generations. 2009.

Olsson T, Toll D, Wingkvist A. *Evolution and Evaluation of the Model-View-Controller Architecture in Games*. IEEE/ACM 4th International Workshop on Games and Software Engineering. 2015.

Olsson T, Toll D, Wingkvist A. *Evaluation of a Static Architectural Conformance Checking Method in a Line of Computer Games*. Proceedings of the 10th international ACM Sigsoft conference on Quality of software architectures. 2016.

Peker A, Can A. *A Design Goal and Design Pattern Based Approach for Development of Game Engines for Mobile Platforms.* 6th International Conference on Computer Games (CGAMES). 2011.

Qu J., Song Y., Wei Y. *Design Patterns Applied for Game Design Patterns*. 17th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD). 2016.

Reenskaug T. *The Model-View-Controller (MVC) Its Past and Present*. University of Oslo. 2003.

Zavcer G, Mayr S, Petta P. *Design Pattern Canvas: Towards Co-Creation of Unified Serious Game Design Patterns*. 6th International Conference on Games and Virtual Worlds for Serious Applications (VS-GAMES). 2014.

Muut lähteet

Ahex. *MVC Design Pattern for IOS Apps*. 2018. Viitattu 22.4.2019. https://ahex.co/mvc-design-pattern/

Asp.net. *Learn about ASP.NET MVC.* Viitattu 18.11.2018. https://www.asp.net/mvc

Backbone.js. *FAQ-MVC.* Viitattu 18.11.2018. http://backbonejs.org/#FAQ-mvc

Django project. *FAQ.* Viitattu 18.11.2018. https://docs.djangoproject.com/en/2.1/faq/general/#django-appears-to-be-a-mvc-framework-but-you-call-the-controller-the-view-and-the-view-the-template-how-come-you-don-t-use-the-standard-names

Game from scratch. *Gamedev glossary: Library vs framework vs engine.* 2015. Viitattu 18.11.2018. https://www.gamefromscratch.com/post/2015/06/13/GameDev-Glossary-Library-Vs-Framework-Vs-Engine.aspx

Game career guide. *What is a Game Engine?* 2008. Viitattu 29.4.2018. http://www.gamecareerguide.com/features/529/what_is_a_game_.php

Gamma E., Helm R., Johnson R. & Vlissides J. *Design patterns: Elements of Reusable Object-Oriented Software.* Massachusetts: Courier Westworld, 2009, 37. painos.

Hoey P. *Mastering LibGDX Game Development.* Birminghan: Packt Publishing, 2015.

Libgdx. *Libgdx life cycle*. Viitattu 24.4.2019. https://github.com/libgdx/libgdx/wiki/The-life-cycle.

Pygame. *Pygame docs.* Viitattu 29.9.2018. https://www.pygame.org/docs/

Pygame. *Pygame FAQ*. Viitattu 1.5.2019. https://www.pygame.org/wiki/FrequentlyAskedQuestions

Pygame. *Pygame wiki.* Viitattu 29.9.2018. https://www.pygame.org/wiki/about

Ruby on Rails. *Getting started with Ruby on Rails.* Viitattu 18.11.2018. https://guides.rubyonrails.org/getting_started.html

Tutorialsteacher. MVC-arkkitehtuurin rakenne, jossa näkymä tuntee mallin. Viitattu 17.11.2018. http://www.tutorialsteacher.com/mvc/mvc-architecture

Unity. *Unity execution order*. Viitattu 1.5.2019. https://docs.unity3d.com/Manual/ExecutionOrder.html