MOBIILIPELIN TOTEUTTAMINEN UNITY-PELIMOOTTORILLA: CAST INTO HELL

Jari Mäkelä

Opinnäytetyö
Toukokuu 2016
Tietojenkäsittelyn koulutusohjelma
Ohjelmistotuotanto

TIIVISTELMÄ

Tampereen ammattikorkeakoulu
Tietojenkäsittelyn koulutusohjelma
Ohjelmistotuotanto

MÄKELÄ, JARI:
Mobiilipelin toteutus Unity-pelimoottorilla: Cast into Hell

Opinnäytetyö 31 sivua
Toukokuu 2016

Opinnäytetyön tavoitteena oli saada kokemusta mobiilipelien suunnittelemisesta ja teknisestä toteuttamisesta Unity-pelimoottorilla, tarkoituksena puolestaan oli tuottaa toimeksiantajan esittämästä peli-ideasta toimiva prototyyppi. Toimeksiantaja on nuori tamperelainen peli- ja ohjelmistoalan yritys *Rimeforge Entertainment*. Työn vaatimukseksi asetettiin toimiva peliprototyyppi, joka sisältää peli-ideassa esitetyt keskeiset pelimekaniikat ja osan lopulliseen peliin tulevista kentistä.

Opinnäytetyössä esitellään prototyypin suunnitteluprosessia, kuvataan sen keskeisten ominaisuuksien tekniset ratkaisut ja toteutustavat sekä tehdään lopuksi ehdotuksia sisällön lisäämisestä ja pelattavuuden parantamisesta. Työ täytti sille asetetut vaatimukset ja siitä tullaan kehittämään Androidille julkaistava peli.

Asiasanat: unity, mobiilipelit, pelinkehitys, ohjelmointi

ABSTRACT

Tampereen ammattikorkeakoulu
Tampere University of Applied Sciences
Degree Programme in Business Information Systems
Software Development

MÄKELÄ, JARI:
Mobile Game Development with Unity: Cast into Hell

Bachelor's thesis 31 pages
May 2016

The purpose of this thesis was to gain experience in the design and implementation of mobile games using the Unity game engine, with the additional objective of developing a fully functional prototype based on the concept document provided by the client, Rimeforge Entertainment. The project requirements state that the prototype must implement all the features presented in the concept document and support the possibility of further development by the client.

The thesis explores the prototype's design process and the principles concerning mobile game development in general, depicts the core features of the game and provides suggestions for possible further development options.

The project met all the requirements set for it and will be further developed with the aim of eventually publishing the finished product for Android phones. Personally, the project presented an invaluable opportunity to learn of game creation and about Unity as a development platform.

Key words: unity, mobile games, game development, programming

SISÄLLYS

1 JOHDANTO 6
2 UNITY 7
3 PELIKONSEPTI 8
4 MOBIILIPELIN SUUNNITTELU 10
4.1 Käyttöliittymä 10
4.2 Käyttäjän syöte ja kontrollit 10
4.3 Kenttäsuunnittelu 11
4.4 Optimointi 13
4.4.1 Optimoinnista yleisesti 13
4.4.2 Optimointi osana kehitysprosessia 13
4.4.3 Unity Profiler 14
4.4.4 Object pool 15
4.4.5 Reference caching 16
4.4.6 Tyhjien funktioiden poistaminen 17
4.4.7 Find() ja SendMessage() 18
5 TOTEUTUS 19
5.1 Grafiikat 19
5.2 Animaatio 20
5.3 Ohjelmointi 21
5.3.1 Pelihahmo 22
5.3.2 Ympäristö 24
5.3.3 Viholliset 25
6 POHDINTA 29
LÄHTEET 30

ERITYISSANASTO

Platformer	Peli jossa tarkoituksena on pelihahmon hyppyyttäminen tasanteelta toiselle.
Infinite runner	Peli jossa pelihahmo juoksee jatkuvasti eteenpäin.
Scene	Unityn tiedosto, joka sisältää yhden kohtauksen peliobjektit.
Prefab	Varastoitu peliobjekti, josta on mahdollista luoda kopioita.
Collider	Unityn peliobjekteissa törmäyksen havaitseva komponentti.
Coroutine	Funktio jonka toiminta voi tauota määrätyksi ajaksi.

1 JOHDANTO

Mobiilipelit ovat osoittautuneet nykypäivänä suosituksi ja varteenotettavaksi peliteollisuuden osaksi. Näitä pelejä pelaa yhä suurempi osa väestöstä ja laitteiston kehittymisen myötä mobiilipeleistä on tullut hyvin näyttäviä ja tunnettuja. Tämä opinnäytetyö käsittelee *Cast into Hell*-mobiilipelin suunnitteluun ja toteutukseen liittyviä haasteita ja keskeisten ominaisuuksien teknisiä ratkaisuja. Näitä ratkaisuja voidaan myös monessa tapauksessa hyödyntää yleisesti tulevissa peliprojekteissa.

Opinnäytetyön tavoitteena on saada kokemusta mobiilipelin suunnittelemisesta ja toteuttamisesta, jota voidaan hyödyntää tulevissa projekteissa. Työn yhteydessä luodaan toimeksiantajan käyttöön prototyyppi jota on mahdollista kehittää jatkossa yrityksen omaan käyttöön.

Työn toimeksiantaja on nuori tamperelainen peli- ja ohjelmistoalan yritys Rimeforge Entertainment Osk. Toimeksiantajan vaatimuksena oli toimivan prototyypin tuottaminen, joka sisältää keskeiset pelimekaniikat ja osan pelin kentistä.

Työssä käsitellään ensin projektin lähtökohtia; kehitysympäristöksi valittua Unity-ohjelmistoa ja pelin konseptia. Suunnitteluosuudessa käsitellään mobiilipelien suunnittelussa yleisesti huomioitavia asioita, prototyypin kenttäsuunnittelua ja mobiilipeleille erityisen tärkeää optimointia. Toteutusosuudessa keskitytään käytettyihin tekniikkoihin, toteutuksen kannalta oleellisiin Unityn ominaisuuksin ja pelin ohjelmointiin. Työn onnistumista ja prototyypin tulevaisuutta käsitellään viimeisessä luvussa.

Työn luonteesta johtuen aiheesta saatavilla olevien kirjallisten lähteiden määrä on rajattu. Saatavilla olevien kirjallisten lähteiden lisäksi työssä on hyödynnetty verkkolähteitä, kuten Unityn verkkosivujen dokumentaatiota ja työelämässä kertynyttä käytännön tietoa.

2 UNITY

Unity on Unity Technologies-yrityksen kehittämä pelimoottori ja monialustainen pelinkehitysympäristö. Pelimoottorina Unity on monipuolinen ja soveltuu hyvinkin erilaisten pelityyppien toteutukseen. Unityn visuaalinen editori on kilpailijoihinsa verrattuna helppokäyttöinen ja nopea oppia. Itse editoria on myös mahdollista muokata omilla skripteillä. Toisin kuin Unity monet muut visuaalisen editorin sisältävät kehitysympäristöt tarjoavat vain rajallisen tuen skripteille, mikä rajoittaa niiden ohjelmointimahdollisuuksia (Hocking 2015).

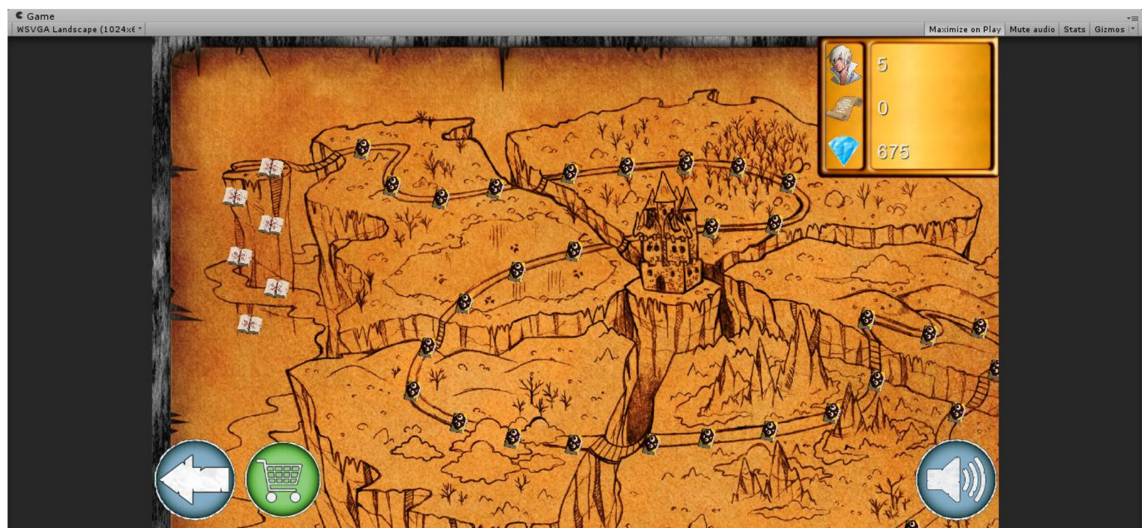
Ensimmäinen versio Unitystä julkaistiin Macille vuonna 2005 ja Windows-tuki lisättiin muutamaa kuukautta myöhemmin. Unityn myöhemmät versiot ovat lisänneet lukuisia uusia julkaisualustoja, kuten Web (2006), iPhone (2008) ja Android (2010). (Hocking 2015.)

Aloittelevan kehittäjän kannalta yksi houkuttelevimmista syistä on Unityn saatavuus ja ilmaisuus. Unity tukee lukuisia eri julkaisualustoja mukaan lukien Windows, Mac, Linux, iOS, Android, Windows Phone 8, Blackberry 10 ja Web. Lisäksi Unitylle on saatavissa maksullisia lisenssejä konsolialustoille kuten Xbox360, Xbox One, PS3, PS4 ja Wii U. (Unity Technologies 2016h.)

3 PELIKONSEPTI

Cast into Hell on nopeatempoinen ja haastava sivunäkymästä kuvattu 2D-toimintapeli joka sekoittaa elementtejä platformer-tyylisistä peleistä (esim. *Super Mario*) ja infinite runner-peleistä (esim. *Temple Run*). Pelin tapahtumapaikkana toimii fiktiivinen helvetti joka perustuu löyhästi italialaisen Dante Alighierin teoksessaan *Inferno* kuvaamaan helvetin yhdeksänteen tasoon.

Pelimaailma on jaoteltu vaihtelevan mittaisiin kenttiin ja sen rakennetta kuvataan karttaruudulla johon pelaaja voi halutessaan palata näiden kenttien välissä. Kentät avautuvat ennalta määrätyssä järjestyksessä sitä mukaan kun pelaaja suorittaa niitä ja karttaruudulta pelaaja voi valita pelattavaksi minkä tahansa jo avatun kentän.

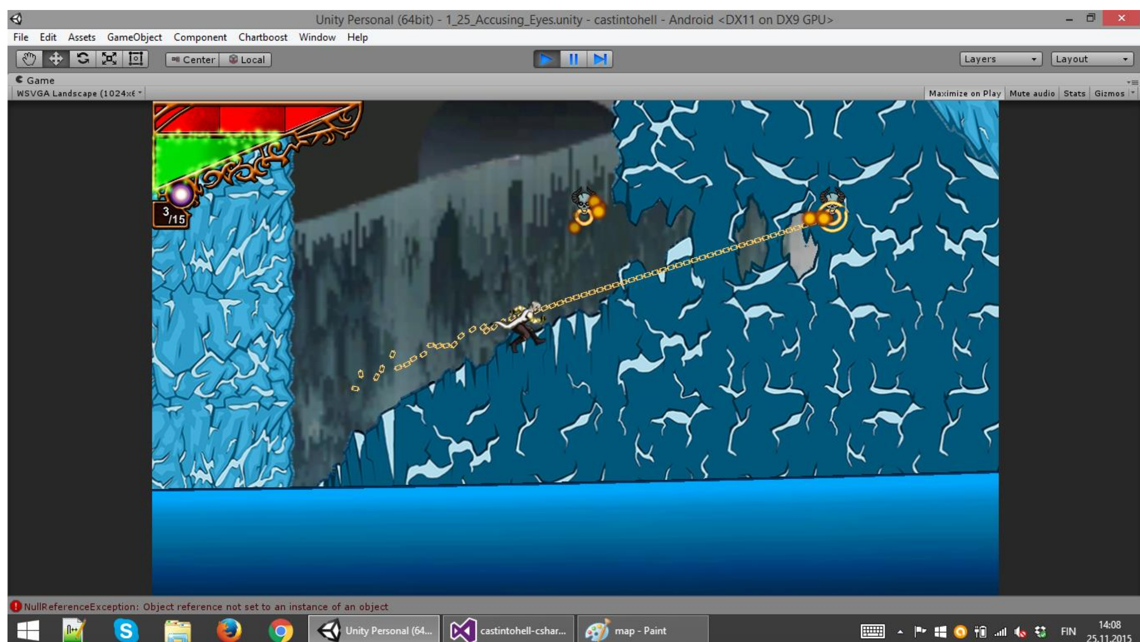


Kuva 1. Karttaruutu

Pelaajan tavoite on ohjata pelihahmo esteratamaisten kenttien loppuun kuolematta. Pelihahmo voi kuolla ottaessaan liian monta osumaa kenttiin sijoitetuista vaarallisista elementeistä tai vihollisista. Näitä vaara-elementtejä ovat esimerkiksi kentän pohjan peittävä hyinen vesi, nopeasti liikkuvat kivenjärkäleet tai terävät piikit. Pelissä on lukuisia erilaisia vihollistyyppejä jotka käyttäytyvät eri tavalla, esimerkiksi paikallaan seisovat jäähirviöt jotka yrittävät heittää pelaajaa kivillä tai nopeat helvetinkoirat jotka hyökkäävät pelaajaa kohti tämän tullessa liian lähelle.

Suoriutuakseen kentistä tulee pelaajan oppia miten eri viholliset käyttäytyvät ja miten ne voi parhaiten välttää tai tuhota. Jokaiseen kenttään on myös sijoitettu vaihteleva määrä löydettäviä esineitä (sieluja), joiden kerääminen tuo peliin ylimääräisen mutta vapaaehtoisen lisähaasteen. Pelaajan onnistuessa keräämään kaikki sielut pelin jokaisesta kentästä pääsee hän taistelemaan ylimääräistä pomohahmoa vastaan.

Infinite runner-pelien tapaan pelihahmo juoksee kentän alettua jatkuvasti eteenpäin, eikä voi pysähtyä. Pelaajalla on tästä huolimatta lukuisia keinoja vaikuttaa pelihahmon liikkumiseen; ruudun koskettaminen saa pelihahmon hyppäämään, sormen liu'uttaminen ruudulla sivusuunnassa saa hahmon tekemään nopean syöksyn, liu'uttaminen pystysuunnassa hidastaa aikaa ja koskemalla yhtä jokaiseen kenttään sijoitetuista "koukuista" pelihahmo vetää itsensä ketjun avulla kosketetun koukun luokse suorassa linjassa. Näiden toimintojen yhdisteleminen ja käyttö eri tilanteissa on olennainen osa pelihahmon hallitsemista ja erittäin keskeinen pelimekaniikka.



Kuva 2. Pelihahmon liikkuminen

4 MOBIILIPELIN SUUNNITTELU

4.1 Käyttöliittymä

Käyttöliittymän toteuttaminen mobiilipeliin tuo mukanaan monia haasteita. Mobiililaitteiden ruudut ovat pääsääntöisesti pieniä, joten pelaajalle tarpeellisen tiedon esittäminen ytimekkäästi ja selkeästi on olennaista. Käyttöliittymän tulee myös skaalautua käytetyn laitteen mukaan, sillä esimerkiksi pelkästään Android-laitteissa käytetään niin monia eri resoluutioita että näiden huomioiminen suunnittelussa yksitellen ei ole kannattavaa. Suunnittelussa tulee myös ottaa huomioon tullaanko laitetta pelin aikana käyttämään sivu- vai pystyasennossa, sillä tämä vaikuttaa oleellisesti käytettävissä olevan tilan määrään.

4.2 Käyttäjän syöte ja kontrollit

Toimivat kontrollit ovat olennainen osa mitä tahansa pelikokemusta ja pelin toteuttaminen mobiililaitteille tuo mukanaan haasteita jotka täytyy ottaa huomioon jo suunnitteluvaiheessa. Armen Ghazarian (2014) listaa kolme suurinta haastetta mobiilipelien käytettävyydessä:

1. Mobiililaitteiden ruutukoko on erittäin pieni verrattuna konsoli- ja PC-peleihin.
2. Ergonomisen käyttöliittymän suunnitteleminen vaikeutuu koska pelaaja pitelee sekä pelinohjainta että ruutua käsissään.
3. Kosketusnäytön painaminen ei anna tuntoaistilla havaittavaa palautetta.

Yksi mobiilipelien käyttämä ratkaisu perinteisen ohjausvälineen puuttumiseen on sisällyttää pelaamiseen tarvittavat painikkeet itse pelin käyttöliittymään. Tämän ratkaisun ongelma on että pelaaja ei fyysisen painikkeen tapaan saa painalluksesta riittävää haptista palautetta joten ohjaaminen on tunnotonta. Toinen yleisesti käytetty tapa on ohjaamisen perustuminen pelaajan eleisiin ja sormen liikkumiseen ruudulla. Hyvä esimerkki tästä on tunnettu *Angry Birds*-pelisarja, jossa lintujen ampuminen ja tähtääminen tapahtuvat vetämällä ruudulla näkyvää ritsaa sormella.

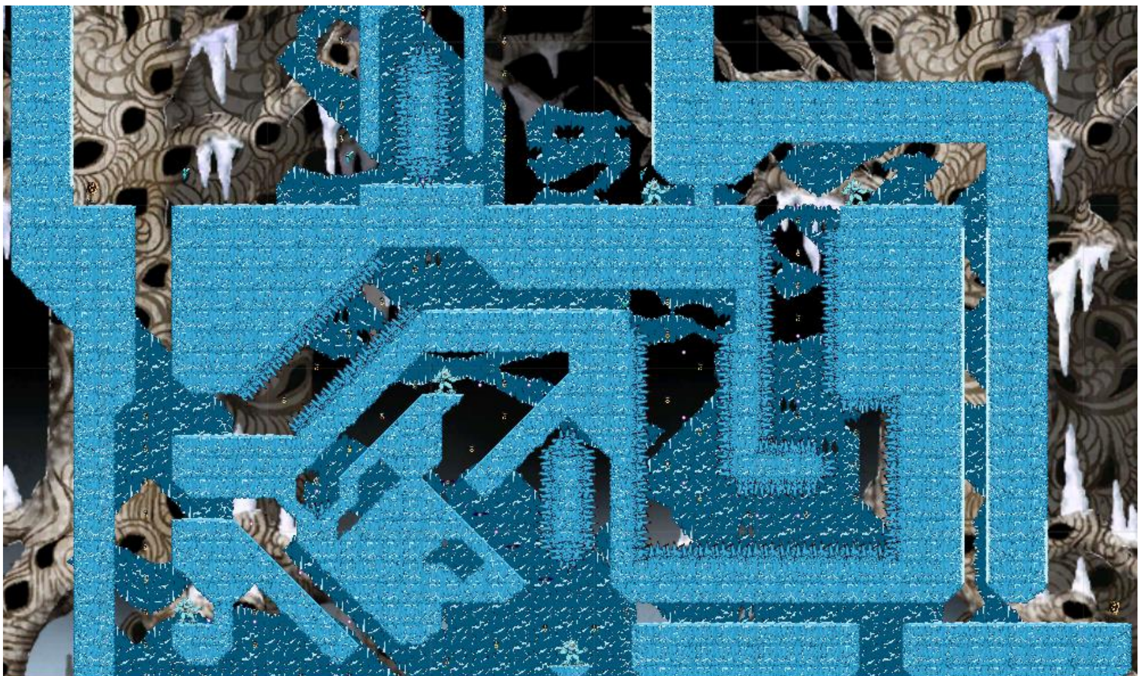
4.3 Kenttäsuunnittelu

Kehitystyön alusta asti tavoitteena oli tehdä *Cast into Hell*-pelistä haastava ja kenttäsuunnittelun täytyi tukea tätä tavoitetta. Tästä huolimatta on tärkeää, että haasteen määrä kentissä ei nouse liian jyrkästi tai ole alussa liian korkea. Liian vaikeat alkukentät voivat estää pelaajaa pääsemästä kunnolla peliin sisälle ja alussa haaste tuleekin yksinkertaisesti pelimekaniikkojen ja ohjauksen opettelemisesta, ei vihollisista tai ympäristöistä. Tätä tuetaan opettamalla pelaajalle yksi uusi taito tai kyky kerrallaan; ensimmäisessä kentässä keskitytään ainoastaan kaikkein oleellisimpiin ohjaustekniikkoihin eli hyppäämiseen ja pelihahmon liikuttamiseen ketjujen ja koukkujen avulla. Pelaajalle annetaan aikaa tutustua näihin toimintoihin omaan tahtiinsa, sillä ensimmäisessä kentässä ei ole vaaratekijöitä jotka voisivat aiheuttaa pelihahmon kuoleman.

Kuva 3. Pelin ensimmäinen kenttä

Vaaroja tuodaan peliin hiljalleen; toisessa kentässä pelaajan on mahdollista pudota hyiseen veteen, mutta siinä ei ole aktiivisia vaaroja kuten vihollisia jotka hyökkäisivät pelaajan kimppuun. Myös passiivisten vaarojen, kuten pudotuksien, haitallisuutta on osin lievennetty ja pelihahmon kuoleman sijaan putoaminen saattaa viedä kentän alempaan osaan. Pelihahmon kaikkein vaativinta kykyä eli tulipallojen heittämistä ajanhidastuksen sisällä pelaaja ei edes pysty käyttämään ennen kuin on päihittänyt pelin ensimmäisen pomovastuksen.

Kaikissa kentissä on myös mukana taitaville pelaajille tarkoitettu vapaaehtoinen haaste kerättävien sielujen muodossa. Nämä keräilyesineet on usein sijoitettu vaikeisiin ja vaarallisiin paikkoihin, mutta niiden kerääminen ei ole edellytys kentän läpäisyyn. Pelaajan mielenkiinnon ylläpitämiseksi on tärkeää, että kentät tuntuvat uniikeilta eivätkä ole toistavia. Pelin edetessä kenttien tuleekin esitellä haasteen lisäksi jatkuvasti uusia pelimekaniikkoja, ympäristöjä ja hahmoja. Pelimaailman läpi kulkiessaan pelaajaa huomaa että kenttien taustat vaihtelevat ja pelin loppua kohden kentät käyvät myös huomattavasti alkua monimutkaisemmiksi ja pidemmiksi.



Kuva 4. Pelin loppupään kenttä

Jotta kenttien rakenteesta saatiin koherentti ja suunnitelmallinen piirrettiin jokaisesta kentästä suunnitelma paperille ennen kentän rakentamista editorissa. Tämä lähestymistapa helpotti myös palautteen antamista mahdollisten ongelmakohtien havaitsemista varhaisessa vaiheessa.

4.4 Optimointi

4.4.1 Optimoinnista yleisesti

Pelien optimointi on laaja ja usein hyvinkin tapauskohtainen aihe, sillä monesti optimoinnin vaatimat toimenpiteet riippuvat suoraan optimoitavan pelin ominaisuuksista, pelimoottorista ja alustasta. Mobiilipeleistä puhuttaessa pelin optimointi on äärimmäisen tärkeä osa kehitysprosessia, sillä huono suorituskyky vaikuttaa käyttäjän pelinautintoon. Huono suorituskyky kaventaa myös suoraan mahdollista asiakaskuntaa, sillä esimerkiksi hieman vanhempien puhelinmallien omistajat eivät välttämättä pysty pelaamaan huonosti optimoitua peliä ollenkaan. Harvat uusivat puhelimensa aina uuden version tullessa myyntiin, joten vanhojen puhelinmallien käyttäjät ovat suuri kohderyhmä jonka huomioiminen on tärkeää.

Puhelimien laskentatehossa voi esiintyä valtavia eroja jo pelkästään yhden ”sukupolven” välillä; uuden puhelinmallin grafiikkaprosessori (GPU) voi olla teholtaan jopa viisinkertainen edeltäjään verrattuna (Unity Technologies 2016e).

4.4.2 Optimointi osana kehitysprosessia

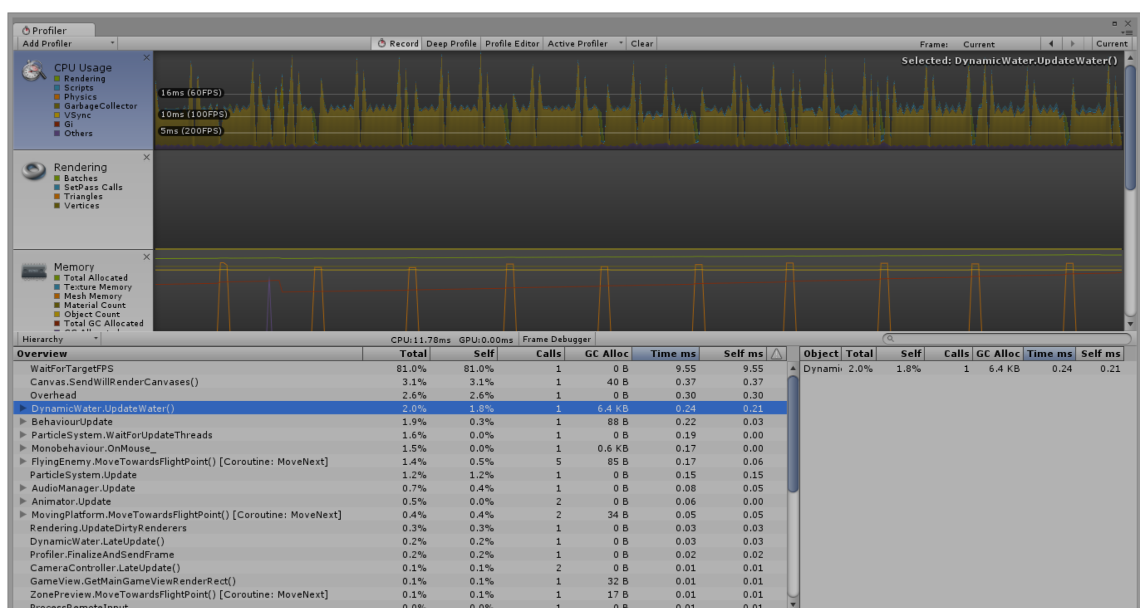
Johtuen mobiililaitteiden rajoituksista verrattuna ”perinteisiin” pelialustoihin kuten pelikonsoleihin ja pöytätietokoneisiin ei optimointia tulisi jättää kehitysprosessin loppupäähän, vaan se on tärkeä ottaa huomioon aina pelin suunnitteluvaiheesta lähtien. Näin ollen optimointi on tavoite jota kohti koko kehitystiimin tulisi työskennellä; vaikka se monesti mielletään etenkin ohjelmoijien vastuualueeksi voivat esimerkiksi graafikot omalta osaltaan vaikuttaa suuresti pelin suorituskykyyn. Ohjelmoijien tavoin tulisi heidän tuntea laitealustan asettamat vaatimukset ja rajoitteet.

Pelin suorituskykyä ja toteutettujen ominaisuuksien vaikutusta siihen on hyvä valvoa pelinkehityksen alkuvaiheista asti. Näin voidaan helpommin havaita minkä verran eri ominaisuudet vaikuttavat suorituskykyyn ja minkä ominaisuuksien optimoinnista on pelille eniten hyötyä.

Optimoinnin kannalta on tärkeää tietää missä suorituskyvyn pullonkaulat ovat ja keskittyä näiden ratkomiseen. Esimerkiksi pelin ollessa raskas puhelimen suorittimelle ei varjostuksia laskevien skriptien optimointi paranna ruudunpäivitysnopeutta, mutta kuormituksen kohdistuessa puhelimen grafiikkaprosessorille ei pelin fysiikkamallinnuksen parantaminen auta tilannetta (Unity Technologies 2016e).

4.4.3 Unity Profiler

Käytettäessä Unityä pelinkehitysalustana yksi parhaista optimoinnin apuvälineistä on Unityn sisäänrakennettu profilointityökalu (Unity Profiler), jonka kautta on mahdollista selvittää kuinka paljon aikaa pelimoottori käyttää eri osa-alueiden suorittamiseen. Näin pystytään tarkastelemaan esimerkiksi pelilogiikan, animaation ja renderoinnin viemän ajan välistä prosenttijakaumaa. Profilointityökalun ollessa päällä testaamisen aikana on mahdollista tarkastella pelin suorituskyvyn tasoa aikajanalla, mikä auttaa testaajaa havaitsemaan peliä erityisesti hidastavia tilanteita (Unity Technologies 2016f). Tämän jälkeen voidaan havaittuja ongelmakohtia selvittää tarkemmin esimerkiksi poistamalla pelin toiminnollisuuksia väliaikaisesti; jos vihollisten tekoälyn sammuttaminen kaksinkertaistaa ruudunpäivitysnopeuden hyötyisi se todennäköisesti optimoinnista (Unity Technologies 2016g).



Kuva 5. Unity Profiler

Profilointityökalun avulla on mahdollista selvittää mille pelin osa-alueelle optimointi tulisi keskittää sekä vertailla suorituskykyä ennen mahdollisia muutoksia ja niiden jälkeen. Koodimuutosten vaikutus pelin toimintaan tulisi aina tarkistaa profilointityökalulta, jotta ennalta-arvaamattomat vaikutukset eri osa-alueisiin voidaan havaita ja ratkaista.

4.4.4 Object pool

Yksi olennainen tekniikka koodin optimoimiseksi etenkin pelien kannalta on niin kutsuttu objektivaranto (object pool). Tällä tekniikalla voidaan parantaa pelin suorituskykyä huomattavasti kun pelissä on lukuisia samankaltaisia objekteja joiden toistuva luominen tai tuhoaminen rasittaa pelimoottoria, esimerkiksi hahmon ampumat ammukset. Objektivarannon toimintaperiaate on, että kun ohjelma tarvitsee uuden objektin yrittää varanto ensin käyttää hyödyksi jo olemassa olevaa objektia joka on luotu aikaisemmin ja käytön jälkeen palautettu varantoon. Uusi objekti luodaan, jos tällaista objektia ei löydy, mikä vähentää tarvetta uusien objektien jatkuvaan luomiseen huomattavasti. (Microsoft Developer Network 2016.)

```
function Update () {
    if(Input.GetButtonDown("Fire1")) {
        var instance : ProjectileWithInstantiate =
            Instantiate(prefab, transform.position, transform.rotation);
        instance.velocity = transform.forward * power;
    }
}
```

Kuva 6. Objektin luominen (Unity Technologies 2016e)

```
function Update () {
    if(Input.GetButtonDown("Fire1")) {
        var instance : ProjectileWithObjectPooling = GetNextAvailiableInstance();
        if(instance != null) {
            instance.Initialize(transform, power);
        }
    }
}
```

Kuva 7. Objektin hakeminen varannosta (Unity Technologies 2016e)

Ohjelmoijan Unityssä kirjoittamat skriptit (C#, UnityScript tai Boo) hyödyntävät automaattista muistinhallintaa, kuten lähes kaikki modernit skriptikielet. Tämä eroaa ”mata-

lan tason” ohjelmointikielistä kuten C tai C++, joissa ohjelmoija on itse vastuussa muistinhallinnasta. Automaattista muistinhallintaa hyödyntävissä kielissä ”roskankeruuksi” (garbage collection) kutsuttu automaattinen prosessi vapauttaa muistia kun objektit käyvät ohjelman kannalta tarpeettomiksi. Objektien jatkuva luominen ja tuhoaminen aiheuttavat roskankeruuprosessille paljon työtä, mikä puolestaan saattaa hidastaa ohjelman toimintaa. (Unity Technologies 2016g.)

Objektivarannon käyttämisen lisäksi tätä ongelmaa voidaan yrittää välttää esimerkiksi roskankeruun kutsumisella tilanteessa, jossa ohjelmalla ei ole muuta raskasta suoritettavaa, tai kutsumalla roskankeruuta usein jotta käyttämätöntä muistia ei kerry käsiteltäväksi paljoa kerralla. (Unity Technologies 2016g.)

4.4.5 Reference caching

Yleinen virhe Unity-skripteissä on GetComponent-kutsun liikakäyttö. Esimerkiksi seuraava koodi hakee aina aktivoituessaan viisi eri komponenttireferenssiä:

```
void TakeDamage() {
  if (GetComponent<HealthComponent>().health < 0) {
    GetComponent<Rigidbody>().enabled = false;
    GetComponent<Collider>().enabled = false;
    GetComponent<AIControllerComponent>().enabled = false;
    GetComponent<Animator>().SetTrigger("death");
  }
}
```

Kuva 8. Referenssien haku (Dickinson 2015)

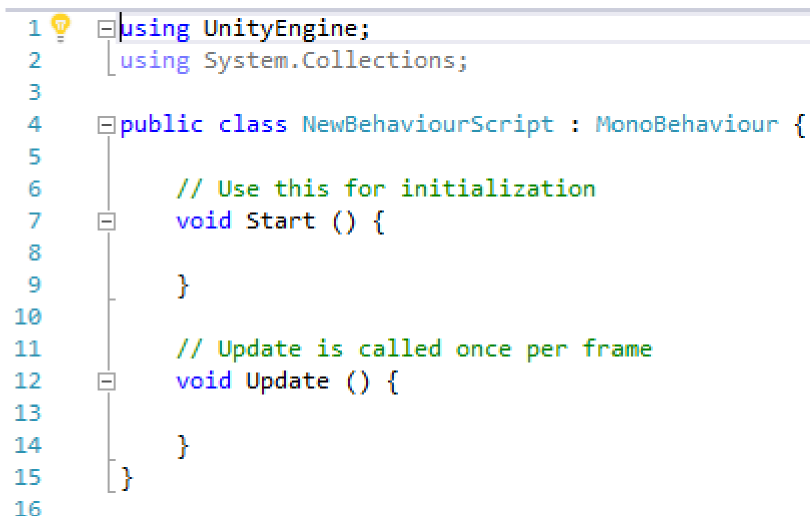
Tämän kaltainen koodi saattaa paljon käytettynä aiheuttaa pelimoottorille runsaasti ylimääräistä ajonaikaista työtä, jos sitä käytetään jatkuvasti kutsuttavien funktioiden yhteydessä. Ellei sovelluksen käytössä olevan muistin määrä ole äärimmäisen rajoitettu, on parempi hakea komponenttireferenssit vain kerran ja säilyttää ne kunnes niitä tarvitaan. (Dickinson 2015.)

```
private HealthComponent _healthComponent;
private Rigidbody _rigidbody;
private Collider _collider;
private AIControllerComponent _aiController;
private Animator _animator;

void Awake() {
  _healthComponent = GetComponent<HealthComponent>();
  _rigidbody = GetComponent<Rigidbody>();
  _collider = GetComponent<Collider>();
  _aiController = GetComponent<AIControllerComponent>();
  _animator = GetComponent<Animator>();
}

void TakeDamage() {
  if (_healthComponent.health < 0) {
    _rigidbody.detectCollisions = false;
    _collider.enabled = false;
    _aiController.enabled = false;
    _animator.SetTrigger("death");
  }
}
```

Kuva 9. Referenssien säilyttäminen (Dickinson 2015)

4.4.6 Tyhjien funktioiden poistaminen

Uuden skriptitiedoston luomisen yhteydessä Unity generoi automaattisesti kaksi tyhjää funktiota:



Kuva 10. Automaattisesti luodut funktiot

Start-funktiota kutsutaan ainoastaan peliobjektin luomisen yhteydessä, joten sen jättäminen aiheuttaa harvoin suurta kuormitusta. Update-funktiota sen sijaan kutsutaan jokaisen

ruudunpäivityksen yhteydessä. Laitteen suorittimelle aiheutuu turhaa rasitusta, jos käytössä on suuri määrä peliobjekteja, joista potentiaalisesti jokaisella on komponentteja joiden skriptit sisältävät tyhjiä funktioita. (Dickinson 2015.) Näiden poistaminen voi parantaa pelin suorituskykyä ilman haittavaikutuksia. Automaattisesti luotujen Start- ja Update-funktioiden lisäksi tämä pätee tyhjiin funktioihin yleisesti.

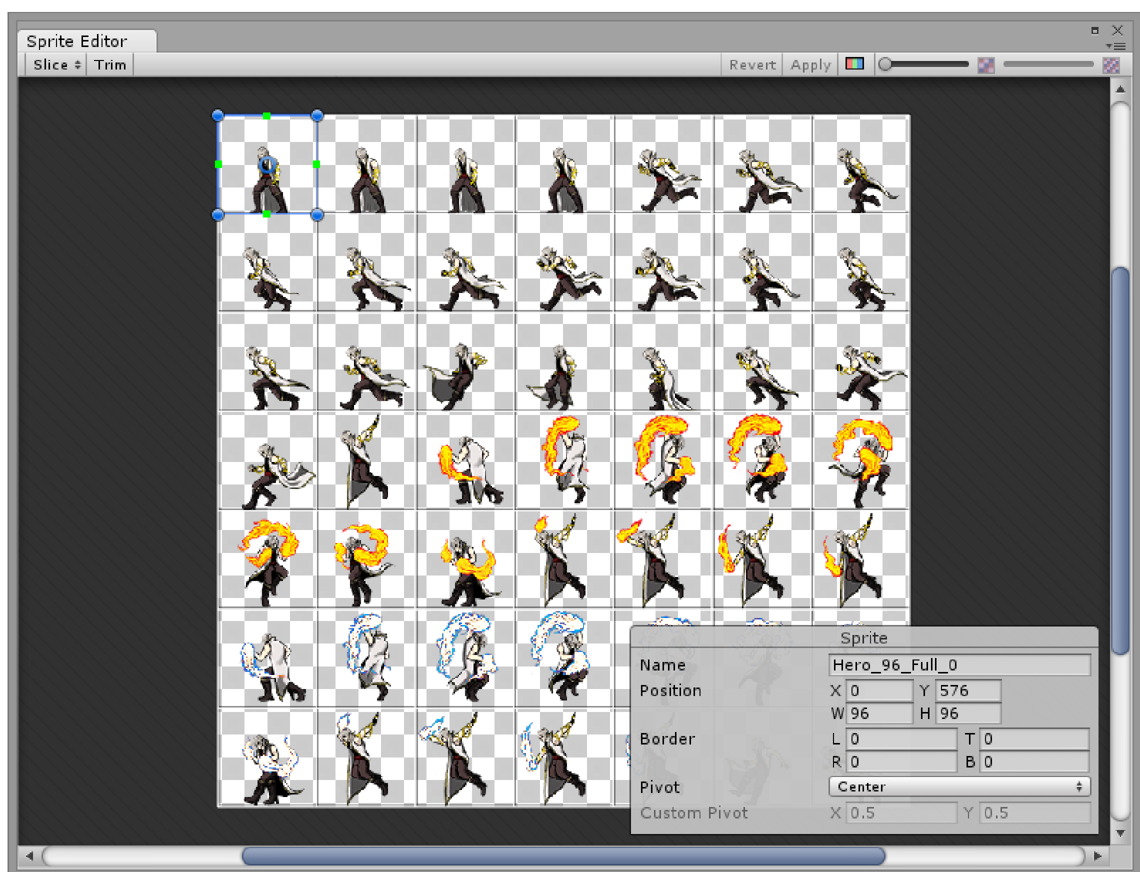
4.4.7 Find() ja SendMessage()

SendMessage- ja GameObject.Find-tyyppiset funktiot ovat suhteellisen raskaita, mistä johtuen niiden käyttöä tulisi mahdollisuuksien mukaan välttää. SendMessage on noin 2000 kertaa hitaampi kuin tavallinen funktiokutsu. Find puolestaan on sitä hitaampi mitä enemmän scenessä on peliobjekteja, sillä se käy ne kaikki läpi etsinnän yhteydessä. Find-funktion käyttö voi olla järkevää scenen alustuksen yhteydessä, esimerkiksi Awake- tai Start-funktioissa, mutta ajonaikaista käyttöä tulisi välttää. (Dickinson 2015.)

5 TOTEUTUS

5.1 Grafiikat

Cast into Hell-pelissä grafiikat muodostuvat suurimmaksi osaksi spriteistä, jotka ovat Unityssä 2D-elementtien esittämiseen käytettyjä bitmap-kuvia (Texture2D), joita peliobjektien SpriteRenderer-komponentit käyttävät grafiikan näyttämiseen (Unity Technologies 2016a). Useimmiten yksi kuvatiedosto sisältää useamman spriten, esimerkiksi kaikki yhden hahmon animaatio-spritet, joka sitten leikataan Unityn Sprite Editor-työkalulla.



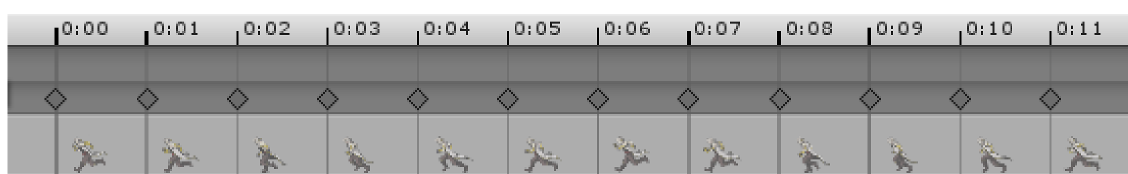
Kuva 11. Sprite Editor

SpriteRenderer-komponenttiin voi vaikuttaa myös skripteistä käsin. Näin voidaan esimerkiksi pelihahmon kääntyessä ympäri kääntää myös sprite, mikä ei vaikuta peliobjektin lapsiobjekteihin tai törmäyksen tunnistaviin collider-komponentteihin toisin kuin hahmon kääntäminen peliobjektin mittasuhteisiin vaikuttamalla. SpriteRendererin avulla säädetään myös spriten sorting layer, mikä määrittää sen mitkä spritet peittyvät niiden

osuessa päällekkäin. Tällaisessa tilanteessa alemman sorting layerin omaavat spritet peittyvät. (Unity Technologies 2016b.)

5.2 Animaatio

Animaation toteuttamisessa Unityssä on kaksi tärkeää osaa: animation clip ja animator controller. Animation clip on pienin Unityssä käytettävä animaation rakennuspalikka; ne edustavat yksittäisiä liikesarjoja kuten juoksemista, hyppäämistä tai ryömimistä ja animaation lopputulos saadaan aikaan niitä yhdistelemällä (Unity Technologies 2016c).



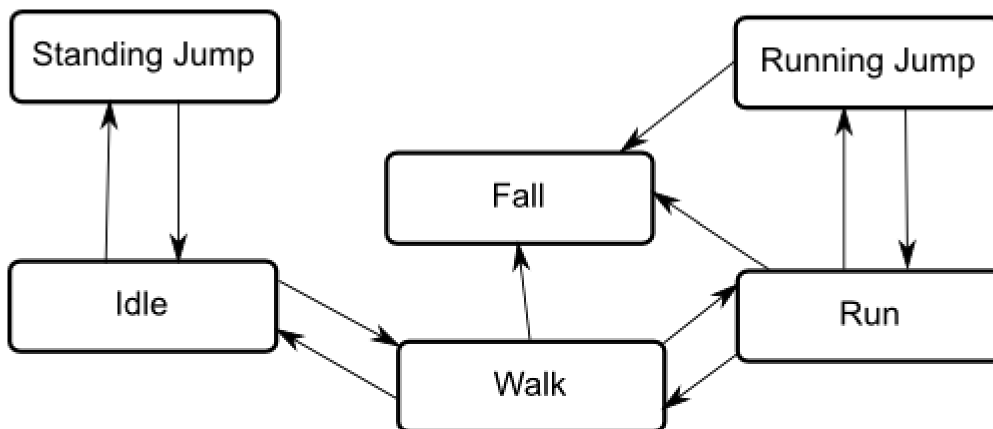
Kuva 12. Animation clip

Animator controller puolestaan mahdollistaa hahmon tai objektin animaatioiden hallinnan ja järjestämisen. Normaalisti hahmolla on lukuisia animaatioita joiden välillä vaihdellaan pelitilanteen ja hahmon toimien muuttuessa tai käyttäjän syötteen yhteydessä; hahmo voi joutua esimerkiksi siirtymään juoksuanimaatiosta hyppyanimaatioon käyttäjän painaessa näppäintä. Vaikka hahmolla tai objektilla olisi vain yksi animaatio, tarvitaan sen käyttämiseksi Unityssä silti animator controller.

Nimensä mukaisesti animator controller hallitsee hahmon animaatiotilaa ja animaatioiden välisiä siirtymiä. Animaatioiden välisiä riippuvuuksia, siirtymiä ja suhteita kuvaa Unityssä ”state machine” jota voidaan ajatella eräänlaisena animaatioiden vuokaaviona. Tämän periaate on, että hahmo on aina yhdessä tilassa (state) joka kuvaa hahmon senhetkistä toimintaa, kuten seisomista, juoksemista tai hyppäämistä.

Yleensä hahmo ei voi siirtyä vapaasti eri tilojen välillä, vaan tila jossa hahmo sillä hetkellä on myös rajoittaa mahdollisia tiloja joihin hahmo voi siirtyä. Hahmo ei voi esimerkiksi siirtyä pitkään juoksuhyppyyn, jos se sillä hetkellä seisoo paikoillaan. Unityssä näitä siirtymämahdollisuuksia kutsutaan nimellä state transition. Unityn state machine muodostuu toisin sanoen näistä tiloista, niiden välisistä siirtymistä ja muuttujista jotka hallit-

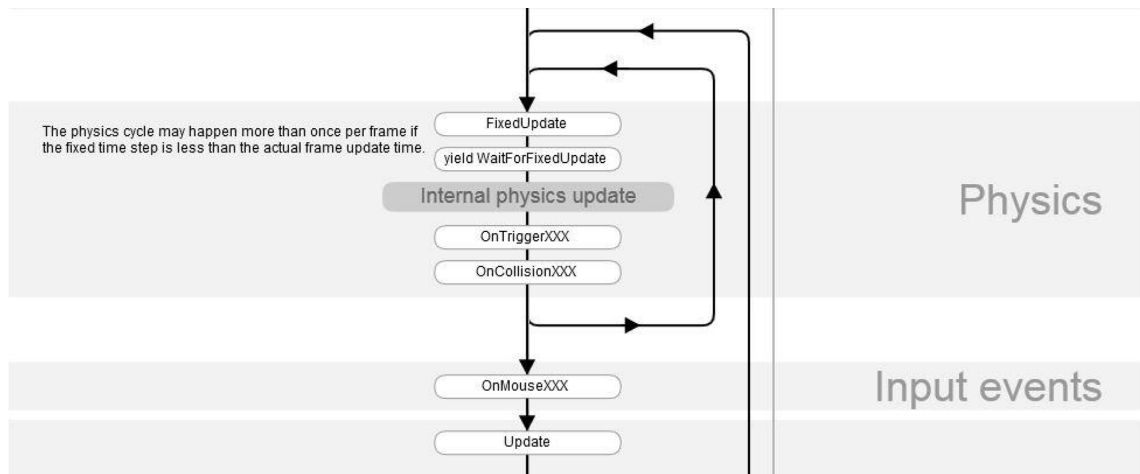
sevat siirtymiä. Animaation kannalta state machine on ketterä työkalu, sillä sen muokkaaminen ja hyödyntäminen eivät vaadi suuria ohjelmointimääriä. (Unity Technologies 2016d.)



Kuva 13. Animaation vuokaavio (Unity Technologies 2016d)

5.3 Ohjelmointi

Funktiot Update ja FixedUpdate lukeutuvat Unityssä skriptien tärkeimpiin funktioihin. Update-funktiota kutsutaan jokaisen ruudunpäivityksen (frame) yhteydessä ja sitä käytetään yleisesti pelilogiikan suorittamiseen (Unity Technologies 2015a). Updateen sijoitetaan usein koodi, joka liittyy yksinkertaisiin ajastimiin, pelifysiikasta riippumattomien objektien liikuttamiseen ja pelaajan syötteen vastaanottamiseen. FixedUpdate-funktiota puolestaan kutsutaan tasaisin väliajoin, yleensä useammin kuin Updatea ja esimerkiksi rigidbody-objektien liikuttamiseen liittyvä koodi tulisi sijoittaa tämän funktion sisään. (Unity Technologies 2015b.) Pelin kuvataajuuden ollessa korkea FixedUpdatea ei välttämättä kutsuta framejen välissä ollenkaan, mutta kuvataajuuden ollessa matala sitä saatetaan kutsua useasti yhtä framea kohden. Kaikki pelin sisäiseen fysiikanmallinnukseen liittyvät laskelmat tapahtuvat heti FixedUpdaten jälkeen. (Unity Technologies 2015.)



Kuva 14. Funktioiden suoritusjärjestys (Unity Technologies 2015)

5.3.1 Pelihahmo

Pelihahmon Update-funktiossa otetaan vastaan hahmon liikuttamiseen liittyvä syöte. Funktio tunnistaa kaksi erilaista kosketustyyppiä: lyhyt kosketus (tap) ja sormen liu'uttaminen ruudulla (swipe). Liu'utuksesta havaitaan myös neljä mahdollista suuntaa: ylös, alas, oikealle ja vasemmalle.

Funktion havaitessa lyhyen kosketuksen tarkistetaan osuiko kosketus pelialueella sijaitsevaan koukkuun. Kosketuksen osuessa koukkuun lasketaan onko pelihahmo etäisyys koukusta hahmon ketjun kantaman sisällä ja tarkistetaan onko näiden välillä esteitä. Reitin ollessa esteetön tuodaan tälle reitille objekteja jotka kuvaavat ketjun lenkkejä ja pelihahmo lähtee siirtymään. Ketju-objektit tuodaan myös esteen, kuten esimerkiksi seinän, osuessa reitille jotta pelaaja saa visuaalista palautetta siitä miksi hahmo ei siirtynyt. Pelihahmon käyttämä ketju ei ole yhtenäinen, vaan jokainen lenkki on oma objektinsa. Tämä mahdollistaa sen, että ketju voi hajota ja pudotessaan jokainen lenkki reagoi esteisiin ja hahmoihin pelifysiikan määräämällä tavalla.

```
IEnumerator DrawLineGraphic(bool breakChain)
{
    Rigidbody2D chainBody;
    Transform chain;
    Vector3 endPosition = pz;
    Vector3 drawPosition = transform.position;
    Vector3 lookPos = endPosition - drawPosition;
    float angle = Mathf.Atan2(lookPos.y, lookPos.x) * Mathf.Rad2Deg;

    while (Vector2.Distance(drawPosition, endPosition) > 0.1f)
    {
        drawPosition = Vector2.MoveTowards(drawPosition, pz, 0.1f);

        chain = ChainPool.instance.Get(chainEffect.transform);
        chain.position = drawPosition;
        chain.rotation = Quaternion.AngleAxis(angle, Vector3.forward);

        activeChainList.Add(chain);

        if (breakChain)
        {
            chainBody = chain.GetComponent<Rigidbody2D>();
            chainBody.isKinematic = false;
            chainBody.AddTorque(100f);

            activeChainList.Clear();

            newPoleTarget.SetActive(false);
        }
    }

    yield return null;
}
```

Kuva 15. Pelihahmon ketjun piirto

Funktion havaitessa liu’utuksen pelihahmo käyttää erikoiskykyä, joka riippuu liikkeen suunnasta. Ylöspäin liu’uttaminen aktivoi lyhytkestoisen ajanhidastuksen, mikä mahdollistaa tulipallojen heittämisen ja antaa pelaajalle lisäaikaa ympäristön havainnointiin tiukassa tilanteessa. Hidastuksen aikana pelihahmo ei tee uusia liikkeitä, mutta pelaajan syöte otetaan silti vastaan ja käskyt suoritetaan hidastuksen loputtua. Tämä mahdollistaa joidenkin liikkeiden samanaikaisen suorittamisen: pelaaja voi esimerkiksi sekä tähdätä tulipallon kohti lähestyvää vihollista että kiskaista itsensä ketjulla pois vaarallisesta kuilusta. Alaspäin liu’uttamalla pelihahmon putoamisnopeus laskee, mikä mahdollistaa hallitun alastulon ja vaarallisten alueiden lähestymisen ylhäältäpäin.

Oikealle tai vasemmalle liu’uttaminen aktivoi syöksykyvyn, mikä siirtää pelihahmoa nopeasti sivusuunnassa. Pelihahmo on jatkuvassa liikkeessä, joten syöksy on erittäin olennainen osa hahmon hallintaa ja ohjattavuutta. Syöksyn aikana pelihahmoa ei voi vahingoittaa, joten sitä voi käyttää esimerkiksi ansojen tai vihollisten hyökkäyksen väistämiseen. Syöksyllä voi myös keskeyttää pelaajan liikkeen ketjua pitkin, mikä tuo koukkuja pitkin liikkumiseen huomattavaa ketteryyttä. Pelaaja ei pysty tekemään useaa syöksyä peräkkäin, mikä osoitetaan pelinäkymän vasemmassa ylälaidassa sijaitsevalla, nopeasti palautuvalla ”energiamittarilla”. Syöksyfunktiota kutsuttaessa tarkistetaan, että liikkeen suorittaminen ei vie pelaajaa kiinteiden objektien sisälle. Tässä tapauksessa syöksy on tavallista lyhempi ja pelihahmo pysähtyy havaitun objektin eteen. Pelaajan on kuitenkin

tärkeää oppia arvioimaan syöksyn pituus, sillä se voi päättyä vihollishahmon ”päälle” mikä saa pelihahmon kimpoamaan vastakkaiseen suuntaan ja aiheuttaa vauriota.

```
pz = dashVector;
direction = (dashVector - transform.position).normalized;
firePosition = transform.position;
invincible = true;

rayHit = Physics2D.Raycast(transform.position, direction, Vector2.Distance(transform.position, dashVector), whatIsGround | whatIsSlippery | whatIsSpike);

if (!grounded && rayHit.collider == null) // Additional ray to check that the player's legs don't collide when in air.
{
    rayHit = Physics2D.Raycast(
        new Vector2(transform.position.x, transform.position.y - colliderRadius),
        (new Vector2(dashVector.x, dashVector.y - colliderRadius) - new Vector2(transform.position.x, transform.position.y - colliderRadius)).normalized,
        Vector2.Distance(new Vector2(transform.position.x, transform.position.y - colliderRadius),
        new Vector2(dashVector.x, dashVector.y - colliderRadius)), whatIsGround
    );
}

if (rayHit.collider != null) // Can't dash full distance.
{
    if (rayHit.point.x < dashVector.x)
    {
        pz = new Vector3(rayHit.point.x - colliderRadius, rayHit.point.y, 0);
    }
    else
    {
        pz = new Vector3(rayHit.point.x + colliderRadius, rayHit.point.y, 0);
    }
}
```

Kuva 16. Syöksyn törmäystarkistus

5.3.2 Ympäristö

Pohjimmiltaan kaikki pelin ympäristöt koostuvat pienistä, leveydeltään ja korkeudeltaan yhden mittayksikön pituisista prefabeista. Kenttien luomisen tehostamiseksi näistä on koottu suurempia, peliympäristön osia kuvaavia prefabeja, joita ovat esimerkiksi tasanteet, seinät, lattiat, pilarit ja taustaseinät. Näitä valmiiksi luotuja kokonaisuuksia pystyy Unityn editorissa asettamaan kentälle helposti raahaamalla ne valikosta haluttuun paikkaan. Kuten muihinkin peliobjekteihin myös näihin voi liittää skriptejä, mikä on tarpeen kun peliympäristön osista halutaan tehdä interaktiivisia.

Monissa pelin kentistä on pelaajalle vaarallisia liikkuvia elementtejä, kuten liikkuvia tai pyöriviä tasanteita. Näitä varten peliin luotiin monikäyttöinen skripti joka ottaa vastaan halutun liikkumis- ja pyörimisnopeuden. Elementille voidaan määrittää liikerata antamalla sille FlightPointeiksi nimettyjä lapsiobjekteja. Kentän alettua skripti tarkistaa onko peliobjektilla johon se on liitetty kyseisiä lapsiobjekteja, joiden koordinaateista luodaan objektin seuraama liikerata. Näin ollen kentän tekijän ei tarvitse liikkuvia elementtejä luodakseen osata itse ohjelmoida.

Monikäyttöisyydestään huolimatta kaikki liikkuvat ympäristön osat eivät voi hyödyntää samaa skriptiä, sillä peliobjektia ei voida liikuttaa vaikuttamalla suoraan sen koordinaatteihin, jos liikkeen halutaan ottavan huomioon pelimaailman fysiikat. Hyvä esimerkki tästä ovat joissain kentissä esiintyvät piikkipallot, joiden on tarkoitus alamäkeen tullessaan vieriä sitä pitkin ja kimmota toiseen suuntaan osuessaan seinään. Tällaisessa tilanteessa täytyy objektin liikuttaminen hoitaa vaikuttamalla sen rigidbody-komponenttiin. Rigidbodyn lisääminen objektiin altistaa sen Unityn fysiikkamoottorille ja mahdollistaa objektin manipuloinnin kohdistamalla siihen voimaa, kuten nopeutta tai vääntöä (Unity Technologies 2015c).

```
transform.position = Vector2.MoveTowards(transform.position, flightPoints[i], maxSpeed * Time.timeScale);
```

Kuva 17. Objektin siirtäminen koordinaatteihin vaikuttamalla

```
rigidbody.velocity = new Vector2(rigidbody.velocity.x + (move * 10), rigidbody.velocity.y);
```

Kuva 18. Objektin siirtäminen rigidbodyyn vaikuttamalla

5.3.3 Viholliset

Pelin sisältämien lukuisien erilaisten vihollistyyppien tarkoitus on luoda vaihtelevia vaaratilanteita jotka pakottavat pelaajan reagoimaan näiden vastustajien toimiin. Toisin kuin staattiset vaara-elementit kuten piikit tai vesiesteet, viholliset yrittävät aktiivisesti tuhota pelihahmon ja reagoivat pelaajaan toimiin. Suoriutuakseen kentistä täytyy pelaajan soveltaa vihollisiin erilaisia taktiikoita näiden käytöksestä ja ominaisuuksista riippuen.

Kivenjärkäleitä heittelevä jäähirviö on ensimmäinen vihollistyyppi jonka pelaaja pelissä kohtaa. Näiden vihollisten on tarkoitus olla jatkuva uhka joten ne ovat tuhoutumattomia, mutta pelaaja pystyy tulipallokyvyllään sekä hetkellisesti lamauttamaan jäähirviön että hajottamaan heitettyjä kiviä. Ohjelmoinniltaan nämä viholliset ovat yksinkertaisia: pelaajan tullessa tarpeeksi lähelle käynnistetään hyökkäysanimaatio, minkä jälkeen kiveä kuvaavasta peliobjektista luodaan kopio joka laukaistaan kohti pelaajaa. Vihollisen OnTriggerEnter-funktio havaitsee siihen osuvat tulipallot, jolloin käynnistetään lamaantumista kuvaava animaatio ja palautumisajastin. Tässä tilassa pelaaja pystyy ohittamaan vihollisen vahingoittumatta, joten funktio sammuttaa myös hetkellisesti vihollisen törmäyksen havaitsevan colliderin. Jotta vihollinen ei colliderin ollessa pois päältä putoa

maaston läpi, säädetään myös vihollisen painovoimakerroin (gravityscale) lamaantumisen ajaksi nollaan.

Helvetinkoirat ovat paljon jäähirviötä nopeampi ja aggressiivisempi mutta myös helpommin tuhottava vihollistyyppi. Koira odottaa kunnes pelaaja tulee tarpeeksi lähelle ja hyppää sitten suoraan tätä kohti, jolloin aloitetaan coroutine ”JumpAtPlayer”. Tässä coroutinessa lasketaan ensin hypyn suunta, joka riippuu sekä pelihahmon asemasta vihollliseen nähden. Jotta koira ei yritä hyökätä pelaajaa kohti esteiden kuten esimerkiksi seinien läpi tarkistetaan kulkureitin esteettömyys Raycastilla, minkä jälkeen lasketaan vihollista kuvaavalle peliobjektille hypyn vaatima kulma ja käynnistetään hyppyanimaatio. Jotta hypyn liikerataa voidaan hallita tarkasti, asetetaan peliobjektin painovoimakerroin hypyn ajaksi nollaan. Coroutine siirtää vihollista hypyn loppupistettä kohti kunnes se saavutetaan, minkä jälkeen peliobjektin kulma ja painovoimakerroin palautetaan normaaleiksi. Laskeuduttuaan koira jää uudelleen odottamaan pelaajan lähestymistä.

```
IEnumerator JumpAtPlayer()
{
    RaycastHit2D rayHit;
    Vector3 target;
    Vector2 direction;
    float angle;

    target = player.transform.position;
    direction = (target - transform.position).normalized;

    rayHit = Physics2D.Raycast(transform.position, direction, Vector2.Distance(transform.position, target), whatIsGround | whatIsSlippery | whatIsSpike);

    if (rayHit.collider == null)
    {
        anim.Play("Werewolf_Jump");

        rigidbody.gravityScale = 0;
        audio.PlayOneShot(audio.clip);
        angle = Mathf.Atan2(direction.y, direction.x) * Mathf.Rad2Deg;
        transform.rotation = Quaternion.AngleAxis(angle, Vector3.forward);

        if (!facingRight)
        {
            Flip();
        }

        CheckAngle(angle);

        while (Vector2.Distance(transform.position, target) > jumpSpeed)
        {
            rigidbody.velocity = Vector2.zero;
            transform.position = Vector2.MoveTowards(transform.position, target, jumpSpeed * Time.timeScale);

            yield return null;
        }

        if (Mathf.Abs(angle) > 90)
        {
            Flip();
        }
```

Kuva 19. Coroutine JumpAtPlayer

Lentävät, hyönteismäiset pirut lukeutuvat pelin helpoimpiin ja lukuisimpiin vihollisiin. Toisin kuin helvetinkoirat ja jäähirviöt ne eivät yritä aggressiivisesti hyökätä pelaajan kimppuun, vaan lentävän pitkin ennalta määrättyä reittiä. Nämä reitit vievät pirut kuitenkin usein pelaajan tielle ja osuessaan ne vahingoittavat pelaajaa. Myöhemmissä kentissä

pirut saattavat kantaa teräviä jääpuikkoja jotka ne yrittävät pudottaa pelaajan päälle. Toisinaan pelaaja kohtaa myös suuremman, hitaasti liikkuvan version pirusta jota ei voi tuhota tulipalloilla. Näiden vihollisten lentoreitti voidaan asettaa reitin pisteet määrittävillä FlightPoint-nimisillä pelaajalle näkymättömillä peliobjekteilla. FlightPointit ovat vihollista kuvaavan peliobjektin lapsiobjekteja, joiden koordinaateista luodaan vihollisen Start-funktiossa lista. Listan luomisen jälkeen käynnistetään MoveTowadsFlightPoint-coroutine, joka liikuttaa vihollista kohti seuraavan pisteen koordinaatteja. Saavutettuaan koordinaatit kohteeksi asetetaan seuraava piste ja coroutine käynnistää itsensä uudestaan mikä toistuu kunnes vihollinen saavuttaa listan viimeisen pisteen, jolloin listaa lähdetään käymään läpi käänteisessä järjestyksessä.

```
while (Vector2.Distance(transform.position, flightPoints[i]) > maxSpeed)
{
    transform.position = Vector2.MoveTowards(transform.position, flightPoints[i], maxSpeed * Time.timeScale);

    yield return null;
}

if (i == flightPoints.Count - 1 && !reverse)
{
    if (stopAtEnd)
    {
        yield break;
    }
    else if (flightPoints[i] == flightPoints[0])
    {
        i = 0;
    }
    else
    {
        reverse = !reverse;
        i--;
    }

}
else if (i == 0 && reverse)
{
    reverse = !reverse;
    i++;
}
else if (!reverse)
{
    i++;
}
else
{
    i--;
}

StartCoroutine("MoveTowardsFlightPoint", i);
```

Kuva 20. Coroutine MoveTowardsFlightpoint

Neljäs pelissä esiintyvä vihollistyyppi on lentävät, siivekkäät kallot. Toisin kuin muut viholliset jotka ovat vain uhkia, kallot ovat tuhoutuessaan hyödyksi sillä ne jättävät aina jälkeensä koukun johon pelaaja pystyy ketjulla tarrautumaan. Joissain kentissä pelaaja joutuu tuhoamaan kalloja voidakseen edetä näin paljastuvia koukkuja pitkin esimerkiksi

kuilun yli. Suurin osa kalloista leijuu paikallaan, mutta osa liikkuu hyödyntäen edellä kuvattua FlightPoint-systeemiä.

6 POHDINTA

Toimeksiantajan vaatimuksena oli toimivan prototyypin tuottaminen pelikonseptin pohjalta, joka sisältää keskeiset pelimekaniikat ja osan pelin kentistä. Nämä vaatimukset täyttyivät, peli on pelattavissa ja toteutetut kentät esittelevät hyvin konseptissa kuvattuja mekaniikkoja, käytännön toimintaa ja antavat kuvan pelin potentiaalista.

Unity osoittautui toimivaksi pelinkehitysympäristöksi, mutta ei täysin ongelmattomaksi valinnaksi. Etenkin kun ottaa huomioon että kyseessä oli 2D-peli, kasvoi projekti tiedostokooltaan huomattavan suureksi, mikä voi olla ongelmallista kauppapaikkojen suhteen. Myös päivitykset loivat toisinaan vaikeuksia, etenkin projektin aikana ilmestynyt Unity 5 aiheutti useita muutoksia ennen kuin peli oli taas toimintakunnossa. Näistä asioista huolimatta on Unity pääsääntöisesti miellyttävä kehitysympäristö käyttää ja tuo mukanaan monia etuja. Ohjelmoijan näkökulmasta erityisen miellyttäviä olivat Unityn verkkosivuilta löytyvä kattava dokumentaatio sekä lukuisat esimerkit ja opetusvideot. Dokumentaation laadukkuudesta ja laajuudesta johtuen sivuilta löytyi myös suuri osa työssä käytetyistä lähteistä.

Nähdäkseni peliä on hyvinkin mahdollista jatkokehittää julkaistavaksi versioksi, mikä vaatisi kuitenkin vielä huomattavasti lisää pelillistä sisältöä ja kenttien ulkopuolisen valikkorakenteen. Suuressa osassa nykyisistä mobiilipeleistä on kuitenkin erittäin matala vaikeusaste, sillä ne on usein tähdätty pelaajille jotka pelaavat pelejä vain ajoittain. Näin ollen ne eivät tarjoa vastusta kokeneelle pelaajalle. *Cast into Hell*-pelistä lähdettiin tämä huomioon ottaen alusta asti tekemään aktiivipelaajille tarkoitettua haastavaa, mutta palkitsevaa kokemusta. Tulevaisuuden kannalta vaarana on, että peli ei tästä lähestymistavasta johtuen löydä kohdeyleisöään.

Pelin tekeminen toi mukanaan lukuisia haasteita ja näiden selvittämisen yhteydessä opin paljon uutta sekä Unitystä että peliohjelmoinnista yleisesti.

LÄHTEET

Ghazarian, A. 2014. The Player Experience: How To Design for Mobile Games. Luettu 21.11.2015. http://designmodo.com/mobile-games-ux/

Unity Technologies. 2015. Execution Order of Event Functions. Luettu 29.11.2015. http://docs.unity3d.com/Manual/ExecutionOrder.html

Unity Technologies. 2015a. MonoBehaviour.Update(). Luettu 29.11.2015. http://docs.unity3d.com/ScriptReference/MonoBehaviour.Update.html

Unity Technologies. 2015b. MonoBehaviour.FixedUpdate(). Luettu 29.11.2015. http://docs.unity3d.com/ScriptReference/MonoBehaviour.FixedUpdate.html

Unity Technologies. 2015c. Rigidbody. Luettu 15.3.2016. http://docs.unity3d.com/ScriptReference/Rigidbody.html

Unity Technologies. 2016a. Sprite. Luettu 23.3.2016. http://docs.unity3d.com/ScriptReference/Sprite.html

Unity Technologies. 2016b. Sprite Renderer. Luettu 23.3.2016. http://docs.unity3d.com/Manual/class-SpriteRenderer.html

Unity Technologies. 2016c. Animation Clip. Luettu 24.3.2016. http://docs.unity3d.com/Manual/class-AnimationClip.html

Unity Technologies. 2016d. State Machine Basics. Luettu 25.3.2016. http://docs.unity3d.com/Manual/StateMachineBasics.html

Unity Technologies. 2016e. Practical Guide to Optimization for Mobiles. Luettu 3.4.2016. http://docs.unity3d.com/Manual/MobileOptimizationPracticalGuide.html

Unity Technologies. 2016f. The Profiler Window. Luettu 26.4.2016. http://docs.unity3d.com/Manual/Profiler.html

Unity Technologies. 2016g. Optimizing Scripts. Luettu 26.4.2016. http://docs.unity3d.com/Manual/MobileOptimizationPracticalScriptingOptimizations.html

Microsoft Developer Network. 2016. How to: Create an Object Pool by Using a ConcurrentBag. Luettu 7.5.2016. https://msdn.microsoft.com/en-us/library/ff458671%28v=vs.110%29.aspx

Dickingson, C. 2015. Unity 5 Game Optimization. Packt Pusblishing.

Hocking, J. 2015. Unity in Action: Multiplatform game development in C# with Unity 5. Manning Publications

Unity Technologies. 2016h. Unity Multiplatform. Luettu 7.5.2016. https://unity3d.com/unity/multiplatform