PELIN TEKOÄLYN SUUNNITTELU

Ensimmäisen persoonan ammuskelupeli

HAMK
HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

Ammattikorkeakoulututkinnon opinnäytetyö

Riihimäki, Tietotekniikan koulutusohjelma

Syksy, 2017

Tommi Väisänen

HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

Tietotekniikan ko.
Riihimäki

Tekijä Tommi Väisänen **Vuosi** 2017

Työn nimi Pelin tekoälyn suunnittelu

Työn ohjaaja Petri Kuittinen

TIIVISTELMÄ

Työn tavoitteena oli luoda toimintaopas aloittelevalle ensimmäisen persoonan räiskintäpelin tekoälyn suunnittelijalle. Samalla tutustuttiin yleisesti pelin suunnitteluun kuuluviin prosesseihin sekä Unity–pelimoottoriin. Käytännön osuuden tavoitteena oli luoda yksinkertainen esimerkkipeli, jonka avulla pystyi mittaamaan pelin suunniteluun kuluvan ajan ja vaivan. Esimerkkipelistä syntyi myös opasteena toimivaa ohjelmakoodia C#-ohjelmointikielellä.

Luodut ohjelmakoodit saatiin toimimaan Unity-pelimoottorilla ja lisäksi pelissä käytettiin yksinkertaisia mutta toimivia mallikappaleita. Tekoälyvirheiden korjaaminen vei odotettua enemmän aikaa, mikä tarkoitti tekoälyn ominaisuuksien vähentämistä.

Pelillisesti reilun tekoälyn ohjelmointi on vaikeaa, sillä tietokoneen äly on erittäin suoraviivaista. Tätä suoraviivaisuutta voi rikkoa muuttamalla tekoälyn saamat tiedot enemmän sattumanvaraisiksi. Tämä saa tietokoneen tekemät päätökset vaikuttamaan enemmän arvauksiksi, mikä vaikuttaa pelaajalle reilulta. Samalla ajateltiin kaikkien toimintojen optimointia, jotta peli ei kuluttaisi liikaa prosessointitehoja.

Avainsanat Tekoäly, Toteuttamissuunnittelu, Pelimoottori, Unity, C#

Sivut 37 sivua, joista liitteitä 5 sivua

HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

Information Technology
Riihimäki

Author Tommi Väisänen **Year** 2017

Subject Designing a game artificial intelligence

Supervisor Petri Kuittinen

ABSTRACT

The aim of this thesis project was to create a tutorial for beginners designing artificial intelligence for a first-person-shooter game. The author also became familiarized with basic game planning processes and the Unity Game Engine during the thesis process. The goal of the empirical part in the project was to produce a simple sample game which was also to measure the time and work required to design a game. The scripts used in the sample game then worked as a guide for the C# scripting in the thesis.

The created scripts worked in the Unity Game Engine and basic but working models were also created and used in the game. Debugging artificial intelligence took more time than was expected which made it necessary to reduce the number of features in the artificial intelligence.

A fair game intelligence is difficult to code because the way computers think is very straightforward. This can be broken by editing the values given to the AI into more random values. This makes the decisions made by the AI seem more like guessing which the player interprets as fair. At the same time scripting the code was also optimized so that the game would not require too much processing power.

Keywords Artificial Intelligence, Designing, Game Engine, Unity, C#

Pages 37 pages including appendices 5 page

SISÄLLYS

1 JOHDANTO

Ensimmäisen persoonan ammuntapelin tekoälyn suunnittelussa pitää ottaa huomioon monia muitakin asioita kuin pelaajaan hyökkääminen. Itse hyökkäämiseen liittyy jo asioita kuten pelikentän analysointi, pelaajan mahdollisten liikkeiden ennakointi sekä näkö- ja kuuloaistin käyttämisen lisäksi mahdollinen kommunikointi muiden hahmojen kesken.

Kaikkeen edeltä mainittuun liittyy myös reiluus. Tekoäly tulee myös muuttaa tyhmemmäksi, jotta pelaaja ei ajattelisi tekoälyn huijaavan. Tällöin Non-Player-Character eli NPC ei reagoi millisekunnin nopeudella ja ammu täydellisellä tarkkuudella pelaajaa, eikä löydä häntä automaattisesti seinien takaa. (Lidén 2004, 41-42.)

Päättääkseen mitä tekoäly tekee milloinkin, on lisättävä jonkinlainen käyttäytymismalli. Kaikki käyttäytymismallit toimivat tekoälyn keskushermostona ja ovat keskeinen osa NPC:n ajattelutapaa. Ero käyttäytymismallien kesken tulee sitä ohjelmoitaessa ja hahmottamisessa. Työssä käydään käyttäytymismalleista läpi äärellinen automaatti ja käyttäytymispuu. Esimerkkityössä käytetään käyttäytymismallina äärellistä automaattia.

Opinnäytetyössä käytetään Unity Game Enginellä tehtyä peliä esimerkkinä sekä käydään ensin läpi Unityn käyttöliittymä ja sitten tekoälyn suunnittelu ja ohjelmointi. Työssä käydään myös läpi A* reitinhaku pinnallisesti ja siihen liittyen pelikentän suunnittelussa huomioitavat asiat kuten tasapuolisuus ja kulkureitteihin liittyvät asiat kuten näköyhteyksien pituus ja vaikeasti kuljettavat alueet.

2 KÄYTTÄYTYMISMALLIT

2.1 Äärellinen automaatti

Äärellinen automaatti on monista tiloista koostuva hypoteettinen kone. Kone vaihtaa tilaansa riippuen saamastaan ärsykkeestä ja vastaa tapahtumiin valitun tilan mukaisesti. Koneella voi olla vain yhdessä tilassa kerrallaan, jolloin tiloja ja tilan vaihtoja tulee yleensä olla vähintään kaksi, mutta on tapauksia joissa yksi tila riittää. (Bevilacqua, F 2013)

Äärellinen automaatti on yksinkertainen vaihtoehto ensimmäisiksi tekoälyiksi, sillä tilat ja niistä siirtymiset ovat koodista helppolukuisia. Tiloista siirtymisien esittäminen diagrammina auttaa tekijää ymmärtämään kokonaisuuden. Diagrammista on apua tekoälyn jatkokehittämisessä sekä jokaiseen tekoälyyn liittyvien tilojen ja siirtymien ymmärtämisessä. (Gesota, R. 2016)

Äärellistä automaattia ohjelmoitaessa on hyvä pitää tilan koodi ja siitä siirtyminen peräkkäisinä, jolloin on helppo nähdä jälkeenpäin, kuinka jokainen tila toimii ja kuinka jokaisesta tilasta siirrytään pois. Tilan ja siirtymisen erottamisessa tulee ongelmaksi nähdä ja laskea jokaisen tilan siirtymiset samanaikaisesti ja tekoälyn kokonaisuuden lukemisesta tulee vaikeampaa. Äärellinen automaatti tulee kasvamaan pelin kehityksen jokaisessa vaiheessa, minkä takia kokonaisuuden lukemisen vaivattomuus on tärkeää. (Fu & Houlette 2004, 286)

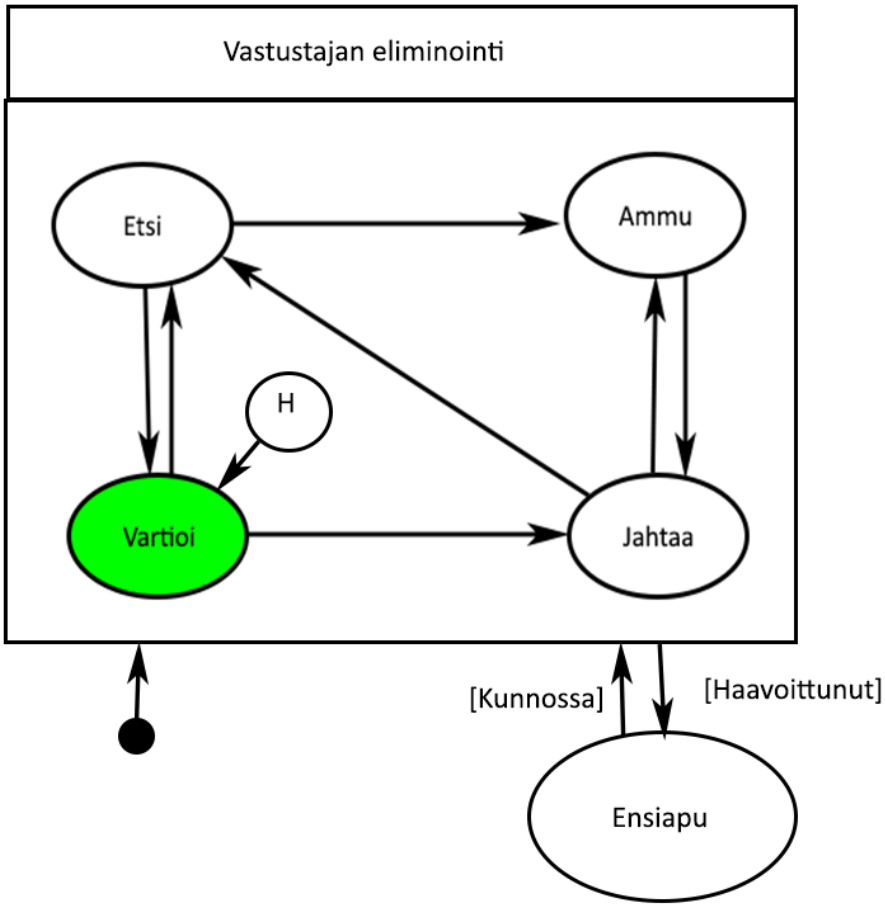
Äärellisen automaatin väistämättömän koon kasvamisen takia on tärkeää ylläpitää muistilistaa pseudokoodina, joka listaa automaatin tilat ja ehdot.

Taulukko 1 Esimerkkipelin pseudokoodi

```
void Update() {
   switch (curState) { //Vaihda tilaa
        case  Patrol: //Tila Patrol
            doPatrol(); //Vartioi aluetta
            if (playerInSight) //Pelaaja näkökentässä
                curState =  Chase; //Vaihda tilaan Chase
            break;
        case  Chase: //Tila Chase
            doChase(); //Mene viimeiseen paikkaan jossa pelaaja
nähty
            if (!playerInSight) //Pelaaja ei ole näkökentässä
                curState =  Search; //Siirry tilaan Search
            if (playerInSight)
                curState =  Shoot; //Siirry tilaan Shoot
            break;
        case  Search: //Tila Search
            doSearch(); //Tutki lähialue
            if (elapsedTime = 5) //Kulunut aika
                curState =  Patrol; //Siirry tilaan Patrol
            if (playerInSight)
                curState =  Shoot; //Siirry tilaan Shoot
            break;
        case  Shoot: //Tila Shoot
            if (playerInSight)
                doShoot(); //Ammu pelaajaa
            else
                curState =  Chase; //Siirry tilaan Chase
            break;
    }
}
```

Muistilista (taulukko 1) tuo etuja ohjelman rakenteeseen, luettavuuteen ja virheiden poistamiseen. Kaikki automaatit tulevat noudattamaan samankaltaista rakennetta, mikä auttaa muita ohjelmoitsijoita. Luettavuuden kannalta kehittäjä voi kirjoittaa tiivistä ja helpommin luettavaa koodia. Lukija pystyy näin keskittymään enemmän tilojen toimintoihin ja siirtymislogiikkoihin. Virheiden korjaaja pystyy keskittymään täysin tehtäviin sekä siirtymisiin, eikä jokaisen tilan kokonaisuuksiin ja laajennuksiin. (Fu & Houlette 2004, 287-288)

Äärellinen automaatti myös toimii monimutkaisempina toimintamalleilla kuten pakka-automaattina ja hierarkkisena automaattina (Kuva 1) jotka pystyvät päättämään joustavammin siirryttävät tilat. Joustavuus vähentää työmäärää huomattavasti kun siirrytään monimutkaisempiin tekoälyihin. Monimutkaisemmissa tekoälyissä tilojen määrä kasvaa, jolloin siirtymien määrä saattaa moninkertaistua. Tilat, jotka kasvattavat eniten siirtymien määrää, ovat eräänlaiset hätätilat joihin pitäisi pystyä siirtymään kaikista tiloista. Esimerkiksi NPC:n ollessaan haavoittunut, ei hahmo pyri taistelemaan tai etsimään vastustajaa vaan pyrkii joko piiloutumaan tai etsimään ensiapua. (Bevilacqua, F 2013)



Kuva 1 Hierarkkinen automaatti

Hierarkkisen automaatin tarkoitus on asettaa tiloja sisäkkäin, jolloin on mahdollista asettaa suuremmista tiloista vähemmän siirtymisiä kuin jokaisesta tilasta siirtymät yhteen tilaan. Hierarkkisessa automaatissa alkupiste on kuvassa 1 merkitty mustalla pisteellä. Alkupiste merkitsee mikä on ensimmäinen tila johon tullaan siirtymään. Ympäröity H merkitsee alitilan johon siirrytään kun tilakokonaisuuteen ”Vastustajan eliminointi” siirrytään takaisin.

2.2 **Käyttäytymispuu**

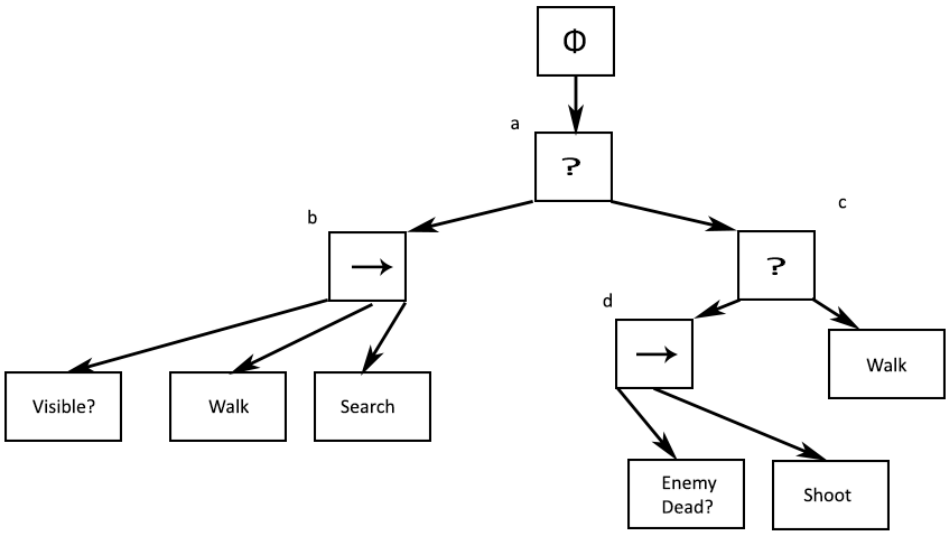
Käyttäytymispuussa tekoälyn tehtävät jaetaan moneen eri osaan ja rakennetaan malli tehtävien suorittamisesta. Malli koostuu juuresta josta siirrytään suurin tehtäviin joista taas siirrytään niitä pienempiin tehtäviin niin pitkään että jokainen matkalla ollut tehtävä voidaan merkitä suoritetuksi.

Käyttäytymispuun solmut koostuvat Leaf-, Decorator- ja Composite-kategorioista, jotka päättävät tai tarkentavat, kuinka tehtävät suoritettaan. Leaf-solmu on joko ehdon varmistava tai tehtävän tekevä solmu, joka muuttaa tilaa tekemällä esimerkiksi ampumisanimaation. Tehtävä

palauttaa solmulle joko onnistumisen tai epäonnistumisen. Suurin osa käyttäytymispuusta koostuu näistä solmuista.

Decorator-solmu tarkentaa yhden lapsensa toimintaa esimerkiksi kääntämällä lapsen tuloksen päinvaistaiseksi tai mukauttamalla lapsen tehtävää toimimaan hitaammin. Hahmon kävelyanimaation voisi siis muuttaa Decorator-solmulla hitaammaksi kun hahmo on loukkaantunut ilman itse animaatioon koskemista.

Composite-solmu voi sisäistää monia lapsia ja on siten suurin rakennuspalikka käyttäytymispuussa. Solmu päättää toimintajärjestyksen lapsilla sekä palautusarvon, jonka lähettää takaisin vanhemmalle. Composite-solmuja ovat kuvan 2 valitsijat ja sekvenessit. Kolmantena vaihtoehtona on rinnakkaissolmu, joka suorittaa kaikkia lapsia samaan aikaan. Mikäli lapsien palauttama onnistumisien määrä on suurempi kuin valittu parametriarvo, palauttaa solmu epäonnistumisen, muuten solmu jatkaa toimintaa. Valitsijan (merkitty ?) tehtävänä on suorittaa ensimmäinen solmu. Ensimmäisen solmun palauttaessa epäonnistuminen, valitsija siirtyy seuraavaan solmuun. Sekvenssi (merkitty laatikkoon nuolella) suorittaa jokaisen lapsen yksitellen vuorollaan niin pitkään että jokin lapsista palauttaa jonkin muun kuin onnistumisen.



Kuva 2 Käyttäytymispuu

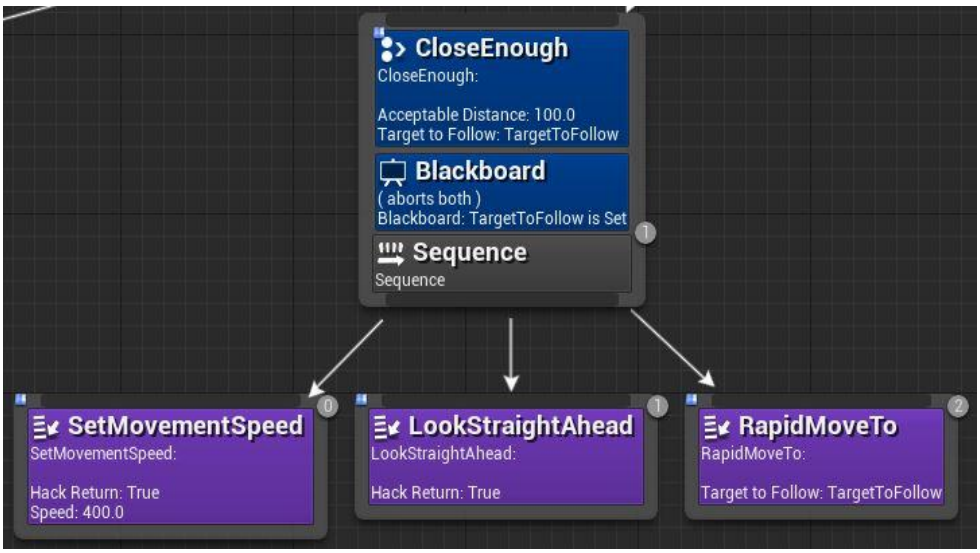
Kuvassa 2 käyttäytymispuussa valitaan (a) aluksi vasemmanpuoleinen etsintäsekvenssi (b) kunnes kohde tulee näkyviin, jolloin valitsijoiden a ja c kautta suoritetaan ammuntasekvenssiä (d) kunnes kohde on kuollut. Valitsija c suorittaa lopuksi kävelemisen kohteelle, jolloin koko käyttäytymispuu on käyty läpi ja voidaan palata juureen.

Kuvan 2 jokainen tehtävä pystyy jakautumaan pienempiin puihin kuten animaatioon tai muihin ehtoihin ja tehtäviin. Esimerkkinä ampuminen voi vielä jakaantua ampumisanimaatioon ja ehtoon jossa varmistetaan ammusten määrä.

Toimintojen lisääminen eri tehtäviin muuttuu helpoksi kun toiminnot itsessään sisältävät jo tarvittavat animaatiot, toiminnot ja ehdot. Tällöin toiminnot on helppo muodostaa puiksi, eikä ohjelmoijan tarvitse miettiä, kuinka jokainen alatoiminto suoritetaan. (Millington, I. & Funge, J. 2009)

Käyttäytymispuun arkkitehtuuria voi myös muokata tapahtumiin perustuvaksi, jolloin joka ruutupäivityksen aikana ei tarvitse ajaa koko puuta läpi muutoksien varalta vaan käyttäytymispuu voi kuunnella tapahtumaa, joka laukaisee muutoksen puussa. Tästä seuraa suuri suorituskyvyn parantuminen. (Unreal Engine, How Unreal Engine 4 Behavior Trees Differ, n.d.)

Unreal Enginen muunnelma peruskäyttäytymispuusta on helpommin luettava suuremmissa mittakaavoissa kuin peruskäyttäytymispuu. Tämä perustuu muun muassa Leaf-solmun ehdon muuttamisen Decorator-solmuksi. Tällöin arkkitehtuuri muuttuu myös tapahtumiin perustavaksi, optimoiden prosessointia. Toisena suurena muutoksena on rinnakkaissolmujen muuttaminen sallimaan vain toimijan ja mahdollisesti kokonaisen alapuun. (Unreal Engine, How Unreal Engine 4 Behavior Trees Differ, n.d.)



Kuva 3 Unreal Enginen käyttämä käyttäytymispuu (Unreal Engine (n.d.). How Unreal Engine 4 Behavior Trees Differ)

3 PELITYYPPI

Ensimmäisen persoonan räiskintäpelin laji saattaa olla keskittynyt aikakausiin, kuten maailmansotiin tai nykyaikaan. On mahdollista aikakauden olevan myös futuristinen, jolloin hahmoilla voisi olla käytössä mielikuvituksellisia laseraseita tai hahmo pystyisi liikkumaan epäluonnollisen nopeasti. Maailmansotiin ajoitettu peli voisi keskittyä realistisiin, vanhoihin aseisiin ja ajoneuvoihin. (Wikipedia, First Person Shooter, n.d.)

Valitsin pelityypiksi nykyaikaisen, realistisen räiskinnän, jolloin pelien hahmot tuntuvat liikkuvan nopeuksilla, jotka tuntuvat oikeilta ja käyttävät nykyaikaisia aseita. Tällöin pelaaja pystyy arvioimaan mikä kaikki on mahdollista hahmoille. Nykyaikaisella sotilaalla on yleensä pääaseena isomman kaliiperin ase kuten rynnäkkökivääri ja sivuaseena pienemmän kaliiperin pistooli.

Hahmojen kestävyys myös vaihtelee huomattavasti pelityypistä riippuen. Esimerkiksi futuristisessa pelissä kuten Halo: Combat Evolved:issa hahmot kestävät monia osumia pelin rynnäkkökivääristä, koska hahmoilla on suojakenttä. Enemmän realistisessa pelissä kuten Counter Strike: Global Offensessa pelaaja kestää vain muutaman osuman kehoon ennen kuolemaa. Counter Strikessä hahmon kestävyyttä voi hieman nostaa rintapanssarilla ja kypärällä. Pääosumat ilman kypärää kaataa pelaajan yleensä yhdellä osumalla mutta kypärän kanssa tarvitaan muutama osuma pienemmällä kaliiperilla.

Kuva 4 Halo: Combat Evolved (Wikipedia (n.d.). Halo: Combat Evolved)

Kuva 5 Counter Strike: Global Offensive (Wikipedia (n.d.). Counter Strike: Global Offensive)

Räiskintäpelien pelattavuuteen yleensä liittyy tiiviisti pelin ase- ja tavaravalikoima. Pelaajille jää mieleen helposti aseet jotka tuntuvat hyödyttömiltä sekä aseet jotka toimivat liian tehokkaasti. Pelkässä yksinpelissä näiden arvojen hiominen ei ole niin tärkeää mutta verkonvälityksellä pelattavissa, monen pelaajan pelissä on tärkeää muokata kaikkien aseiden arvoja siten etteivät ääripäät ole liian kaukana toisistaan. Yleensä tämä tarkoittaa arvojen hiomista pelin julkaisunkin jälkeen. (Polson, T. 2012)

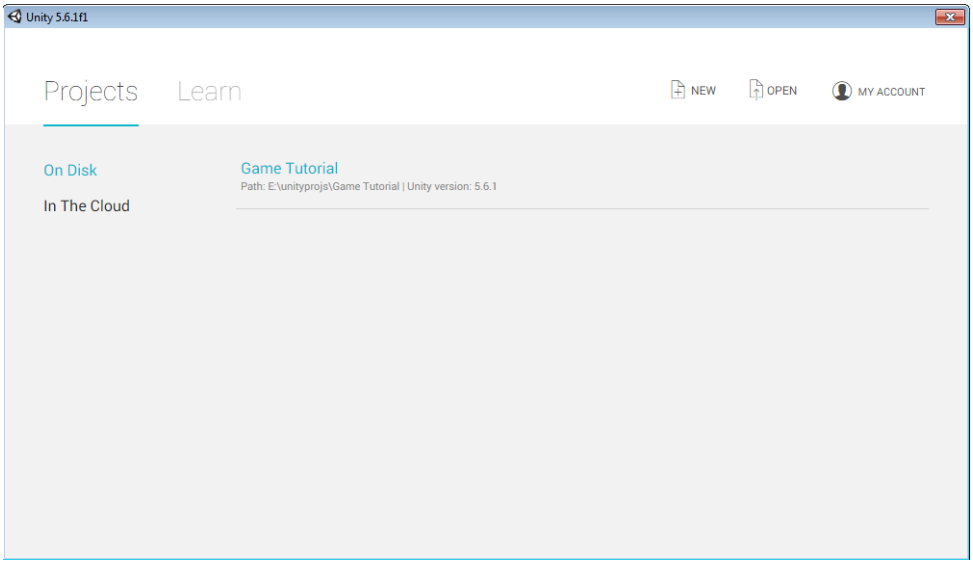
4 UNITY GAME ENGINE

4.1 Esittely

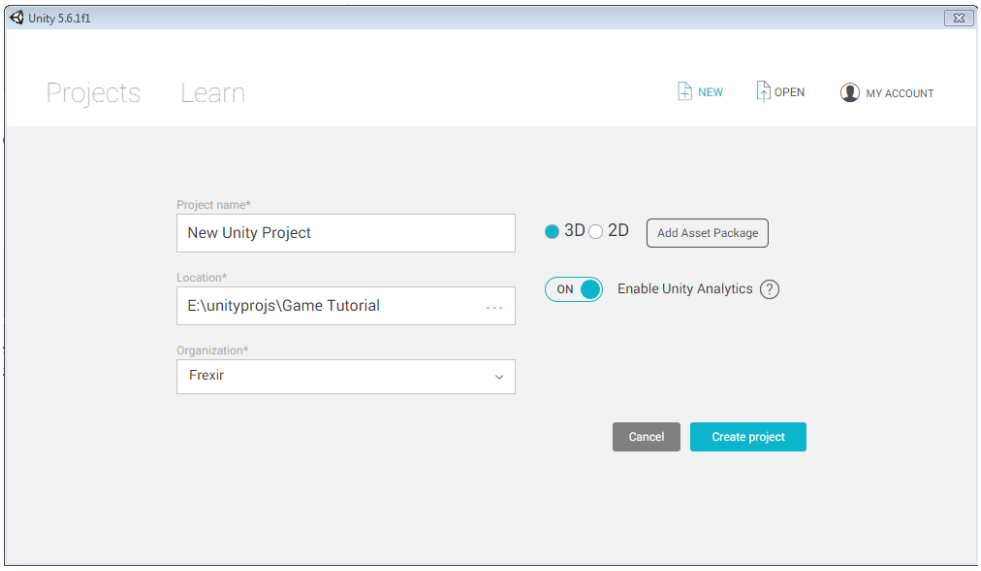
Käytössä oli Unityn Personal Edition, joka vaatii vain ilmaisen rekisteröitymisen. Personal Editioniin sisältyy ominaisuudet kuten Core Analytics, mainosten upottaminen, pelinsisäisten ostoksien teko, pelin rakentaminen pilveen, 20 pelaajan yhteispelin ja tuen kaikille alustoille. Core Analytics ominaisuudella pystyy valvomaan, kuinka tehtyä ohjelmaa käytetään sekä keräämään tärkeää tietoa kuten pelaajien edistymistä. (Unity, Unity Analytics, n.d.)

4.1.1 Aloittaminen

Personal Editionin asennettua ja käyttäjän tehtyä aloita uusi projekti valitsemalla ”NEW”. Nimeä projekti, valitse pelityypiksi 3D sekä valitse tallennuspaikka (kuva 7). Myöhemmin avattaessa Unityn luodun projektin saa auki Projects-välilehdestä (kuva 6).



Kuva 6 Unity projektit

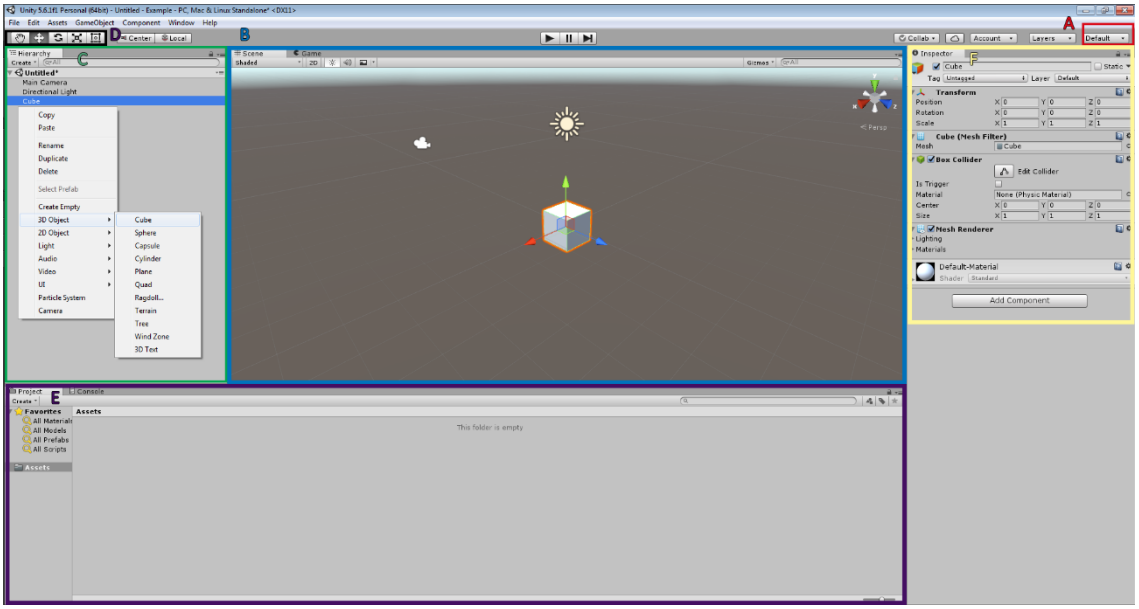


Kuva 7 Uusi projekti

Projektin luomisvaiheessa (kuva 7) on myös mahdollista lisätä Unityn omia valmiita toimintoja ja mallikappaleita valitsemalla ”Add Asset Package”, josta voi valita haluamansa paketit. Tämä lataa koneelle valmiit paketit ja lisää ne projektin Assets-kansioon.

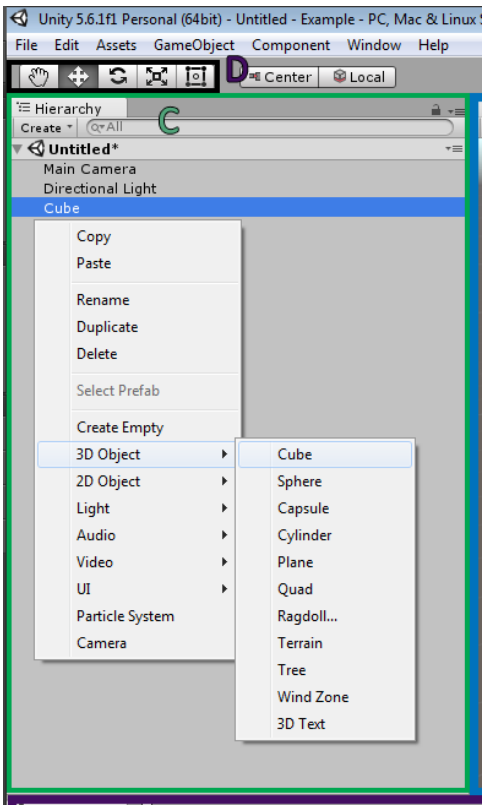
4.1.2 Unityn käyttöliittymä

Ruudulle aukeaa kuvan 8 mukainen käyttöliittymä, jos mitään muutoksia ei ole tehty. Käyttöliittymää voi muokata joko käsin siirtelemällä osioita tai käyttämällä valmiita pohjia Layout-valikosta (Kuvan 8 punainen A).



Kuva 8 Unityn peruskäyttöliittymä

Keskellä käyttöliittymää on Scene- ja Game-katseluikkunat (Kuvan 8 sininen B). Scene-ikkunassa kootaan yhteen sekä siirrellään pelin osia, kuten kamerat, mallikappaleet ja valot. Ikkunaa voi käyttää myös testattaessa peliä, esimerkiksi mallikappaleita voi siirrellä testauksen aikana. Testauksen loputtua kaikki tehdyt muutokset palautuvat samoiksi kuin testauksen alussa. Game-katseluikkuna näyttää pelaajaksi merkityn hahmon kameran. Tässä ikkunassa myös ohjataan pelaajaa testipelauksen aikana.

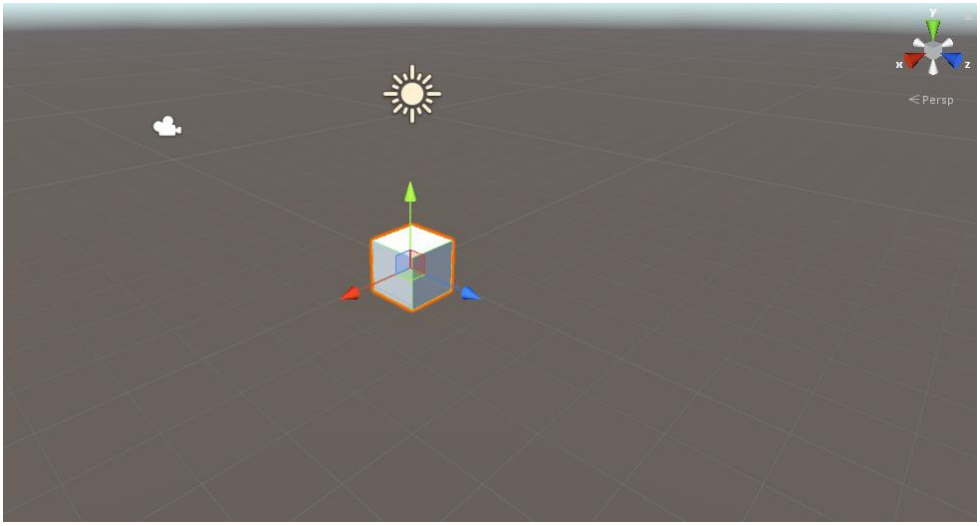


Kuva 9 Hierarkia ja työkalut

Hierarchy-ikkuna (Kuvan 9 vihreä C) näyttää listan pelin sisältävistä kohteista. Listan sisällä pystyy luomaan eri kohteita painamalla hiiren 2. nappulaa ja valitsemalla listasta haluamansa kohteen. Kohde sijoitetaan pelin sisälle keskelle nykyistä Scene-kameraa. Saman tekee yläpalkin GameObject-pudotusvalikko. Tässä valikossa kohteet voidaan myös asettaa sisäkkäin. Yleistä on luoda tyhjiä kohteita, jotka nimetään yleisesti kartaksi tai pelaajaksi, joihin kaikki toisiinsa liittyvät kohteet kerätään.

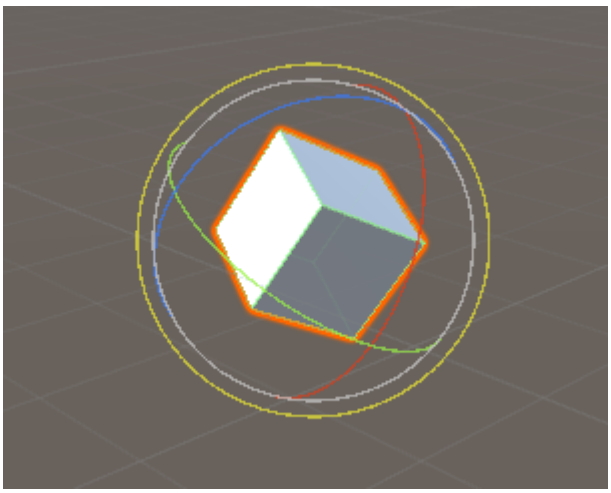
Hierarchy-ikkunan yläpuolella on kohteita sekä katseluikkunaa manipuloivat työkalut (Kuvan 9 D-laatikko). Ensimmäisenä on Hand-työkalu jonka avulla vieritetään tai käännetään kameraa ensimmäisellä tai toisella hiiren painikkeella. Samalla voi zoomata näkymää painamalla pohjaan ctrl- ja alt-painikkeet sekä toista hiirenpainiketta ja liikuttamalla hiirtä ylös tai alas.

Toisena on Translate-työkalu, jolla valittua kohdetta pystyy siirtämään käyttämällä joko x-, y-, z–akseleita tai kahden akselin väliä käyttämällä akselien välisiä neliöitä.



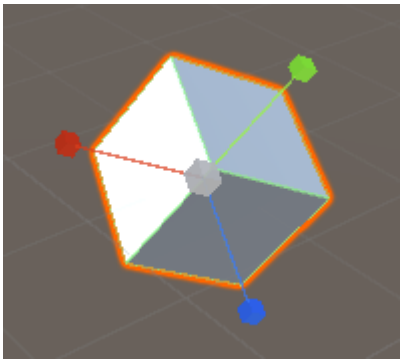
Kuva 10 Akselit

Kolmantena työkaluna on Rotate-työkalu, jolla pyöritetään kohdetta akselien mukaisesti. Kohdetta voi myös pyörittää vapaasti valitsemalla akselien välisen tyhjän alueen. Uloimmaisin reuna, joka on merkitty keltaisella, pyörittää kohdetta katselukulman mukaisesti 2D-alustalla.



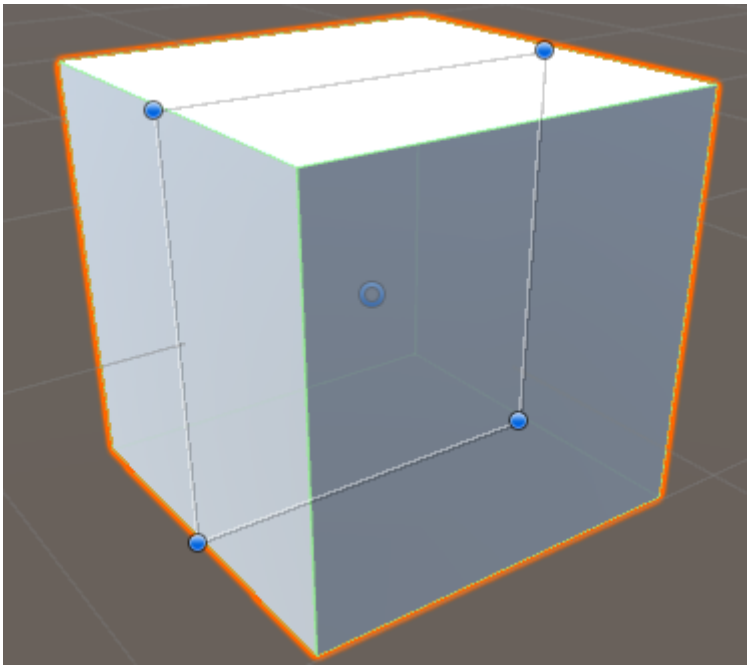
Kuva 11 Kohteen pyöritys

Neljäntenä työkaluna on Scale, jonka avulla voi skaalata kappaletta joko yhden akselin pituudella tai kaikilla akseleilla samanaikaisesti. Skaalaaminen onnistuu vetämällä yhtä kohteen akseleista tai vetämällä kohteen keskellä olevaa harmaata laatikkoa ylös tai alas muuttaakseen koko kohteen kokoa.

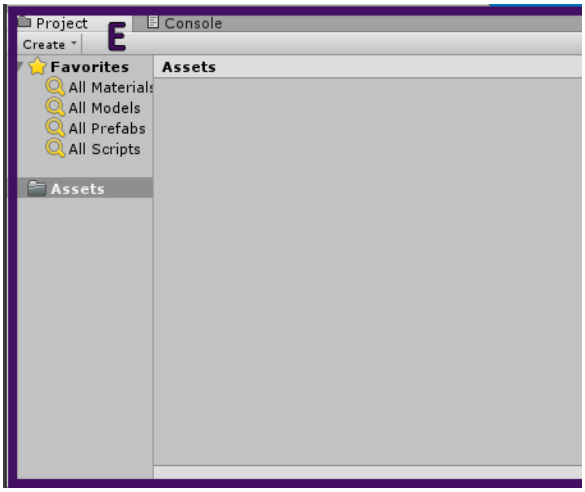


Kuva 12 Kohteen skaalaaminen

Viimeisenä työkaluna on Rect. Työkalu toimii 2D-tasolla, mutta sitä voi käyttää 3D-malleissa. Työkalulla voi venyttää tai supistaa kohteen reunoja sekä nollata kohteen Pivot-pisteen. Pistettä kuvaa sininen ympyrä kohteen päällä.

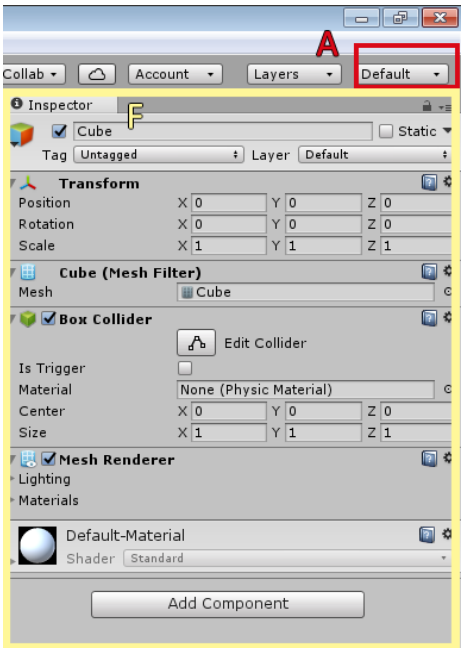


Kuva 13 ”Rect” työkalu



Kuva 14 Projekti ja konsoli ikkunat

Projektiin sisälle tuodut kaikki materiaalit ja tehdyt ohjelmakoodit kerääntyvät Project-välilehteen. Assets-kansion alle luodaan kansioita, joiden sisälle kerätään haluamat tiedostot. Yleisimmin luodaan kansiot materiaaleille, animaatioille, mallikappaleille, ohjelmakoodeille ja prefab-kappaleille. Ne ovat kohteita joihin on valmiiksi liitetty kaikki tarvittavat osat, kuten ohjelmakoodit ja mallikappaleet. Tällöin on helpompaa muuttaa kaikkia kentän kohteita muuttamalla prefab-kappaletta, sillä kentälle tuodut prefab-kappaleet ottavat kaikki sille kappaleelle tehdyt muutokset vastaan. Kentän yksittäistä kappaletta voi kuitenkin muuttaa yksinään mutta tämä rikkoo yhteyden prefab-kappaleeseen.



Kuva 15 Inspector-välilehti

Inspector-välilehdellä voi muuttaa kohteen eri parametreja kuten koordinaatteja. Kaikki julkiseksi määritetyt parametrit tai erikseen [SerializeField]-koodilla merkityt parametrit näkyvät ohjelmakoodin

kohdalla välilehdessä. Kohteen pyörityksen ja skaalaamisen voi myös tehdä Transform-kohdassa. Tärkeimpänä kohtana on Add Component -toiminto, jolla kohteeseen voi lisätä ominaisuuksia kuten animaattorin tai ohjelmakoodin.

4.2 **Mallikappaleet**

Esimerkkipelin mallikappaleina käytetään yksinkertaisia muotoja, jotta työn suuruus säilyisi kohtuullisena. Työssä käytetään muutamia valmiita mallikappaleita Unity Storesta.

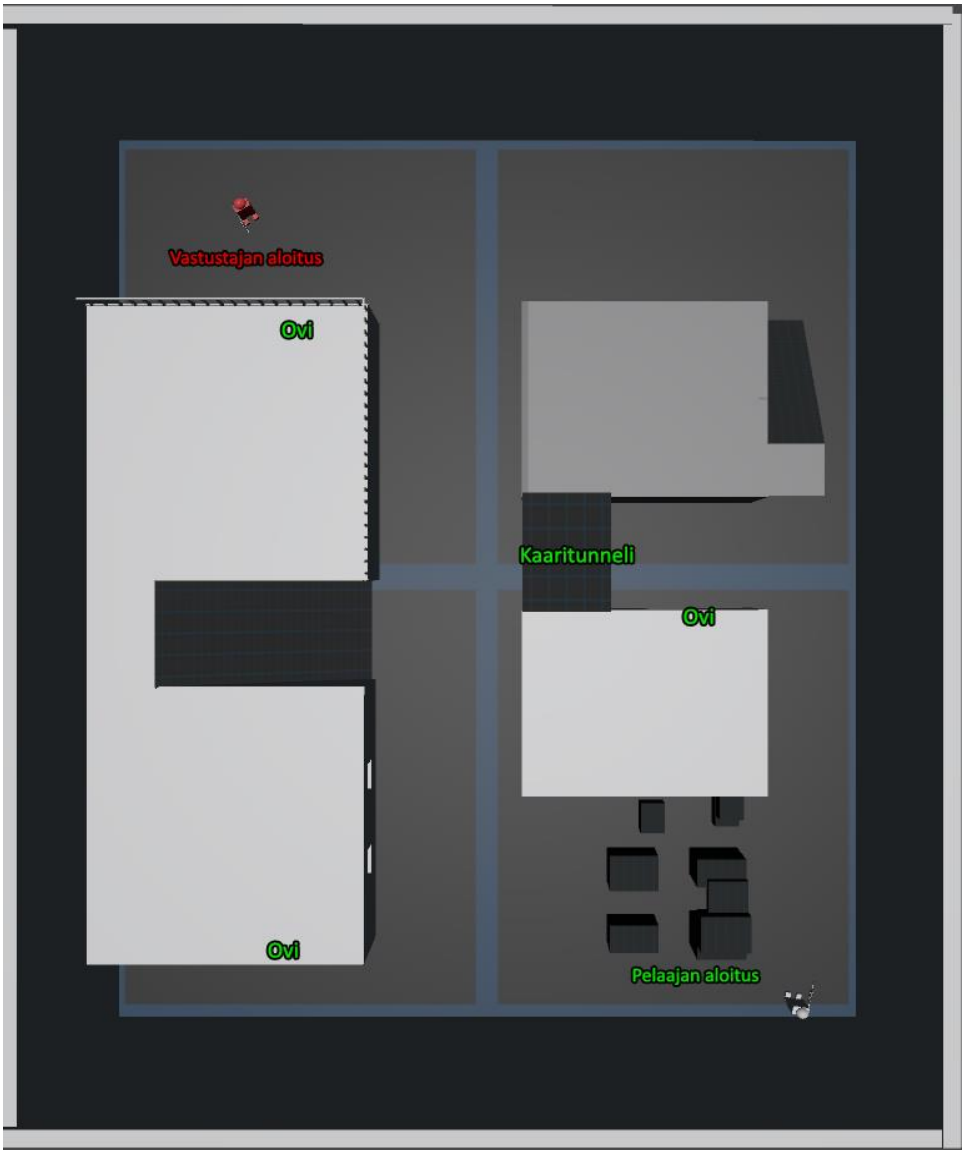
Mallikappaleista riippuen tekoälyn muokkaaminen saattaa muuntua vaikeammaksi, esimerkiksi näköaistin suhteen. Tekoälyn ei tulisi nähdä ruohon tai lehvistön väleistä pelaajaa välittömästi vaan huomioon tulisi ottaa myös kuinka paljon pinta-alaa pelaajasta näkyy. (Tvtropes (n.d.). The Computer Is a Cheating Bastard)

Pelikentän mahdolliset yksityiskohdat kuten ikkunat ja ovet tulee myös valjastaa tekoälyn käyttöön. Lisätessä yksityiskohtia pelikenttään tulisi aina ottaa huomioon mahdolliset vaikutukset tekoälyyn ja siihen liittyvän ohjelmoinnin työmäärän lisääntymiseen.

4.3 **Pelikenttä**

Kenttää suunnitellessa otetaan huomioon ensimmäisenä mihin tarkoitukseen se on. Yleensä ensimmäisen persoonan räiskintä peleissä on kaksi joukkuetta joiden tarkoituksena on eliminoida toisen joukkueen jäsenet. Tässä tapauksessa on kenttä rakennettava tasapuoliseksi molemmille joukkueille. Tasapuolisuutta yleensä lisätään vaihtamalla aloituspuolia määritetyin väliajoin.

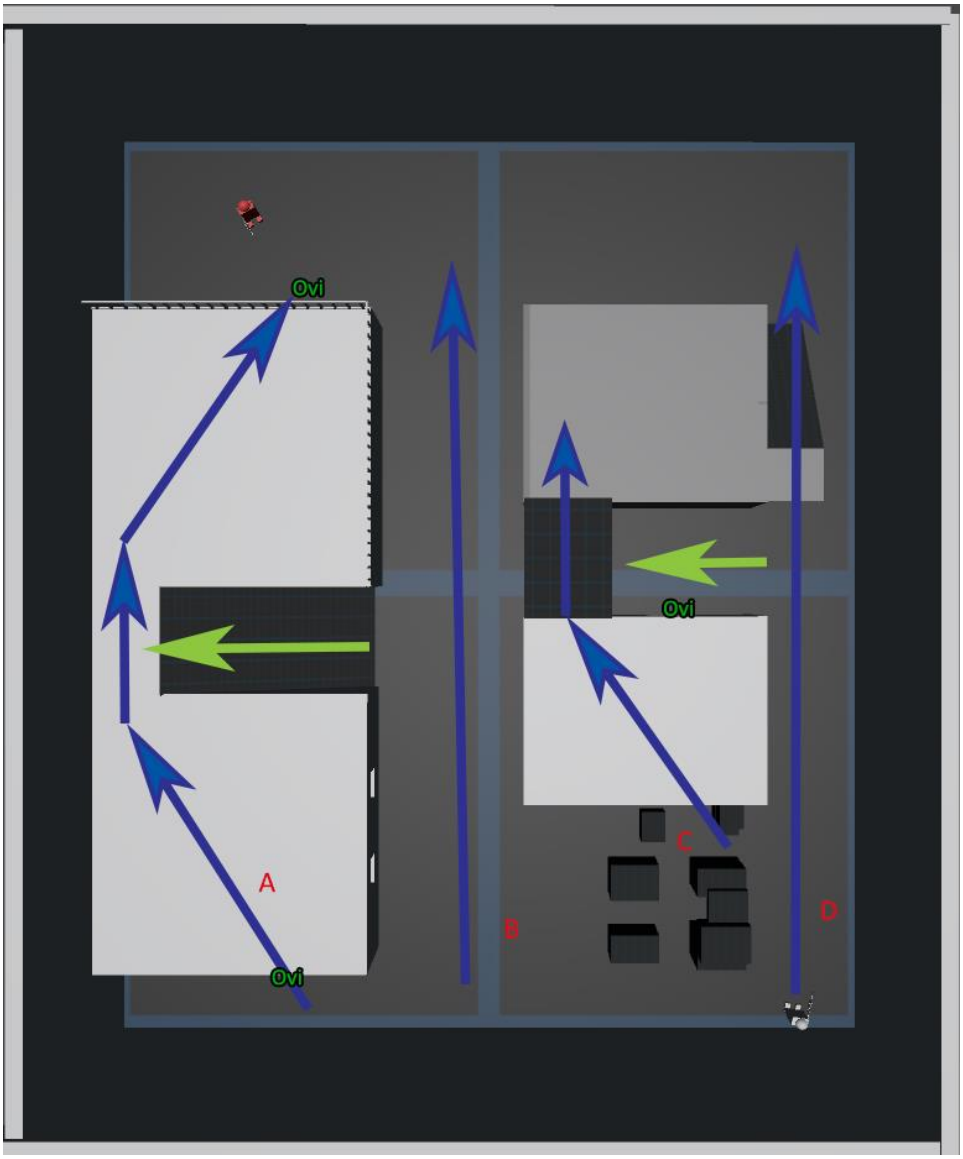
Pelikenttä muodostettiin kolmesta talosta joiden katoille johtaa kaksi ramppia ja laatikkokasa. Kahden talon sisätilat ovat käytössä ja ison talon sisätila on yhdistetty, joka antaa pelaajalle siis yhteensä neljä eri vaihtoehtoa siirtyä kentän toiselle puolelle.



Kuva 16 Pelikenttä

Suunnitellussa tulee ottaa huomioon kulku reittien määrä, vaikeasti kuljettavien reittien paikat ja näköyhteyksien pituudet. Kuvassa 17 näkyvät pääreitit A, B, C ja D sininä nuolina. Sivureittejä kuvaavat vihreät nuolet. Koska pääreittejä on monia, ei ole vaaraa tilanteesta joissa molemmat joukkueet ovat asemasodassa vaan pelaajat pystyvät pysymään liikkeessä ja vaihtamaan taktiikkaansa. Tämä myös tarkoittaa tekoälyn pystyvän valitsemaan eri taktiikan suuremmasta, ennalta määrätystä valikoimasta.

Koska kentässä on paljon reittejä sekä korkeampi taso, tulee näköyhteyksistä hyvinkin pitkiä. Kenttään kuitenkin sisältyy muutamia lyhyen välimatkan näköyhteyksiä kuten suuren talon sisätilat sekä mahdollisen sivureitin käyttö B- ja D-reiteissä. Vaikeasti kuljettavina reitteinä ovat B- ja C-reitti, johtuen reittien pitkistä näköyhteyksistä moneen eri suuntaan.



Kuva 17 Pelikentän reitit

4.4 **Pelaaja**

Pelaajaan liittyvät ohjaimet kuten liikkuminen, katsominen, ampuminen ja lataaminen. Liikkumisessa tulee ottaa huomioon eri liikkumisnopeudet ja hyppy- ja kyykkykorkeudet. Ampumiseen yleensä liittyy voimakkaasti valitun aseen parametrit, eli eri aseet saattavat käyttäytyä eri tavoin johtuen voimakkaammasta rekyylistä tai nopeammasta tulinopeudesta.

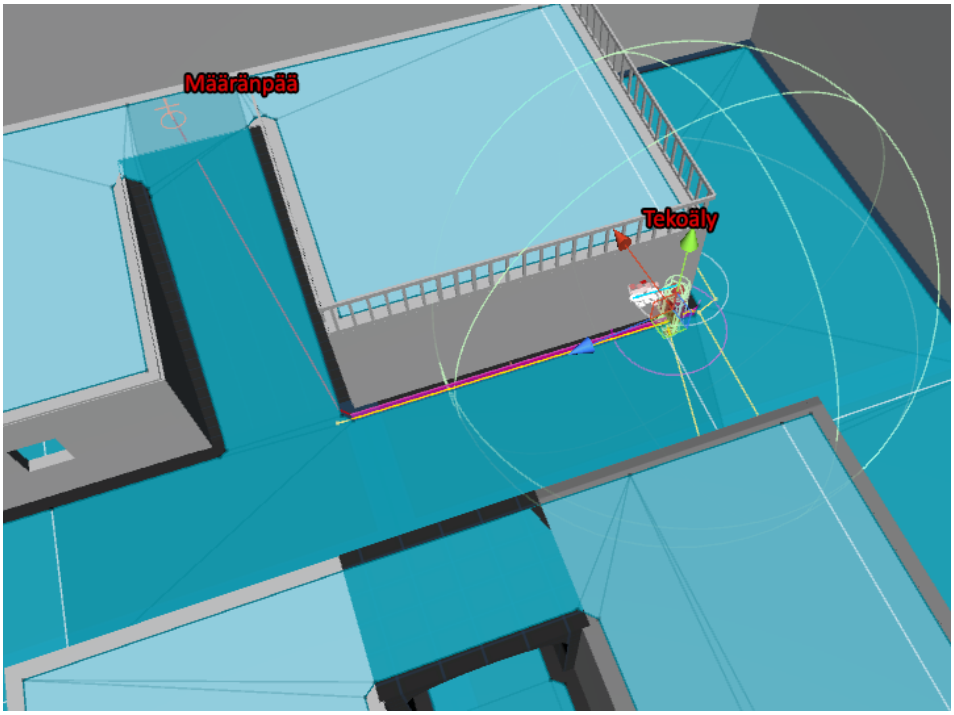
Pelaajalle tulisi myös lisätä graafinen käyttöliittymä, joka kertoo pelaajalle tärkeitä tilastoja kuten elinvoiman määrä tai aseen ammusten määrä. Pelilajista riippuen pelaajalla saattaa olla käytössä kartta ja muita kentällä tärkeitä kohtia kertova merkki kuten kuvassa 18.

Kuva 18 Tom Clancy's Rainbow Six Siege (Wikipedia (n.d.) Tom Clancy's Rainbow Six. Siege)

5 REITINHAKU

Reitinhaussa käytettiin Unityn navigation meshiä, joka puolestaan käyttää A*-algoritmia. Algoritmi määrittää painot jokaiselle avoimelle solmulle, eli viereiselle ruudulle johon ei ole vielä siirrytty. Paino perustuu ruutujen välisen reunan painolle johon lisätään arvioitu matka lähtöpisteestä loppupisteeseen. Matkan arvio perustuu heuristiikkaan, joka hylkää mahdolliset solmut joiden matkat ovat pidempiä kuin lähtö- ja loppupisteen matkat. (Unity (n.d.). Inner Workings of the Navigation System)

Navigation mesh luo kartan alueista jotka määritettään käveltäviksi ja esteiksi toimijoille. Toimijoille määritetään parametreja kuten koko ja askelkorkeus. Näiden perusteella navigation mesh luo pinnan yhdistämällä alueet, joissa toimitsija voi seistä. Alueet muutetaan kuperiksi monikulmioiksi, jolloin pystytään tunnistamaan monikulmioiden reunat sekä viereiset monikulmiot. Tällöin monikulmioiden alueet voidaan tulkita toimitsijalle vapaasti liikuttaviksi. (Unity (n.d.). Inner Workings of the Navigation System)



Kuva 19 Reitinhaku

Kuvassa 19 toimitsijalla on määränpäänä rampin yläpää. Tekoälyn valitsema reitti on vaaleanpunainen ja tekoälyn laskemat solmut reittivalinnoille ovat keltaisella. Vaaleansininen alue on navigation meshin liikuttavaksi laskettu alue.

6 TEKOÄLYN REILUUS

Tekoälyä tehdessä on helppoa valmistaa epäreilu, täydellinen vastustaja, joka tietää pelaajan kaikki liikkeet sekä ampuu luonnottoman tarkasti. On tärkeää pitää mielessä, mitä kukin tekoäly tarvitsee ja pyrkiä karsimaan turhat, monimutkaiset ratkaisut ja pyrkiä yksinkertaisempaan malliin.

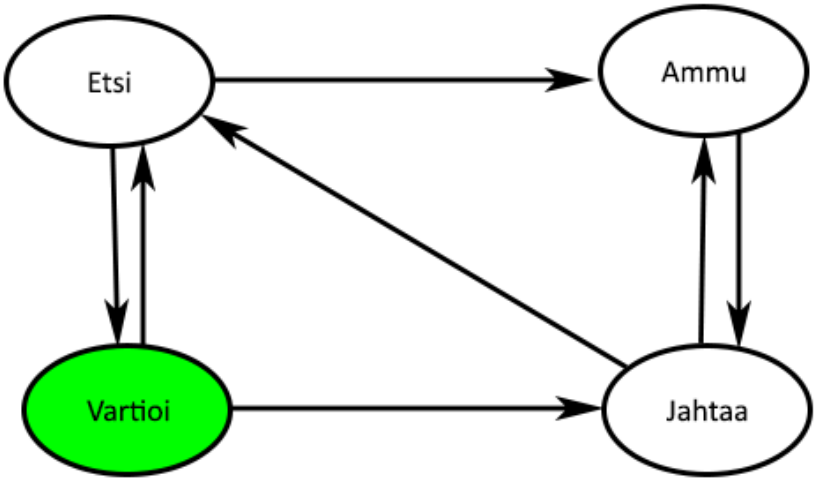
Tekoälyn tarkoitus on matkia ihmistä samankaltaisessa tilanteessa. Ihmiset ovat kuitenkin välillä arvaamattomia, mikä saattaa johtaa matemaattisesti epäloogisiin ratkaisuihin. Tällaisen arvaamattomuuden ohjelmointi on vaikeaa. On kuitenkin mahdollista luoda illuusio tällaisesta arvaamattomuudesta tekemällä tarpeeksi sääntöjä joiden perusteella tekoäly toimii, mutta lisätä sääntöjen sisään satunnaisia arvoja. (Scott 2002, 16-18)

Pelaaja yleensä pystyy havaitsemaan tilanteet joissa tekoäly toimii luonnottomasti tai epäilyttävästi. Tästä syystä on tärkeää miettiä mikä on pelaajan kannalta reilua ja ennen kaikkea viihdyttävää. Tämä tarkoittaa minkä tahansa ohjelmakoodin käyttäminen olevan hyväksyttävää, jos sen lopputulos on viihdyttävä pelaajalle. (Lidén 2004, 41-42.)

Suuri tekijä NPC:n viihdyttävyyden kannalta on sen ennalta-arvattavuus. Kun tekoäly toimii joka pelikerta samoin, laskee pelin viihdyttävyys huomattavasti. On siis asetettava jonkinlainen persoonallisuus. Tekoälylle voi asettaa erilaiset taitotasot muuttamalla yksilöiden osumistarkkuutta tai väistämistaitoja. Toisia tapoja yksilöidä eri NPC:eitä on antaa jokaiselle oma suosikki ase tai alue johon hakeutua mieluummin. (Jamieson 2015)

7 TEKOÄLYN OHJELMOINTI JA SUUNNITTELU

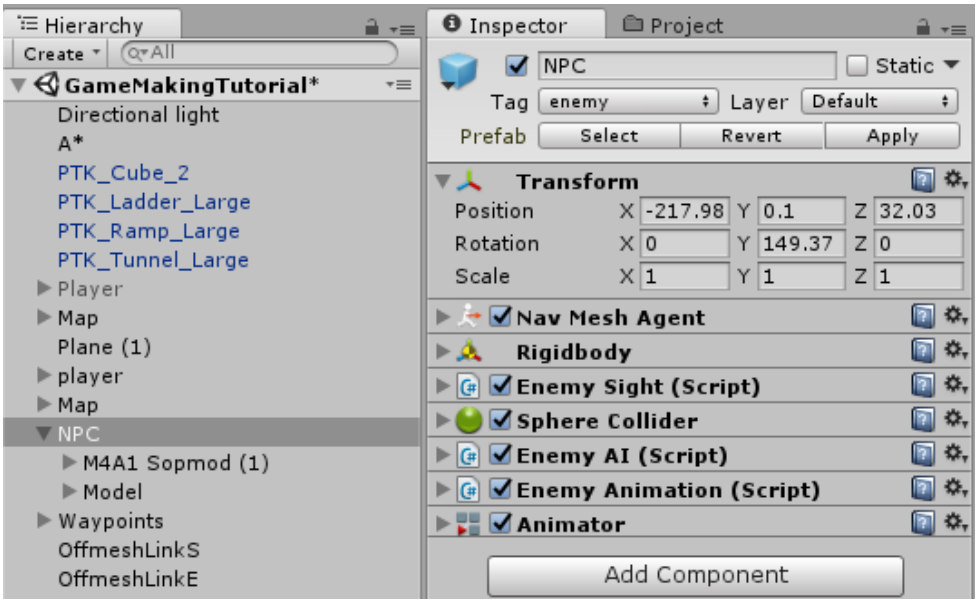
Hahmon tekoälyä suunnitellessa tulee aluksi arvioida mikä hahmon tarkoitus on pelissä. Tekoälyn tavoite esimerkkipelissä on vartioida aluetta. Tavoitteen saavuttamiseksi tekoälylle tarvitsee antaa joukko toimintoja, jotka kannattaa lajitella diagrammiksi.



Kuva 20 Äärellisen automaatin diagrammi

Kuvan 20 tekoälyn perustila on merkitty vihreällä. Vartioi-tilasta siirrytään joko kuuloaistin mukaan Etsi-tilaan tai näköaistin mukaan Jahtaa-tilaan. Näistä tiloista siirrytään Ammu-tilaan kun pelaaja on näkökentässä ja kannattavalla etäisyydellä ampua. Etäisyyden ollessa liian suuri tekoäly pyrkii lähemmäksi pelaajaa juoksemalla Jahtaa-tilassa. Tällöin pelaaja on voinut juosta pois näköyhteydestä, jolloin tekoälyn tulee tutkia lähiympäristö Etsi-tilassa. Kun pelaajaa ei löydy, tekoäly palaa Vartioi-tilaan ja liikkuu ympäri pelikenttää käyttäen sijoitettuja etappeja.

Unity Game Enginen ohjelmointi tapahtui käyttämällä MonoDevelop ohjelmaa ja C#-ohjelmointikieltä. Ohjelmakoodit lisättiin NPC-hahmoon, jonka lapsiobjekteina olivat ase sekä hahmon oma mallikappale.



Kuva 21 NPC hierarkkia

7.1 Reitinhaku ja liikkuminen

Reitinhaun päättää Unityn Navigation -mesh toimija, jolle määrätään eri ohjelmakoodeissa haluamat koordinaatit johon hahmoon liitetty toimija liikkuu.

Taulukko 2 Vartiointi ohjelmointikoodi

```
void doPatrol() {
    nav.speed = patrolSpeed;
    if (nav.destination == lastPlayerSighting.resetPosition || nav.remai
ningDistance < nav.stoppingDistance) {
        patrolTimer += Time.deltaTime;
        if (patrolTimer >= patrolWaitTime) {
            if (wayPointIndex == patrolWaypoints.Length - 1)
               wayPointIndex = 0;
            else wayPointIndex++;
                patrolTimer = 0 f;
        }
    } else
         patrolTimer = 0 f;
    if (enemySight.playerInSight || lastPlayerSighting.position != lastP
layerSighting.resetPosition) {
        curState = AIstate.Chase;
    }
    nav.destination = patrolWaypoints[wayPointIndex].position;
}
```

Vartiointi toimii asettamalla pelikentälle etappeja joihin toimija suunnistaa. Päästyään tarvittavalle etäisyydelle hahmo tulkitsee päässeensä perille ja odottaa hetken, minkä jälkeen hahmo asettaa seuraavan etapin koordinaatit kohteeksi. Ohjelmassa on myös siirtyminen jahtaamiseen, jos pelaaja tulee jonkin tekoälyn näkökenttään.

7.2 Havainnointi

Tekoälylle tulee ohjelmoida tapa tunnistaa pelaaja eli näköaisti sekä kuuloaisti. Taulukon 3 mukaan kun pelaaja on NPC:n edessä ja sille määritetyn alueen sisällä, NPC merkkaa pelaajan olevan näkyvissä. Ohjelmakoodi sisältää myös lastPlayerSighting-kohdan, joka päivittää yleiseen ohjelmakoodiin pelaajan sijainnin, jolloin muutkin NPC:t pystyvät tulemaan pelaajan luokse. Tämä kuvaa NPC:n kykyä kommunikoimaan pelaajan sijainnin muille. Pelaajan poistuessa alueen sisältä merkitään pelaajan olevan pois näkökentästä.

Taulukko 3 NPC:n näkökyky (Unity. 2014. Tutorials – Stealth Project)

```
void OnTriggerStay(Collider other) {
    if (other.gameObject == player) {
        playerInSight = false;
      Vector3 direction = other.transform.position - transform.po
sition;
        float angle = Vector3.Angle(direction, transform.forward)
;
        if (angle < fOVAngle * 0.5 f) {
            RaycastHit hit;
            if (Physics.Raycast(transform.position + transform.up
, direction.normalized, out hit, col.radius)) {
                if (hit.collider.CompareTag("player"))  {
                    playerInSight = true;
              lastPlayerSighting.position = player.transform.posi
tion;
                }
            }
        }
    }
}
void OnTriggerExit(Collider other) {
    if (other.gameObject == player) playerInSight = false;
}
```

Pelaajan tehdessä ääntä on laskettava, kuinka pitkä matka äänellä on kuljettavana. Etäisyyden mittaamiseksi merkitään matka navigation mesh -toimijan kuljettavaksi. Kulkureitin pituuden mittaamiseksi jokainen käännös merkitsee matkalle etapin joiden välinen matka lasketaan yhteen. Kun palautettu matka pathLength on pienempi kuin NPC:lle määritetty kuuloetäisyys, voidaan merkitä pelaajan sijainti sille NPC:lle tutkittavaksi.

Taulukko 4 Kuulomatkan mittaus (Unity. 2014. Tutorials – Stealth Project)

```
public float CalculatePathLength(Vector3 targetPosition) {
    NavMeshPath path = new NavMeshPath();
    if (nav.enabled)
       nav.CalculatePath(targetPosition, path);
    Vector3[] allWayPoints = new Vector3[path.corners.Length + 2]
;
    allWayPoints[0] = transform.position;
    allWayPoints[allWayPoints.Length - 1] = targetPosition;
    for (int i = 0; i < path.corners.Length; i++) {
        allWayPoints[i + 1] = path.corners[i];
    }
    float pathLength = 0 f;
    for (int i = 0; i < allWayPoints.Length - 1; i++) {
        pathLength += Vector3.Distance(allWayPoints[i], allWayPoi
nts[i + 1]);
    }
    return pathLength;
}
```

Koska ohjelmakoodi käyttää navigation meshiä on syytä varmistaa hahmon navigation mesh -toimijan olevan käytössä. Toimijan ollessa pois käytöstä, ei matkaa tarvitse laskea, jolloin vähennetään pelin prosessointi taakkaa.

7.3 **Kääntyminen**

NPC:n nähtyään tekoälyn tulisi kääntyä pelaaja kohti. Tähän tarvitaan edellisen osion playerInSight-parametria, joka kertoo onko pelaaja näkyvissä.

Taulukko 5 Kääntyminen (Unity. 2014. Tutorials – Stealth Project)

```
void NavAnimSetup() {
    float speed;
    float angle;
    if (enemySight.playerInSight) {
        speed = 0 f;
        angle = FindAngle(transform.forward, player.position - transform
.position, transform.up);
        transform.LookAt(player.position + nav.desiredVelocity);
        enemyWeapon.transform.LookAt(player.position);
    } else {
        speed = Vector3.Project(nav.desiredVelocity, transform.forward).
magnitude;
        angle = FindAngle(transform.forward, nav.desiredVelocity, transf
orm.up);
        enemyWeapon.transform.LookAt(transform.forward);
        if (Mathf.Abs(angle) < deadZone) {
            transform.LookAt(transform.position + nav.desiredVelocity);
            angle = 0 f;
        }
    }
}
float FindAngle(Vector3 fromVector, Vector3 toVector, Vector3 upVector)
{
    if (toVector == Vector3.zero)
      return 0 f;
    float angle = Vector3.Angle(fromVector, toVector);
    Vector3 normal = Vector3.Cross(fromVector, toVector);
    angle *= Mathf.Sign(Vector3.Dot(normal, upVector));
    angle *= Mathf.Deg2Rad;
    return angle;
}
}
```

7.4 **Jahtaaminen**

NPC:n liikkumisen ja havainnoinnin lisäämisen jälkeen tekoäly pystyy nyt liikkumaan pelaajaa kohti kun hänet on nähty. Jahtaamisen tarkoituksena on siirtyä ampumisetäisyydelle ja siirtyä sen jälkeen ampumiseen. ”Jahtaa” tilalla on myös toinen tarkoitus, siirtyä paikkaan jossa pelaaja nähtiin viimeksi. Tämä tapahtuu vain kun pelaaja on poistunut ampumisetäisyydeltä tai näkyvistä.

Taulukko 6 Jahtaaminen (Unity. 2014. Tutorials – Stealth Project)

```
void doChase() {
    Vector3 sightingDeltaPos = enemySight.personalLastSighting - transfo
rm.position;
    if (sightingDeltaPos.sqrMagnitude > 12 f)
       nav.destination = enemySight.personalLastSighting;
    nav.speed = chaseSpeed;
    if (nav.remainingDistance < nav.stoppingDistance) {
            StartCoroutine(SearchCoroutine());
        chaseTimer += Time.deltaTime;
        transform.LookAt(transform.position + nav.desiredVelocity);
        if (chaseTimer > chaseWaitTime) {
            lastPlayerSighting.position = lastPlayerSighting.resetPositi
on;
            enemySight.personalLastSighting = lastPlayerSighting.resetPo
sition;
            chaseTimer = 0 f;
            curState = AIstate.Search;
        }
    } else {
        chaseTimer = 0 f;
    }
    if (enemySight.playerInSight) curState = AIstate.Shoot;
}
```

Etsintäohjelma myös laukaisee operaation ”SearchCoroutine”, joka auttaa tekoälyä arvioimaan mahdollisen suunnan josta etsiä pelaajaa, mikäli hän katoaa näkyvistä jahtaamisen jälkeen.

Taulukko 7 SearchCoroutine

```
IEnumerator SearchCoroutine() {
        while (curState == AIstate.Chase) {
            pos3SecAgo = pos2SecAgo;
            pos2SecAgo = pos1SecAgo;
            pos1SecAgo = posNow;
            posNow = player.transform.position;
            pos3SecFromNow = posNow + (posNow - pos3SecAgo); // Wait for
 1 second:
            yield return new WaitForSeconds(1);
        }
```

Operaatio merkitsee pelaajan koordinaatit sekunnin välein ja laskee paikan jossa pelaaja mahdollisesti olisi kolmen sekunnin jälkeen kun jahtaaminen on loppunut. Vaikka tekoälylle kerrotaan suoraan pelaajan paikka, on mahdollista naamioida se älykkyytenä kun arvojen tarkkuuksia vähennetään. Tässä tapauksessa pelaaja voi tulkita tekoälyn nähneen suunnan johon pelaaja juoksi ja näin ollen tarkistaa onko pelaaja mahdollisesti piiloutunut lähettyville.

7.5 **Etsintä**

Tekoälyn kadottaessa pelaajan jahtauksen jälkeen on syytä antaa jonkinlainen arvio pelaajan mahdollisesta olinpaikasta. Saatuaan arvion tekoäly liikkuu arvioidulle paikalle ja katselee ympärilleen. Tämän jälkeen voidaan merkitä pelaaja kadotetuksi ja käskeä kaikkia tekoälyjä siirtymään takaisin vartiointi tilaan.

Taulukko 8 Etsiminen (Unity. 2014. Tutorials – Stealth Project)

```
void doSearch() {
    nav.speed = searchSpeed;
    nav.destination = pos3SecFromNow;
    StopCoroutine(SearchCoroutine());
    if (enemySight.playerInSight)
      curState = AIstate.Shoot;
    if (nav.remainingDistance < nav.stoppingDistance && !searchCoStart)
      curState = AIstate.Patrol;
}
```

Etsintätila myös sammuttaa ”SearchCoroutine” operaation, jotta operaatio ei turhaan kuluttaisi prosessointitehoja toimimalla aina taustalla.

7.6 **Päätöksenteko**

EnemyAI päättää kaikkien toimintojen väliltä mitä toimintoa suoritettaan. Äärellinen automaatti siirtyy eri tilojen välillä käyttäen luetteloa mahdollisista tiloista, josta tilat voivat valita haluamansa tilan johon siirtyä käyttäen curState-parametria.

Taulukko 9 Lista tiloista

```
public enum AIstate {
Shoot,
Chase,
Patrol,
Search,
}
```

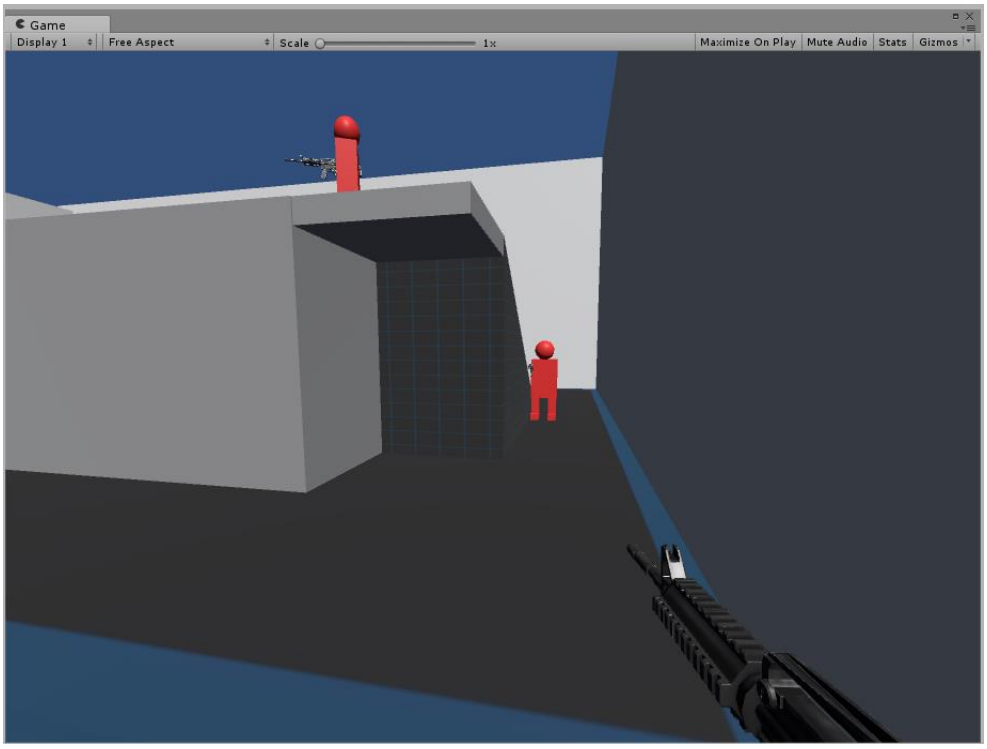
Taulukko 10 Tilojen toimintojen päivittäminen

```
void Update() {
    switch (curState) {
        case AIstate.Patrol:
            doPatrol();
            break;
        case AIstate.Chase:
            doChase();
            break;
        case AIstate.Search:
            doSearch();
            break;
        case AIstate.Shoot:
            doShoot();
            break;
    }
}
```

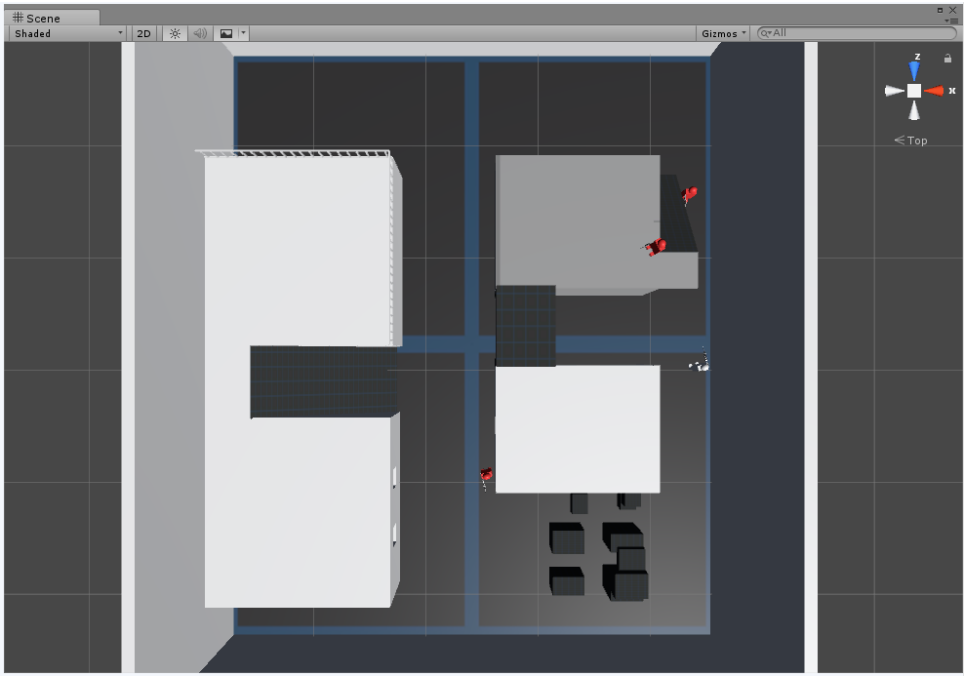
Ohjelma päivittää valittua curState-tilaa johon on mahdollista liittää monia toimintoja. Tässä tapauksessa tiloilla on vain yksi toiminto, mutta esimerkiksi Shoot-tila voisi olla yleinen hyökkäystila, joka valitsisi hyökkäystavan riippuen pelaajan ja tekoälyn välisestä matkasta. Tällöin hyökkääminen voisi olla joko lyöminen tai ampuminen.

8 ESIMERKKIPELIN LÄPIKÄYNTI

Kun esimerkkipelin osat olivat kasattu ja ohjelmakoodit liitetty, tuli hahmoihin liittyviä parametreja säätää testipelaamalla. Asetin tekoälyille muutaman etapin jota vartioida. Testipelaamisessa tuli helposti huomattua hahmojen nopeus ja tärkeys kyseisessä kentässä. Jos liikkumisnopeus oli suuri, oli mahdotonta pysyä paikoilla sillä vastustaja pääsi helposti hyökkäämään sivusta. Kun ammuin samalla tasolla olevaa vastustajaa (Kuva 22), ylemmällä tasolla sekä selkäpuolelle suuntaava vastustaja (Kuva 23) näkivät nopeasti ampua pelaaja monesta eri suunnasta.



Kuva 22. Pelaajan näkökulma



Kuva 23 Yleiskuva

Tämä antoi hyviä vihjeitä, kuinka tasapainottaa peliä. Esimerkiksi hahmojen nopeutta tuli pudottaa ja aseiden tulivoimaa tuli lisätä. Tulivoiman lisäämisen tarkoituksena on muuttaa peliä nopeampi tempoiseksi sekä antaa suurempi etu hahmolle, joka huomaa ja ampuu ensimmäisenä vastustajaa. Muina vaihtoehtoina tasapainottamiseen olivat suojien määrän lisääminen, jotta näköyhteyksien katkaiseminen olisi helpompaa tai vastustajien vähentäminen.

Kehittämiseksi tulisi seuraavaksi vahingonmäärän vähentäminen tai lisääminen, riippuen mihin kehonosaan hahmo ottaa osuman. Tekoäly tähtäämiseen tulisi myös liittää epätarkkuusarvo, sillä tällä hetkellä käytössä oli vain aseen oma rekyyli.

9 YHTEENVETO

Arvioitavaksi tulee käyttäytymismallina ollut äärellinen automaatti, pelimoottorina käytetty Unity Game Engine sekä valmiiksi saatu tekoäly. Huomioitavana oli helppokäyttöisyys, kulutettu aika ja toimivuus.

Äärellinen automaatti toimi käyttötarkoitukseeni hyvin sen yksinkertaisuuden ansiosta. Tekoälyn tilojen vähäisyyden sekä yksinkertaisuuden takia tilasiirtymät eivät kuluttaneet liikaa aikaa eikä siirtymät olleet liian monimutkaisia. Pystyin kuitenkin näkemään, kuinka nopeasti tilojen määrän kasvaessa tilasiirtymistä tulisi suuri päänvaiva yksinkertaisella äärellisellä automaatilla. Tästä syystä en suosittelisi pysymään yksinkertaisessa äärellisessä automaatissa vaan siirtyisin käyttämään mieluummin hierarkkista äärellistä automaattia tai käyttäytymispuuta.

Unity Game Engine toimi pelimoottorina hyvin. Unitylla on paljon dokumentaatiota sen toiminnollisuuksista sekä suuri valikoima kursseja. Unityn toiminnot olivat jo ennen tuttuja itselleni, mikä nopeutti työn edistymistä. Unity Storen mallikappaleet nopeuttivat myös työtä, sillä tarkoituksena ei ollut tehdä kokonaan omaa peliä vaan toimivaa esimerkkiä opinnäytetyön pohjaksi. Unityn ohjelmakoodi käytti myös ennestään tuttua C#-ohjelmointikieltä, joka muutenkin mielletään helppolukuiseksi ohjelmointikieleksi. Toisena vaihtoehtona olisi ollut javascript.

Tekoälyn tilat jäivät perustasolle, eikä aikaa jäänyt lisätä monia tärkeitä asioita tekoälyn reiluuden kannalta, kuten reagointinopeutta tai tähtäämistarkkuutta. Näiden asioiden lisääminen vaatisi paljon enemmän aikaa ja kokemusta kuin alun perin arvioin.

Opinnäytetyö antaa kuitenkin eri perusteita käyttää äärellistä automaattia tai käyttäytymispuuta eikä toimi pelkkänä ohjelmointikurssina. Työ myös tuo esille huomioitavia asioita ensimmäisen persoonan peliä suunnitellessa, kuten tekoälyn reiluus ja pelikentän rakenne.

LÄHTEET

Bevilacqua, F. 2013. Finite-State Machines: Theory and Implementation. Envatotuts+. Haettu 27.9.2017. https://gamedevelopment.tutsplus.com/tutorials/finite-state-machines-theory-and-implementation--gamedev-11867

Fu, D. & Houlette R. (2004). The Ultimate Guide to FSMs in Games. Teoksessa S. Rabin (toim.) AI Game Programming Wisdom 2. Hingham: Charles River Media, 286 – 288.

Gesota, R. 2016. How to ace the FINITE State Machine Model in Unity AI Implementation. Haettu 27.9.2017. http://www.theappguruz.com/blog/ai-implementation-using-finite-state-machine-model

Jamieson, D. 2015. Making AI Fun: When Good Enough is Good Enough. Envatotuts+. Haettu 27.9.2017. https://gamedevelopment.tutsplus.com/articles/making-ai-fun-when-good-enough-is-good-enough--cms-23460

Lidén, L. (2004). Artificial Stupidity: The Art of Intentional Mistakes. Teoksessa S. Rabin (toim.) AI Game Programming Wisdom 2. Hingham: Charles River Media, 41 − 42.

Millington, I. & Funge, J. 2009. Artificial Intelligence for Games. CRC Press.

Polson, T. (2012). *What makes a video game fun: An investigation into the expectations of playing First Person Shooter video games.* Opinnäytetyö. Liiketoimintajohtamisen maisteritutkinto. University of Otago. Haettu 28.9.2017. http://hdl.handle.net/10523/2238

Scott, B. (2002). The Illusion of Intelligence. Teoksessa S. Rabin (toim.) AI Game Programming Wisdom. Hingham: Charles River Media, 16 − 18.

Tvtropes. The Computer Is a Cheating Bastard. Haettu. 28.9.2017. http://tvtropes.org/pmwiki/pmwiki.php/Main/TheComputerIsACheatingBastard

Unity. 2014. Tutorials – Stealth Project. Päivitetty 30.6.2014. Haettu 22.9.2017. https://www.youtube.com/playlist?list=PLX2vGYjWbI0QGyfO8PKY1pC8xcRb0X-nP

Unity. Inner Workings of the Navigation System. Haettu 27.9.2017. https://docs.unity3d.com/Manual/nav-InnerWorkings.html

Unity. Unity Analytics. Haettu 30.9.2017. https://unity3d.com/unity/features/analytics

Unreal Engine. How Unreal Engine 4 Behavior Trees Differ. Haettu 29.9.2017. https://docs.unrealengine.com/latest/INT/Engine/AI/BehaviorTrees/HowUE4BehaviorTreesDiffer/index.html#specialhandlingforconcurrentbehaviors

Wikipedia (n.d.). Counter Strike: Global Offensive. Haettu 28.9.2017. https://en.wikipedia.org/wiki/Counter-Strike:_Global_Offensive

Wikipedia (n.d.). First Person Shooter. Haettu 28.9.2017. https://en.wikipedia.org/wiki/First-person_shooter

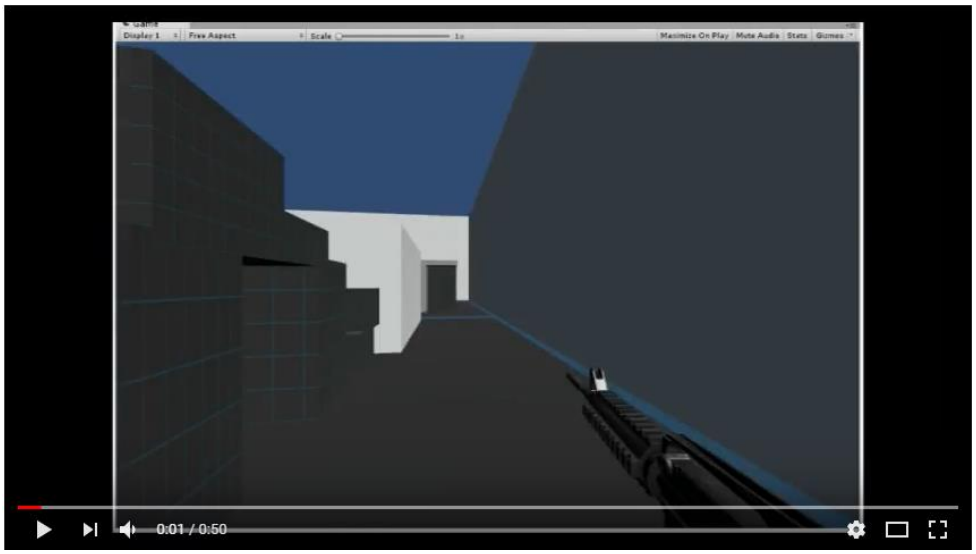
Wikipedia (n.d.). Halo: Combat Evolved. Haettu 28.9.2017. https://en.wikipedia.org/wiki/Halo:_Combat_Evolved

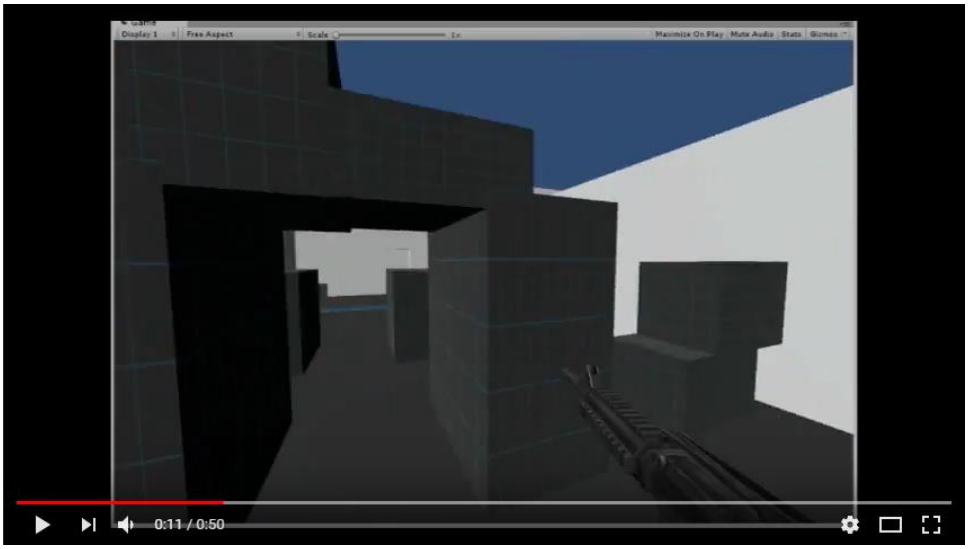
Wikipedia (n.d.). Tom Clancy's Rainbow Six. Siege. Haettu 29.9.2017. https://en.wikipedia.org/wiki/Tom_Clancy%27s_Rainbow_Six_Siege

World of Level Design. 2013. CS:GO How to Design Gameplay Map Layouts (Complete In-Depth Guide). Haettu 27.9.2017. http://www.worldofleveldesign.com/categories/csgo-tutorials/csgo-how-to-design-gameplay-map-layouts.php

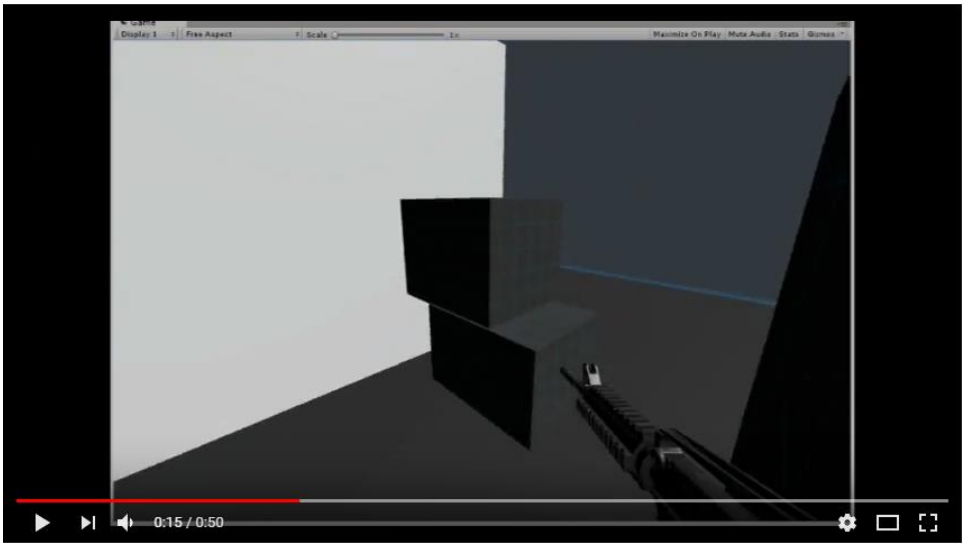
Liite 1/1

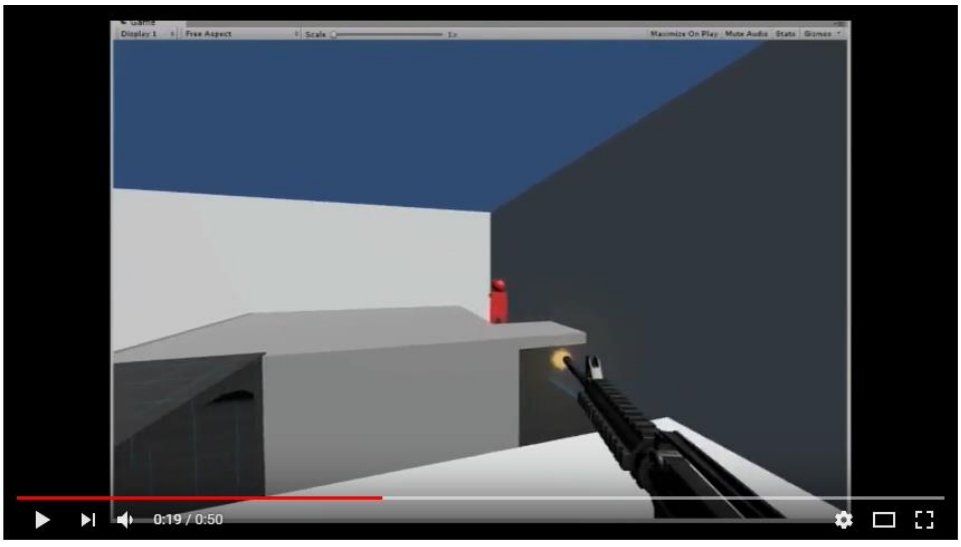
Esimerkkipelin testipelaus
https://www.youtube.com/watch?v=eKVHXrW-Og4&feature=youtu.be



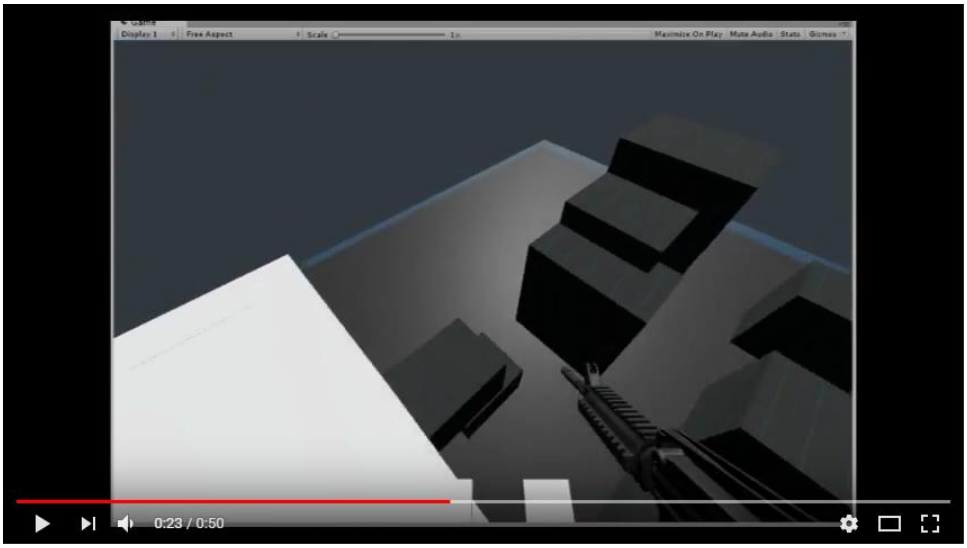


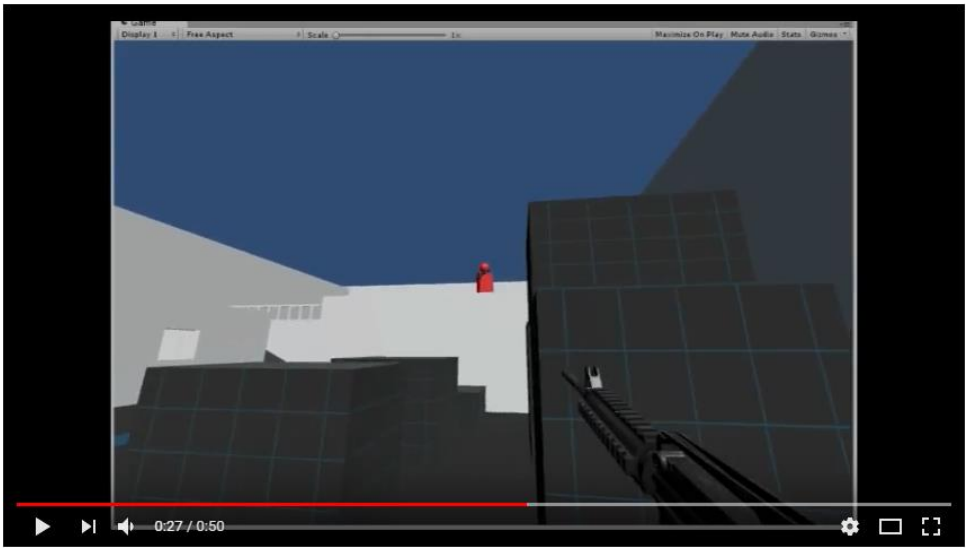
Liite ½

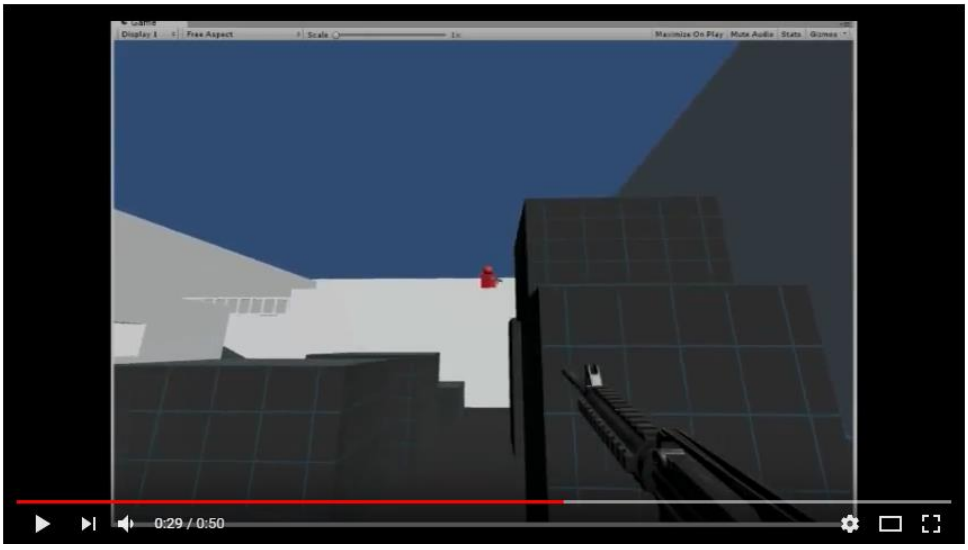




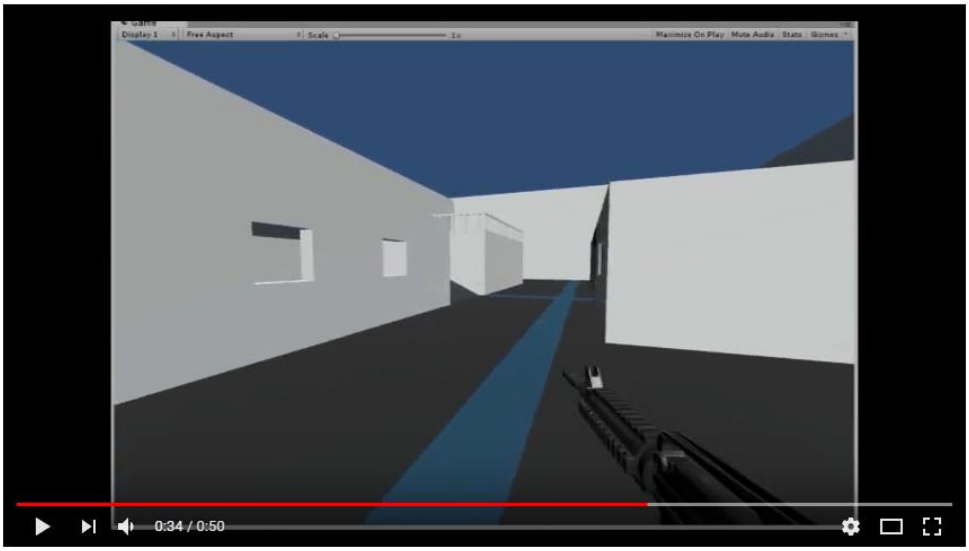
Liite 1/3

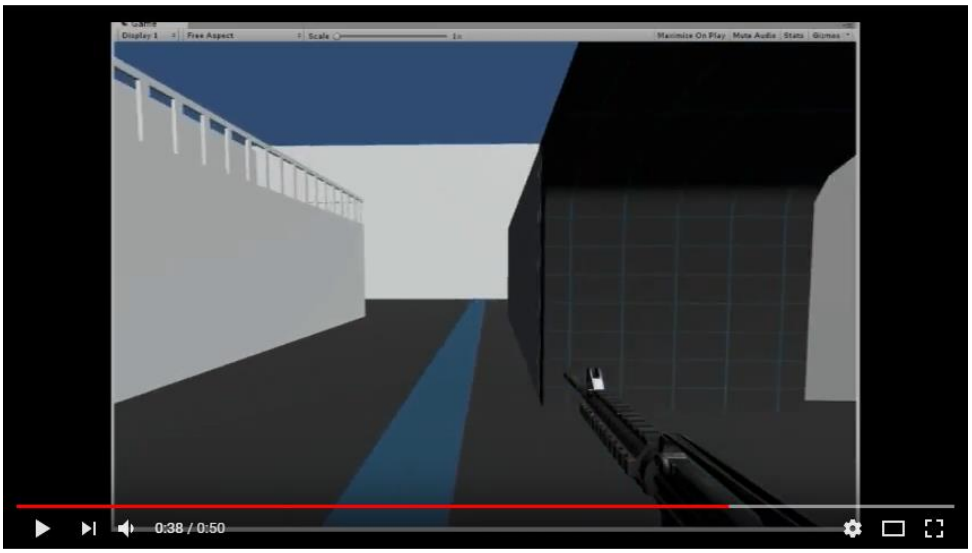


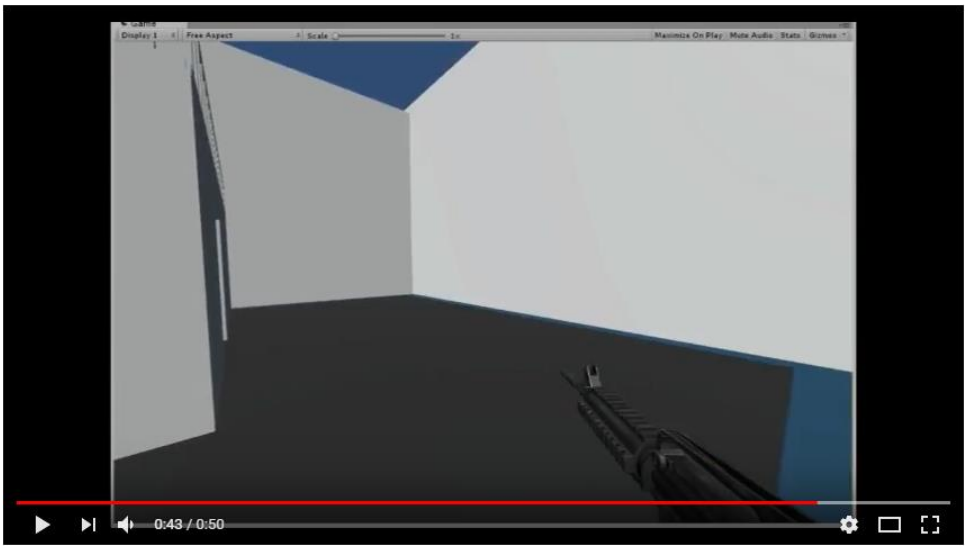




Liite 1/4







Liite 1/5