Miettinen Teemu

2D-pelin kehittäminen Godot Enginellä

GODOT
Game engine

Insinööri (AMK)

Tieto- ja viestintätekniikka

Kevät 2019

Tiivistelmä

Tekijä: Miettinen Teemu

Työn nimi: 2D-pelin kehittäminen Godot Enginellä

Tutkintonimike: Insinööri (AMK), tieto- ja viestintätekniikka

Asiasanat: Godot Engine, pelimoottori, 2D, videopeli

Godot Engine on melko nuori ja huomiotta jäänyt pelimoottori. Silti se varsinkin vuosina 2017−2019 on kehittynyt melkoisesti ja on pian varteenotettava kilpailija jo tunnettujen Unity3D:n ja Unreal Engine 4:n rinnalla. Godot Engine on kevytrakenteinen ja monimuotoinen 2D- ja 3D-pelimoottori, mutta erityisesti se on keskittynyt 2D-pelien kehittämiseen. Kuitenkin Godot Enginellä on mahdollista tehdä hyvinkin näyttäviä 3D-pelejä, mutta grafiikan taso ei yllä kuitenkaan vielä ns. AAA-pelimoottorien tasolle. Godot Enginen suurin vahvuus onkin sen kevytrakenteisuus, helppokäyttöisyys ja laaja alustatuki. Kokemattomankin ohjelmoijan on helppo ottaa pelimoottori käyttöön. Helppokäyttöisyys näkyy myös GDScriptissä, eli Godot Enginen omassa ohjelmointikielessä, sillä se on helppo oppia yksinkertaisen syntaksinsa ansiosta. Godot Enginessä on myös tuki muille ohjelmointikielille, kuten C# ja C++.

Godot Engine koostuu pääosin kahdesta peruskomponentista; *scene* eli näkymä ja *node* eli solmu. Näkymän on suurempi kokonaisuus, joka koostuu useasta eri solmusta. Solmu taas on näkymän oma komponentti, joka voi sisältää oman skriptinsä ja omia ominaisuuksiaan. Näkymiin voi kuitenkin instansoida muita näkymiä ja näin ollen saada myös instansoidun näkymän solmut käyttöön. Näkymien instansointi toisiinsa johtaa siihen, että projektit saavat puumaisen rakenteen. Puumaisen rakenteen myötä projekteja on helppo havainnollistaa, että suunnitella.

Pelinkehitykseen Godot Engine tarjoaa monenlaisia ratkaisuja ja oikeanlainen toteutustapa riippuu täysin meneillään olevasta peliprojektista. Lähtökohtaisesti projekti on syytä jakaa eri näkymiin, kuten pelaajanäkymään, kenttänäkymään, käyttöliittymänäkymään ja AI-näkymään. Tämänkaltainen lähestymismalli mahdollistaa näkymien instansoimisen omiin puuhaaroihin, mikä taas helpottaa komponenttien hahmottamista ja toteutusta. Solmutyyppien suunnittelu on myös syytä toteuttaa huolella, sillä tietyntyyppiset solmut tarjoavat eri ominaisuuksia, kuten erilaisia signaaleja. Signaalien avulla esimerkiksi törmäystarkastukset on helppo toteuttaa. AI:n osalta Godot Engine ei tarjoa vielä kovin laajoja kokonaisuuksia, vaan tekoälyn joutuu suunnittelemaan suurilta osin itse. Reitinhaun osalta Godot Engine tarjoaa kuitenkin hyviä toteutusvaihtoehtoja, kuten navmesh-navigoinnin ja A*-reitinhaun.

Godot Engine on kokonaisuutena kelpo pelimoottori käytettäväksi 2D-peliprojekteihin ja aloitteleville peliohjelmoijille helppo oppia. Kaupallisiin projekteihin Godot Engine tarjoaa myös kustannustehokkaan pelimoottorin, sillä MIT-lisenssin alla oleva moottori on käytännössä ilmainen. Suunnitteluvaiheessa on vain syytä muistaa Godot Enginen erilaisuus muihin moottoreihin nähden, ja suunnitella peliprojektit moottorin ominaisuuksien mukaan ja niitä hyödyntäen.

Abstract

Author: Miettinen Teemu

Title of the Publication: 2D Game Development with Godot Engine

Degree Title: Bachelor of Engineering, Game Technologies

Keywords: Godot Engine, game engine, 2D, videogame

Godot Engine is quite young and disregarded game engine. Still it has been improving a lot in 2017 – 2019 and it will be soon worthy competitor for Unity3D and Unreal Engine 4. Godot Engine is light weighted and diverse 2D and 3D game engine, but it is built especially for 2D games. However, it is possible to do quite polished 3D games with it, yet the graphical output is not so fantastic as with the triple-A game engines. In this Bachelor's thesis, the main focus was in 2D game development functionalities and how these functionalities work in Godot Engine. This game engine provides also its own programming language GDScript, but it also supports more commonly used programming languages like C++ and C#.

Godot Engine is composed of two main components; Scene and Node. Scene is a bigger entity, which contains one or multiple Nodes. Node is its own component in the Scene, and it can contain its own unique script and other functionalities. However, there is a possibility to instance a Scene to another Scene. In this way, Scene can get other Scenes Nodes and their functionalities. This instance system results a tree like structure to the project, which makes projects easy to plan and demonstrate.

Godot Engine offers many solutions to build games and there is not one right solution. But as a baseline, a project should spare to different scenes, like a player scene, a map scene, a UI scene and AI scene. This manner of an approach allows these scenes to be instanced in the tree like hierarchy. Node types should also be designed in carefully, because different type of nodes allows different properties like special signals. For example, collision detections are very easily done by these signals. Thus, for fast and easy approaches nodes should be planned with care. AI systems are lacking in Godot Engine, so developers have to build these kind of systems themselves. However, for AI-navigation Godot Engine provides many easy in build solutions like navmeshes and A*-navigation.

Overall, Godot Engine is a very flexible and modern game engine, at least for 2D games. 3D side is, however, improving all the time. In future, by today's statics, Godot Engine may not replace, but become a considerable alternative along with Unity3D, Unreal Engine and other triple-A game engines.

Sisällys

Sanasto

AI	Artificial Intelligence, tekoäly.
Kuvatiilimalli	Tileset, 2D-kuvakkeista luotu kokoelma.
Kuvatiiliruudukko	Tilemap, Tilesetin 2D-kuvakkeista luotu kartta.
NPC	Non-player Character, tekoälyn käyttämä hahmo.
Näkymä	Scene, Godot Enginen komponentti, joka sisältää solmuja.
Olio-ohjelmointi	Object-oriented programming, ohjelmoinnin ohjelmointiparadigma.
Open GL	Open Graphics Library, avoin ohjelmointirajapinta.
Reittipiste	Waypoint, tekoälyn käyttämä koordinaatiston piste, johon tekoäly pyrkii siirtymään.
Re-parentointi	Reparent, komponentin uudelleen järjestäminen hierarkiassa.
Sapluuna	Template, valmiita komponentteja tai projektin pohjia.
Shader	Shader, grafiikkaa muokkaava varjostin.
Skripti	Script, koodia sisältävä tiedosto.
Solmu	Node, Godot Enginen komponentti.
Sprite	Sprite, 2D-kuva.
Säilö	Repository, ohjelmistoprojektien versionhallinta.

1 Johdanto

Pelialalla on runsaasti erilaisia pelimoottoreita, mutta silti suurin suosio menee Unity3D:lle ja Unreal Engine 4:lle. Godot Engine on vuosina 2017-2019 alkanut nousta näiden kahden pelimoottorin rinnalle ominaisuuksiensa puolesta, mutta käyttäjämäärät eivät silti ole nousseet kyseisten pelimoottoreiden tasolle. Tämän opinnäytetyön ajatuksena on tuoda Godot Enginelle näkyvyyttä ja tarjota perusteltuja vaihtoehtoja perinteisille pelimoottoreille. Mikään pelimoottori ei ole yksiselitteisesti toista pelimoottoria parempi, mutta eri osa-alueilla, kuten käytettävyydessä, komponenttien toteutuksessa ja käyttäjälisensseissä, on suuriakin eroja.

Opinnäytetyön tavoitteena on tutkia, miten Godot Engine soveltuu 2D-pelien kehittämiseen ja miten 2D-pelien ominaisuuksia voidaan kyseisellä pelimoottorilla toteuttaa. Opinnäytetyön ajatuksena on esitellä Godot Enginen peruselementit, 2D-pelin rakenteen suunnittelu Godot Enginellä sekä 2D-pelien peruskomponenttien toteutus kyseisellä pelimoottorilla. Opinnäytetyön aikana kehitettiin 2D-peliprojektia, mikä mahdollisti Godot Enginen ominaisuuksien testaamisen käytännössä. Opinnäytetyön ajatuksena on myös tuoda esille Godot Enginen vahvuuksia ja heikkouksia. Opinnäytetyö tutkii myös, mitä uutta Godot Engine tarjoaa videopelien kehittämiseen. Opinnäytetyön tarkoituksena ei ole olla ohje 2D-pelin toteuttamiseen Godot Enginellä, mutta se tarjoaa vinkkejä, miten 2D-peliin liittyviä komponentteja voidaan toteuttaa kyseisellä moottorilla.

2 Godot Engine

2.1 Yleistä

Godot Engine on 2D- ja 3D-pelimoottori, jonka kehitystyö on aloitettu kaupallisena ja suljettuna vuonna 2007, mutta se julkaistiin MIT-lisenssillä vuonna 2014 [1.]. Godot Enginen luojana voidaan pitää Ariel Manzuria ja Juan Linietskyä, mutta moottorin kehitystyöhön on osallistunut myös lukuisa joukko vapaaehtoisia eli ns. Godot-yhteisö. Kyseinen pelimoottori toimiikin pääosin yksittäisten sponsorien ja lahjoitusten myötä sekä tietysti vapaaehtoisella toiminnalla. Pelimoottorin ollessa MIT-lisenssin alla sitä voi käyttää myös suljetuissa kaupallisissa projekteissa. Huomioitavaa on myös, että pelimoottorin käytöstä ei tarvitse maksaa, eikä pelimoottorilla toteutettujen pelien myynnistä tarvitse maksaa osuuksia tekijöilleen.

Godot Enginen sponsoriksi voi kuka tahansa ryhtyä, ja pienin mahdollinen summa on viisi euroa per kuukausi. Sponsorina toimiminen avaa rahasumman suuruuden mukaan erilaisia etuuksia. Esimerkiksi viiden euron kuukausisummalla saa ennakkoon pelimoottorin päivitykset ja Discord-keskustelupalveluun oman *”Donor*-roolin” Godot Enginen keskustelukanavalle [2.]. On kuitenkin huomioitavaa, että lahjoitukset ovat täysin vapaaehtoisia, eikä moottorin tavallinen käyttö niitä edellytä. Lisäksi, jos on kiinnostunut auttamaan muuten kuin rahallisesti, on GitHub-palvelusta mahdollista hakea Godot Enginen säilön ja aloittaa moottorin ohjelmointi.

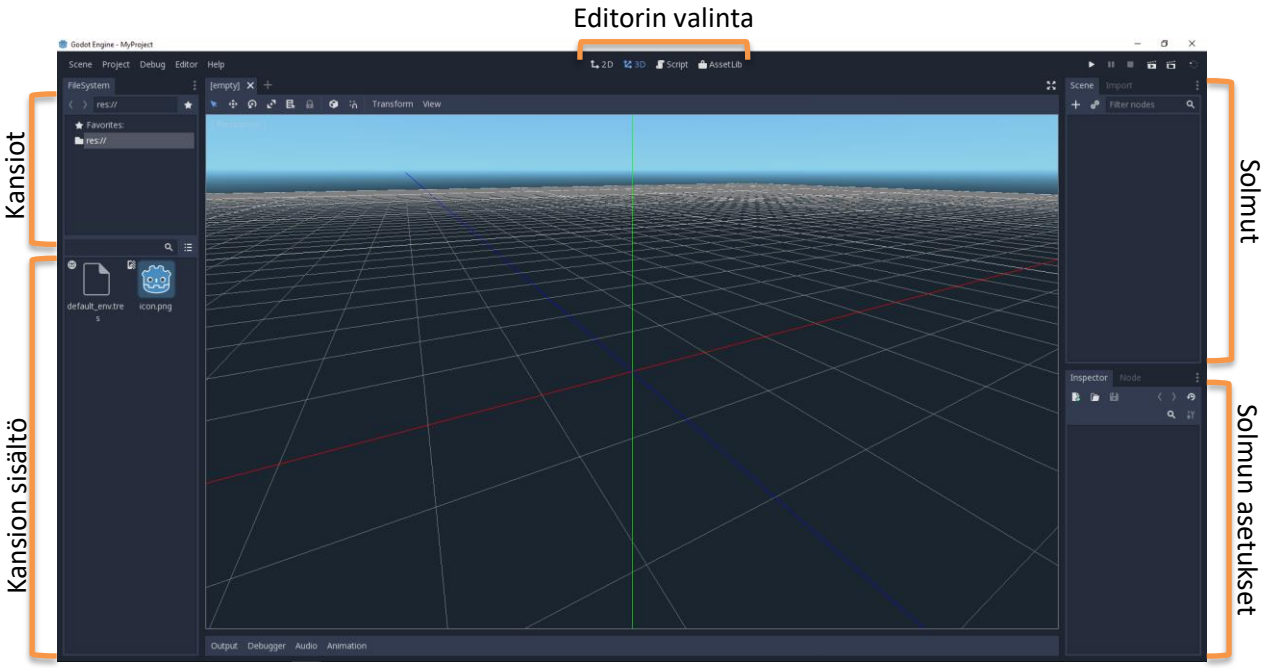
Godot Engine tukee yleisimpiä tietokonekäyttöjärjestelmiä (Linux, MAC OS, Windows), joten moottorin käyttö ei edellytä minkään tietyn käyttöjärjestelmän asentamista. Godot Engine on myös erityisen kevyt pelimoottori, mikä mahdollistaa sen käytön hyvin vaatimattomalla tietokoneellakin. Ainoana edellytyksenä on Open GL 3:n tuki, jota ei vanhemmista tietokoneista löydy. Tällaisessa tilanteessa on kuitenkin mahdollista käyttää Godot Engine 2.1-versiota, joka käyttää vielä Open GL 2-ohjelmointirajapintaa. Tätä ei voi kuitenkaan suositella, sillä 2.1 on hyvin rajallinen ja vaatimaton versio pelimoottorista. Kaikista moottorin versioista on saatavilla 32-bittiset sekä 64-bittiset versiot.

Godot Enginen rakenne eroaa hieman perinteisistä pelimoottoreista, sillä kaikki pelin komponentit koostuvat näkymistä eli *skeneistä* ja solmuista eli *nodeista.* Tällainen toimintamalli mahdollistaa hyvin erilaiset rakenteet eri projekteille, ja suunnitellessa peliprojektia tämä on syytä ottaa huomioon. Godot Engine sisältää myös oman ohjelmointieditorin, joka on tarkoitettu nimenomaan Godot Enginen oman ohjelmointikielen GDScriptin käyttöön.

2.2 Rakenne

Godot Engine käyttää pohjimmiltaan olio-ohjelmointirakennetta, mutta se muodostuu myös joustavasta *skene*(näkymä)-systeemistä ja *node*(solmu)-hierarkiasta. Godot Engine pyrkii pois ohjelmointiin perustuvasta mallista ja haluaa tarjota intuitiivisen pelimoottorin. [3.]

Pelimoottorin käyttö rakentuu pääosin kahdesta palasesta; solmusta ja näkymästä. Nämä palaset esitellään tarkemmin omissa kappaleissaan. Nämä osaset ovat ”ylempää arkkitehtuuria” ja loppukäyttäjä työskentelee näiden osa-alueiden kanssa pääsääntöisesti. ”Alempaa arkkitehtuuria” edustavat taas OS eli Operating System ja Godot Enginen omat serverit. [4.] Mutta näihin loppukäyttäjä harvemmin tarvitsee tutustua tai perehtyä. Itse editorikin on osa pelimoottoria ja se käyttää pelimoottoriin sisään rakennettuja käyttöliittymäpalasia. Editorin käyttöliittymä esitellään kuvassa 1.



Kuva 1. Godot Engine -editori perus-3D-näkymällä.

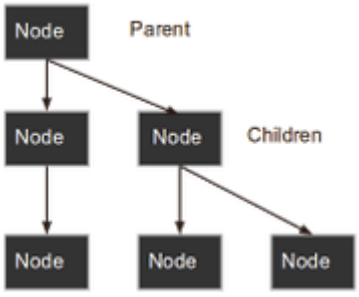
Editori sisältää kaikki pelimoottorin tarvittavat ominaisuudet, ja sen ulkoasua ja ikkunoiden paikkaa voi muokata mielensä mukaan. Teemoja editoriin löytyy valmiiksi asennettuna kuusi kappalaetta, mutta halutessaan voi luoda täysin oman teeman. Editorin käyttöliittymän napit ja informaatio ovat hyvin selkeitä ja yksinkertaisia, joten mitään informaatiotulvaa ei pääse tulemaan, mutta kaikki on kuitenkin hyvin nopeasti saatavilla.

2.3 Solmun ominaisuudet ja toiminta

Käytännössä kaikki, mitä Godot Enginellä tehdään, liittyy nodeihin eli solmuihin. Solmut ovatkin Godot Enginen perusrakennuspalikoita ja sisältävät aina seuraavat ominaisuudet: [5.]

- Solmun nimi
- Editoitavat muuttujat
- Oma skripti
- Oma päivitysfunktio
- Periytettävät ominaisuudet
- Omat lapsisolmut

Lapsisolmujen antaminen onkin erityisen tärkeää Godot Enginessä, sillä tämä muodostaa puumaisen rakenteen, joka esitetään kuvassa 2. Tämä puumainen rakenne mahdollistaa projektin organisoimisen helposti.



Kuva 2 Solmujen puurakenne. Hierarkiassa ylimpänä äitisolmu (*Parent*) ja alempana lapsisolmuja (*Children).* [5.]

Solmuhierarkian suunnittelussa tuleekin ottaa huomioon se, että minkä lapsi kukin solmu on. Lapsisolmut perivät äitisolmulta tiettyjä ominaisuuksia. Esimerkiksi sijainti näkymässä on suhteessa äitisolmun sijaintiin, eli jos äitisolmu liikkuu, liikkuu lapsisolmukin samassa suhteessa. Myös jos äitisolmu poistetaan, poistetaan myös kaikki sen alla olevat lapsisolmut ja mahdolliset ”lapsenlapsi”-solmut.

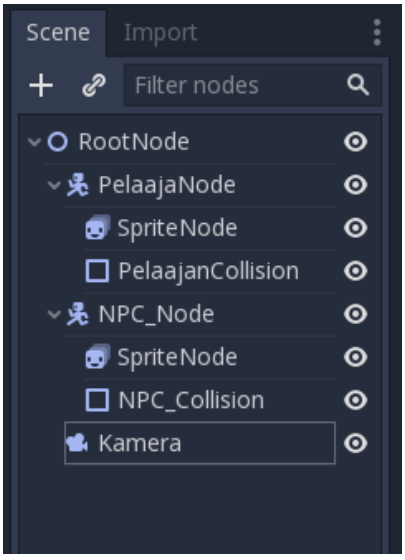
Solmuja voidaan kuitenkin järjestellä ajon aikana uudelleen eli ”*re-parentoida*”, joten hierarkiasysteemissä on pientä joustoa. ”*Re-parentoinnissa*” on kuitenkin huomioitava mahdolliset ”orposolmut” ja näitä solmuja tulee välttää projektin toiminnan varmistamiseksi.

2.4 Näkymän ominaisuudet ja toiminta

Skene eli näkymä on Godot Enginessä solmuja sisältävä kokonaisuus, ja projekti tulee sisältämään ainakin yhden näkymän, koska peliprojektiin tulee aina määritellä aloitusnäkymä. Näkymät voivat solmujen tapaan olla mitä vain ja sisältää mitä vain, eikä Godot Engine estä rakentamasta projektia millään tavalla. Esimerkiksi pelaaja voi olla oma näkymänsä ja sisältää solmuina kaikki siihen kuuluvat komponentit, tai pelaaja voi olla esimerkiksi kenttänäkymään lisättävä solmukomponentti, joka taas sisältäisi lapsisolmuina omat komponenttinsa. Skene-systeemi mahdollistaakin omaan projektiin hyvinkin räätälöidyn lähestymistavan eikä väärää vaihtoehtoa ole.

Samoin kuin solmu, näkymätkin saavat puumaisen rakenteen, koska näkymällä tulee aina olla yksi juurisolmu eli *root node*. Tämän juurisolmun alle taas tulevat kaikki näkymään sisältyvät komponentit. Näkymät ovat myös mahdollista tallentaa levylle kaikkine solmuineen ja näin ollen myös lataaminen levyltä on mahdollista.

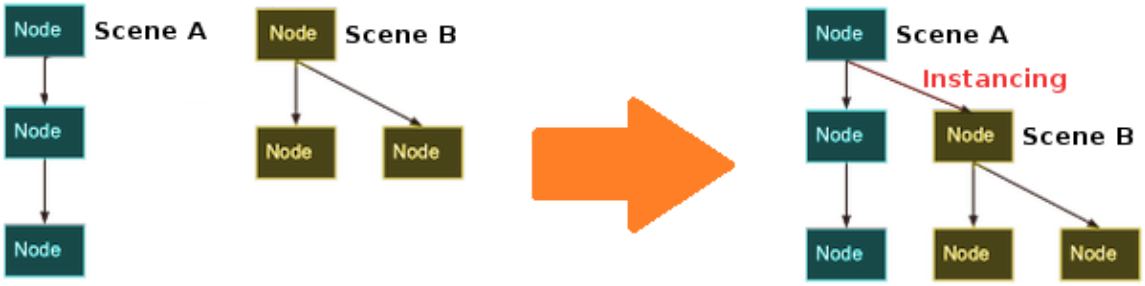
Godot Enginen editori onkin käytännössä näkymäeditori, jossa aktiivista näkymää voidaan muokata. Kuvassa 3 esitellään esimerkkitapaus näkymässä olevista solmuista.



Kuva 3. Yksinkertaisen projektin näkymä ja sen solmut. Ylimpänä juurisolmu (*RootNode)* ja tämän alla pelin perusominaisuuksia lapsisolmuina.

Yksinkertaisessa projektissa aikaisemmin esitellyn kuvan 2 mukainen rakenne voi toimia, mutta kun projekti alkaa kasvaa ja laajentua, on suositeltavaa alkaa käyttää useampaa eri näkymää ja ladata ne yhden päänäkymän alle. Tästä päästäänkin näkymän instansoimiseen eli alustamiseen.

Kun projekti kasvaa, eri komponenttien laittaminen omiin näkymiin lisää selkeyttä projektin hierarkiaan ja helpottaa myös projektin ylläpitämistä, kun komponentit on jaettu pienimpiin osioihin. Kuvan 4 mukaisesti näkymä B alustetaan näkymän A:n alle ja näkymän A:n juurisolmusta tulee myös näkymän B:n juurisolmu. Kuvan 4 mukaista tilannetta voidaan jatkaa loputtomiin.

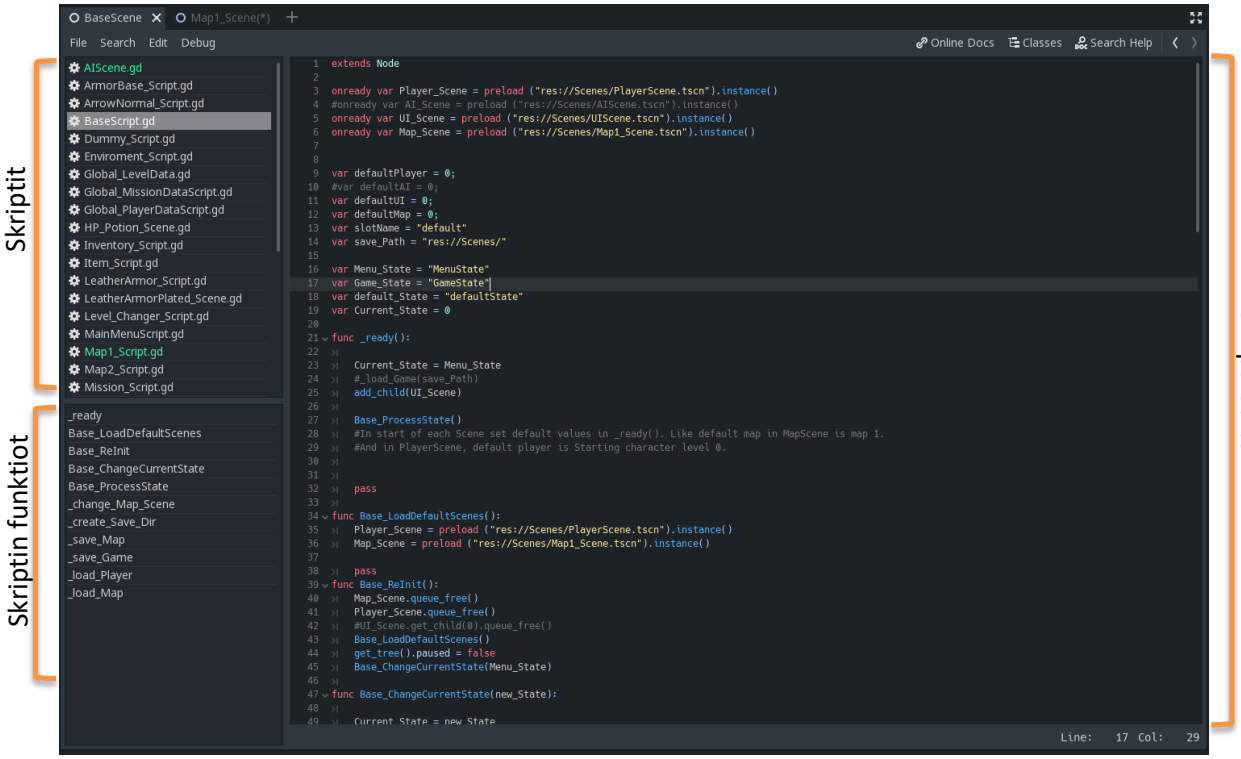


Kuva 4. Näkymän instansointi toiseen näkymään [5.]

Tällainen toimintamalli mahdollistaa helpon kommunikoinnin eri näkymien ja solmujen välillä, koska kaikilla näkymillä ja solmuilla on sama juurisolmu. Yhteinen juurisolmu mahdollistaa käytännössä sen, että kaikista solmuista ja näkymistä päästään kaikkiin toisiin solmuihin ja näkymiin aina tarvittaessa, joten ns. yhteyksien kovakoodaaminen on helppoa. Pienellä hierarkian suunnittelulla voi kuitenkin välttyä kovakoodaamiselta ja kokonaisuudesta saadaan toimivampi.

2.5 Ohjelmointi

Itse pelin ohjelmointiin ja kehitykseen Godot Engine lupaa monia ohjelmointikieliä. Moottoriin on rakennettu sisäisesti oma ohjelmointikieli GDScript, mikä vastaa hieman Pythonia. GDScript on syntaksiltaan varsin helppo oppia, mutta sen käyttö rajoittuu vain Godot Engineen. Lisäksi käytettävissä ovat C# 7.0-, C++-ohjelmointikielet ja graafinen ohjelmointi sekä yhteisön tarjoamia ohjelmointikieliä: Python, Nim ja D. Mikäli käytetään Godot Enginen omaa ohjelmointikieltä, ei kolmannen osapuolen kehitystyökalua tarvita, koska pelimoottori sisältää sisäänrakennetun ohjelmointieditorin GDScriptille. (Kuva 5.)



Kuva 5. Godot Enginen oma ohjelmointieditori.

Suositeltavaa on käyttää kyseistä sisään integroitua ohjelmointikieltä, koska tämä ohjelmointikieli on suunniteltu erityisesti Godot Enginen käyttöön. Godot Enginessä on 3.0.x-versioissa mahdollista myös toteuttaa graafista ohjelmointia, eli tavallista koodia ei tarvitse kirjoittaa riviäkään. Tämä toimintamalli on tuttu Unreal Enginen vastaavasta toimintamallista, *blueprinteistä*. Tässä opinnäytetyössä ei kuitenkaan tätä tapaa toteuttaa ohjelmointia käsitellä.

GDScript on dynaaminen skriptausohjelmointikieli, ja sen syntaksi on hyvin lähellä Python-ohjelmointikieltä. Sen tarkoitus on olla hyvin integroitu Godot Enginen kanssa ja olla joustava ohjelmointikieli, joka näkyy muun muassa seuraavissa asioissa:

- GDScript sulautuu hyvin solmuihin, jotka ovat Godot Enginen pääominaisuuksia [6.].
- GDScript sisältää sisäänrakennettuja 2D- ja 3D-matematiikkakirjastoja, muut kielet eivät näitä sisällä [6.].
- Godot Engine käyttää useaa säiettä (*multi-threading*) datan hakemisessa ja lukemisessa, GDScript tukee tätä ominaisuutta [6.].
- GDScriptissä on oma muistinhallinta [6.].

GDScript on ohjelmointia aloittelevalle sekä ohjelmointia osaavalle hyvä ohjelmointikieli Godot Engineä käytettäessä. Aloittelevan ohjelmoijan on helppo sisäistää GDScriptin yksinkertainen syntaksi, ja jo kokenut ohjelmoija sisäistää tämän kyseisen syntaksin hyvin nopeasti. Yksinkertaisen syntaksin ansiosta ohjelmoidessa aika ei mene syntaksin ja toiminnallisuuden hahmottamiseen, vaan itse projektityöhön.

2.6 Godot Enginen vahvuudet ja heikkoudet

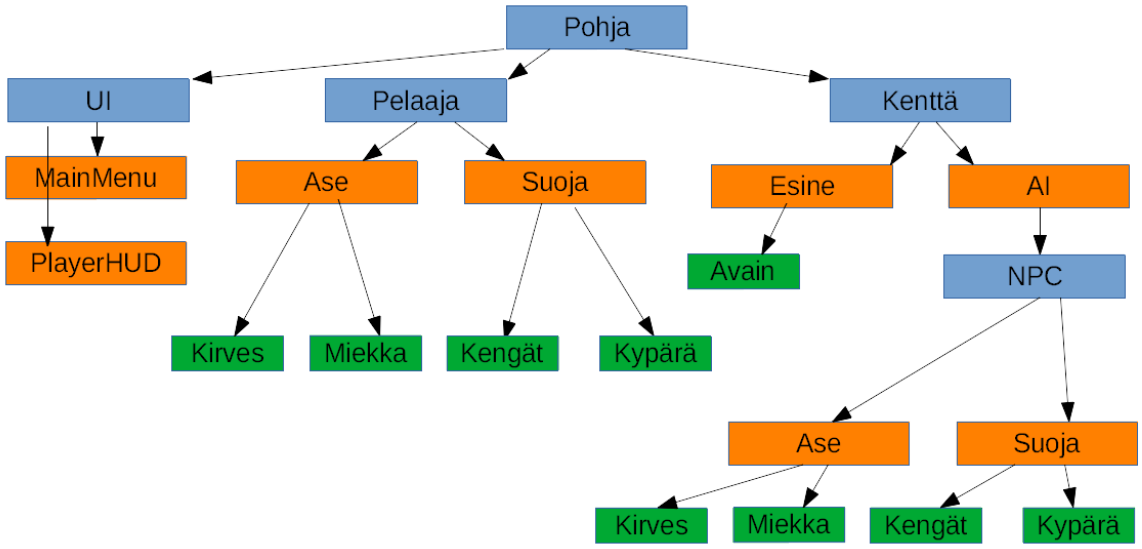
Suurimmat vahvuudet Godot Enginessä ovat kevytrakenteisuus, laaja alustatuki ja helppokäyttöisyys. Moottorin lataamisessa kotikoneelle ei mene kauan, eikä muutaman muun ns. *Indie*-pelimoottorien tavoin tarvitse alustuksia tai CMake-kääntämistä. Yleensä AAA-pelimoottorien asennus ja päivitys vie useita tunteja, mutta Godot Enginellä ei tätä ongelmaa ole. Dokumentaatio on helppolukuista, ja dokumentaatiota myös käännetään usealle eri kielelle. Suomenkielinenkin versio on mahdollista saada, tosin se on vielä keskeneräinen. [7.] Alustatuki on myös laaja, pois lukien pelikonsolit. Pelimoottoriin sisään rakennetun GDScriptin ansiosta ei kolmannen osapuolen koodikääntäjiä tai -editoreita tarvita. Puumainen rakenne projekteissa on myös helppo havainnollistaa esimerkiksi luokkakaavioilla. Kaupallisia projekteja suunnitellessa lisenssit myös toimivat edukseen, sillä pelimoottorin käyttö ei maksa mitään, eikä omien pelien myynnistä mene prosenttiakaan Godot Enginelle. [8.]

Huonojakin puolia on, esimerkiksi pelikonsolitukea ei ole. Tämä johtunee lähinnä pelikonsolien lisenssimaksuista, ja koska Godot Enginen rahoitus pohjautuu vapaaehtoisuuteen, ei pelikonsolitukea ole ainakaan näillä näkymin tulossa. Ainoa vaihtoehto on käyttää kolmannen osapuolen yrityksiä, jotka tekevät konsolikäännöksen Godot Engine-projektista. [9.] GDScriptin opettelu ja käyttö jää ainoastaan Godot Enginen sisälle, mikä tietysti rajoittaa muihin pelimoottoreihin siirtymistä. Mikäli ei kyseistä ohjelmointikieltä koe tarpeelliseksi, Godot Engine tarjoaa onneksi useampaa eri ohjelmointikieltä käytettäväksi. Grafiikassakin Godot Engine jää vielä varsinkin AAA-pelimoottorien jalkoihin ja VR-pelituki on vielä varhaisessa vaiheessa.

3 Pelin kehittäminen

3.1 2D-peli arkkitehtuurin suunnittelu Godot Enginelle

Omaa projektia suunnitellessa Godot Enginelle on tärkeää muistaa Godot Enginen puumainen rakenne. Tällöin oma projektikin tulee suunnitella puumaisesti. Suunnittelu kannattaa aloittaa *Base Noden* eli juurisolmun suunnittelulla. Tämä kyseinen solmu on koko projektin pohja tai juuri, jonka alle muut alisolmut rakentuvat. Tämä juurisolmu myös huolehtisi pelin eri tilojen vaihdon. Käytännössä siis juurisolmu on iso tilakone, jossa on myös tarvittaessa muita perusominaisuuksia. Tämän juurisolmun alle on suunniteltu muita pelin peruskomponentteja. (Kuva 6.)



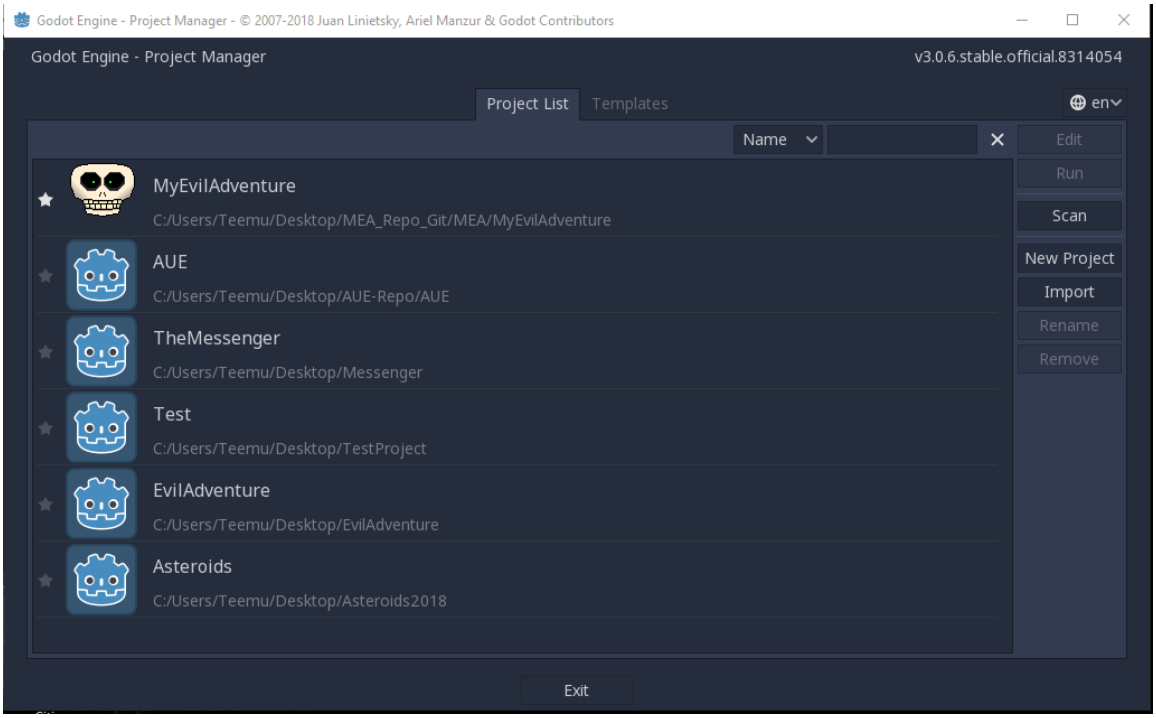
Kuva 6. Yksinkertaistettu luokkakaavio projektista.

Luokkakaavio ei kuitenkaan kuvaa täysin tarkasti toteutetun peliprojektin rakennetta. Selvyyden vuoksi globaalit luokat, dialogisysteemi ja inventaarioluokka on jätetty pois luokkakaaviosta. Luokkakaavio on kuitenkin pääpiirteittäin peliprojektin mukainen, ja se selventää Godot-projektien rakennetta.

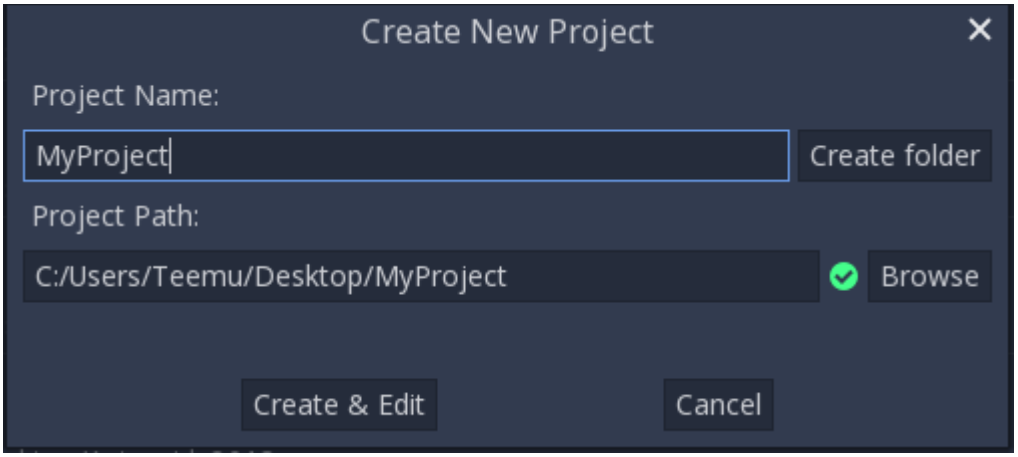
Godot Enginessä projektin rakennetta voi myös muuttaa helposti, sillä solmun tyyppiä voidaan vaihtaa myöhemmin. Ainoana asiana on muistaa, että mikäli solmuun on linkitetty signaaleja, jotka reagoivat tiettyihin tapahtumiin ajon aikana, niin nämä signaalit tulee aktivoida uudelleen. Godot osaa kuitenkin automaattisesti löytää jo olemassa olevan signaalin koodin, joten signaalin skriptiä ei tarvitse uudelleen kirjoittaa.

3.2 Oman projektin luonti

Oman projektin aloitus on hyvin suoraviivainen ja yksinkertainen prosessi. Godot Enginen käynnistyessä avautuu aloitusikkuna, mistä voi aloittaa uuden projektin tai avata jo olemassa olevan projektin. (Kuva 7.) Lisäksi tässä ikkunassa on mahdollista valita pelimoottorin käyttökieli sekä valita mahdollisia *sapluunoita* (*template*). Nämä *sapluunat* ovat yleensä jo valmiiksi luotuja videopelien ominaisuuksia tai esimerkkiprojekteja, joita voi halutessaan tarkastella tai lisätä jo olemassa olevaan projektiin. Tästä ikkunasta voidaan myös skannata tietokoneen kansioita ja tällä tavoin lisätä vanhoja projekteja pelimoottorin aloitusruutuun. Skannauksessa on huomioitava, että esimerkiksi 2.1.-versiolla tehdyt projektit eivät näy 3.0.x-version aloitusikkunassa, eikä niitä voi siihen lisätä. Ainut tapa on avata vanha projekti sillä Godot Enginen versiolla, millä se on tehty ja toteuttaa projektin konvertointi uudempaan versioon. Tämä konvertointi on hieman riskialtista, sillä toimivuutta uudessa pelimoottorin versiossa ei ole taattu. Aloittaessa uuden projektin tulee sille valita nimi ja tallennussijainti omassa tietokoneessa. (Kuva 8.) Tallennussijainnin valinnassa on suositeltavaa luoda uusi kansio projektille, sillä tallennuskansion tulee olla tyhjä.



Kuva 7. Godot Enginen aloitusikkuna.



Kuva 8. Projektin luonti-ikkuna.

Yleisesti ottaen projektin aloittaminen on helppoa ja vaivatonta, sitä verrata hyvinkin jo vahvan jalansijan saaneisiin Unity3D:n projektin luontiin sekä Unreal Engine 4:n projektin luontiin. Godot Enginen kevytrakenteisuuden vuoksi projektin aloittamisessa ei mene kauan, ja pelinkehityksen voi aloittaa heti.

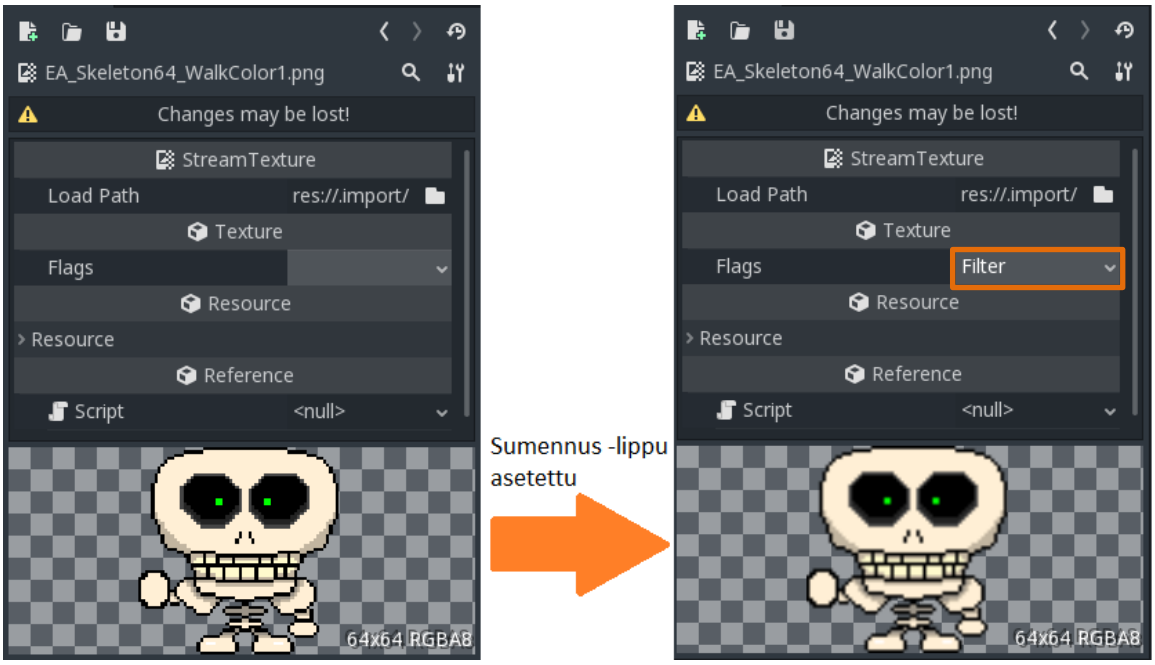
3.3 Grafiikka

Grafiikka on yleisesti ottaen Godot Enginessä keskinkertaista, varsinkin jos verrataan AAA-pelimoottoreiden graafisiin mahdollisuuksiin. Toisiin pienempiin pelimoottoreihin verraten grafiikan voi sanoa olevan laadukasta, erityisesti 3D-grafiikan. Tässä projektissa käytetään vain 2D-spritejä, joten grafiikka muodostuu enemmänkin omista piirustustaidoista kuin Godot Enginen ominaisuuksista. 3D- sekä 2D-grafiikan osalta Godot Engine tarvittaessa kykenee melko näyttävälle tasolle. (Kuva 9. ja 10.)

Kuva 9. Esimerkki 3D-grafiikasta. [10.]

Kuva 10. Esimerkki 2D-grafiikasta. [11.]

Godot Engine ei 2D-grafiikan osalta anna juuri mitään uutta tai mullistavaa. Grafiikkaa voi pelimoottorin puolesta muokkailla *shaderien* eli varjostimien avulla, mutta tässä opinnäytetyössä ei varjostimia juuri esitellä. Mainittakoon, että pelimoottorista löytyy varjostimien ohjelmointiin ja työstämiseen sisäänrakennettu editori. 2D-grafiikkaa voi muokata jo sisääntuontivaiheessa erinäisillä lipuilla. Näillä lipuilla voi esimerkiksi lisätä Godot Enginen luoman automaattisen ”*blur*”-efektin eli sumennuksen. (Kuva 11.)



Kuva 11. Projektiin ladatun kuvan ominaisuuksia. Kuvassa esitettynä lippujen käyttö ja lippujen vaikutukset.

3.4 Globaalit luokat

Globaalit luokat eli Godot Enginessä paremmin tunnetut *singletonit* ovat nimensä mukaisesti globaaleja kaikille solmuille ja näkymille. Globaaleja luokkia voidaan käyttää muun muassa tietyn datan siirtämiseen näkymästä toiseen näkymään helposti. [12.]

Globaalit luokat ladataan usein käyttäen Godot Enginen automaattista latausta *(AutoLoad)*. Automaattinen lataus takaa sen, että globaali luokka on ladattu aina, kun näkymä ladataan. Automaattista latausta voidaan käyttää muuhunkin kuin globaalien luokkien lataamiseen, esimerkiksi tietyn näkymän voi asettaa aina ladattavaksi. [12.]

Seuraava kuva esittää miten globaalit luokat näkyvät ohjelmoinnissa. Projektiin automaattiseen lataukseen on asetettu *Global_PlayerDataScript*-niminen skripti ja täältä haetaan pelaajan nopeus pelaajanäkymän omaan muuttujaan. (Kuva 12.) Tämä tapa on oikeastaan ainut tapa toteuttaa globaaleja muuttujia tai luokkia, koska GDScript ei normaalissa syntaksissaan tunne globaaleja muuttujia. [12.]

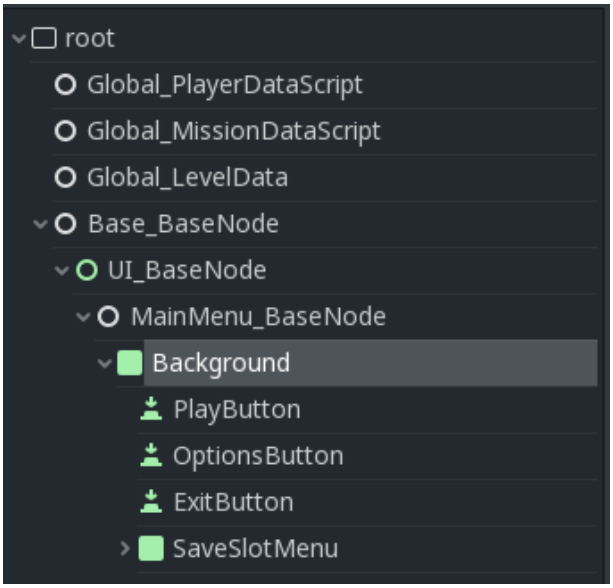
```
#Esimerkki globaalin luokan käytöstä:

var playerSpeed = 0 #Näkymän tai solmun oma muuttuja.

#Asetetaan muuttujaan globaalin luokan muuttujan arvo.
playerSpeed = Global_PlayerDataScript.Global_PlayerSpeed
```

Kuva 12. Globaalien luokkien ja muuttujien käyttö.

Godot Enginessä globaaleja luokkia käsitellessä on huomioita muutamia asioita, esimerkiksi globaaleja luokkia ei suositella käytettäväksi tietyn datan säilömiseen, esimerkiksi pelaajan inventaario. Tällaiset systeemit tarvitsevat omat näkymänsä ja solmunsa. Lisäksi jatkuva lataaminen ja tallentaminen tällaiseen globaaliin luokkaan on raskasta ja hidasta. [12.]

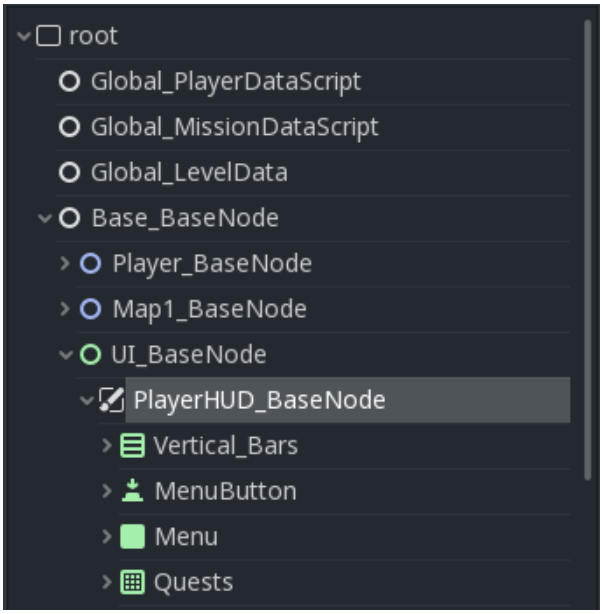
3.5 Käyttöliittymänäkymä

Käyttöliittymänäkymän tehtävänä on päivittää ja tarvittaessa vaihtaa pelaajalle näkyvää käyttöliittymää. Käyttöliittymänäkymää ei juurinäkymän tavoin koskaan poisteta, vaan se vaihtaa tilaansa ja näin ollen poistaa ja lisää lapsisolmuja. Esimerkiksi kun peli käynnistetään, lataa juurinäkymä (*Base_BaseNode*) etukäteen käyttöliittymänäkymän (*UI_BaseNode*), joka puolestaan on asetettu lataamaan aina ensimmäiseksi päävalikkonäkymän (*MainMenu_BaseNode*). (Kuva 13.)



Kuva 13. Pelin hierarkia aloitusruudussa. Ylimpänä Godot Enginen luoma juuri, alempana globaalit skriptit (*Global_xDataScript*). Projektin luokkakaavion mukainen hierarkia alkaa *Base_BaseNode*-solmusta, kyseinen solmu on projektin juurinäkymä.

Pelaajan aloittaessa pelin käyttöliittymänäkymä (*UI_BaseNode*) vaihtaa tilaansa ”pelitilaan” ja poistaa päävalikkonäkymän ja lisää tilalle pelaajan käyttöliittymänäkymän. Kuvassa 14 nähdään pelin rakenteen muutos.



Kuva 14. Pelin hierarkia pelitilassa. Käyttöliittymänäkymä (*UI_BaseNode*) poistaa päävalikkonäkymän ja lataa tilalle pelaajan käyttöliittymänäkymän (*PlayerHUD_BaseNode*).

Tällaisella systeemillä saadaan käyttöliittymä irralleen muusta pelilogiikasta, mikä mahdollistaa käyttöliittymän helpon käsittelyn erillään. Esimerkiksi pelin pysäyttämistilanteessa (*pause*) käyttöliittymää voidaan käyttää, mutta muu peli on pysähtynyt. Tämä systeemi mahdollistaa myös dialogisysteemin käytön muun pelin ollessa pysähtyneenä.

3.6 Pelaajanäkymä

Godot Engine tarjoaa pelaajan luontiin vaihtoehtoisia tapoja, ja on hankala määritellä, mikä toteutustapa olisi paras mahdollinen. Tässä luvussa esitellään yksi tapa toteuttaa pelaajanäkymä ja annetaan perustelut, miksi tällaiseen ratkaisuun päädytään. On kuitenkin muistettava, että tämä tapa ei suinkaan ole ainoa, vaan vaihtoehtoisia tapoja löytyy useita. Kuvassa 15. on esiteltynä pelaajanäkymä projektin tämänhetkisessä muodossaan. Näkymässä on paljon lapsisolmuja ja esiladattuja näkymiä, mutta luvussa keskitytään vain pelaajanäkymän ydinalueisiin.

Koska pelaaja on 2D-hahmo, koko luokan juurisolmun (*Player_BaseNode*) tyyppinä toimii *Node2D*. Tämä *Node2D* antaa perusominaisuuksia koko näkymälle, esimerkiksi sijainnin vektorimuodossa. *Node2D* tunnistaa nimen vasemmalla puolella olevasta sinisestä kierukasta. Juurisolmuun on liitetty skripti ja juurisolmu on myös asetettu omaan ryhmäänsä. Skriptin tunnistaa solmun oikealla puolella olevasta kääröstä ja ryhmän taas neliöstä, jonka sisällä on ympyrä. Ryhmän tarkoitus tässä tilanteessa vain määritellä tallennettavat näkymät, joten tässä luvussa ei kyseistä ryhmää käsitellä. (Kuva 15.) Seuraavana solmuhierarkiassa on *Player_Character* eli itse pelaajan hahmo. (Kuva 15.) Tämä hahmo on muotoa *Area2D*, joka antaa törmäystarkastuksiin liittyviä ominaisuuksia. *Area2D* tarvitsee kuitenkin lapsisolmukseen itse törmäysboksin, josta *Area2D* hakee törmäysdatan aina tarvittaessa.

Pelaajan hahmon alta hierarkiassa löytyy *Character*, joka on tyypiltään ”*animoitu-sprite*” eli pelaajan sprite-hahmo. (Kuva 15.) *Character* ei itsessään omaa mitään sisältöä, vaan se toimii äitisolmuna sen lapsisolmuille. Lapsisolmut on jaettu seuraavasti, *pää, kädet, torso* eli keho ja *jalat.* Näistä solmuista löytyy sprite-kuvakkeet, joiden avulla pelaajan hahmo piirretään näytölle. Nämä solmut sisältävät myös animaatiot. Animaatiot-luvussa esitellään nämä animaatiot tarkemmin ja miten nämä solmut on toteutettu.

Alemmalla hierarkiassa on useita sijaintisolmuja. Niiden tarkoitus on määritellä aseiden ja suojien sijainti aina, kun hahmo kääntyy tai käyttää asetta. (Kuva 15.) Tämä tapa ei välttämättä ole optimaalisin tapa toteuttaa suojien ja aseiden oikea sijoitus, mutta Animaatiot-luvussa on tarkempia syitä, miksi näin on toimittu.

Lisäksi näkymään on sijoitettu kamera, joka seuraa pelaajan hahmoa ja piirtää kuvan näytölle. Alimpana hierarkiassa on *Inventory_Sprite*, joka on tavallinen sprite-kuvake animoidun sprite-kuvakkeen sijaan. Tämän solmun tarkoitus on vain säilöä pelaajan hahmon kuva pelaajan inventaariota varten. (Kuva 15.)

Kuva 15. Pelaajanäkymän hierarkia ja sen solmut.

Luokkakaavion mukaan pelaajanäkymään on liitetty kaksi muutakin näkymää, jotka ovat *Ase*- ja *Suoja*-näkymä. Nämä ovat pelaajan tavoin omia näkymiään ja näin ollen ne on esiladattu pelaajanäkymän skriptissä. Nämä kyseiset näkymät eivät näy pelaajanäkymän näkymäeditorissa, koska lataaminen toteutetaan skriptin puolella, eli lataaminen toteutetaan ns. ajon aikana. Nämä luokat sisältävät toiminnallisuudet aseisiin ja suojiin.

Tämän tyylisen toteutuksen tavoitteena on pyrkiä pitämään pelaajanäkymä mahdollisimman tiiviinä ja jakaa toiminnot muihin luokkiin, joita voitaisiin hyödyntää myös NPC-näkymässä. Vaihtoehtoinen tapa olisi ollut toteuttaa suuri *Character*- eli hahmonäkymä, jonka NPC- ja pelaajanäkymä olisivat perineet. Näin olisi saatu samat toiminnallisuudet suoraan molempiin luokkiin.

Pelaajanäkymän juurisolmuksi voi vaihtoehtoisesti valita *KinematicBody2D*-tyyppisen solmun, mikä helpottaa törmäystarkistusten tekemistä muiden liikkuvien tai staattisten objektien kanssa. *KinematicBody2D* ei kuitenkaan sisällä kaikkia samoja ominaisuuksia kuin *Area2D*, joten osan tarkistuksista voi joutua tekemään *Area2D*:n törmäystarkistuksissa.

3.6.1 Liikkuminen

Liikkuminen pelihahmolla tapahtuu näppäimistöllä perinteisiä WASD-näppäimiä käyttäen. Ennen liikkumisskriptin kirjoittamista täytyy projektille alustaa ns. *Input event* -eli käyttäjäsyöte. Käyttäjäsyötteiden alustaminen on Godot Enginessä helppoa ja muistuttaa hieman Unreal Enginen 4:n tapaa toteuttaa käyttäjäsyötteiden alustukset.

Kuvassa 16 nähdään, miltä käyttäjäsyötteiden alustus näyttää. Vaalean harmaalla oleva osio sisältää käyttäjäsyötteen nimen ja sen alapuolella tummemmalla harmaalla oleva osio on näppäin tai laite, mikä lähettää tätä käyttäjäsyötettä.

Kuva 16. Käyttäjäsyötteet ja niihin sijoitetut painikkeet.

Käyttäjäsyötteen nimeksi voi käytännössä kirjoittaa mitä vain, mutta itse käytin selvyyden vuoksi kaikissa pelaajaan liittyvissä komennoissa alkusanaa *player.* Kun käyttäjäsyötteet on luotu, voidaan skripteissä aloittaa niiden käyttö. Kuvassa 17 näytetään, miten käyttäjäsyötteitä voidaan kontrolloida ja käyttää.

Käyttäjäsyötteitä kutsutaan ns. *päivitys*-funktiossa, joka kulkee Godot Enginessä nimellä *_process(delta)*. Päivitys-funktiota kutsutaan, joka kuvanpäivitys kerta ja se saa parametrikseen *delta*n eli ajan, joka kului edellisestä kuvanpäivityskerrasta. Käytännössä funktio tarkastaa joka kuvanpäivityskerralla, onko mitään näppäintä painettu.

Jos esimerkiksi painetaan näppäintä, johon on liitetty käyttäjäsyöte nimeltä *player_MoveRight*, eli tässä tilanteessa näppäintä *D*, sallii funktio jos-lausekkeen jälkeisen osion suorituksen. Tässä osiossa pelaajan nopeuteen lisätään vektorin x suunnassa pelaajan nopeuden verran. Tämän jälkeen asetetaan oikeat animaatiot kaikille pelaajanäkymän lapsisolmuille ja pelaajan suunta päivitetään. Tämän jälkeen *Ase*- ja *Suoja*-luokat asettavat mahdolliset aseet ja suojat oikeille paikoilleen, jonka jälkeen tilanteesta riippuen asetetaan oikeat z-indeksit, mitkä vaikuttavat sprite-kuvien piirtojärjestykseen. (Kuva 17.)

```
func _process(delta):

#Player Movement:
playerVelocity = Vector2(0, 0)

if (Input.is_action_pressed("player_MoveRight")):
      playerVelocity.x += playerSpeed
      playerAnimation_Head = "Walk_Right"
      playerAnimation_Torso = "Walk_Right"
      playerAnimation_Hands = "Walk_Right"
      playerAnimation_Legs = "Walk_Right"
      playerDirection = "Right"
      WeaponBase_Scene._set_weaponPosition(get_node(
      "Player_Character/Right_WeaponPos").position)
      ArmorBase_Scene._set_ArmorDirection(playerDirection)
      _set_zIndex_Value(get_node("Player_Character/Character/Head"), 1)
      _set_zIndex_Value(WeaponBase_Scene, 2)
      isIdle = false

if (Input.is_action_pressed("player_MoveLeft")):
      playerVelocity.x -= playerSpeed
      playerAnimation_Head = "Walk_Left"
      playerAnimation_Torso = "Walk_Left"
      playerAnimation_Hands = "Walk_Left"
      playerAnimation_Legs = "Walk_Left"
      playerDirection = "Left"
      WeaponBase_Scene._set_weaponPosition(get_node(
      "Player_Character/Left_WeaponPos").position)
      ArmorBase_Scene._set_ArmorDirection(playerDirection)
      _set_zIndex_Value(get_node("Player_Character/Character/Head"), 1)
      _set_zIndex_Value(WeaponBase_Scene, 2)
      isIdle = false
```

Kuva 17. Käyttäjäsyötteiden käyttö skriptissä. Ylhäällä pelaajan liikkuessa oikealle toteutettava koodi. Alempana taas vasemmalle liikkuessa toteutettava koodi.

Seuraavana tarkastetaan pelaajan nopeusvektorin pituus. Jos arvot ovat suurempia kuin nolla, pelaajan nopeusvektorin normaali kerrotaan pelaajan nopeudella ja asetetaan tämä pelaajan nopeusvektoriksi. Tämän jälkeen animaatiot käynnistetään. Mikäli pelaajan nopeusvektorin pituus on nolla, asetetaan pelaajan suunnan mukaan paikallaanolo-animaatiot. (Kuva 18.)

```
if (playerVelocity.length() > 0):
      playerVelocity = playerVelocity.normalized() * playerSpeed
      _select_animation(playerAnimation_Head, playerAnimation_Hands,
      playerAnimation_Torso, playerAnimation_Legs)
else:
      if (!isIdle):
            match playerDirection:
            "Down":
                  if (WeaponBase_Scene.WeaponInUse == true):
                        playerAnimation_Hands = "Idle_Down_Weapon"
                        playerAnimation_Torso = "Idle_Down_Weapon"
                  else:
                        playerAnimation_Hands = "Idle_Down"
                        playerAnimation_Torso = "Idle_Down"

                        playerAnimation_Head = "Idle_Down"
                        playerAnimation_Legs = "Idle_Down"
                        _select_animation(playerAnimation_Head, playerAnima-
                        tion_Hands, playerAnimation_Torso, playerAnima-
                        tion_Legs)
            "Up": ...
            "Right": ...
            "Left": ...
      isIdle = true
```

Kuva 18. Pelaajan nopeusvektori ja alempana *idle*-animaatioiden asettaminen.

Päivitys-funktion viimeisessä vaiheessa päivitetään pelaajan sijainti. Tämän funktion jälkeen *päivitys*-funktio aloitetaan alusta. Mikäli käytetään *KinematicBody2D*-tyyppistä juurisolmua, tulee liikkuminen toteuttaa kyseisen solmun jäsen funktion *move_and_colliden* kautta.

Tämä tapa toteuttaa pelaajan liikkuminen on helpoin ja suoraviivaisin, sillä kaikki toiminnot toteutetaan yhden funktion sisällä ja tämän funktion päivittämisestä ei ohjelmoijan tarvitse huolehtia, sillä pelimoottori pitää huolen päivityksestä. Huonona puolena voi nähdä jatkuvan jos-lauseiden tarkastuksen *päivitys*-funktiossa, sillä tämä rasittaa hieman prosessoria. Kuitenkin tarkistukset ovat yksinkertaisia, ja kun pidetään huoli, että samankaltaisia tilanteita ei muissa luokissa ilmene, niin projekti pysyy kevytrakenteisena. Hyvänä lähtökohtana on, että *päivitys*-funktioissa ei olisi mitään jatkuvia tarkistuksia, mutta joskus tulee vastaan tilanteita, joissa niiltä ei voi välttyä. Projektin prosessorisyklejä sekä niihin kuluvia aikoja on hyvä tarkkailla aika ajoin Godot Enginen *debug*-ikkunasta, sillä jos syklien ajat alkavat kasvaa, on syytä tarkastaa *päivitys*-funktioiden tarkastukset ja ehkä miettiä uutta ratkaisutapaa.

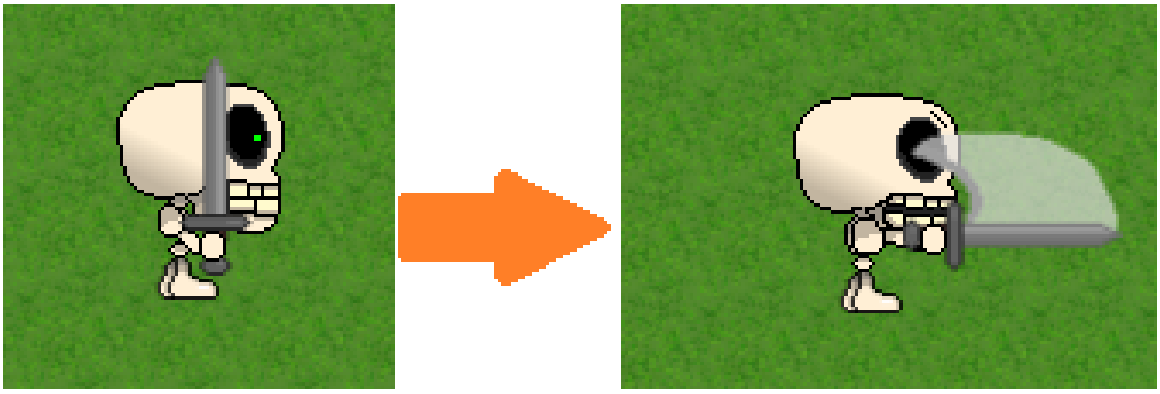
3.6.2 Taistelu ja toiminnot

Kuten aikaisemmin näytetystä kuvasta 18 voidaan nähdä, pelaajan lyöntinäppäimet tulee myös alustaa käyttäjäsyötteeksi. Prosessi on täysin samanlainen kuin jo aikaisemmin 4.6.1 Liikkumisluvussa esitelty liikkumisnäppäinten alustaminen, eli annetaan käyttäjäsyötteelle nimi ja näppäin. Taistelu ja aseiden käyttö tulee siis tapahtumaan joko hiiren vasemmasta painikkeesta tai välilyöntinäppäimestä. Nämä toiminnot tarkastetaan *Ase*-luokassa, joka on esiladattuna pelaajanäkymään. *Ase*-näkymässä näppäintarkastus toteutetaan *_input(event)* -funktiota käyttäen (Kuva 19).

```
func _input(event):

	if (Input.is_action_just_pressed("player_UseWeapon")):
		if (!EnableInput && event is InputEventMouseButton):
			#Tarkastetaan onko pelihahmon käyttäjäsyöte
			#aktivoituna. Esimerkiksi valikoissa
			#pelihahmon käyttäjäsyötteet ovat ei-aktiivisia.
			return

		PlayerDirection = get_parent().playerDirection

		if (MeleeWeapon): #Lähitaisteluaseen toiminta.
			match PlayerDirection: #Tarkastetaan pelaajan suunta
			"Up":
				#Löynti -function toteutus.
				_meleeHit("Hit_Up", PlayerDirection, 0)
			"Down":
				_meleeHit("Hit_Down", PlayerDirection, 3.14)
			"Right":
				_meleeHit("Hit_Right", PlayerDirection, 1.57)
			"Left":
				_meleeHit("Hit_Left", PlayerDirection, -1.57)
			return

		if (RangedWeapon): #Ampuma-aseen toiminta.
			_rangedShot(PlayerDirection)
			return
		else: #Käytetyllä aseella ei ole toimintalogiikkaa.
			print("NO MELEE OR RANGED WEAPON")
```

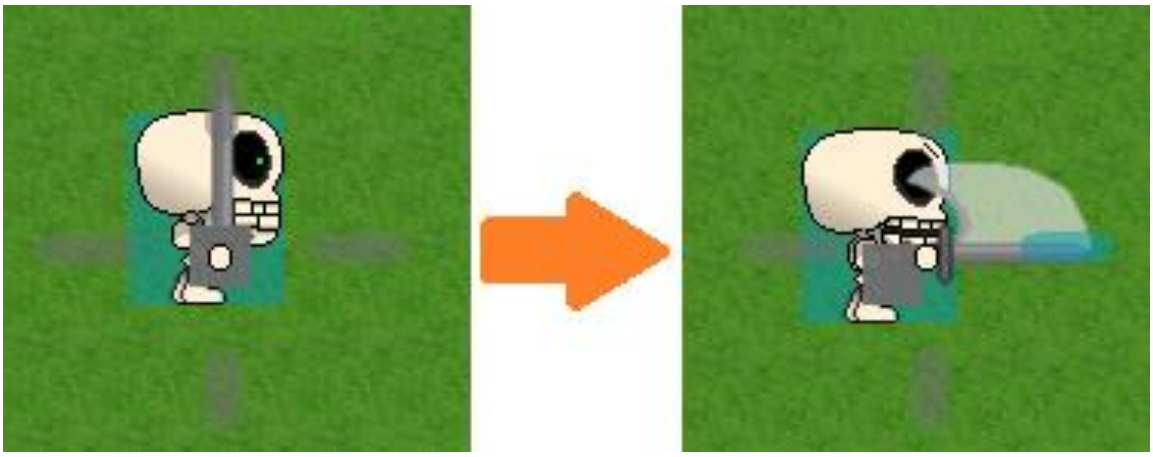
Kuva 19. *Ase*-luokan *_input(event)* -funktio.

_input(event)-funktio eroaa aikaisemmin esitetystä *_process(delta)*-funktiosta siten, että se aktivoituu vain, kun Godot Enginen käyttäjäsyöte alkaa saamaan tapahtumia, esimerkiksi hiiren liikuttelu tai näppäinten painelu aktivoi *_input(event)*-funktion. Siksi *_input(event)*-funktio sopii paremmin lyönti- ja toimintonäppäinten tarkastamiseen. Pelaajan painaessa lyöntinäppäintä näkyy lyöntitilanne hyvin lyhyen animaation muodossa. (Kuva 20.)



Kuva 20. Lyöntitapahtuma ilman törmäysbokseja.

Kuvassa 21 on sama tilanne, mutta törmäysboksit on asetettu näkyväksi. Näin ollen voidaan tarkastella, miten törmäystarkastukset saadaan toteutettua, kun pelaaja päättää painaa lyöntipainiketta. Kuvassa 21 harmaat alueet ovat epäaktiivisia törmäystarkastusbokseja ja siniset alueet ovat taas aktiivisia törmäystarkastusbokseja.



Kuva 21. Lyöntitapahtuma törmäysbokseilla.

Pelaajan painaessa lyöntinäppäintä aktivoituu pelaajan suunnan mukainen törmäyslaatikko, jotka ovat kaikilla aseilla omanlaisensa. Aseen törmäysbokseihin on taas linkitetty *signaalit,* jotka käsittelevät osumapisteiden aiheuttamat vahingot mahdolliseen kohteeseen. *Signaaleista* kerrotaan enemmän luvussa 4.6.3. Törmäystarkistukset.

3.6.3 Törmäystarkistukset

Godot Enginellä 2D-peliä tehdessä on törmäystarkistuksiin neljä vaihtoehtoista tapaa, sillä moottori tarjoaa neljä erilaista solmutyyppiä, jotka omaavat kaikki hieman eri toiminnallisuudet. Nämä vaihtoehdot ovat:

- *Area2D*, joka periytyy *CollisionObject2D*-luokasta. *Area2D* tarjoaa törmäystarkastukset ja niihin reagoinnin signaalien avulla. Lisäksi tähän solmutyyppiin voidaan asettaa tarvittaessa fysiikkaominaisuuksia. [13.]

Seuraavat kolme vaihtoehtoa periytyvät kaikki *PhysicsBody2D*-luokasta [13.]

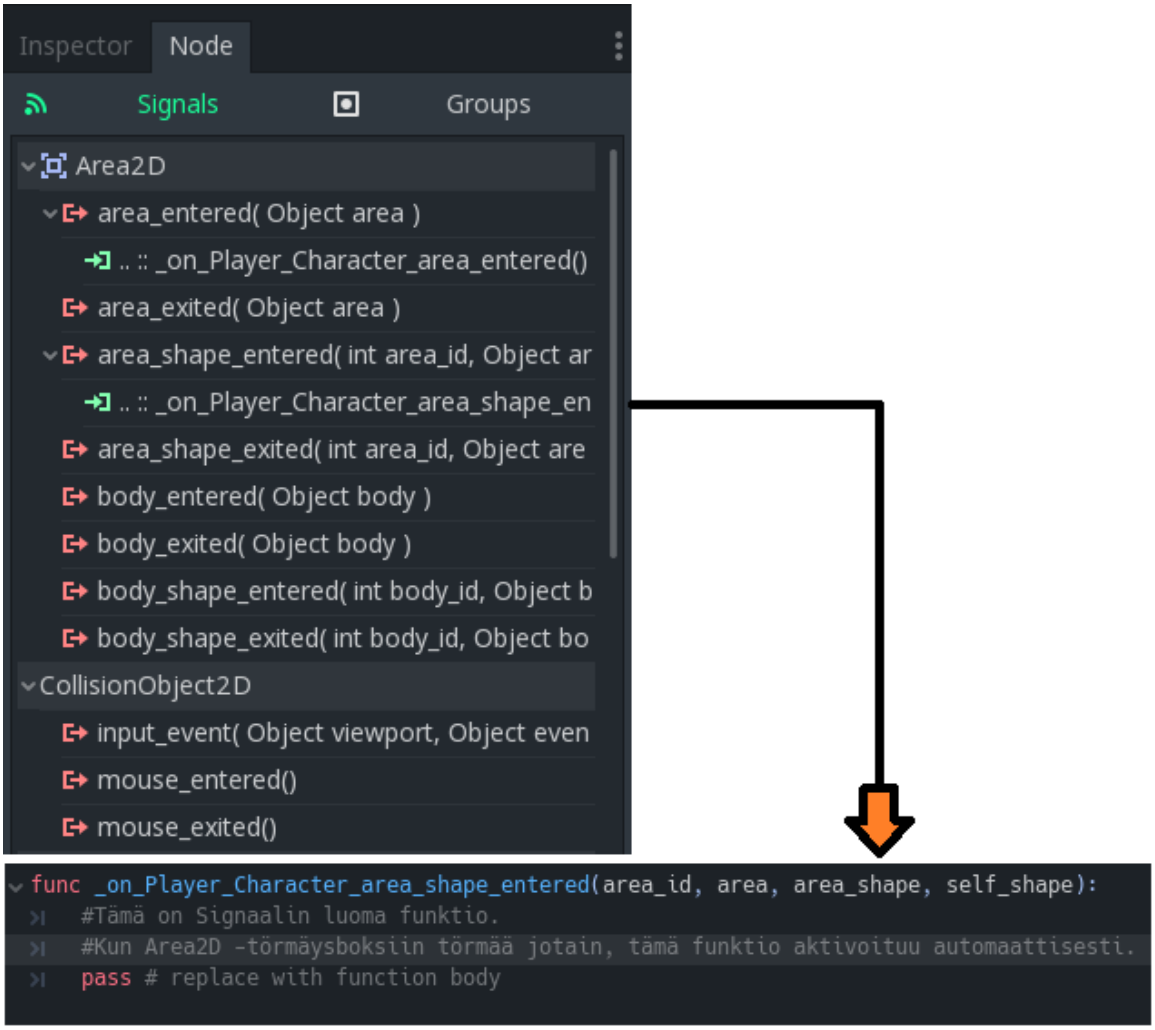
- *StaticBody2D*, tätä komponenttia ei voida liikuttaa fysiikkamoottorin toimesta, mutta tarjoaa se kuitenkin törmäysten tarkastelun [13]. Käytetään usein staattisissa komponenteissa, esimerkiksi seinissä.

- *RigidBody2D*, tätä komponenttia voidaan liikutella ainoastaan käyttämällä fysiikkamoottorin ominaisuuksia. [13.]

- *KinematicBody2D,* tämä komponentti tarjoaa törmäystarkastukset, mutta ei fysiikkaa. Lisäksi kaikki törmäykseen reagoinnit täytyy kirjoittaa skriptiin itse. [13.]

Opinnäytetyöprojektissa käytetään *Area2D*-tyyppistä ratkaisua, koska se tarjoaa tähän projektiin eniten ominaisuuksia, sekä signaalien avulla törmäysten käsittely on helppoa ja itse signaalin lisääminen hyvin suoraviivaista. Lisäksi näihin komponentteihin on lisättävä *CollisionShape2D*-komponentti, joka määrää törmäysboksin muodon ja koon. *CollisionShape2D*-komponentin muotoa on mahdollista muokata mielensä mukaan, tai voi vaihtoehtoisesti valita valmiiksi tehdyistä muodoista, jotka noudattelevat peruskappaleiden muotoja, esimerkiksi neliö tai ympyrä.

Area2D-komponentin törmäystarkistuksien tarkastus onnistuu helposti *signaaleja* käyttäen. *Signaalit* voidaan linkittää reagoimaan esimerkiksi silloin, kun törmäysboksiin osuu jokin muu komponentti, jolla on myös törmäysboksi. *Signaalien* linkitys tapahtuu helposti valitsemalla *Area2D* -komponentti ja valitsemalla haluttu *signaali.* (Kuva 22)



Kuva 22. *Signaalien* luonti ja linkitys. Kuvassa luodaan *signaali*, joka tarkastaa *Area2D*:n törmäysboksiin tulevia törmäyksiä. Nuolen mukainen funktio aktivoituu aina, kun törmäys havaitaan. *Signaali* on linkitetty pelaajanäkymän skriptiin.

Signaalin linkittyessä Godot Engine osaa automaattisesti luoda *signaalin* tarvitseman funktion. Skripti, johon funktion haluaa luoda, valitaan linkitysvaiheessa. Skriptin tulee kuitenkin sijaita samassa näkymässä. *Signaalista* riippuen saa funktio tiettyjä parametrejä, joista sitten voidaan saada tietoa törmääjästä ja näin ollen käsitellä myös törmääjää.

3.6.4 Pelaajan käyttöliittymä

Pelaajan käyttöliittymänäkymä on UI-näkymän lapsi, joka on ladattu, kun pelaaja aloittaa pelin. Kyseinen näkymä hakee pelaajanäkymän hahmon attribuutteja, esimerkiksi *”elämäpisteet”,* ja tu-

lostaa tiedon näytölle käyttäen omia lapsisolmujaan. Lisäksi pelaajan käyttöliittymänäkymä sisältää käyttöliittymän nappeja, kuten *Menu*-napin, josta pelaaja voi pysäyttää pelin, tallentaa pelin tai palata takaisin päävalikkoon. (Kuva 23)

Kuva 23. Pelaajalle näkyvä ikkuna ja pelaajan käyttöliittymä.

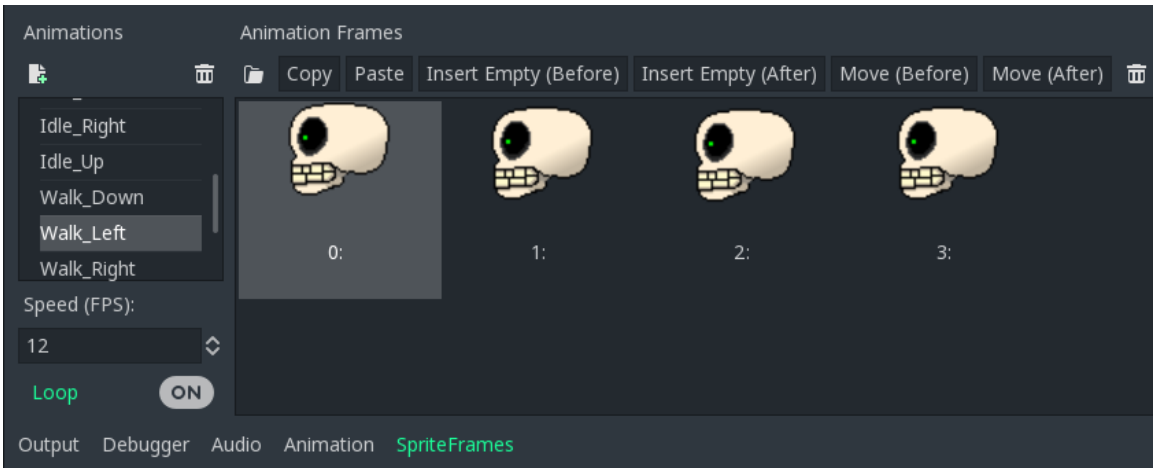
Projektin puumaisen rakenteen ansiosta käyttöliittymänäkymistä on helppo päästä pelaajanäkymään juurisolmun kautta, ja näin ollen esimerkiksi pelaajan ”elämäpisteitä” ei tarvitse päivitellä *päivitysfunktiossa*, vaan päivitykset voidaan toteuttaa komentopohjaisesti juuri silloin, kun siihen on tarvetta. Näin saadaan reaaliaikainen lopputulos, mutta projektia ei rasiteta turhilla *päivitysfunktioilla*.

3.6.5 Animaatiot

Godot Engine tarjoaa 2D-peleille kaksi vaihtoehtoa toteuttaa animaatiot. Mahdollista on käyttää Godot Enginessä olevaa sisään rakennettua animaatioeditoria ja tehdä animaatiot pelimoottorin sisällä. Vaihtoehtoisesti voi käyttää *animoituja sprite*-kuvakkeita, joita on käytetty tässä opinnäytetyössä. Tällöin animaation kaikki kuvakkeet joudutaan piirtämään itse kolmannen osapuolen piirtosovelluksella ja Godot Engine ainoastaan toistaa näitä kuvia aina tarvittaessa. Kyseinen rat-

kaisu on yksinkertaisempi toteutukseltaan, mutta animaatiotyökaluja osaavan kannattaa toteuttaa animaatiot käyttäen Godot Enginen animaatioeditoria, sillä työn määrä on animaatioeditoria käytettäessä pienempi. Animaatioeditorista on myös saatavilla enemmän ohjeita eri nettisivuilta.

Animaatioiden teko aloitetaan avaamalla *animoidun spriten* (*Animated Sprite*) kautta *animaatiokuvakkeet*, ja tällöin päästään käsittelemään kyseisen solmun dataa. (Kuva 24.) Animaatioita voidaan luoda lukuisia määriä ja niille kaikille voidaan asettaa omat kuvakkeet, joita pelimoottori sitten käy läpi aina kutsuttaessa. *Animoitu sprite* ei kuitenkaan ole monipuolinen kokonaisuus ja animaatiot tällä tavalla toteutettuna sisältävät hyvin rajatusti ominaisuuksia, mutta yksinkertaisten animaatioiden tekeminen on mahdollista. Käytännössä tätä tapaa voisi kuvata vanhanaikaiseksi tavaksi tehdä 2D-animaatioita, sillä kaikki piirretään käsin ja tietokone vain toistaa kuvasarjaa halutulla nopeudella tai vaihtaa kuvasarjan.



Kuva 24. Animoidun spriten editointi-ikkuna. Animaatio toistetaan vasemmalta oikealle lukujen mukaisessa järjestyksessä.

Skriptissä animaatioita voidaan käsitellä helposti, sillä animaatioihin voidaan linkittää *signaaleja,* niiden avulla saadaan esimerkiksi tietää, milloin animaatio on loppunut. On myös mahdollista käyttää animaatioiden jäsenmuuttujia ja -funktioita. Nämä funktiot tarjoavat perusominaisuuksia animaatioiden käsittelyyn, esimerkiksi animaation käynnistäminen. (Kuva 25.)

```
func _set_animation_play(head, hands, torso, legs):

	if (!get_node("Player_Character/Character/Head").is_playing()):
		get_node("Player_Character/Character/Head").play(head)
		playAnimation = true

	if (!get_node("Player_Character/Character/Hands").is_playing()):
		get_node("Player_Character/Character/Hands").play(hands)
		playAnimation = true

	if (!get_node("Player_Character/Character/Torso").is_playing()):
		get_node("Player_Character/Character/Torso").play(torso)
		playAnimation = true

	if (!get_node("Player_Character/Character/Legs").is_playing()):
		get_node("Player_Character/Character/Legs").play(legs)
		playAnimation = true
pass
```

Kuva 25. Animaation käynnistäminen skriptissä.

3.6.6 Dialogisysteemi

Godot Engine ei sisällä mitään omaa systeemiä käsitellä dialogeja. *Popup*-luokassa on muutamia dialogeihin viittaavia ominaisuuksia [14.], mutta tässä projektissa näitä käyttöliittymän ominaisuuksia ei käsitellä. Projektissa dialogit on toteutettu rakentamalla dialogeja käsittelevä systeemi käyttäen muita pelimoottorin tarjoamia käyttöliittymäominaisuuksia ja on suositeltavaa tutustua eri käyttöliittymäsolmuihin ja miettiä omaan projektiin sopiva ratkaisu.

Tässä projektissa dialogit muodostuvat useasti tekstitiedostosta, joita sitten ladataan NPC-hahmon alustuksen yhteydessä. Dialogit on säilötty NPC-hahmon sisälle omiin kirjastoihin (*Dictionary)* ja nämä kirjastot sisältävät myös pelaajan vastaukset. (Kuva 26.) Näin ollen NPC-hahmolla on kaikki tieto siitä, missä tilassa dialogi on ja mitkä ovat pelaajan mahdolliset vastaukset.

```
func _load_Mission_Dialogues():
	d_NPC_DummysQuest = {
		"Intro" : get_parent().get_node("NPC_Dialogue")._load_Dialogue(
			MISSION_DIALOGUE_PATH + "NPC/Mission1.txt"),
		"Complete" : get_parent().get_node("NPC_Dialogue")._load_Dialogue(
			MISSION_DIALOGUE_PATH + "NPC/Mission1_Complete.txt"),
		"Pass" : get_parent().get_node("NPC_Dialogue")._load_Dialogue(
			MISSION_DIALOGUE_PATH + "NPC/Mission1_Pass.txt"),
		"Failed" : get_parent().get_node("NPC_Dialogue")._load_Dialogue(
			MISSION_DIALOGUE_PATH + "NPC/Mission1_Failed.txt") }

	d_Player_Intro_Answers = {
		"Pass" : get_parent().get_node("NPC_Dialogue")._load_Dialogue(
			MISSION_DIALOGUE_PATH + "Player/Mission1_A1.txt"),
		"Failed" : get_parent().get_node("NPC_Dialogue")._load_Dialogue(
			MISSION_DIALOGUE_PATH + "Player/Mission1_A2.txt") }

	d_Player_Pass_Anwsers = {
		"Complete" : get_parent().get_node("NPC_Dialogue")._load_Dialogue(
			DEFAULT_DIALOGUE_PATH + "OK.txt") }

	d_Player_Failed_Answers = {
		"Complete" : get_parent().get_node("NPC_Dialogue")._load_Dialogue(
			DEFAULT_DIALOGUE_PATH + "OK.txt") }

	d_Player_Complete_Answers = {
		"Quit" :  get_parent().get_node("NPC_Dialogue")._load_Dialogue(
			DEFAULT_DIALOGUE_PATH + "Quit.txt") }

	d_Player_DummysQuest = {
	"Intro" : d_Player_Intro_Answers,
	"Pass" : d_Player_Pass_Anwsers,
	"Failed" : d_Player_Failed_Answers,
	"Complete" : d_Player_Complete_Answers }
```

Kuva 26. Dialogi-tiedostojen lataus NPC:lle. Dialogit ladataan omina tekstitiedostoinaan *dictionary*-tyyppisiin säiliöihin.

Godot Enginessä kirjastot (*Dictionary)* toimivat samoin kuin C++:ssa *map*-säiliöt. Kirjastolle annetaan *key*-avain, joka tässä tapauksessa on esimerkiksi *”Intro”*. Kirjastoon annetaan myös *value* eli arvo (tässä tilanteessa tekstitiedosto), jota *key*-avain osoittaa. Oikean dialogin ja vastausten näyttämisestä vastaa NPC-näkymään toteutettu *NPC_Dialogue*-solmu, josta tieto lähetetään pelaajan dialoginäkymään. (Kuva 27.) Pelaajan dialoginäkymä sijaitsee käyttöliittymänäkymän alla, sillä se sisältää ainoastaan käyttöliittymän ominaisuuksia. *NPC_Dialogue* vastaa kaikesta tiedon siirtämisestä NPC:n ja pelaajan välillä ja pitää huolen myös dialogin tilan oikeasta vaihtamisesta. (Kuva 28.).

```
func _check_Answer(currentState) :
	if (Player_Main.has(currentState)) :
		_add_Answer(Player_Main[currentState])
		Player_Main_Dialog = Player_Main[currentState]
	pass

func _add_Answer(dictionary) :
	var dialogKeys = dictionary.keys()
	for i in range(dialogKeys.size()) :
		get_tree().get_root().get_node(
		"Base_BaseNode/UI_BaseNode/PlayerHUD_BaseNode").Player_Dia-
		logue_Scene._init_Player_Text(dictionary[dialogKeys[i]])
	pass

func _add_NPC_Mission_Text(currentState) :
	print("Current State ", currentState)
	get_tree().get_root().get_node(
	"Base_BaseNode/UI_BaseNode/PlayerHUD_Base-
	Node").Player_Dilogue_Scene._init_NPC_Text(NPC_Dialog[currentState])
	pass

func _add_Optional_Answer(dictionary) :
	var dialogKeys = dictionary.keys()
	for i in range(dialogKeys.size()) :
		get_tree().get_root().get_node(
		"Base_BaseNode/UI_BaseNode/PlayerHUD_BaseNode").Player_Dia-
		logue_Scene._init_Player_Text(dictionary[dialogKeys[i]])
	pass
```

Kuva 27. Dialogin lisääminen pelaajan dialoginäkymään.

```
match answerID:
	"Start Mission":
		_start_Mission()
		_update_Dialog()
		pass
	"State 1":
		Dialog_State = "State 1"
		_update_Dialog()
		_process_NPC()
		pass
	"Pass":
		Dialog_State = "Pass"
		_update_Dialog()
		_send_Data_toGlobal()
		_process_NPC()
		pass
	"Failed":
		Dialog_State = "Failed"
		_update_Dialog()
		_send_Data_toGlobal()
		_process_NPC()
		pass
	"Complete":
		Dialog_State = "Complete"
		_update_Dialog()
		_send_Data_toGlobal()
		_process_NPC()
		pass
	"Quit":
		_end_Dialogue()
		_set_toDefaults()
		pass
	null:
```

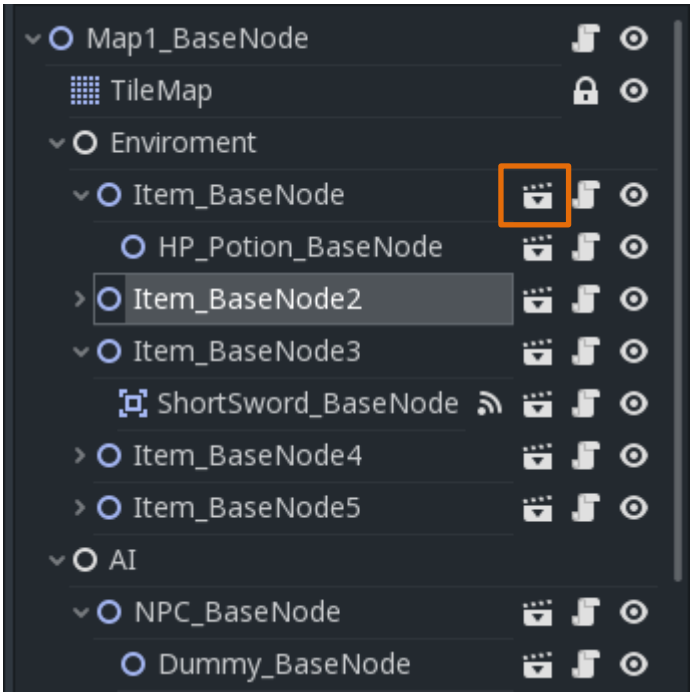
Kuva 28. Dialogin tilakone.

Pelaajan dialoginäkymä hoitaa dialogin tulostamisen pelaajalle. Tässä näkymässä toteutetaan myös pelaajan reagointi dialogiin ja vastaukset lähetään takaisin NPC-näkymän *NPC_Dialogue*–solmuun, joka taas reagoi dialogin vastauksen perusteella. (Kuva 29.)

Kuva 29. Dialogi NPC:n kanssa.

3.7 Kenttänäkymä

Kenttänäkymä koostuu perinteisestä kuvatiiliruudukosta (*Tilemap*) ja kenttään instansoiduista näkymistä. Kenttänäkymään luokkakaavion mukaan instansoidaan (*instanced*) NPC-näkymä ja esinenäkymä (kuva 6). Nämä näkymät instansoidaan suoraan kenttänäkymän editorissa, jossa näiden käsittely olisi helpompaa. Alustetun näkymän tunnistaa pienestä klaffimerkistä, skriptimerkin vieressä. (Kuva 30.)

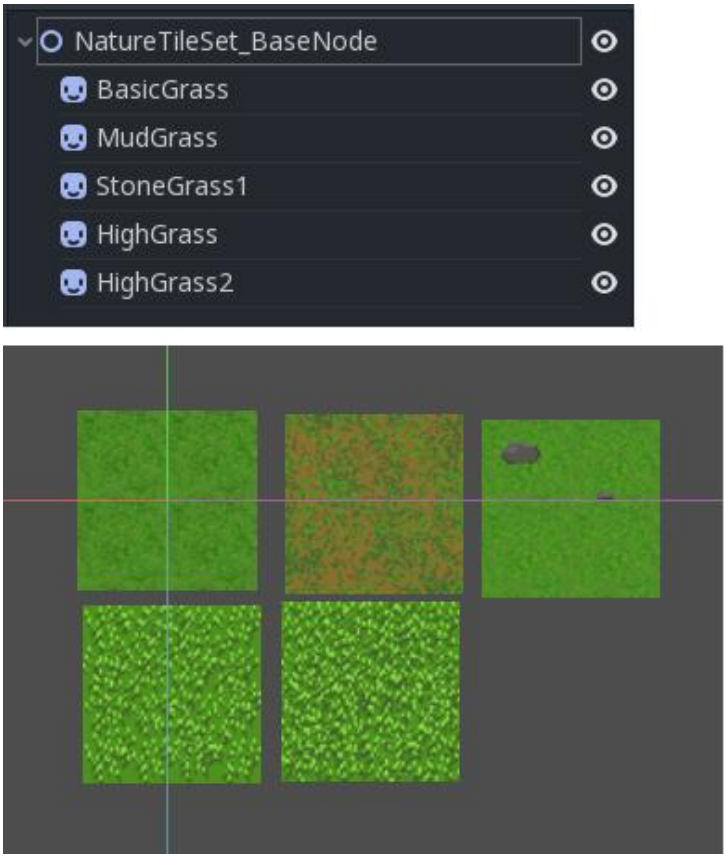


Kuva 30. Kenttänäkymä. Kaikki solmut *Enviromentin*, sekä *AI:n* alla ovat instansoituja solmuja.

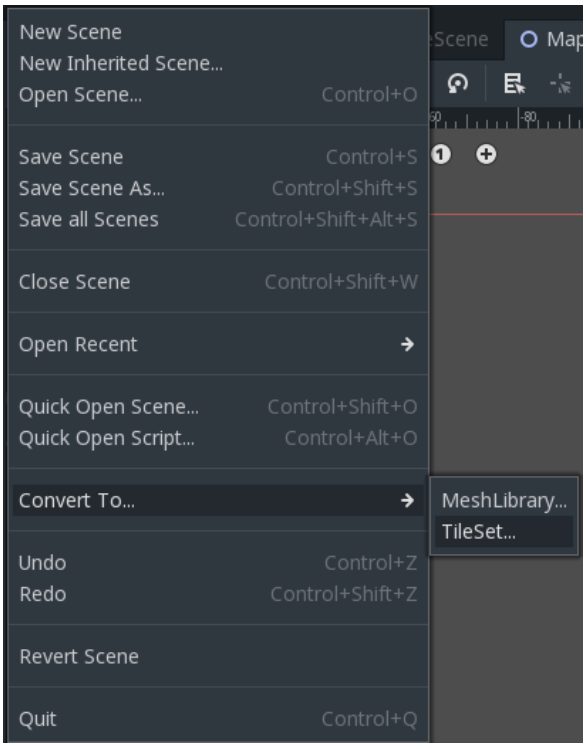
Tällä tavoin kenttään voidaan helposti lisätä haluttuja objekteja ja tekoälyvihollisia ja niiden sijaintia kartalla on helppo vaihtaa tarvittaessa. (Kuva 31.) Godot Engine mahdollistaa myös instansoitujen näkymien helpon muokkaamisen, koska klaffipainiketta painamalla pääsee suoraan siihen instansoituun näkymään tai näkymän omaa skriptiä voi muokata skriptipainiketta painamalla ja näin ollen säästytään editorissa oikean näkymän etsimiseltä. Kenttä-skripti ei itsessään paljon sisällä ominaisuuksia, ainoastaan kentän perustiedot kuten nimi löytyvät skriptistä, sekä tallennusfunktio, joka pakkaa kentän oikein tallennettaessa.

Kuva 31. Kenttä editorinäkymässä. Oikealla NPC-hahmo, vasemmalla pelaajan poimittavissa olevia esineitä.

Karttatiiliruudukon ja karttatiilimallin (*Tileset)* tekeminen onnistuu Godot Enginellä eikä tähän tehtyjä kolmannen osapuolen sovelluksia tarvita. Ennen karttatiiliruudukko-solmun luontia tulee tehdä karttatiilimalli-tiedosto, jota käytetään sitten kenttään lisätyssä karttatiiliruudukko -solmussa. Karttatiilimallin luonti tapahtuu tavallisen näkymän luonnin tavoin. Pari eroavaisuutta on kuitenkin hyvä huomioida, kuten se, että karttatiilimallinäkymä sisältää ainoastaan sprite-lapsisolmuja (kuva 32) ja se, että karttatiilimallin-näkymä tulee muuttaa Godot Enginen karttatiilimalli-tietotyypiksi. (Kuva 33.)

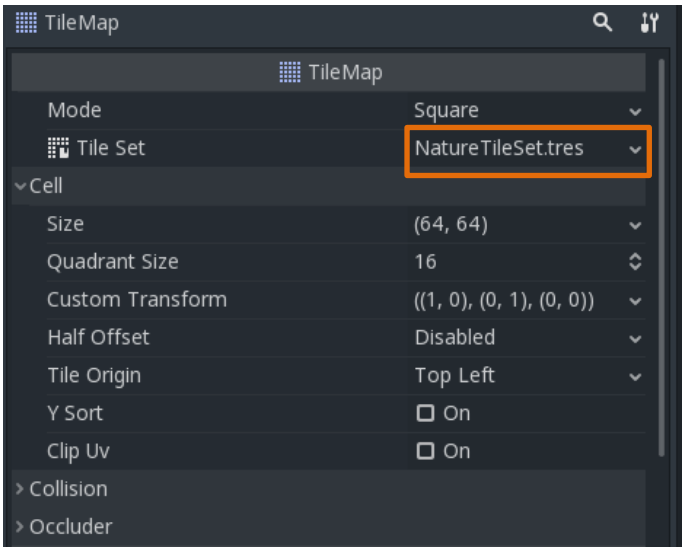


Kuva 32. Karttatiilimalli. Ylhäällä näkymän solmut ja alempana solmujen sisältämät tekstuurit.

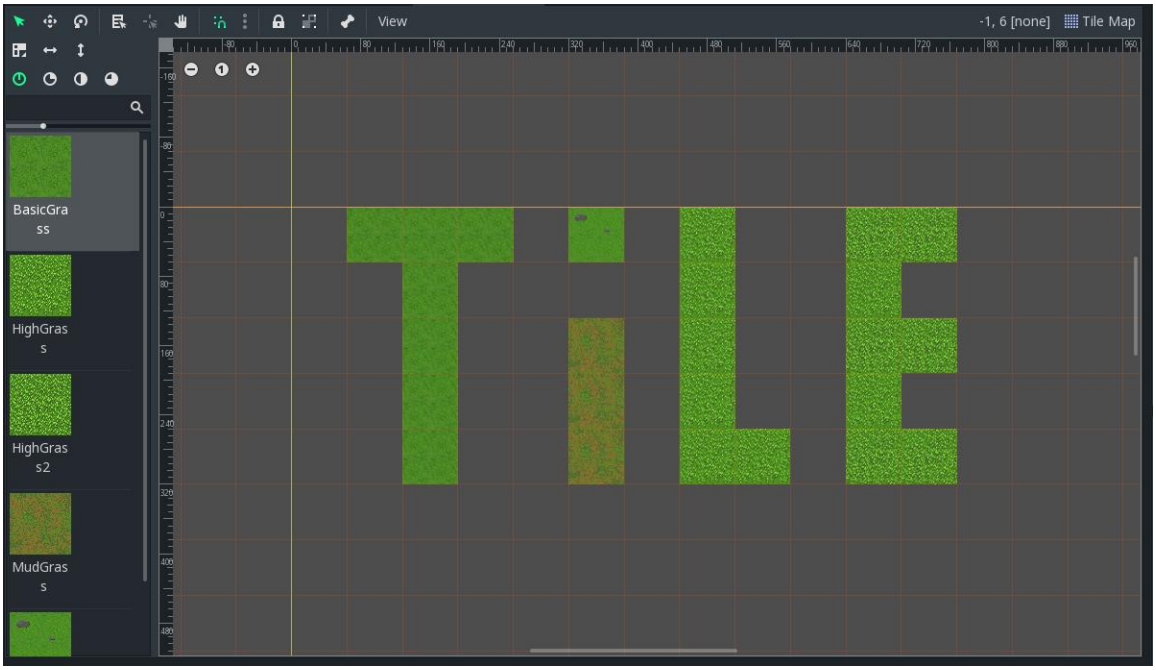


Kuva 33. Karttatiilimallin konvertointi oikeaan tiedostomuotoonsa.

Kun konvertointi on tehty, voidaan haluttuun näkymään lisätä lapsisolmu tyypiltään karttatiiliruudukko ja tämän solmun asetuksiin ladata luotu karttatiilimalli-tiedosto. Asetuksista voidaan myös muuttaa karttatiilimallinkuvakkeiden perusasetuksia, kuten kuvakkeen kokoa. (Kuva 34.). Karttatiiliruudukon käyttö ja kenttien luonti on yksinkertaista, karttatiilimallin sprite-kuvakkeet latautuvat karttatiiliruudukon editoriin, jossa niitä voi asetella haluttuihin paikkoihin hiirtä käyttäen. (Kuva 35.) Mikäli karttatiilimalliin haluaa lisätä uusia kuvakkeita, tulee karttatiilimalli-näkymä konvertoida uudestaan uusien kuvakkeiden kanssa.



Kuva 34. Kuvatiilimallin asetukset kuvatiiliruudukko solmussa.



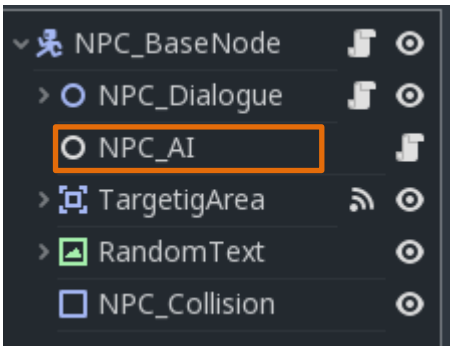
Kuva 35. Kuvatiiliruudukon editorinäkymä.

3.8 AI

AI:n luontiin Godot tarjoaa vielä varsin rajallisia ominaisuuksia pelimoottorin puolesta. Reitinhakuun löytyy hyviä ominaisuuksia, kuten pelimoottoriin sisäänkirjoitettu A*-reitinhaku [15.], mutta muut ominaisuudet täytyy itse suunnitella ja toteuttaa. Varsinkin hyvin monimutkaisten AI-vihollisten toteutus voi osoittautua hankalaksi.

3.8.1 Tilakone

Yleisesti tekoälyn toteuttaminen kannattaa aloittaa tilakoneen suunnittelulla. Tilakoneen skriptin sijoitus pitää suunnitella oman projektin puitteissa, mutta tässä projektissa se sijoitettiin NPC-näkymän lapsisolmuksi nimeltä *NPC_AI*. (Kuva 36.)



Kuva 36. NPC-näkymän solmut.

Tilakoneessa käytetään *enum*-tyyppisiä muuttujia määrittelemään AI:n sen hetkinen tila. (Kuva 37.) Tilalla on tarkoitus määritellä, mitä AI:n tulisi kulloinkin tehdä ja näyttää mahdollisimman ”ihmismäiseltä”. Esimerkiksi *Move*- eli liikkumistilassa AI:n tarkoitus on liikkua ja hakea omaa reittiään reitinhaun kautta, kun taas *Idle*- eli toimettomassa tilassa AI on vain paikallaan sille määrätyn toimettomuusajan verran. (Kuva 38.) Tilakoneen tehtävä on myös hoitaa oikeat animaatiot AI:lle.

```
func _NPC_State_Machine(state):
	match state:
		States.Idle:
			_NPC_Idle()
			return
		States.Move:
			_NPC_Movement()
			return
		States.Speak:
			_NPC_Speak()
			return
		States.Attack:
			_NPC_Attack()
			return
		States.Hit:
			_NPC_Hit()
			return
		States.FallBack:
			_NPC_FallBack()
			return
		States.Die:
			_NPC_Die()
			return
```

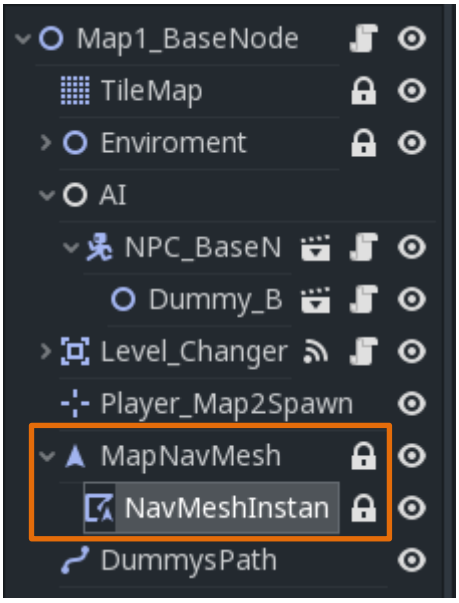
Kuva 37. Tilakone

```
func _NPC_Idle():

	if (IdleTime == 0):
		CurrentState = States.Move
		return

	match get_parent().NPC_Direction:
		"Down":
			get_parent().npcAnimation_Hands = "Idle_Down"
			get_parent().npcAnimation_Torso = "Idle_Down"
			get_parent().npcAnimation_Head = "Idle_Down"
			get_parent().npcAnimation_Legs = "Idle_Down"

		"Up": ...

		"Right": ...

		"Left": ...

	get_parent()._select_animation()

	if (!isIdle):
		Idle_Timer = Timer.new()
		Idle_Timer.set_one_shot(true)
		Idle_Timer.wait_time = IdleTime
		Idle_Timer.connect("timeout", self, "_NPC_Idle_Done")
		add_child(Idle_Timer)
		Idle_Timer.start()
		isIdle = true
		pass
pass
```

Kuva 38. Tilakoneen *Idle*-tila

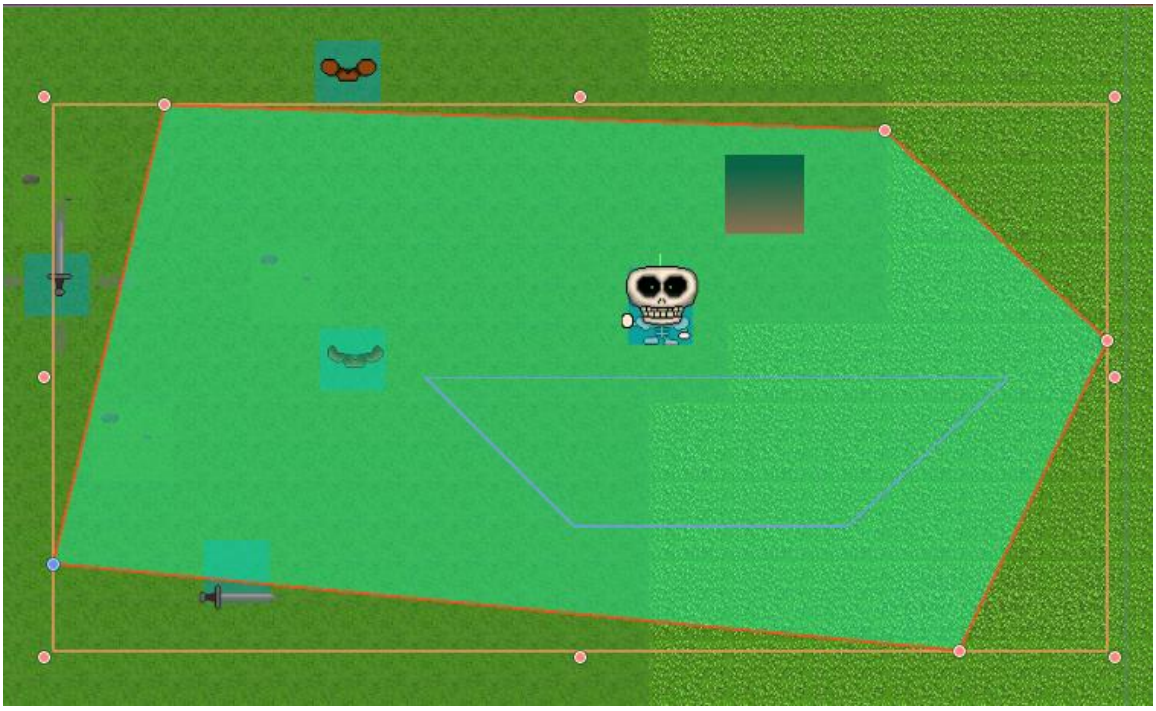
3.8.2 Liikkuminen

Opinnäytetyöprojektissa liikkuminen toteutettiin *Navmesh*-navigointiominaisuutta käyttäen. Tämä *Navmesh*-navigointi muistuttaa Unreal Engine 4:ssa olevaa samankaltaista *Navmesh*-ominaisuutta ja onkin käytöltään yhtä yksinkertainen. Kenttään lisätään lapsisolmu, tyypiltään *Navigation2D,* ja tälle solmulle lisätään *NavigationPolygonInstance*-tyyppinen lapsisolmu, joka pitää sisällään itse navigaatiokartan. (Kuva 39.)

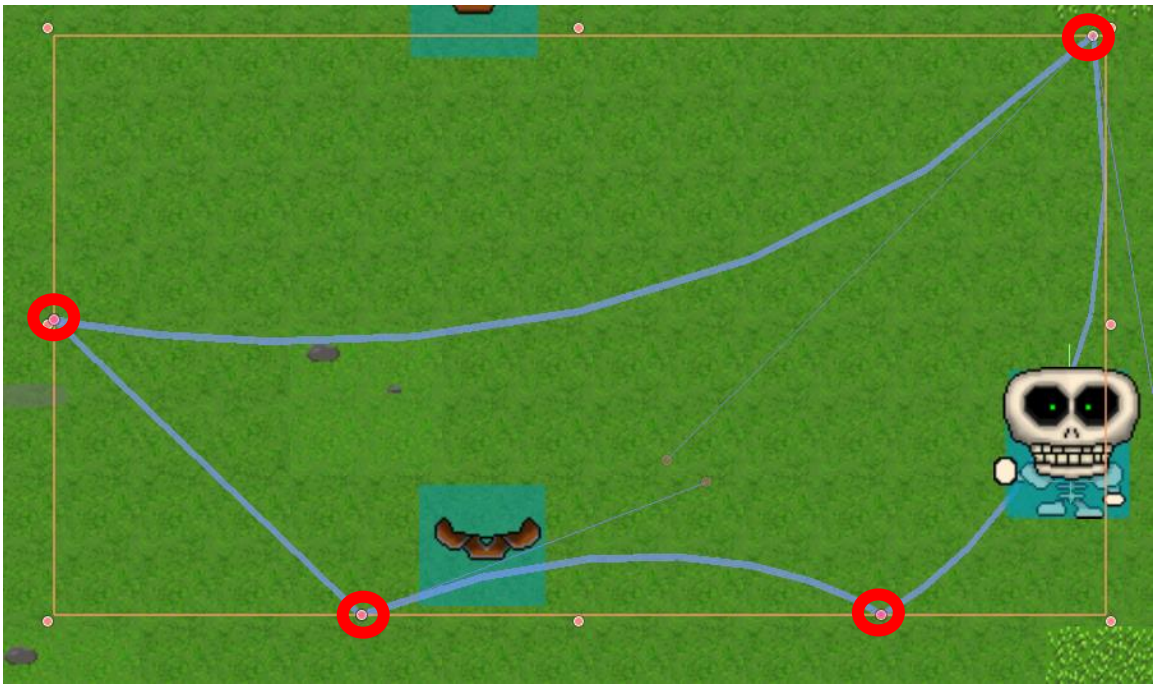


Kuva 39. Navigointikomponentit kenttänäkymässä.

*NavigationPolygonInstance*a on mahdollista muokata mielensä mukaan, ja näin ollen navigointialueiden määrääminen NPC-hahmoille yksilöllisesti on mahdollista. (Kuva 40.) Vaihtoehtoisesti voidaan käyttää A*-reitinhakua tai *waypointeja* eli reittipisteitä. Reittipisteitä voidaan käyttää *navmesh*-navigoinnissa tukevana komponenttina tai kokonaan erillään ilman *navmesh*-navigointia. Reittipisteitä käytettäessä erillään on huomioitava reitin selvyys ja mahdolliset ongelmakohdat, sillä reittipisteet eivät itsessään sisällä mitään reitinhakua, vaan reitti haetaan suoraan pisteeltä toiselle. Reittipisteiden välistä rataa on mahdollista hieman muokata, esimerkiksi tekemällä reitistä kaarimaisen. (Kuva 41.)



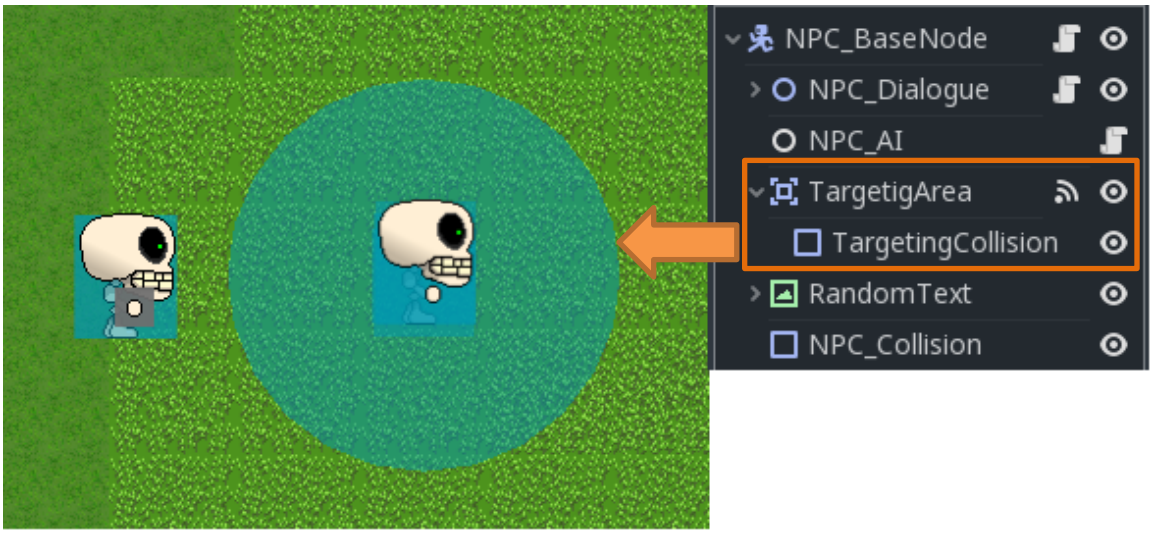
Kuva 40. *Navmesh*-komponentti näkyy sinisen vihreänä alueena.



Kuva 41. Reittipiste-komponentti. Reittipisteet ympyröity punaisella. AI:n käyttämä reitti kuvattu sinisellä viivalla reittipisteiden välillä.

3.8.3 Pelaajan seuraaminen

AI:n reagointi eri tilanteisiin toteutetaan tilakoneen kautta tekemällä tarkistuksia ja AI:n tilaa vaihtamalla. Esimerkiksi pelaajan seuraaminen aloitetaan, kun pelaaja astuu NPC-näkymässä sijaitsevaan *TargetingArea*-solmuun, joka on tyypiltään *Area2D*. (Kuva 42.) Tätä solmua voi pitää *triggerinä* eli aktivointiboksina tälle tilanvaihdolle.



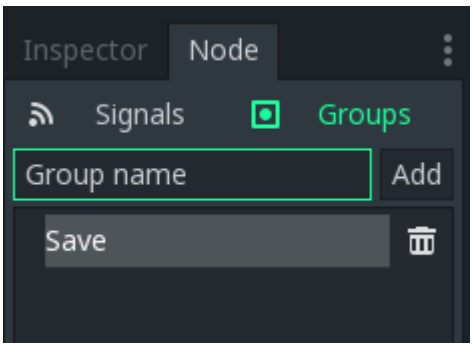
Kuva 42. AI:n *triggeriboksi*. (Pelaaja vasemmalla, NPC oikealla).

Pelaajan astuessa *Area2D*:n törmäysboksiin saa AI pääkohteekseen pelaajan hahmon, ja tällöin AI:n tilakone vaihtaa tilan liikkumistilaan ja päämääränä toimii pelaajan koordinaatit. Pelaajan poistuessa tästä *triggeristä* eli aktivointiboksista AI käynnistää ajastimen, jonka tarkoituksena on tietyn ajan kuluessa palata takaisin normaaliin liikkumistilaan, jossa NPC seuraa omia reittipisteitään. Tällä tavoin NPC ei seuraa pelaajaa loputtomiin, vaan ajastimen ajan kuluessa loppuun luodaan illuusio siitä, että AI luovuttaisi pelaajan jahtaamisen.

3.9 Pelitilanteen tallentaminen ja lataaminen

Godot Enginessä videopelin sisäisen tilanteen tallennuksen voi suorittaa usealla eri tavalla ja parasta onkin miettiä omaan projektiin sopivin rakenne. Tässä luvussa esittelen pari yksinkertaista vaihtoehtoa, joita opinnäytetyössä on käytetty.

Yksi tapa on käyttää solmuihin liitettyä ryhmää, jonka tarkoituksena on määritellä tallennettavat solmut. [16.] Tässä mallissa luodaan ryhmä, johon lisätään solmuja, joita tallennustiedostoon tulee tallentaa. (Kuva 43.)



Kuva 43. Tallennusryhmän luonti.

Solmussa itsessään tulee toteuttaa datan sarjallistaminen eli funktio, jossa käsitellään tallennettava data. Tämä funktio tulee olla kaikissa tallennusryhmään kuuluvissa solmuissa. Funktio voi olla esimerkiksi kuvan 44 mukainen.

```
func save() :
	var save_dict = {
		"filename" : get_filename(),
		"parent" : get_parent().get_path(),
		"pos_x" : position.x, # Vector2 is not supported by JSON
		"pos_y" : position.y,
		"attack" : attack,
		"defense" : defense,
		"current_health" : current_health,
		"max_health" : max_health,
		"damage" : damage,
		"regen" : regen,
		"experience" : experience,
		"tnl" : tnl,
		"level" : level,
		"attack_growth" : attack_growth,
		"defense_growth" : defense_growth,
		"health_growth" : health_growth,
		"is_alive" : is_alive,
		"last_attack" : last_attack
	}
return save_dict
```

Kuva 44. Datan sarjoittamisfunktio. [16.]

Tallennusfunktiossa taas käydään kaikki tähän ryhmään kuuluvat solmut läpi, joten suurien kokonaisuuksien tallentaminen on helppoa. Tässä funktiossa kutsutaan myös solmussa sisältävän datan sarjoittamisfunktiota. (Kuva 45.)

```
#Haetaan kaikki tallennusryhmään kuuluvat solmut.
var save_nodes = get_tree().get_nodes_in_group("Save")
	for i in save_nodes :
		#Kutsutaan datan sarjoittamis - funktiota.
		var save_data = i.call("save")
```

Kuva 45. Tallennusfunktio

Tallennusfunktiossa data tallennetaan esimerkiksi tekstitiedostoon (kuva 46), mutta on suositeltavaa käyttää jotain muuta kuin txt-päätettä, esimerkiksi save-päätettä.

```
func save_game() :
	var save_game = File.new()
	save_game.open("save_path + file.save", File.WRITE)
	var save_nodes = get_tree().get_nodes_in_group("Save")
	for i in save_nodes :
		var node_data = i.call("save");
		save_game.store_line(to_json(node_data))
	save_game.close()
```

Kuva 46. Tallennustiedoston kirjoittaminen. [16.]

Kun tiedoston pääte vaihdetaan txt-päätteestä joksikin muuksi, käyttöjärjestelmä ei vakiona pysty avaamaan tiedostoa ja käyttäjä ei pääse muokkaamaan tallennustiedostoaan itse. Kuitenkin hieman kokeneempi käyttäjä pystyy vaihtamaan tallennustiedoston päätteen itse ja näin ollen myös muokkaamaan tiedostoa. Siksi onkin suositeltavaa kryptata tallennustiedosto. Kryptausta ei tässä opinnäytetyöstä käsitellä, mutta se on Godot Enginessä varsin yksinkertaista. [17.]

Esitellyn tallennustiedoston lataaminen on samankaltainen prosessi kuin tallentaminenkin. (Kuva 47.) Muistettava kuitenkin on, että kuvassa 47 esitetty lataamisfunktio ei välttämättä sovellu kaikkiin projekteihin, mutta yksinkertaisimmillaan prosessi on kuvan 47 mukainen, jossa käytännössä vain avataan tallennustiedoston kansio ja luetaan sen sisältämät komponentit sen hetkiseen näkymään.

```
func load_game() :

#Avataan kansio
var save_game = File.new()
	if not save_game.file_exists(Save_Path) :
		return

#Koska pelissä yleensä on jo olemassa nämä samat komponentit,
#tulee jo olemassa olevat komponentit poistaa.
var save_nodes = get_tree().get_nodes_in_group("Save")
	for i in save_nodes :
		i.queue_free()

	#Avataan tallennustiedosto
	save_game.open(Save_Path, File.READ)
	while not save_game.eof_reached() :
		var current_line = parse_json(save_game.get_line())

		#Luodaan objekti ja asetetaan positio
		var new_object = load(current_line["filename"]).instance()
		get_node(current_line["parent"]).add_child(new_object)
		new_object.position = Vector2(current_line["pos_x"],
		current_line["pos_y"]))

		#Asetetaan puuttuvat data objektin muuttujiin.
		for i in current_line.keys() :
			if i == "filename" or i == "pos_x" or i == "pos_y" :
				continue
			new_object.set(i, current_line[i])
	save_game.close()
```

Kuva 47. Tiedoston lataaminen. [16.]

Godot Enginessä on myös mahdollista pakata kokonainen näkymä ja tallentaa tämä sellaisenaan kuin se on. Tällä tyylillä esimerkiksi kenttien tallentaminen on helppoa, eikä kaikkia kentän komponentteja tarvitse erikseen sarjoittaa, vaan pakattukenttä hoitaa tämän itse. (Kuva 48.)

```
func _save_Scene() :
	var packed_Scene = PackedScene.new()
	packed_Scene.pack(self)
	ResourceSaver.save("Save_Path + Scene_Name" + ".tscn", packed_Scene)
	pass
```

Kuva 48. Näkymän pakkaus ja tallennus.

Pakatun näkymän lataaminen on hyvin suoraviivainen operaatio, samoin kuin tallentaminen. Lataamisfunktiossa avataan kansio, mihin pakattu näkymä on tallennettu ja suoritetaan alustus. (Kuva 49.)

```
func _load_Map(mapName) :

	var map = null
	var dir = Directory.new()

	#Avataan kansio
	dir.open(save_Path)

	#Jos tallennettu kartta löytyy
	#Ladataan se muuttujaan "map".
	#Map_Scene on tämän hetkinen kenttä - skene,
	#joten alustetaan nykyiseen kenttään ladattu kenttä.

	if (dir.file_exists(save_Path + mapName + ".tscn")) :
		map = load(save_Path + mapFile + ".tscn")
		Map_Scene = map.instance()
	else:
		print("Packed Scene not found!")

pass
```

Kuva 49. Pakatun näkymän lataaminen.

Godot Enginellä on yleisesti katsoen helppo toteuttaa tallennus- ja lataamisominaisuudet, mutta näiden asioiden helppous riippunee suuresti oman projektin rakenteesta. Opinnäytetyö projektissa käytettiin näiden kahden esiteltyjen ominaisuuksien yhdistelmää ja lataaminen suoritettiin useassa näkymässä tilannekohtaisesti, joten ratkaisutapoja on useita erilaisia.

3.10 Muut työkalut

Projektin grafiikka toteutettiin käyttäen Aseprite-nimistä grafiikkasovellusta. Aseprite on erikoistunut juuri pixel-tyyppiseen piirtämiseen ja on myös hyvin kevytkäyttöinen sovellus, minkä vuoksi se soveltui parhaiten projektiin. Joitakin kokeiluja grafiikan parantamiseksi tehtiin ShaderMap-ohjelmistolla, joka on erikoistunut tekstuurien ja 3D-mallien tekoon. Esimerkiksi kuvan 41 hahmon pää on jälkikäsitelty käyttäen ShaderMap-ohjelmistoa. Versionhallinnassa käytettiin Github-palvelua ja git-pohjaista versionhallintaa.

4 Yhteenveto

Godot Engine on nuoresta iästään ja pienestä huomiostaan huolimatta kelpo ja hyvin toimiva pelimoottori. Godot Enginellä työskentely oli vaivatonta, johtuen suurelta osin kolmannen osapuolen sovelluksien puuttumisesta, pelimoottorin kevytrakenteisuudesta ja pelimoottorin pienestä tiedostokoosta. Kevytrakenteisuuden sekä pienen tiedostokoon voisi olettaa olevan toissijaista, mutta Unity3D:tä ja Unreal Engine 4:ta käyttäneenä voin sanoa, että raskaiden pelimoottorien käyttö, päivitys ja asentaminen vievät paljon aikaa pois itse pelinkehityksestä. Mikään pelimoottori ei toimi moitteetta, ja myös uudelleen käynnistyksen menevä aika voi vaikuttaa peliprojektin etenemiseen. Godot Enginellä näitä ongelmia ei ollut, ja vaikka pelimoottori ”kaatuili” silloin tällöin, ei kulunut kauan, kun pelimoottorin jo saattoi käynnistää uudelleen ja tästä syystä työnkulku (*workflow*) ei päässyt katkeamaan. Nopea ja vaivattomasti toimiva sovellus on yksi Godot Enginen parhaista puolista.

Suurin Godot Enginen ongelma, oli näkymä editorin käyttöliittymässä, kun yritin liikutella 2D-komponentteja, sillä jos useita komponentteja on päällekkäin, oli halutun komponentit siirtäminen vaikeaa. Ongelman voi ratkaista käyttämällä lukkoja, mikä estää tietyn komponentin siirtelyn. Lukkojen käyttö voi osoittautua etenkin suurissa projekteissa hyvinkin vaivalloiseksi. Helpoin tapa ”korjata” ongelma oli valita kursori ”Liikuttelu”-tilaan, jolloin tätä ongelmaa ei juurikaan ilmennyt. Myös ohjelmoitaessa haluttua skriptiä on muistettava pitää skriptin näkymä auki, sillä muutoin skripti ei löydä näkymän solmuja, joihin skriptissä halutaan päästä käsiksi. Koodi kyllä toimii, mikäli virheitä ei ohjelmoitaessa synny, mutta virheiden minimoimiseksi on hyvä pitää oikeaa näkymää aktiivisena. Tämä ominaisuus aiheutti välillä sekaannuksia, sillä kävi tilanteita, joissa olin valinnut väärän näkymän aktiiviseksi ja yritin skriptissä päästä käsiksi tiettyyn solmuun, mutta ohjelmointieditori ei solmua löytänyt. Voisi siis sanoa, että usean näkymän ohjelmointi kerralla voi osoittaa omia haasteitaan.

Juurisolmun käyttöä myös kannattaa miettiä, sillä se tarjoaa hyvinkin helppoja ratkaisuja ohjelmointiin, mutta en itse pidä niitä optimaalisina. Koska koko projekti tulee lähtökohtaisesti saman juurisolmun alle, on mahdollista tämän solmun kautta päästä käsiksi kaikkiin projektissa oleviin solmuihin ja näkymiin. Tämä siis mahdollistaa helpon, mutta kovakoodatun ratkaisun ja tähän sorruin itsekin varsinkin uusia ominaisuuksia ohjelmoitaessa. Tämä ohjelmointitapa ei ole epätoimiva, mutta jälkikäteen asioita päivittäessä tai projektin rakenteen muuttuessa koodin voi joutua uudelleenkirjoittamaan suurelta osin. Jos juurisolmun kautta joutuu projektia rakentamaan, on suunnitteluvaiheessa sattunut virhe ja kannattaa tarkastaa esimerkiksi luokkakaavio uudelleen.

Projektina oli toteuttaa 2D-seikkailu- ja roolipelin alpha-versio, joka sisältää 2D-pelin peruskomponentteja. Pääpaino oli siis core-elementtien eli peruselementtien havainnoinnissa ja niiden käytännön suunnittelussa ja toteutuksessa käyttäen Godot Engineä. Projektiin oli lisätty myös elementtejä, joita en ollut aikaisemmin toteuttanut, esimerkiksi tallennusmahdollisuus ja dialogisysteemi. Näin ollen pääsin myös kehittämään omia taitojani pelinkehityksessä.

Projekti onnistui tyydyttävästi, ja kaikki opinnäytetyössä kertomani ominaisuudet toimivat niin kuin pitääkin. Suurin virhe oli inventaarion suunnittelussa, sillä itse inventaario sijaitsee nyt pelaajan käyttöliittymänäkymässä, eikä sen sijainti tällöin käyttöliittymänäkymän lapsena ole aiheellista. Inventaarion tulisi ennemmin sijaita pelaajanäkymässä omana solmunaan. Grafiikan osalta peliprojekti on kohtalaista, mutta ajan puutteen vuoksi en hienosäätöä grafiikan osalta ehtinyt tekemään. Grafiikka täyttää kuitenkin Alpha-vaiheen vaatimukset ja on peruskuvioiden sijaan paljon parempi vaihtoehto. Peliprojekti löytyy GitHub-palvelusta: https://github.com/TheAspen/TM_Thesis.git

Godot Enginestä ilmestyi myös 3.1. opinnäytetyön tekemisen aikana. [18.] Tätä versiota voin alustavien tietojen mukaan suositella käytettäväksi 3.0.:n sijaan, sillä se tarjoaa uusia ominaisuuksia etenkin 3D-peliohjelmoinnin puolelta, mutta myös 3.0. on varsin pätevä versio käytettäväksi. Uusia versioita Godot Enginestä on viime vuosina ilmestynyt kerran vuodessa, mikä on mielestäni kohtalainen vauhti uusille pelimoottoriversioille. Yhteisö järjestää myös omia tapahtumiaan ympäri maailmaa yleensä isompien pelialan tapahtumien yhteydessä.

Lähteet

(1) Wikipedia. 2018; Available at: https://fi.wikipedia.org/wiki/Godot_Engine.

(2) Godot Engine Sponsor web page. Available at: https://www.patreon.com/godotengine/overview.

(3) Juan Linietsky, Ariel Manzur and the Godot community. Godot Engine Documentation, Godot's design philosophy. 2014; Available at: https://docs.godotengine.org/en/3.0/getting_started/step_by_step/godot_design_philosophy.html.

(4) Juan Linietsky, Ariel Manzur and the Godot community. Godot Engine Documentation, SceneTree. 2014; Available at: https://docs.godotengine.org/en/3.0/getting_started/step_by_step/scene_tree.html?highlight=scene%20tree.

(5) Juan Linietsky, Ariel Manzur and the Godot community. Godot Engine Documentation, Scenes and nodes. 2014; Available at: http://docs.godotengine.org/en/3.0/getting_started/step_by_step/scenes_and_nodes.html.

(6) Juan Linietsky, Ariel Manzur and the Godot community. Godot Engine Documentation, GDScript. 2014; Available at: https://docs.godotengine.org/en/3.0/getting_started/scripting/gdscript/gdscript_basics.html.

(7) Godot community. Godot Engine - Finnish documentation. Available at: https://hosted.weblate.org/languages/fi/godot-engine/.

(8) Juan Linietsky, Ariel Manzur and contributors Godot. Godot Engine web page, License. 2007; Available at: https://godotengine.org/license.

(9) Juan Linietsky, Ariel Manzur and the Godot community. Godot Engine Documentation, Console Support. 2014; Available at: https://docs.godotengine.org/en/3.0/tutorials/platform/consoles.html.

(10) Hugo Locurcio and Juan Linietsky. Godot 3.0: Introducing the New and Outstanding Features. YouTube 2017 Aug 10,.

(11) Juan Linietsky, Ariel Manzur and contributors Godot. Godot homepage, features. 2017; Available at: https://godotengine.org/features.

(12) Juan Linietsky, Ariel Manzur and the Godot community. Godot Engine Documentation, Singletons. 2014; Available at: http://docs.godotengine.org/en/3.0/getting_started/step_by_step/singletons_autoload.html.

(13) Juan Linietsky, Ariel Manzur and the Godot community. Godot Engine Documentation, Physics introduction. 2014; Available at: https://docs.godotengine.org/en/3.0/tutorials/physics/physics_introduction.html?highlight=area2d.

(14) Juan Linietsky, Ariel Manzur and the Godot community. Godot Engine Documentation, Popup. 2014; Available at: https://docs.godotengine.org/en/3.0/classes/class_popup.html#class-popup.

(15) Juan Linietsky, Ariel Manzur and the Godot community. Godot Engine Documentation, A-star. 2014; Available at: https://docs.godotengine.org/en/3.1/classes/class_astar.html?highlight=astar.

(16) Juan Linietsky, Ariel Manzur and the Godot community. Godot Engine Documentation, Saving games. 2014; Available at: https://docs.godotengine.org/en/3.0/tutorials/io/saving_games.html?highlight=save%20game.

(17) Juan Linietsky, Ariel Manzur and the Godot community. Godot Engine Documentation, Encrypting save games. 2014; Available at: https://docs.godotengine.org/en/3.0/tutorials/io/encrypting_save_games.html.

(18) Juan Linietsky. Godot Engine 3.1. Release. 2019; Available at: https://godotengine.org/article/godot-3-1-released.