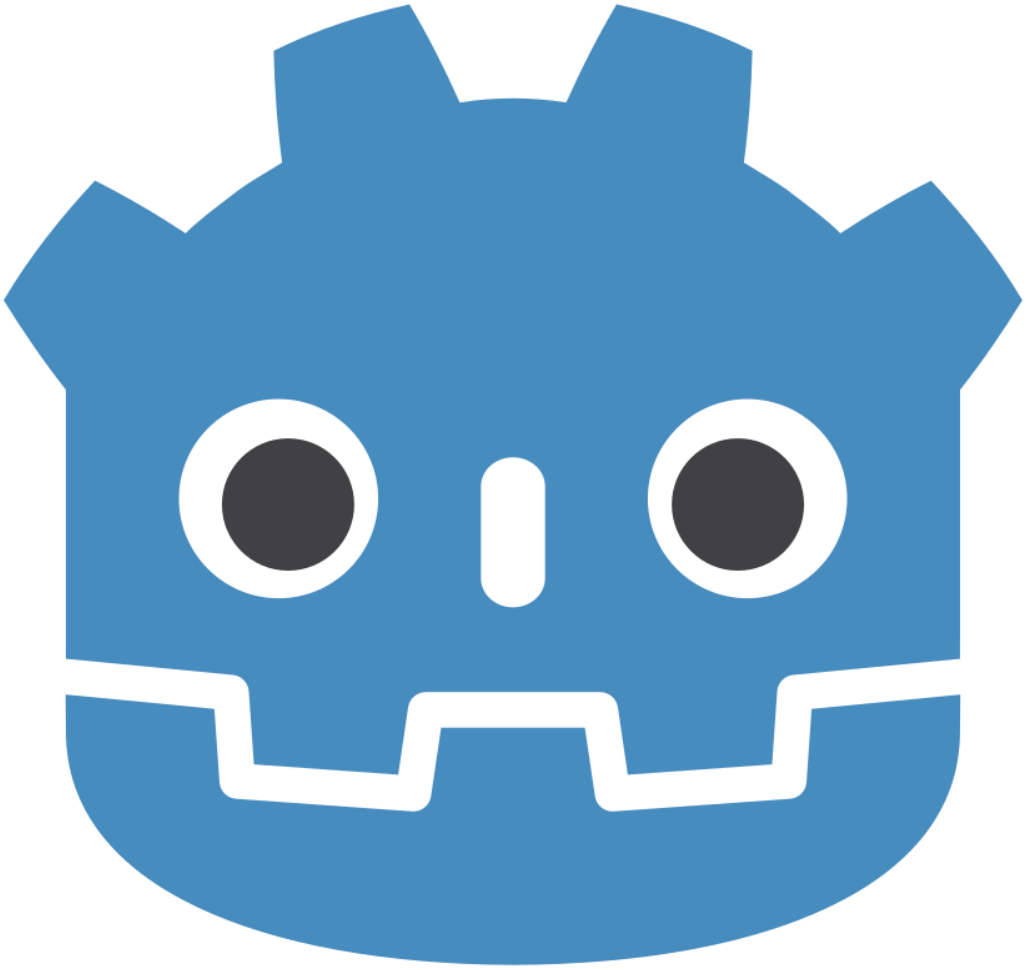
Miika Saajanne

Varjostimien sekä efektien toteutus ja käyttö Godot Engine -pelimoottorissa



Insinööri (AMK)
Tieto- ja viestintätekniikka
Kevät 2021

KAMK • University of Applied Sciences

KAMK • University of Applied Sciences

Tiivistelmä

Tekijä(t): Saajanne Miika

Työn nimi: Varjostimien sekä efektien toteutus ja käyttö Godot Engine -pelimoottorissa

Tutkintonimike: Insinööri (AMK), tieto- viestintätekniikka

Asiasanat: Godot, varjostin, efekti, peliohjelmointi, pelikehitys

Opinnäytetyö on jatkoa peliprojekti Litter Runille. Työn tarkoituksena oli tutkia, kuinka Godot Engine toimii yleisellä tasolla ja kuinka siinä voidaan toteuttaa varjostimia tai efektejä. Opinnäytetyössä käytettiin pelimoottorin versiota Godot Engine 3.2.3. Työssä keskityttiin kaksiulotteisten efektien luomiseen.

Godot Engine on pelimoottori, jolla voi toteuttaa kaksiulotteisia sekä kolmeulotteisia pelejä. Pelimoottoria voi käyttää kaikilla tunnetuimmilla käyttöjärjestelmillä, sekä sillä voi toteuttaa pelejä tietokoneen lisäksi puhelimelle Android sekä IOS-käyttöjärjestelmille. Pelimoottori tukee useampaa ohjelmointikieltä, ja vielä useampaa voidaan käyttää yhteisön tukemilla ohjelmointikielillä. Pelimoottorin suositellut ohjelmointikielet ovat pelimoottorin omat GDScript ja VisualScript. Varjostinohjelmointiin pelimoottori tarjoaa GLSL ES 3.0:n kaltaisen varjostinohjelmointikielen sekä visuaalisen ohjelmointikielen. Varjostinohjelmointikielet ovat vielä hieman puutteellisia.

Litter Run on hyötypeli, ja se edustaa tasoloikka- ja seikkailugenrejä. Pelissä on tarkoituksena löytää erilaisia roskia, joista pelaaja voi niitä löydettyään lukea niiden haitoista ympäristössä. Peliin toteutetut efektit on suunniteltu ajatellen ympäristöä. Kaksi efektiä on luonnollisia ilmiöitä ja kolmas efekti on ihmisen tuottamaa saastetta. Varjostinohjelmointi on pelinkehittäjän hyvin hyödyllistä opetella. Pelejä voidaan elävöittää niiden avulla jopa niin paljon, että peli muistetaan ainoastaan sen upeista varjoista ja heijastuksista. Kaksiulotteisissa peleissä pelinkehittäjä voi opetella vain muutaman tekniikan, joita muokkaamalla voidaan luoda erilaisia efektejä tai varjostimia.

KAMK • University of Applied Sciences

Abstract

Author(s): Saajanne Miika

Title of the Publication: Implementation and Use of Shaders in the Godot Engine.

Degree Title: Bachelor of Engineering, Information and communication technology

Keywords: Godot, Shaders, Game Programming, Game development

The thesis is a continuation of the game project Litter Run. The purpose of this work was to investigate how the Godot Engine works on a general level and how shaders can be implemented in it. The thesis uses the game engine version Godot Engine 3.2.3. The work focuses on creating two-dimensional effects.

Godot is a game engine that can be used to implement two-dimensional as well as three-dimensional games. The game engine can be used with all the most well-known operating systems to develop games, and it can be used to made games for Android and iOS operating systems.

The game engine supports several programming languages, and even more can be used with community-supported programming languages. The recommended programming languages for the game engine are the game engine's own GDScript and VisualScript. For shading programming, the game engine provides similar shading programming language such as GLSL ES 3.0, also it provides visual shading programming language. Godot shading programming languages are still a bit flawed. Shading programming is very useful for a game developer to learn. Games can be used to enliven them so much that the game is remembered only for its stunning shadows, reflections, or effects. In 2D games, the game developer can learn only a few techniques that can be modified to create different effects or shadows.

The project works for the thesis were designed for the Litter Run game, and three different effects have been designed and implemented for it.

Sisällys

1 Johdanto

Opinnäytetyön idea kehittyi työharjoittelun aikana, jonka suoritin Kajak Gamesille. Harjoittelun aikana toteutettiin pienellä ryhmällä mobiilipeliä nimeltä Litter Run. Pelissä pelaaja löytää luonnosta roskia, ja pelin tarkoituksena on opettaa, mitä haittaa niistä on luonnolle. Peliä Kehitettiin Godot Engine -pelimoottorilla. Vastasin peliä kehittäessämme pelin teknisestä puolesta sekä ohjelmoinnista. Pelin kehityksen aikana kiinnostuin entuudestaan tutusta pelimoottorista ja sain idean tutkia, kuinka pelimoottorilla voidaan toteuttaa ja käyttää varjostimia tai efektejä pelimoottorissa.

Opinnäytetyössä käydään läpi Godot Engine -pelimoottoria yleisellä tasolla ja kaksiulotteisia efektejä. Opinnäytetyössä tutkitaan, miten niitä voidaan käyttää pelien kehityksessä pelimoottorissa. Työn aikana toteutetaan myös kolme erilaista efektiä, joiden suunnittelussa käydään läpi, miten ne elävöittävät Litter Run -pelin maailmaa.

Käytän opinnäytetyön tekoon opinnäytetyön aloitushetkellä uusinta vakaata versiota pelimoottorista, Godot Engine 3.2.3.

2 Godot Engine -pelimoottori

2.1 Yleistä

Godot Engine on täysin ilmainen pelimoottori, jolla voidaan toteuttaa kaksi- ja kolmeulotteisia pelejä sekä sovelluksia. Godot on avoimen lähdekoodin alainen ja käyttää hyvin joustavaa MIT-lisenssiä, mikä tarkoittaa sitä, että voi toteuttaa pelejä sekä sovelluksia ilman tarvetta maksaa rojalteja tai muita maksuja Godot Enginen kehittäjille. [1.] Godotin ilmaisuuden mahdollistavat erilliset yritysten tekemät lahjoitukset sekä kuukausittaiset Patreon rahoittajat.

Godot mainostaa itseään ensimmäisenä ilmaisena ja avoimen lähdekoodin omaavana pelimoottorina nykyisten isojen kaupallisten joukossa. Godot on kevyt, vain 30 Mb kokoinen. [2.] Godot on suunniteltu sekä kaksi- että kolmeulotteisten pelien ja sovellusten suunnitteluun ja toteutukseen.

Godot Engine on alustariippumaton, joten sillä voi kehittää pelejä sekä sovelluksia Windows-, macOS-, Linux- ja BSD-käyttöjärjestelmillä. Godot tarjoaa mahdollisuuden kehittää ohjelmistoja edellä mainittujen käyttöjärjestelmien lisäksi myös selaimelle HTML5 ja WebAssemblyn avulla. Kolmannen osapuolen tarjoajat mahdollistavat myös Nintendo Switchin, PlayStation 4 Xbox Onen sovellusten kehittämisen. [3.]

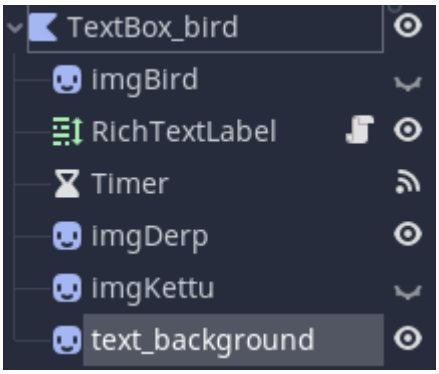
Godot Engine on myös suunniteltu hyväksi tiimitöiden toteuttamiseen. Se tarjoaa laajan mahdollisuuden versionhallintaan esimerkiksi Git:n ja Subversionin avulla. Näkymät on suunniteltu siten, että jokainen voi työskennellä helposti omassa näkymässään, häiritsemättä toisten työtä. Esimerkiksi yksi voi työskennellä pelaajan näkymässä ja toinen kentän suunnittelussa, eivätkä he häiritse toistensa työtä. [3.]

2.2 Solmut ja näkymä

Godot on suunniteltu käytettävyydeltään helpoksi, jotta ammattilaiset sekä aloittelevat voivat tehdä helposti ja mutkattomasti töitä. Tätä varten Godot Enginen peruselementtejä ovat solmut ja näkymät.

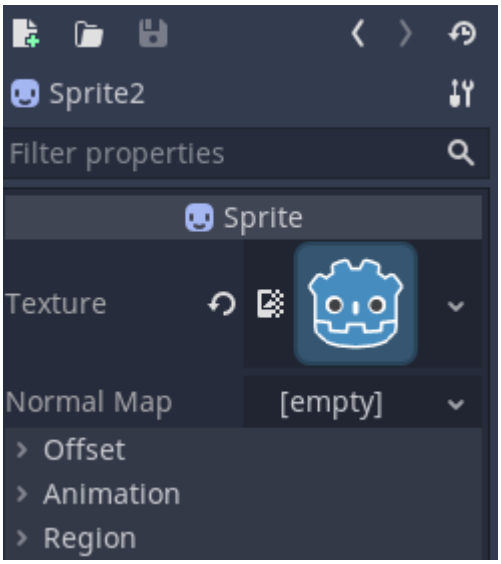
2.2.1 Solmut

Godot Enginellä tehdessä peliä kehittäjän täytyy osata erilaisten solmujen käyttäminen. Jokainen solmu omistaa erilaisia toimintoja. Jokaisella on omat erikoisuutensa. Jokaisella solmulla on oltava nimi, muokattavia ominaisuuksia, funktioon palautuva argumentti, sitä voidaan laajentaa ja sille voidaan lisätä lapsisolmu. [5.] Kuvassa 1 TextBox_bird on vanhempisolmu ja kaikki sen alla ovat lapsisolmuja. Lapsisolmutkin voivat saada omia lapsisolmujaan. Jokainen solmu voidaan myös tallentaa näkymäksi. [6.]



Kuva 1. Esimerkkikuva solmuista [4]

Solmujen ominaisuuksiin päästään käsiksi editorin kautta kuvan 2 tapaisesti. Kuvaa 2 on leikattu siten, että siinä näkyy vain Sprite-solmun omat erikoiset ominaisuutensa. Lisäksi jokainen kaksiulotteinen solmu saa Node2D-solmun ominaisuudet. Ominaisuuksiin päästään myös käsiksi koodin kautta.



Kuva 2. Sprite-solmun ominaisuuksia.

Kuvassa 3 rivillä neljä nähdään, kuinka päästään lapsisolmun koodista vanhempisolmun ominaisuuksiin kiinni, rivillä viisi nähdään, miten päästään solmuun kiinni, mihin koodi on yhdistetty ja rivillä kuusi voidaan nähdä, miten päästään lapsisolmuun kiinni.

```
3 func _physics_process(delta):
4     get_parent().visible = true
5     position = Vector2(200,400)
6     $Sprite2.position = Vector2(100,200)
7     pass
```

Kuva 3. Solmujen ominaisuuksien käyttö koodissa

Laajentamisella tarkoitetaan sitä, että jokainen pelimoottoria käyttävä voi luoda mille tahansa solmulle lisää funktioita. Käyttäjällä on myös oikeus tehdä omia vaihtoehtoisia solmuja pelimoottoriin ja jakaa tekemiään solmujen laajennuksia ja täysin omia solmuja. [5.]

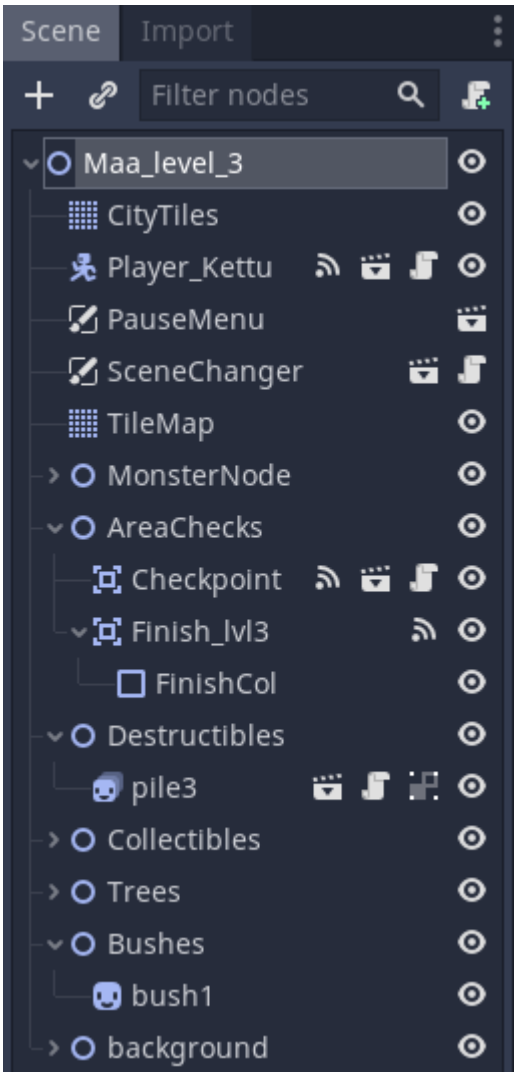
2.2.2 Näkymät

Seuraavaksi solmujen jälkeen Godotin käyttäjien tulisi opetella näkymät. Jokainen peli koostuu vähintään yhdestä näkymästä, niin sanotusta päänäkymästä. Päänäkymä on aina se, jonka kehittäjä haluaa käyttäjän näkevän ensin. Päänäkymä on myös kehittäjän valittava, jotta peli voi avata. [5.] Näkymiä kuitenkin voi olla useampia, niin paljon kuin kehittäjä haluaa. Päänäkymä voi olla valikko, josta siirrytään seuraavaan näkymään, mikä voisi olla ensimmäinen kenttä ja niin edelleen.

Näkymät rakentuvat solmuista. Jokaisessa näkymässä on oltava yksi juurisolmu, joka rakentaa niin sanotusti puuhierarkian lapsisolmuistaan [5].

Kuvasta 4 voidaan nähdä juurisolmu eli Maa_level_3, ja muut solmut näkymässä muodostavat aiemmin mainitun puuhierarkian. Näkymästä voidaan nähdä myös solmut, jotka on tallennettu omiksi näkymiksi.

Jokaisesta näkymästä voidaan myös luoda instanssi [5]. Päänäkymien sisällä olevat mahdolliset muut näkymät ovat niin sanottuja instansseja. Kuvassa 4 nähdään muutamia instansseja, kuten Player_Kettu ja SceneChanger. Jokainen instanssi määrittää omat muuttujat ja metodit. Kun instanssi ladataan, voidaan sen arvoja muokata tai kutsua sen metodeja päänäkymässä. Päänäkymään voidaan ladata samaa instanssia niin monta kertaa kuin halutaan. Instanssien lataamisesta on hyötyä silloin, kun halutaan ladata samaa objektia useampaan kertaan. Esimerkiksi pelissä voi olla monta samaa tekoälyä. Tämä saataisiin helpoiten toteutettua lataamalla päänäkymään tekoälystä tehty näkymä instanssina, eikä luomalla sitä aina uudestaan.



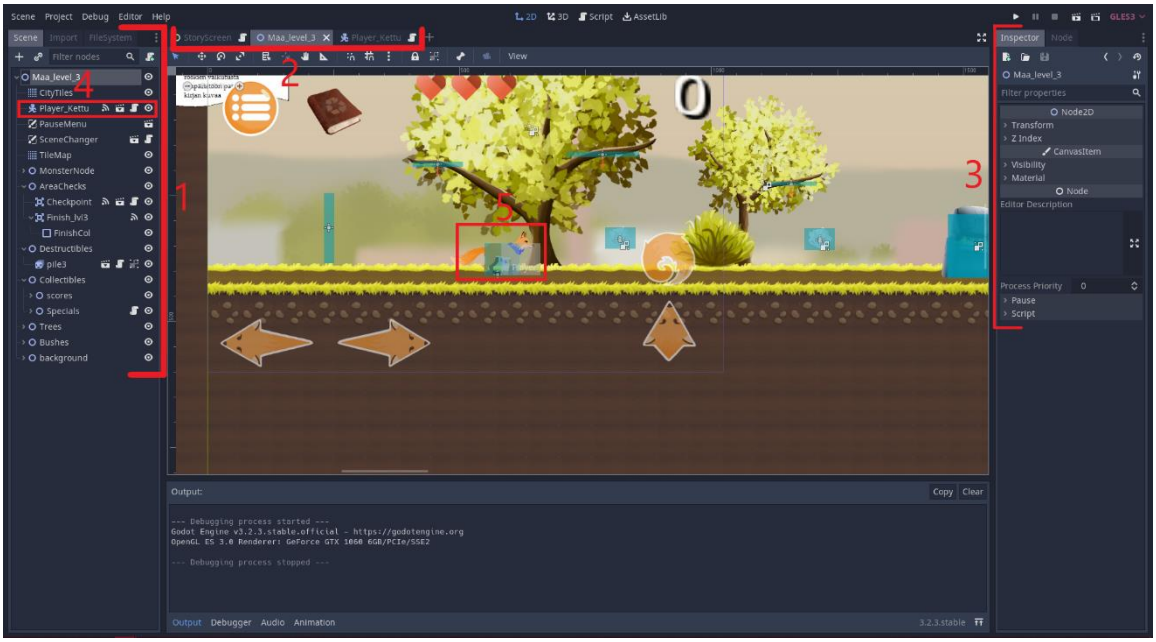
Kuva 4. Havainnollistava kuva, miltä kentän näkymä näyttää. [4]

Kuva 5. Solmujen vieressä oleva kuvake tarkoittaa, että solmu on tallennettu omaksi näkymäksi.

Tilanteessa, jossa solmu vastaanottaa useamman lapsisolmun, tulee kehityksestä paljon helpompaa, kun solmu on tallennettu omaksi näkymäksi. Tästä on huomattavasti hyötyä myös tiimityöskentelyssä; kaksi ihmistä voi työskennellä samassa päänäkymässä. Toinen voi testata pelaajan fysiikoita päänäkymässä ja toinen testata tekoälynäkymää päänäkymässä häiritsemättä toisen työtä. Edellisessä esimerkissä oletetaan, että kumpikin pelaaja sekä tekoäly on tallennettuna

omiin näkymiinsä. Myös molempien projektien liittäminen pääprojektiin on helpompaa, sillä molemmat ovat työskennelleet omissa näkymissään, eikä suoraan päänäkymässä sijaitsevissa solmussa.

Kuvasta 6 voi nähdä Godot Engine -pelimoottorin kokonaisuus näkymineen, solmuineen ja solmujen ominaisuuksineen.



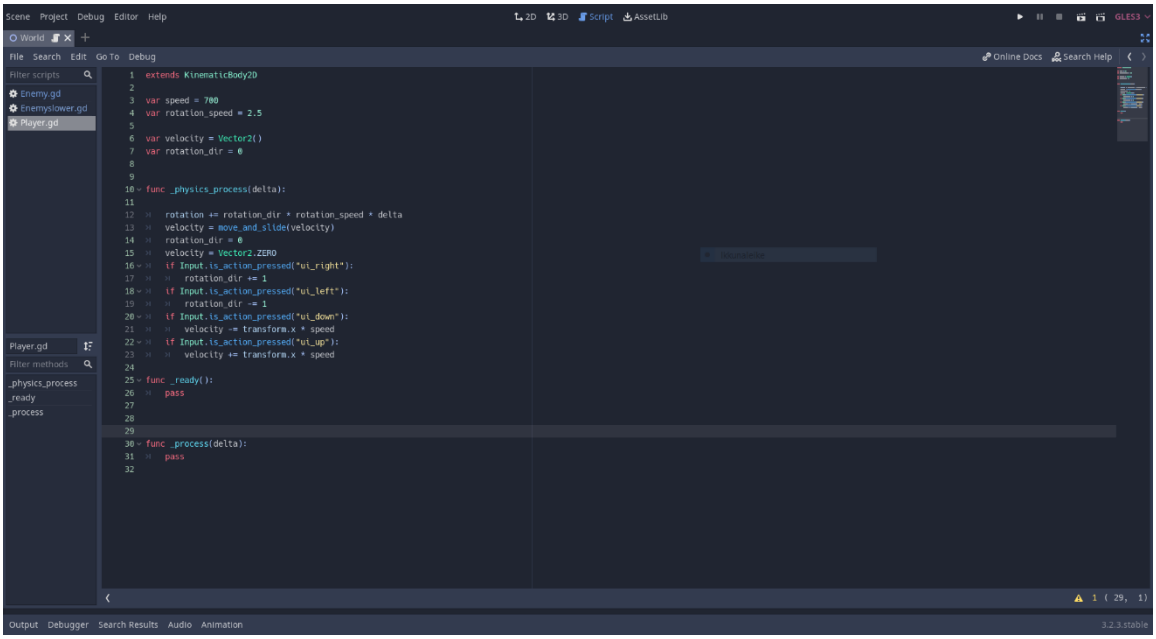
Kuva 6. 1: Avoin näkymä, 2: Muut avoimet näkymät, 3: Pääsolmun ominaisuudet, 4: Pelaaja eli Player_Kettu instanssi ja 5: Näkymän graafisessa editorissa instanssi Player_Kettu [4]

2.3 Ohjelmointi

Godot Engine tarjoaa mahdollisuuden käyttää useampaa ohjelmointikieltä. Virallisia Godotin kehittäjien tukemia kieliä ovat C# 8.0 ja C++. C++ on nopeakäyttöinen, sillä sitä ei tarvitse uudelleen kääntää, kun taas C#-käyttäjiä huomautetaan sen uutuudesta ja mahdollisista ongelmista kehityksen aikana [7]. Godotin tukemat ja kehittämät omat skriptikielet ovat GDScript ja VisualScript [3]. Godotin yhteisö mahdollistaa myös seuraavien ohjelmointikielten käytön: D, Kotlin, Nim, Python ja Rust [8]. Käytetyimmät ohjelmointikielet Godotissa ovat sen omat viralliset kielensä. Suosituin kieli onkin pelimoottorin oma GDScript.

Godot tarjoaa oman sisäänrakennetun ohjelmointieditorin. Editori on hyvä, mutta puutteellinen. Siitä puuttuu toistaiseksi mahdollisuus irrottautua erilliseksi ikkunaksi, mahdollisuus katsoa samanaikaisesti useampaa skriptiä ja muokata useampaa skriptiä yhtä aikaa. Godot on luvannut tuoda tulevaisuudessa edellä mainitut ominaisuudet omaan editoriinsa. Jos kuitenkin haluaa mahdollisuuden erillisille ominaisuuksille tällä hetkellä, joutuu turvautumaan ulkopuoliseen ohjelmointieditoriin. Yhteisö onkin tehnyt useamman lisäosan eri ohjelmointieditoreihin. Suosituimpana ulkopuolisena editorina mainittakoon Visual Studio Code, johon yhteisön jäsenet ovat tehneet lisäosia koodin täydentämiseen ja syntaksin korostamiseen.

Ohjelmointieditorinäkymässä nähdään koodin lisäksi myös avoimet ohjelmointitiedostot ja myös näytettävän ohjelmointitiedoston kaikki funktiot. Funktionäkymä nopeuttaa huomattavasti pitkän koodin lukemista, korjaamista sekä yleisesti yksinkertaistaa koodin tekemistä. (Kuva 7.)

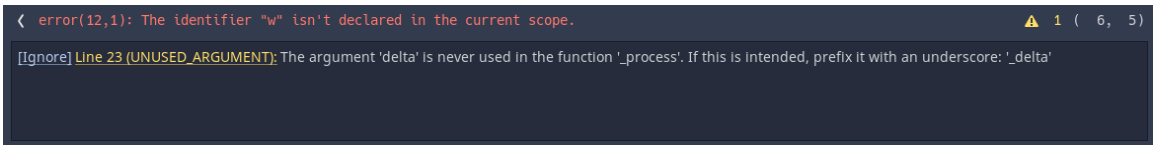


Kuva 7. Godot Enginen sisään rakennettu ohjelmointieditori.

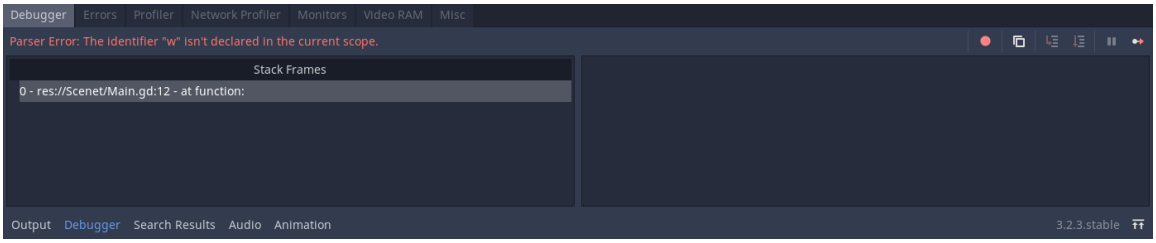
Godot Engine tarjoaa myös hyvän mahdollisuuden virheiden ehkäisyyn ja poistamiseen. Kehittäjä voi valita projektin asetuksista laajan valikoiman varoituksia ja virheitä, joista pelimoottori varoittaa jo ennen ohjelman ajamista.

Kuvasta 8 nähdään pelimoottorin varoittavan virheestä ennen, kun sovellusta on ajettu. Kuvan 8 esimerkkivirheessä ei ole alustettu muuttujaa w. Kuvassa on myös varoitus käyttämättömästä

parametristä. Kuvasta 9 nähdään jos kyseisen ohjelma ajetaan debuggerin läpi, debuggeri antaisi saman virheen, rivin, tiedostonimen sekä sijainnin.



Kuva 8. Pelimoottori varoittaa ohjelmointivirheestä ennen ohjelman ajamista.



Kuva 9. Debuggerin näkymä

2.3.1 GDScript

GDScript on suunniteltu olemaan helposti opittavissa ja helppokäyttöinen. Se on luotu, jottei kehittäjän tarvitsisi kirjoittaa määrällisesti paljoa koodia päästäkseen käsiksi Godot Enginen ominaisuuksiin sekä sen vahvuuksiin. GDScript onkin suunniteltu sekä kokeneille että vasta aloitteleville ohjelmistokehittäjille. GDScript on pelimoottorin oma natiivikieli ja myös pelimoottorin pääkieli. GDScript on syntaksiltaan suunniteltu samanlaiseksi kuin Python. Godotin kehittäjien mielestä yksi tärkeä syy miksi tulisi käyttää GDScriptiä on projekteissa monimutkaisuuden väheneminen [7]. (Kuva 10.)

```
1  extends Node2D
2  var x = 10
3  var y = 20
4  var z
5
6  var a = "hei "
7  var b = " moi "
8
9  func _ready():
10     print ("hello world")
11     z = x + y
12     print(z)
13     print(a+b)
14
15     if x < y:
16         print("pieni")
17     pass
```

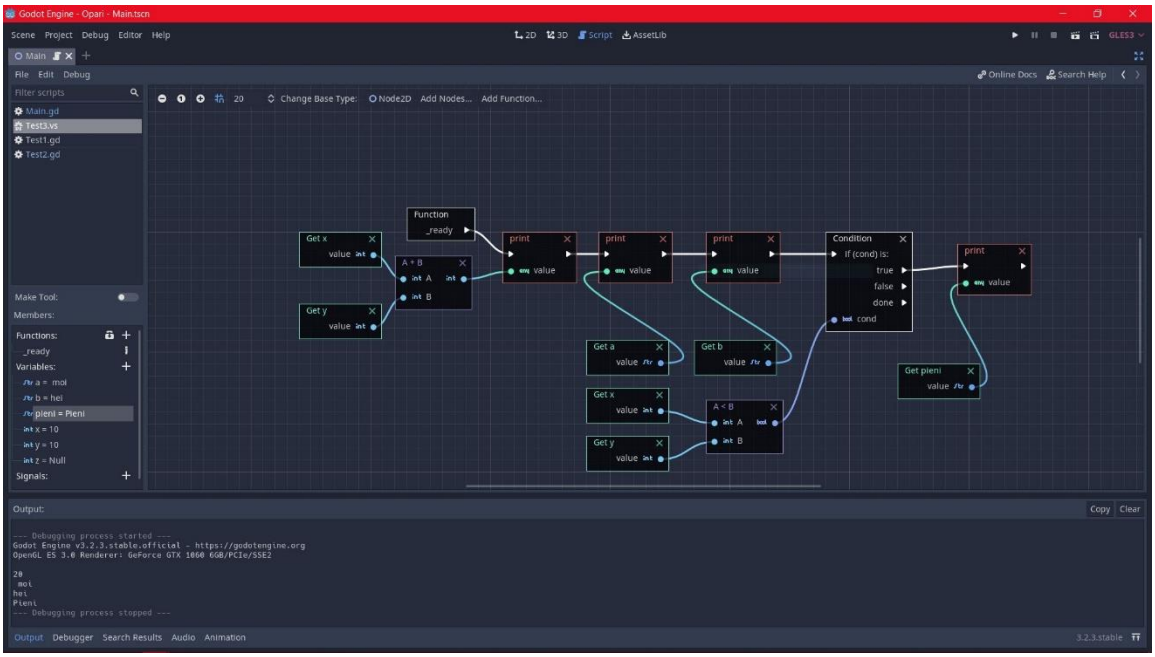
Kuva 10. GDScriptillä tehty esimerkkiohjelma.

2.3.2 VisualScript

VisualScript on Godot Enginen työkalu tuoda ohjelmoiminen visuaaliseksi. VisualScript on suunniteltu ohjelmoinnin aloittamisen helpommaksi. Ohjelmoinnin aloittaminen visuaalisella ohjelmointikielellä on helpompaa, koska se on helppolukuisemmaksi kuin tavallinen koodi. [9.]

Vaikka VisualScript on aluksi helppolukuista ja aloittelevalla kehittäjällä helppo oppia, on siinä kuitenkin isoja heikkouksia. Visuaalinen ohjelmointikieli vie huomattavan paljon tilaa verrattuna normaaliin tekstiin ja täten isompaa ohjelmistoa kehittäessä se muuttuu hyvin hankalalukuiseksi. Visuaalista koodia on myös hitaampi tuottaa.

Kuvasta 11 voidaan nähdä saman pienen ohjelman vievän tilan. Kun GDScript vei 17 riviä koodia väleineen, VisualScript vie lähes näytöllisen verran tilaa. Tästä voidaan päätellä, että pelin visuaalinen skripti kasvaa, tulee siitä hyvin hankalalukuista.



Kuva 11. Esimerkkiä Godot Enginen tarjoamasta VisualScriptistä

Visuaalisessa ohjelmoinnissa on hyviäkin puolia. Aiemmin mainitusti todella helppo kynnys aloittelevalle pelikehittäjälle. Tämä antaa mahdollisuuden esimerkiksi artisteille ja pelisuunnittelijoille tehdä helposti prototyyppejä. VisualScriptiä ja GDScriptiä voi käyttää sekaisin ohjelmissa. Pelimoottori myös antaa mahdollisuuden nähdä VisualScriptin GDScriptinä. Tämä on myös hyvä asia, sillä aiempaan esimerkkiin palaten artisti voi testata omaa prototyyppiään tiimin projektissa. Ei siis tarvitse olla omassa pienessä projektissaan.

Nykyään visuaaliset ohjelmointityökalut alkavat olemaan normi isommissa pelimoottoreissa. Pelimoottorit, joiden kanssa Godot Engine yrittää kilpailla, mahdollistavat myös visuaalisen skriptaamisen, Unreal Engine tarjoaa Blueprintin ja Unity tarjoaa Boltin. Blueprint ja Bolt ovat hyvin samankaltaisia Godot Enginen tarjoaman VisualScriptin kanssa.

Itse suosittelen aloittelevaa ohjelmisto- ja pelikehittäjää valitsemaan GDScriptin VisualScriptin sijaan. GDScript on lähes yhtä yksinkertainen, joten se on helppo oppia ja sillä on myös nopeampi kehittää isompaa projektia. GDScriptin oppiminen vie loppujen lopuksi vain hieman enemmän aikaa.

3 Varjostimet ja efektit

Tässä luvussa kerrotaan lyhyesti, mitä varjostimet ja efektit ovat. Työssä keskitytään peleissä käytettävien varjostimien sekä efektien toimintaan.

3.1 Mitä varjostimilla tarkoitetaan?

Lähes jokainen ihminen, joka on pelannut 2000-luvulla tuotettua videopeliä, on törmännyt tietoisesti tai tietämättään varjostimiin tai efekteihin. Varjostimet ja efektit ovat pieniä ohjelmia, jotka yleensä suoritetaan grafiikkaprosessorissa, mutta kaikki varjostinohjelmointikielet eivät ole grafiikkaprosessorille.

Varjostin ja efekti ovat suomessa käyttöön yleistyneitä sanoja englannin sanasta shader. Suomenkieliset varjostin ja efekti kuvaavatkin hyvin mitä ne ovat. Varjostin kuvaa pelimaailmoista usein löydettävien hahmojen varjoja. Kuvasta 12 nähdäänkin hyvä esimerkki varjostimesta. Efekti taas erilaisia efektejä, oli se kuvan vääristymää tai luoda pelaajan teleportaatiolle efekti. Varjostin- ja efektiohjelmoinnilla voidaan kuitenkin luoda muutakin kuin varjoja tai kuvan vääristymää. Ne voivat myös olla valaistusta, esineestä valon heijastumista tai voidaan luoda taivaalle pilviä tai sumua ilmaan. Toisin sanoen toteutetaan kuvan muokkaamista ennen sen piirtämistä näytölle. [10]

Kuva 12. Kuvasta voidaan nähdä useampaa esimerkkiä varjostimista ja efekteistä. Kuvasta voidaan nähdä valaistus, pelaajan varjo ja jään pinnasta voidaan nähdä valon heijastuminen [11].

3.2 Prosessoritoiminnot Fragment ja Vertex

Opinnäytetyön aikana tulen käyttämään seuraavia prosessoritoimintoja: fragment ja vertex. Prosessoritoimintoja hyödynnetään varjostimien ja efektien ohjelmoimisessa, ja kerron luvussa 4.1, kuinka niitä luodaan.

Fragment-prosessoritoiminto antaa Godot Enginessä mahdollisuuden muokata materiaalin jokaisen pikselin arvoja. Tällä voidaan esimerkiksi vaihtaa satunnaisesti jokaisen pikselin väriä. Prosessoritoiminto ajaa jokaisen näkyvän pikselin. [12.]

Vertex-prosessoritoiminto kutsuu vertex-käsittelyfunktion jokaiselle verteksille kaksi- ja kolmeulotteisissa varjostintyypeissä lukuun ottamatta particles-varjostintyyppiä. Jokaisella verteksillä on ominaisuuksia kuten sijainti ja väri. Vertex-käsittelyfunktio muokkaa näitä arvoja ja lähettää ne fragment-käsittelyfunktiolle. [12.] Näitä arvoja muokkaamalla voidaan esimerkiksi venyttää materiaalia.

4 Varjostinohjelmointi Godot Engine -pelimoottorissa

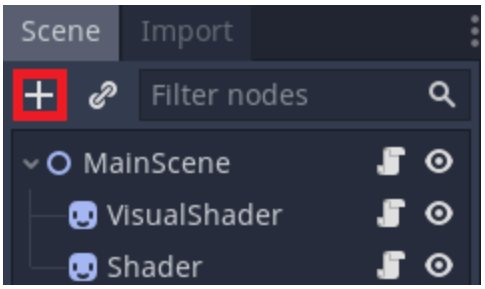
Varjostinohjelmointi Godot Engine -pelimoottorissa ei juuri eroa muista varjostinohjelmointikielistä. Godot tarjoaa oman hyvin samankaltaisen varjostinohjelmointikielen kuin GLSL ES 3.0. Godotin Engine tarjoama varjostinkieli tarjoaa enemmän ominaisuuksia, mutta verrattuna GLSL:ään Godotin varjostinkieli ei ole läheskään niin joustava perinteiselle käyttäjälle. Varjostinkielen joustamattomuus johtuu pitkälti siitä, että pelimoottori pyrkii toteuttamaan monimutkaiset asiat varjostinkielen ominaisuuksilla. Pelimoottori pakkaa käyttäjän luoman varjostinkoodin omaan koodiinsa. Tällä tavalla pelimoottoria käyttävän varsinkin aloittelevan pelinkehittäjän ei tarvitse miettiä turhia matalan tason teknisiä asioita. [13.]

VisualShaders on samankaltainen kuin VisualScript, eli visuaalinen varjostinohjelmointikieli. VisualShadersissä on kuitenkin enemmän heikkouksia kuin tavallisessa Godot Enginen varjostinkielessä. VisualShadersissä ei pysty tekemään kaikkea samaa kuin skriptipohjaisissa varjostimissa ja efekteissä sen puutteellisten ominaisuuksien vuoksi. Kuten GDScriptiä ja VisualScriptiä, voi myös pelimoottorissa yhdistää sekä tavallista varjostinkieltä että visuaalista varjostinkieltä. Voi siis olla välttämätöntä joutua käyttämään yhdistämään sekä VisualShaderseja että varjostinohjelmointikieltä saavuttaakseen tietynlaisia varjostimia tai efektejä [14].

4.1 Kuinka luoda varjostin tai efekti

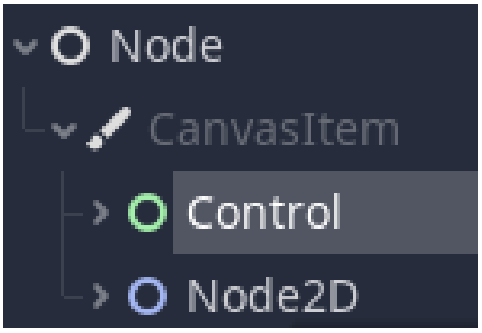
Godot tarjoaa mutkattoman tavan luoda varjostimia sekä efektejä. Tässä luvussa käytän esimerkkeinä Canvas_Item-varjostintyyppiä eli kaksiulotteisille objekteille suunnattua varjostintyyppiä.

Aluksi täytyy luoda solmu, ja kuten aiemmin on mainittu, esimerkit toteutetaan kaksiulotteisille suunnatulla Canvas_item varjostintyypillä. Solmun siis täytyy olla kaksiulotteinen ja CanvasItem ominaisuudet perivä solmu, jotta voidaan käyttää Canvas_Item-varjostintyyppiä. (Kuva 13.)



Kuva 13. Punaisella merkitään painiketta, josta päästään valitsemaan uusi solmu.

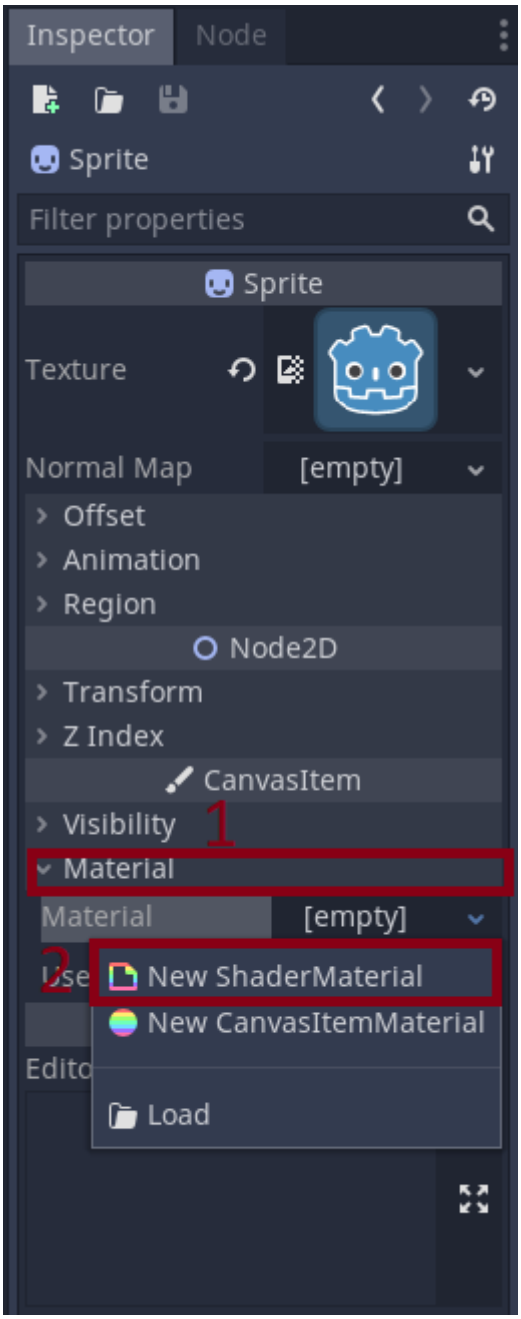
Kuvasta 14 nähdään, minkä solmujen alta voidaan etsiä toimivat solmut CanvasItem-solmutyypille. Node2D:n alla olevat solmut ovat kaksiulotteisia peliobjekteja, joista jokainen perii sijainnin, rotaation, skaalauksen ja z-indeksin. Control-solmun alaisuudessa olevat solmut ovat kaikki käyttöliittymään kohdistuvia solmuja. Käytän Node2D:n alaisuudesta löytyvää Sprite-solmua.



Kuva 14. Noden oltava tässä tapauksessa CanvasItemin alainen.

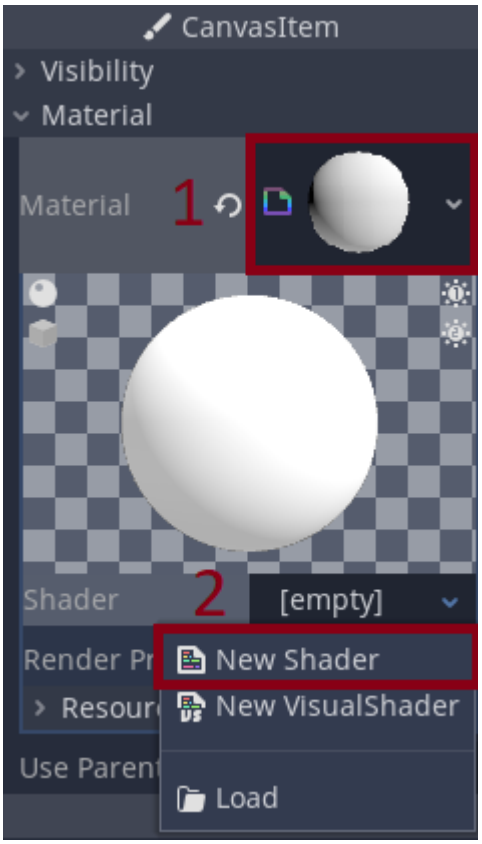
Seuraavaksi luodaan uusi materiaali. Materiaali voidaan luoda editorista solmun ominaisuuksista kohdasta Material. Aukaistaan näkymään Materiaalin ominaisuudet ja painetaan materiaalin vierestä painiketta, jossa kuuluisi lukea [empty].

Kuvassa 15 kohdan kaksi mukaisesti seuraavaksi voidaan luoda jo uusi varjostinmateriaali. Kun uusi materiaali on luotu, tulee näkyville pieni valkoinen pallo. Painetaan pallosta, ja se avaa näkymään kohdan, josta päästään luomaan uusi varjostin.



Kuva 15. Materiaalin luominen.

Kuvan 16 mukaisesti seuraavaksi avataan Shader ja valitaan varjostinohjelmointikieleksi tavanomainen Godot Enginen ohjelmointikieli tai visuaalinen VisualShader-kieli.



Kuva 16 Varjostimen luominen.

4.2 Varjostinohjelmointi

Kun materiaaliin on luotu tyhjä varjostinohjelma, voidaan aloittaa varjostinohjelmointi. Varjostinohjelmointi alkaa aina valitsemalla yhden kolmesta varjostintyypistä. Valittavat varjostintyypit ovat seuraavat: Spatial, Canvas_item ja Particle. [14.] (Kuva 17.)

```
1  shader_type canvas_item;
```

Kuva 17. Varjostinohjelmoinnin ensimmäinen vaihe on valita varjostintyyppi. Tässä tapauksessa kaksiulotteinen Canvas_item.

Spatial-varjostintyyppi on kaikista monimutkaisin, se on tarkoitettu kolmeulotteisille varjostimille ja efekteille. Spatial tyyppi on myös kaikista eniten muunneltavissa oleva varjostintyyppi. Se

omistaa useamman renderöintitilan ja renderöintivalinnan. Nämä voivat olla esimerkiksi materiaalin heijastuminen ja objektin reunan valaisu. [15.]

Canvas_item-tyyppiä käytetään kaikkien kaksiulotteisten varjostimien luomiseen. Canvas_item ei sisällä läheskään niin paljoa ominaisuuksia ja valmiita muuttujia kuten Spatial varjostintyyppi. Canvas_item- ja Spatial-varjostintyypit kuitenkin jakavat saman perusrakenteen seuraavien prosessoritoimintojen avulla: vertex, fragment ja light. [16.]

Particle eli partikkelivarjostintyyppi on siitä erilainen muihin varjostintyyppeihin, ettei sitä ole luotu piirtämään objektia, vaan laskemaan objektin ominaisuuksien arvoja, joita sitten Canvas_item tai Spatial varjostintyyppi käyttää. Partikkelivarjostin myös ajetaan jo ennen objektin piirtämistä. [17.]

Seuraavaksi varjostinohjelmoinnissa käyttäjä voi valita renderöintitilan. Jokaisella varjostintyypillä on omat renderöintitilansa. Renderöintitiloja voi esimerkiksi olla aiemmin mainitut materiaalin heijastuminen ja objektin reunuksen valaisu tai voidaan antaa pelimoottorin sisäisen valon mennä objektin läpi. Renderöintitilaa ei kuitenkaan ole pakko valita itse. Jokaiselle varjostintyypille on asetettu vakio renderöintitila. (Kuva 18.)

```
2  render_mode blend_mix;
```

Kuva 18. Renderöintitilan valitseminen. Kuvan tapauksessa Blend_mix on vakio arvo. Blend_mix antaa mahdollisuuden läpinäkyvyydelle. [16.] Eli Alfa-arvoa voidaan muuttaa.

Tässä vaiheessa on hyvä tietää muuttujien alustamisesta. Tärkeitä asioita on tietää pelimoottorin varjostinohjelmointikielestä se, että muuttujan täytyy olla samaa tyyppiä.

Kuvan 19 esimerkistä voi nähdä, että float eli liukuluvussa täytyy olla seuraavat asiat: kokonaisluku, desimaalipiste ja desimaaliluku. Kaksi ensimmäistä riviä on sallittuja tapoja luoda liukuluku. Editori antaakin viimeisen rivin itsestään virheeksi, sillä siinä yritetään alustaa liukuluku pelkällä kokonaisluvulla. Myös integereissä eli kokonaisluvuissa pätee sama tapa. Kokonaislukua ei saa alustaa liukuluvulla [18]. (Kuva 19.)

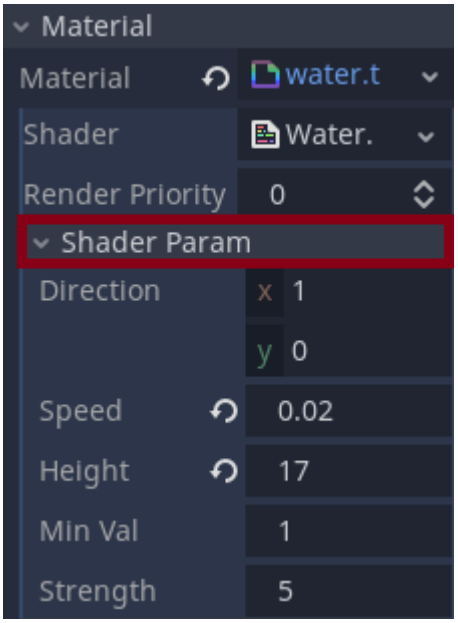
```
    float x = 1.0;
    float y = float(1);

    float z = 1;
```

Kuva 19. Esimerkki liukuluvun alustamisesta.

Muuttujan voi myös alustaa varjostimeen globaaliksi Uniformin avulla. Uniform-muuttujat alustetaan käsittelyfunktion ulkopuolella. (Kuva 20.) Varjostimeen voi myös syöttää arvoja varjostimen ulkopuolelta Uniformin avulla. (Kuva 21.) Uniform mahdollistaa editorista sekä tavanomaisesta koodista arvojen syöttämisen varjostimeen. [18.] (Kuva 22.)

```
3  uniform vec2 Direction = vec2(1.0,0.0);
4  uniform float speed = 0.01;
5  uniform float height = 10.0;
6  uniform float minVal = 1.0;
7  uniform float strength  = 5.0;
```

Kuva 20. Muuttujien alustaminen käsittelyfunktion ulkopuolella. Muuttujat myös globaaleja ja niiden arvoja voidaan muokata editorista.



Kuva 21. Solmun Material kohdasta löytyy Uniform-arvot Shader Param alaisuudesta.

```
1  extends Sprite
2  var Direction = Vector2(1.0,0.0)
3  func _physics_process(delta):
4      material.set_shader_param("Direction",Direction)
5      pass
```

Kuva 22. Arvojen muuttaminen GDScriptiä käyttäen.

Tätä voi myös hyödyntää esimerkiksi efektin käsittelyfunktiossa fragment, jos halutaan luoda sellainen efekti, jossa halutaan nähdä pelaaja veden takana tummempana hahmona, eikä kokonaan piiloteta hahmoa veteen. Myös vertex-käsittelyfunktiossa voi antaa veteen arvon, kun pelaaja on törmäämässä veteen. Se laskisi verteksien uudet sijainnit, pelaajan tuottamat arvot törmäyskohdasta ja tuottaisi pienet aallokot.

Varying arvo on kolmas asia, mikä olisi hyvä tietää ennen itse varjostinohjelmointia. Varying arvolla voidaan lähettää dataa käsittelyfunktiosta toiseen [15]. (Kuva 23.)

```
4  varying float variable1;
5  void vertex()  {
6      variable1 = 1.0;
7  }
8  void fragment(){
9      float variable2 = variable1;
10 }
```

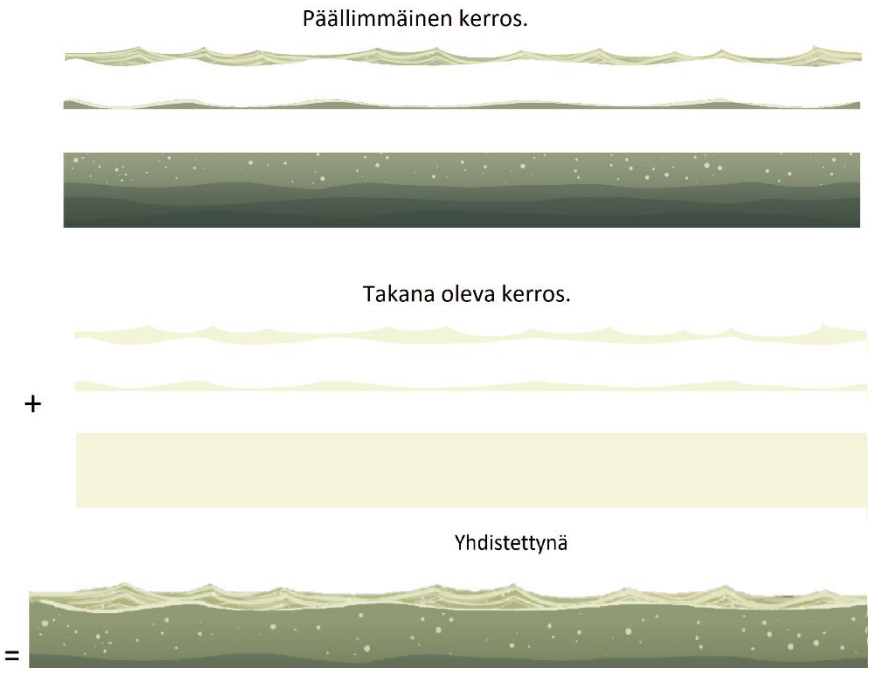
Kuva 23. muuttujan arvojen jakaminen vertexistä fragmenttiin.

Seuraavaksi voidaan aloittaa valitsemalla, mitä prosessoritoimintoja halutaan käyttää. Nyt kun olen käyttänyt esimerkissä varjostintyyppiä Canvas_item, joudun valitsemaan vertex, fragment tai light-prosessoritoiminnon. Kaikkia prosessoritoimintoja voidaan myös käyttää rinnakkain. Jos haluaa käyttää Spatial-varjostintyyppiä, voi käyttää samoja prosessoritoimintoja, mutta Particle-varjostintyyppi mahdollistaa vain vertex-prosessoritoiminnon käyttämisen.

Valitsin esimerkin luomiseen efektin, jossa yhdistyy vertex-prosessoritoiminto ja fragment-prosessoritoiminto. Esimerkissä luodaan useammasta kuvasta muodostavalle aallokolle liike sekä kuvien liikkuminen oikealta vasemmalle huomaamattomasti. Esimerkki on hyvin yksinkertainen, ja

tällä yritetään näyttää, miten pienellä varjostimella tai efektillä voidaan elävöittää pelimaailmaa houkuttelevammaksi.

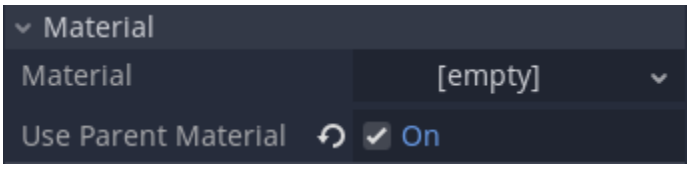
Vesi on suunniteltu useammasta kerroksesta, jotta efekti olisi yksinkertaisempi ja pelissä pelaajan tippuessa kerroksien väliin ei tarvitse ohjelmoida erillistä läpinäkyvyyttä. Huomioitavaa on myös, jos halutaan efektin kohdistuvan useampaan solmuun, joista kaikki on samassa näkymäpuussa, voidaan antaa materiaali ainoastaan vanhempisolmulle. Täten voidaan jokaiselle halutulle lapsisolmulle antaa vanhempisolmun materiaali. (Kuva 24.)



Kuva 24. Kuvasarjat, joista koostuu kokonainen kuva vedestä [4].

Ainoastaan efektin materiaali esimerkkityössä liitetään kuvan 25 vanhempisolmuun eli Vesi. Jokainen lapsisolmu perii tällöin materiaalin Vesi-solmulta. Tämä myös pitää valita itse kuvan 26 tapaisesti.

Kuva 25. Efektin solmupuu.



Kuva 26. Solmun ominaisuuksista materiaalin alaisuuksista valitaan Use parent material halutessaan lapsisolmulle saman materiaalin.

Aloitetaan vertex-prosessoritoiminnasta ja luodaan sille käsittelyfunktio. Tässä käsittelyfunktiossa pyritään venyttämään ja kutistamaan kuvaa niin, ettei kuva veny tai kutistu sivuista ulos eikä myöskään pohjasta ulos. Ainoa sallittu venymä ulos alkuperäisen kuvan rajoista on ylöspäin. Tämänkaltainen efekti saadaan, kun muunnetaan verteksien arvoja y-akselilla eli pystysuoraan sinilausekkeella ja lisäämällä siihen varjostintyypin sisäänrakennetun ajan. (Kuva 27.)

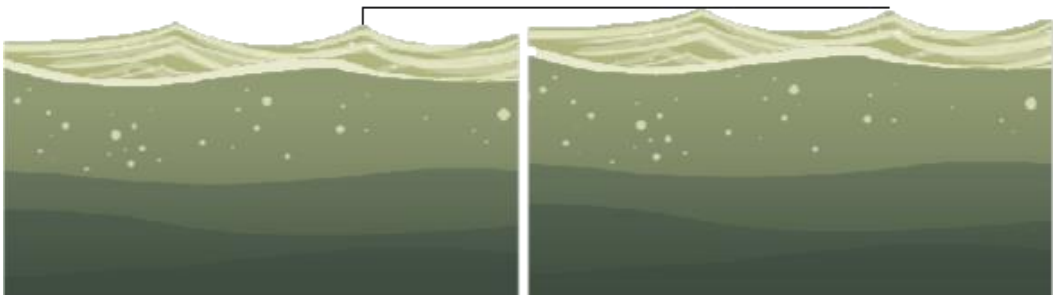
Kuva 27. Kuva venyy pystysuorassa ja toinen kulmakuvasta on aina korkeammalla. Korkeusero ei ole niin huomattavissa lopullisessa efektissä.

Kuvassa 28 voidaan nähdä, miten pienellä määrällä koodia pystytään luomaan pieni efekti, jossa kuva liikkuu aaltomaisesti. Rivillä 12 luodaan sinilausekkeella arvo, kuinka kuvasarja venyy pystysuunnassa. Seuraavalla rivillä tarkistetaan, ettei kuva veny pohjan läpi. Jos tarkistus menee läpi voi nähdä kuvan venyvän suunnitellulla tavalla.

```
10  void vertex()
11  {
12      float vertexY = minVal + sin(TIME + VERTEX.x) * strength;
13      if (VERTEX.y < 1.0)
14      {
15          VERTEX.y += vertexY;
16      }
17  }
```

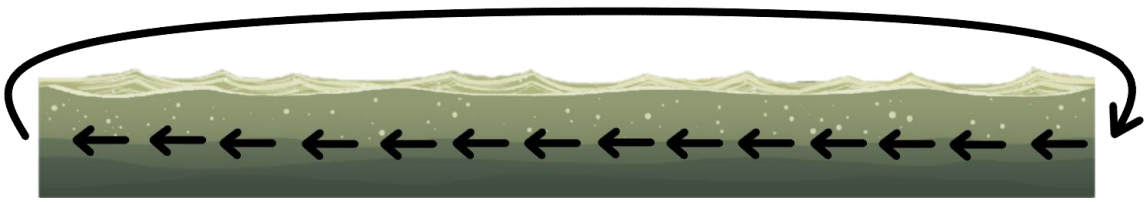
Kuva 28. Yksinkertaisen vesiefektin verteksien venytys.

Kuvasta 29 nähdään, miltä korkeusvaihtelu näyttää lopullisessa efektin versiossa. Kuva näyttää, ettei siinä ole suurta vaihtelua, mutta peliä pelatessa sen huomaa.



Kuva 29. Kuvasta voi nähdä veden korkeusvaihtelun. Pelin sisällä helpommin havaittavissa [4].

Seuraavaksi työssä toteutetaan efektille silmukka, joka huomaamattomasti liikuttaa kuvaa oikealta vasemmalle. Aina reunan ylitettyä pikselit vaihtuvat edelliseen reunaan. (Kuva 30.)



Kuva 30. Fragment-käsittelyfunktion kuvaus. Vasemmalta ylittyvät pikselit renderöityvät oikeaan reunaan.

Kuvasta 31 nähdään, kuinka voidaan luoda edellä mainittu efekti

```
19  void fragment()
20  {
21      COLOR = texture(TEXTURE,UV + (Direction * (TIME * speed)));
22  }
```

Kuva 31. Kuva pyörii silmukassa samankaltaisesti kuin kaksiulotteisissa peleissä taustalla toistuva maisema.

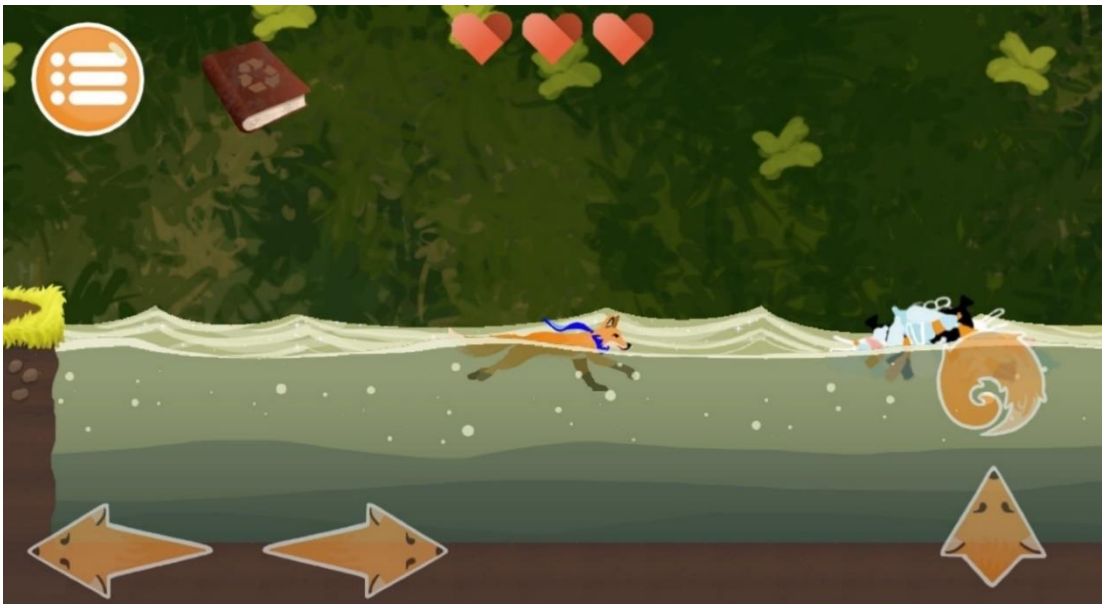
Jos COLOR-arvoa ei muokata, olisi värin vakioarvona solmun tekstuuri. Tässä tapauksessa kuvan 24 kuvat. Haluamme kuitenkin muokata tekstuurin sijaintia kuvan 30 tapaisesti. Tämä voidaan toteuttaa antamalla COLOR-väriarvoiksi TEXTURE, joka on aiemmin mainitut vakioarvoiset kuvat. Seuraavaksi halutaan muokata tekstuurin koordinaatteja. Tässä UV keskustelee vertex-prosessoritoiminnon kanssa ja lisää siihen aina yhtälön: Direction*(Time*speed). Annetaan siis yhtälöön ensimmäiseksi kertojaksi suunnan, joka kertoo, halutaanko kuvasarjan liikkuvan vasemmalle vai oikealle. Suunnalla kerrotaan aikaa kerrottuna nopeudella. Aika määrittää kuvan jatkuvan liikkumisen ajan vaihtuessa. Nopeus ja suunta on alustettu uniform-arvoiksi, jotta voidaan helposti säätää efektin liikettä.

Kuvasta 32 nähdään, kuinka pienellä määrällä ohjelmointia voidaan toteuttaa peliä elävöittävä efekti

```
shader_type canvas_item;
render_mode blend_mix;

uniform vec2 Direction = vec2(1.0,0.0);
uniform float speed = 0.01;
uniform float height = 10.0;
uniform float minVal = 1.0;
uniform float strength  = 5.0;
varying float variable;

void vertex()
{
    float vertexY = minVal + sin(TIME + VERTEX.x) * strength;
    if (VERTEX.y < 1.0)
    {
        VERTEX.y += vertexY;
    }
}
void fragment()
{
    COLOR = texture(TEXTURE,UV + (Direction * (TIME * speed)));
}
```

Kuva 32. Koko varjostimen ohjelmointiosuus. Efekti on hyvin yksinkertainen, mutta samaan aikaan sillä on huomattava merkitys pelimaailmaan.

Efektissä myös varjostinohjelmoinnin lisäksi on kuvat suunniteltu siten, että takana oleva kerros on myös pelaajan takana. Edessä oleva kerros on pelaajan päällä ja on itsessään hiukan läpinäkyvä. Tällä saadaan toteutettua efektiin ilman ylimääräistä koodia efekti, joka tummentaa vesikerroksien välissä olevia objekteja ja antaa hiukan syvyyttä veteen. (Kuva 33.)

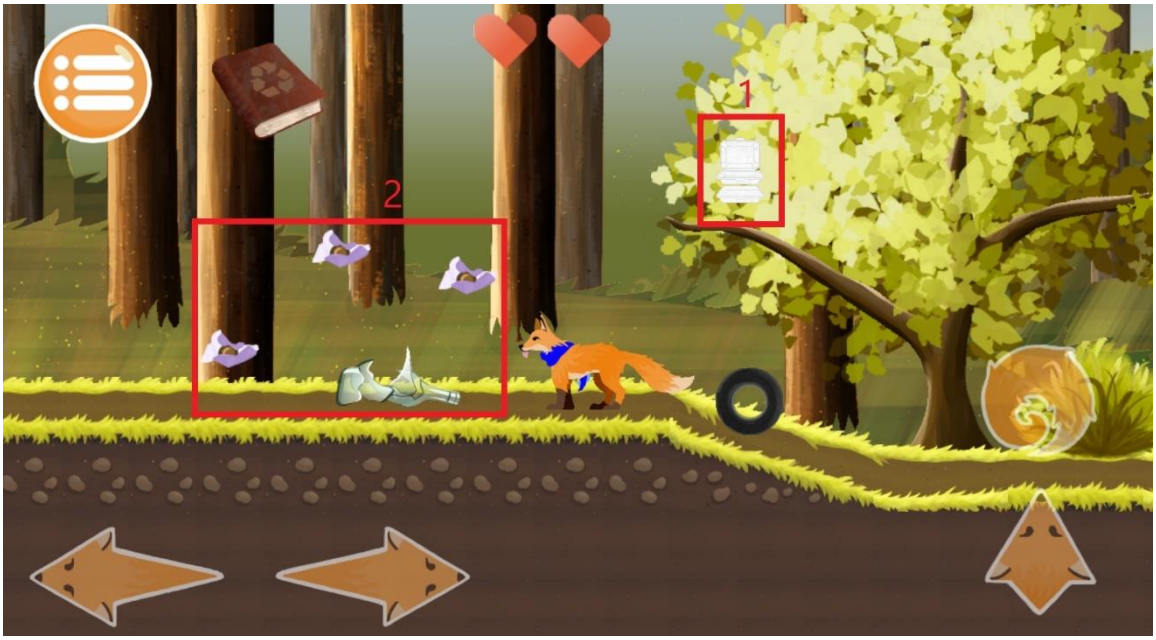


Kuva 33. Efektin näkymä pelissä Litter Run [4].

5 Projekti

Projektityönä jatkoin hyötypeliprojektia, jota toteutettiin työharjoittelun aikana pienessä kolmen hengen ryhmässä. Ryhmään kuului lisäkseni peliohjelmoija ja graafikko. Minun vastuualueenani projektissa oli vastata teknillisestä toteutuksesta ja ohjelmoinnin rakenteesta sekä efektien toteutuksista. Toinen peliohjelmoija vastasi pääsääntöisesti pelin suunnittelusta ohjelmoinnin lisäksi.

Pelin nimi on Litter Run, ja se on suunniteltu lapsille ja nuorille. Peli on kaksiulotteinen, ja se edustaa seikkailu- ja tasoloikkagenrejä. Pelissä pelataan ketulla, joka on ymmärtänyt ihmisten jättämien roskien haitan eläimille ja myös ympäristölle. Pelin tarkoituksena on opettaa roskien haitasta luonnossa. Pelissä pisteitä saa keräämällä roskia luonnosta. Roskat ovat yleisiä luontoon heitettäviä tavaroita, esineitä ja asioita. Samalla pelaajan täytyy huomata, että jotkin roskat voivat vahingoittaa pelaajaa, esimerkiksi lasinsirut rikkoutuneesta lasipullosta. (Kuva 34.)



Kuva 34. Numerolla 1 on merkitty luontoon useasti ihmisten heittämä styroksinen nakkikioskin ruokailuastia. Numerolla 2 merkityssa laatikossa on karkkipapereita ja rikkinäinen lasipullo. Karkkipapereita keräämällä saa pisteitä ja lasinsiruihin astumalla menettää elämäpisteitä [4].

Kentät päättyvät aina, kun pelaaja löytää roskakorin. Roskakorin löydettyään peli toteuttaa animaation, missä pelaaja nojaa roskakoriin tiputtaakseen roskat. (Kuva 35.)

Kuva 35. Kettu tiputtaa kentän päätteeksi löytämänsä roskat roskakoriin [4].

Pelin oppiminen tulee löytämällä erikoisroskia pelimaailmasta. Näitä voivat esimerkiksi olla auton akku ja styroksi. Löydettyään roskan pelissä voi avata kirjan, josta pelaaja voi lukea roskan haitoista ympäristölle. (Kuva 36.) Peli ei kuitenkaan pakota pelaajaa lukemaan roskien haittoja. Ainoastaan ensimmäinen kerta on pakollinen, jotta pelaaja tietäisi pelin edetessä voivansa lukea löytämiensä roskien haitoista ympäristölle.

Kuva 36. Kun pelaaja löytää erikoisroskan, voi hän lukea kirjasta roskien haitoista ympäristössä. Kirja alkaa vaihtamaan silmukassa läpinäkyvyyttään edestakaisin pelaajan löydettyä roskan, josta voi lukea tietoa [4].

Pelissä tulee myös vastaan vihollisia, joista yleisimpiä vihollisia on kuvitteelliset liikkuvat roskapussit. Roskapussit ovat erivärisiä ja jokainen toimii eri tavalla. Maailmasta voi löytää pomppivan roskapussin, ilmassa lentävän roskapussin, pienen roskia ampuvan roskapussin sekä tavallisen kävelevän roskapussin. Roskapussit voi tuhota hyppäämällä päälle tai tekemällä häntäiskun pussiin. (Kuva 37.)

Kuva 37. Erilaisia vihollisroskapusseja, jotka vahingoittavat pelaajaa [4].

Pelimaailmasta löytyy myös toisenlainen vihollinen, ihminen. Ihminen on suunniteltu heittämään ympäristöön lasipulloja. Ihmisen keho ei tee pelaajalle vahinkoa, mutta sen heittämät lasipullot pelaajan osuessa ovat vaarallisia ja pelaaja menettää vahinkopisteitä. (Kuva 38.)

Kuva 38. Ihmishahmo [4].

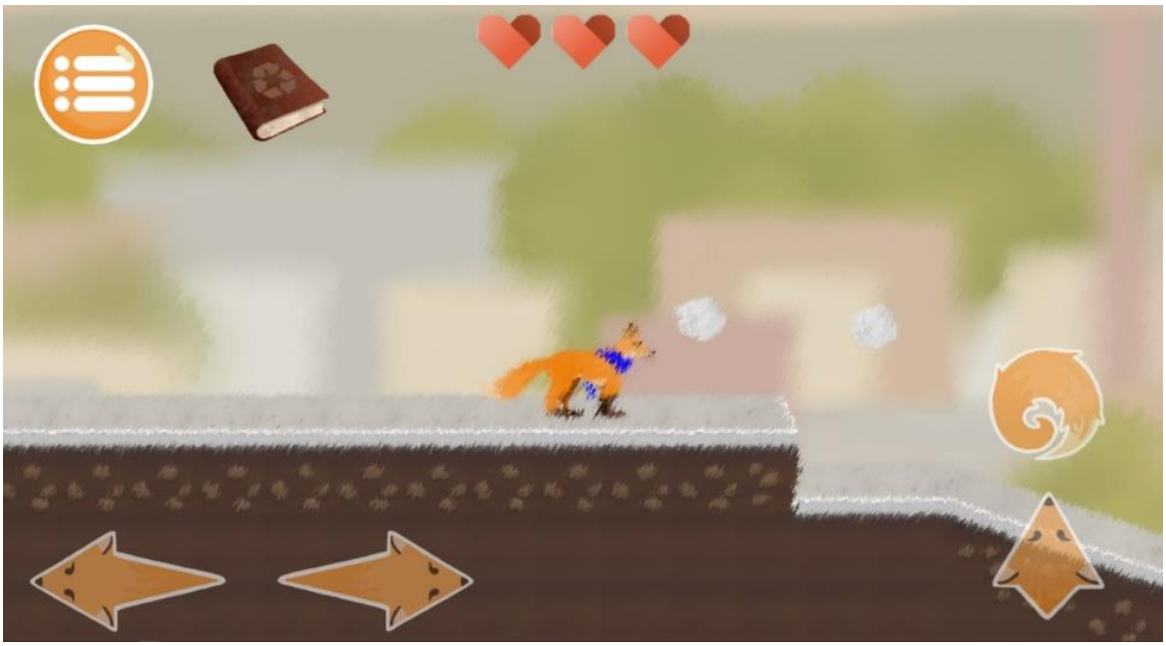
Peli on suunniteltu Android-laitteille, mutta pelinkehityksen aikana sitä on myös testattu Windows ja Linux käyttöjärjestelmillä. Peli on pyritty pitämään mahdollisimman pienenä tiedostokooltaan. Tästä syystä siitä karsittu hiukan animaatioiden kuvaruutuja ja peli käyttää OpenGL 2.0 versiota jotta peliä voisi mahdollisimman moni lapsi tai nuori pelata.

5.1 Maailman elävöittäminen efekteillä

Sain vastuulleni suunnitella ja toteuttaa peliprojektiin useamman efektin, jolla saataisiin luotua pelimaailmaan elävöitymistä. Efektit täytyi suunnitella kevyiksi, jotta aiemmin mainitusti peli pyörisi mahdollisimman monella Android-laitteella, ja mahdollisimman moni nuori ja lapsi pääsisi pelaamaan ja oppimaan roskaamisen vaikutuksesta ympäristössä. Seuraavaksi esitän kaksi muuta suunnittelemaani ja toteuttamaani efektiä Litter Run -peliin. Molempien efektien on tarkoitus tuoda myös peliin eri tavalla tunnelmaa.

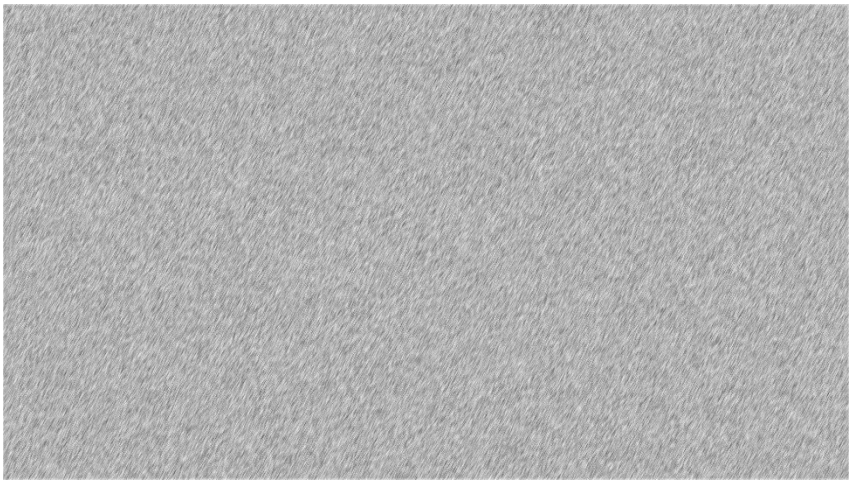
Lähes jokaisessa pelissä, missä seikkaillaan luonnossa, halutaan sen tuntuvan elävältä. Tästä syystä suunnittelin ja toteutin peliin luonnollisia ilmiöitä efekteillä. Aiemmin kerroin varjostinohjelmointi-luvussa veden liikkeestä, joka oli hyvin yksinkertainen mutta merkityksellinen pelin suunnittelun kannalta. Kaksiulotteisten pelien luontoon onkin mahdollista tuottaa hyvin yksinkertaisesti elävöitymistä vain opettelemalla varjostinohjelmoinnin perusteet.

Ensimmäiseksi esittelen efektin, joka toteuttaa ympäristöön kuuluvan luonnollisen ilmiön, vesisateen. (Kuva 39.)



Kuva 39. Vesisade pelissä [4].

Pelin vesisade on toteutettu käyttämällä pelkästään fragment-käsittelyfunktiota. Käsittelyfunktiolle annetaan aluksi tekstuuri. Tekstuurina toimii GIMP-kuvanmuokkausohjelmalla toteutettu melukartta. (Kuva 40.)



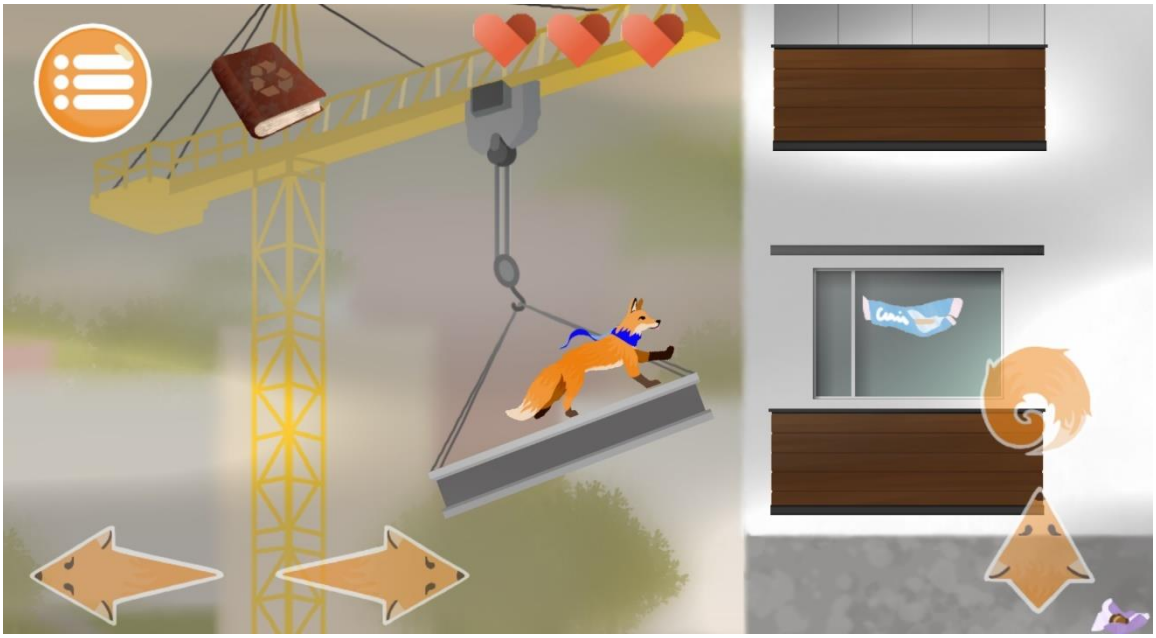
Kuva 40. Efektissä käytetty melukartta [4].

Melukarttaa luetaan fragment-käsittelyfunktiossa. Käsittelyfunktiossa säädetään kaikkia mustia arvoja läpinäkyviksi. Tämän jälkeen melukartta laitetaan vesiefektin tapaisesti silmukalle.

Melukartta on kuitenkin silmukassa siten, että se liikkuu sekä vaaka- että pystysuorassa. Pystysuorassa liikkeessä ylhäältä alas ja vaakasuorassa liikkeessä oikealta vasemmalle. Vaakasuoralla akselilla liikkuminen ei ole välttämätöntä olla oikealta vasemmalle, mutta annoin melukarttaa luodessani GIMP-kuvanmuokkausohjelmalle ehdon, että mustien ”sadepisaroiden” täytyy olla vasemmalle alaviistoon. Tällä saadaan pelaajan tuntemaan sade todellisemmalta, kun se sataa päin pelaajan menosuuntaa.

Efekti tuntuu tällä hetkellä hiukan liian kaatosademaiselta. Tämä kuitenkin saadaan korjattua helposti tekemällä uuden melukartan, missä toteutettaisiin hiukan paksummat pisarat, samalla myös enemmän väliä pikseleiden välissä.

Toinen efekti on suunniteltu enemmän pelin teemaa ajatellen eli ympäristön saastuminen. Ilmiöksi suunnittelin ja toteutin efektinä kaupunkialueelle ilmaston saasteen. (Kuva 41.)

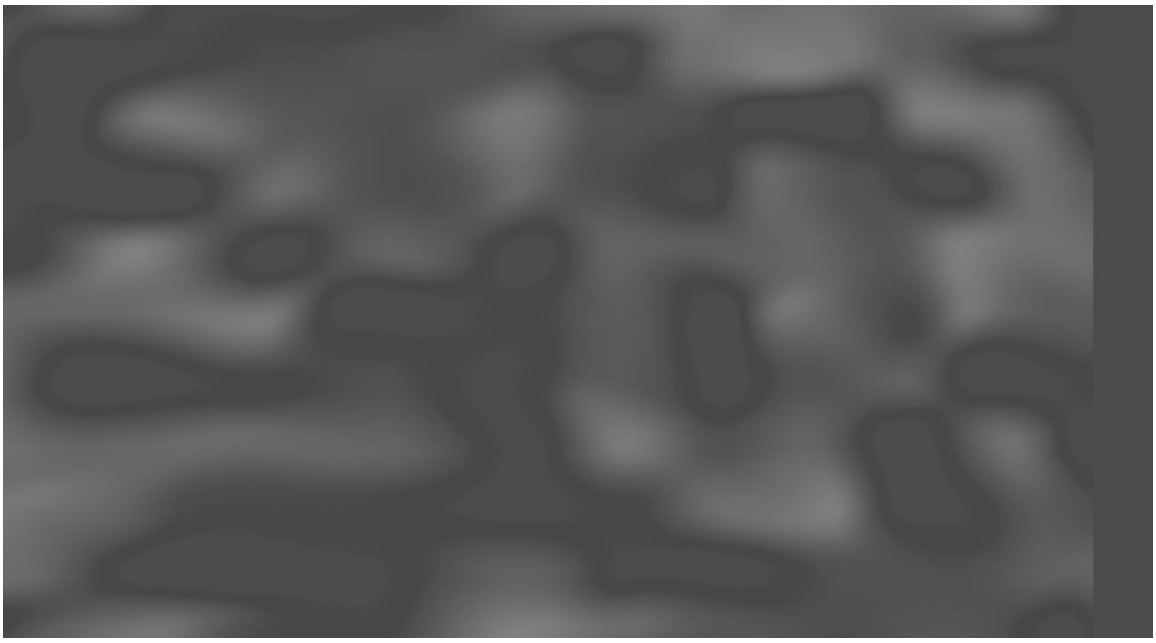


Kuva 41. Kuvassa tummemman sävyiset alueet. Saastetta venytetty, jotta lukija voisi havainnollistaa paremmin sen [4].

Saaste on toteutettu hyvin samankaltaisesti kuin vesisadekin. Satunnaisen melukartan sijaan toteutin aliohjelman, joka luo satunnaista melua. Toinen aliohjelma luo reunoja lähestyessä saasteen tummumisen.

Saasteen toteutuksessa ongelmaksi koitui sen äkillinen loppuminen reunoissa. Kuvasta 42 lukija voi nähdä katsomalla oikeaa reunaa. Tämän voisi korjata luomalla saasteelle pyöreämmän muodon minkä sisällä kaikki tapahtuisi. Toinen lähestymistapa parantamiseen voisi olla luomalla samankaltainen melukartta kuin sadepisaroille, tietenkin käyttäen isompia pyöreitä muotoja. Tällöin saasteesta tulisi säännöllisempää.

Saaste-efektistä voi luoda hyvin pienillä muutoksilla erilaisia muita efektejä. Sitä voi käyttää pohjana myös esimerkiksi savuun ja tuleen.



Kuva 42. Saaste kuvattuna tummaa taustaa vasten auttaakseen lukijaa havainnollistamaan saasteen paremmin [4].

5.2 Projektin yhteenveto

Peliprojekti on opinnäytetyötä kirjoittaessani keskeneräinen, ja siihen on vielä useampi efekti suunniteltavana. Kuitenkin opinnäytetyössä esittelemäni efektit onnistuivat hyvin luomaan pelimaailman elävöitymistä. Molemmilla efekteillä oli jokin tarkoitus. Ensimmäinen, eli vesisade, on suunniteltu luomaan luontoon luonnollista elävöitymistä, ettei maailma olisi tylsä ja seisoisi vain

paikallaan. Toinen efekteistä, eli saaste, on suunniteltu luomaan saasteista tunnelmaa kaupungissa, niin sanotusti ihmisen tuottamaa saastetta elinympäristöön. Molemmat efekteistä onnistuivat luomaan halutunlaisen tunnelmaa pelimaailmaan.

6 Yhteenveto ja pohdinta

Varjostimien ja efektien merkitys on kiistämättä nykypäivänä tärkeä videopeleissä. Niitä voidaan nähdä suurimmassa osassa nykypäiväisistä peleistä. Monissa peleissä on hyvin realistisiakin ja henkeäsalpaavia varjostimia ja efektejä. Sen takia jotkin pelit jopa muistetaan ainoastaan niiden yksityiskohtaisista varjostuksista, valaistuksista tai efekteistä, eikä niinkään pelimekaniikkojen tai tarinan takia.

Godot Engine on vielä toistaiseksi hiukan tuntematon pelimoottori, mutta tämä ei tarkoita sitä, ettei sillä voisi tehdä hyviä pelejä. Pelimoottori on kolmiulotteisten ominaisuuksien kanssa vielä jäljessä, jos verrataan isompiin ja tunnetumpiin markkinoilla oleviin pelimoottoreihin, mutta kaksiulotteisten pelien tekemiseen Godot Engine on loistava valinta. Godot Engine on hyvin aloittelijaystävällinen, mutta se sopii myös erittäin hyvin kokeneemmallekin pelin kehittäjälle. Pelimoottorin valinnassa myös Godot Enginen MIT-lisenssi on houkutteleva. Pelimoottori sopii myös hyvin kehittäjille, jotka ovat kiinnostuneita varjostimien ja efektien ohjelmoimisesta ja haluaavat kokeilla erilaisia tekniikoita. Suosittelen vahvasti antamaan Godot Enginelle mahdollisuuden.

Opinnäytetyötä varten toteutin kolme erilaista kaksiulotteista efektiä, veden, sateen ja saasteen. Kaikki efektit ovat onnistuneita ja olenkin tyytyväinen, kuinka ne toivat peliin maailman elävöitymistä.

Lähteet

(1) Juan Linietsky, Ariel Manzur and Contributors (2007–2021). Godot Engine. Available at: https://godotengine.org Viitattu 16.3.2021

(2) Juan Linietsky, Ariel Manzur and Contributors (2014-2020). Godot Engine. Available at: https://www.patreon.com/godotengine Viitattu 16.3.2021

(3) Juan Linietsky, Ariel Manzur and Contributors (2007-2021). Features. Available at: https://godotengine.org/features Viitattu 16.3.2021

(4) Miika Saajanne, Laura Nevala, Suvi Vikstedt (2020–2021). Litter Run.

(5) Juan Linietsky, Ariel Manzur and the Godot community (2014-2020). Scenes and nodes. Available at: https://docs.godotengine.org/en/stable/getting_started/step_by_step/scenes_and_nodes.html Viitattu 16.3.2021

(6) Juan Linietsky, Ariel Manzur and the Godot community (2014-2020). Instancing. Available at: https://docs.godotengine.org/en/stable/getting_started/step_by_step/instancing.html

Viitattu 16.3.2021

(7) Juan Linietsky, Ariel Manzur and the Godot community (2014-2020). Usein kysytyt kysymykset. Available at: https://docs.godotengine.org/fi/stable/about/faq.html#doc-faq-what-is-gdscript Viitattu 16.3.2021

(8) Juan Linietsky, Ariel Manzur and the Godot community (2014-2020). What is GDNative. Available at: https://docs.godotengine.org/en/latest/tutorials/scripting/gdnative/what_is_gdnative.html#supported-languages Viitattu 17.2.2021

(9) Juan Linietsky, Ariel Manzur and the Godot community (2014-2020). What is Visual Script. Available at: https://docs.godotengine.org/fi/stable/getting_started/scripting/visual_script/what_is_visual_scripting.html Viitattu 17.2.2021

(10) Joey de Vries (2014). Shaders. Available at: https://www.cs.vu.nl/~eliens/download/literatuur-shaders.pdf Viitattu 22.3.2021

(11) Bethesda Game Studios (2011). The Elder Scrolls V: Skyrim.

(12) Juan Linietsky, Ariel Manzur and the Godot community (2014-2018). Introduction to shaders. Available at: https://docs.godotengine.org/en/latest/tutorials/shaders/introduction_to_shaders.html Viitattu 23.3.2021

(13) Juan Linietsky, Ariel Manzur and the Godot community (2014-2018). Shading reference. Available at: https://docs.godotengine.org/en/stable/tutorials/shading/shading_reference/shaders.html Viitattu 5.4.2021

(14) Juan Linietsky, Ariel Manzur and the Godot community (2014-2020). Visual shaders. Available at: https://docs.godotengine.org/en/stable/tutorials/shading/visual_shaders.html Viitattu 5.4.2021

(15) Juan Linietsky, Ariel Manzur and the Godot community (2014-2020). Spatial Shader. Available at: https://docs.godotengine.org/en/stable/tutorials/shading/shading_reference/spatial_shader.html#doc-spatial-shader Viitattu 5.4.2021

(16) Juan Linietsky, Ariel Manzur and the Godot community (2014-2020). Canvas Item Shader. Available at: https://docs.godotengine.org/en/stable/tutorials/shading/shading_reference/canvas_item_shader.html#doc-canvas-item-shader Viitattu 5.4.2021

(17) Juan Linietsky, Ariel Manzur and the Godot community (2014-2020) Particle Shader. Available at: https://docs.godotengine.org/en/stable/tutorials/shading/shading_reference/particle_shader.html Viitattu 5.4.2021

(18) Juan Linietsky, Ariel Manzur and the Godot community (2014-2020). Shading language. Available at: https://docs.godotengine.org/en/stable/tutorials/shading/shading_reference/shading_language.html Viitattu 1.4.2021