Ville Suokas

Dynaaminen muodon uudelleenmuotoilu ajon aikana Godot-pelimoottorissa

Tradenomi
Tietojenkäsittely
Kevät 2024

KAMK • University of Applied Sciences

Tiivistelmä

Tekijä: Suokas Ville

Työn nimi: Dynaaminen muodon uudelleen muotoilu ajon aikana Godot-pelimoottorissa

Tutkintonimike: Tradenomi(AMK), tietojenkäsittely

Asiasanat: muoto, uudelleenmuotoilu, Godot

Opinnäytetyön lähtökohtana oli muodon muokkaamistekniikoiden selvittäminen ja miten niitä voisi soveltaa Godot-pelimoottorissa. Tavoitteeksi muodostui tutkia eri tekniikoita ja niitä soveltamalla toteutettiin yksinkertainen toteutus.

Työssä pohjustettiin aihetta aluksi tutustumalla muodon perusteisiin ohjelmoinnin näkökulmasta ja Godottin ominaisuuksiin ja kehityshistoriaan. Tämän jälkeen käsiteltiin eri tekniikoita muodon uudelleenmuotoiluun, lyhyesti käsitellen niiden toimintaa.

Käytännön osuudessa ohjelmoitiin edellä käytyihin tekniikoihin pohjaten Godot-pelimoottorissa GDScriptillä. Toteutuksessa käytettiin ohjelmoitua muotoa. Tarkasteltiin ohjelman rakennetta ja toimintaa havainnollistaen kuvilla. Kommentointiin toteutusta ja toimintaa huomioiden kehityskohtia.

Lopun yhteenvedossa käsiteltiin, miksi ei tätä toiminnallisuutta niin tänä päivänä ole. Opinnäytetyön tuloksissa osoitettiin kuitenkin, että toteuttamiseen on monia tekniikoita. Moderneilla työkaluilla voidaan hyvin yksinkertaisesti toteuttaa realismia tuovaa muodon uudelleenmuotoilua. Työ itsessään tarjoaa lisäksi syventymistä tietokonegrafiikkaan ja Godotin vahvuuksiin.

KAMK • University of Applied Sciences

Abstract

Author: Suokas Ville

Title of the Publication: Dynamic Mesh Deforming in Godot Game Engine During Runtime

Degree Title: Bachelor of Business Administration,Business Information Technology

Keywords: mesh, deforming, Godot

The start point of the thesis was to become familiar of deforming mesh and how to implement it in Godot game engine. The objective became to study different techniques and use them to create simple implementation.

The thesis set ups topic at start by examine of basics of mesh from perspective of programming and with Godots features and development history. After this different techniques to deform mesh were examined, by shortly summarizing.

In the practical part of the thesis, program using previously examined techniques was programmed inside Godot game engine using GDScript. Programmed mesh was used in implementation. The architecture of the program was reviewed with demonstrating with pictures. Different parts and function of it was reviewed, noting possible improvements.

In the end summary, was discussed why this feature isn't utilized nowadays. The thesis results suggest that implementation can be done with various techniques. Nevertheless, with modern tools simple implementation can create realism increasing mesh deforming. The thesis itself provides insight into computer graphics and Godot game engine strengths.

Sisällys

Symboliluettelo

Omaisuuskirjasto – AssetLibrary, asioiden jakelupaikka, jota voi hyödyntää omissa projekteissa, kuten muotoja tai varjostimia.

Syntaksi – Ohjelmointikielen kirjoitustapa ja sen piirteet

Tesselaatio – Tekniikka, jossa liikutamme verteksejä ylös ja alas, luoden muodon pintaan vertikaalisuutta

RigidBody – Fysiikat omaava objekti Godotissa. Käyttäytyy kuin pallo, ei voi ohjata suoraan, mutta voi ohjailla ulkoisilla voimilla.

CharacterBody – Fysiikat omaava objekti Godotissa. Pääkäyttökohde on hahmot, kuten pelaaja. Ohjataan ohjelmoinnin kautta, fysiikat tulevat painovoimasta ja törmäyksistä.

1 Johdanto

Tässä opinnäytetyössä tutkitaan, miten käsitellä muotoja ajon aikana ja miten se tapahtuu Godot-pelimoottorissa.

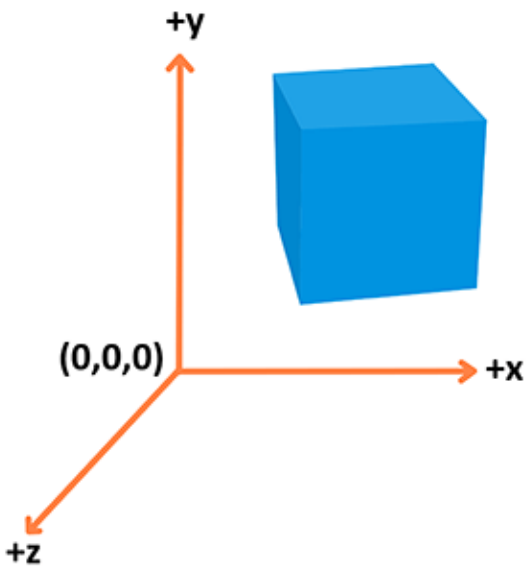
Projektin idea syntyi pelijameilla, kun mietin yksinkertaisen oloista ongelmaa. Miten lytätä esinettä pelin aikana? Olin joskus aiemmin jo miettinyt asiaa eri pelikonsepteja miettiessäni. Godot-pelimoottori valikoitui sen nousevan suosion myötä valtavirran pelimoottoreissa ja henkilökohtainen kiinnostus Godottia kohtaan. Tavoitteena olikin opinnäytetyössä selvittää tapoja joilla muokata muotoa pelinkehityksessä ja miten se toteutetaan Godot-pelimoottorissa.

Aluksi käydään perusteet läpi, mitä muoto on pelinkehityksessä. Sen jälkeen otetaan yleiskatsaus Godot-pelimoottoriin. Tämän jälkeen tutkitaan eri tapoja muodon muokkaamiseen. Käytännön osassa tarkastellaan Godotissa tehtyä toteutusta ja kommentoidaan sitä.

2 3D-grafiikka peleissä

3D-grafiikan ydin on muotojen esittäminen 3D-avaruudessa, kordinaatiojärjestelmän määrittäessä niiden sijainnin.

Alla kuvassa 1. yleisesti 3D-perusmuoto kuutio, ja se sijaitsee 3-ulotteisessa avaruudessa kordinaatistolla, yleensä tässäkin käytössä oleva X,Y,Z.



Kuva 1. 3D-kordinaatisto (MDN Web Docs, n.d.)

2.1 Muodon rakenne

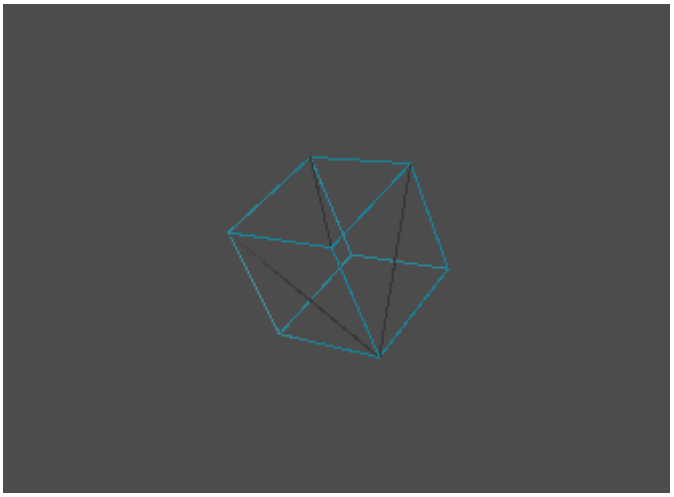
Erinäiset objektit rakentuvat vertekseistä. Verteksi on piste 3D-avaruudessa ja sisältää yleensä lisätietoja, esimerkiksi normaalin katsomissuunnan sekä värin määrittäminen. (MDN Web Docs, n.d.)

Pelinkehityksessä muotojen mallinnus on perinteisesti tehty kolmiomuodoilla. Tämä valinta on tehty sen omaavien ominaisuuksien vuoksi: Yksinkertaisin muoto, jolla on pinta, sen pinta on aina tasainen ja se säilyttää muotonsa pääasiassa, vaikka sitä muokattaisiin.

Periaatteessa tänä päivänä lähes kaikki renderointiratkaisut on toteutettu kolmiorasteroinnilla ja näin tulee olemaan toistaiseksi. Tämän ratkaisun lähtökohdat löytää 3D-pelien alkuajoilta. (Gregory, 2014, 447.)

Kolmio muodostetaan kolmesta verteksistä. Sivut kolmiolla saadaan yhdistämällä vierekkäiset verteksit toisiinsa. Kolmion suunnan määrittämiseksi, meidän tarvitsee tietää kolmion etu- ja takapuoli. Tämä määrittyy kiertosuunnalla, joko myötä- tai vastapäivään. (Gregory, 2014, 449.)

Alla olevassa havainnekuvassa kuution kulmat muodostuu kahdeksasta verteksistä. Kuution sivuista voi nähdä poikki viistosti menevän viivan. Tämä on kolmion sivu. Näin huomataan, että yhden kuution sivun, neliön luomiseksi tarvitaan kaksi kolmiota. Ne voivat kuitenkin jakaa verteksidataa samoissa pisteissä 3D-avaruudessa, jolloin verteksejä on kuution sivulla neljä vaikkakin kolmioita on kaksi. Sivujen kanssa sama juttu, kolmiot jakavat yhteisen sivun, jolloin sivuja on viisi.



Kuva 2. Esimerkkikuutio rautalankamallina

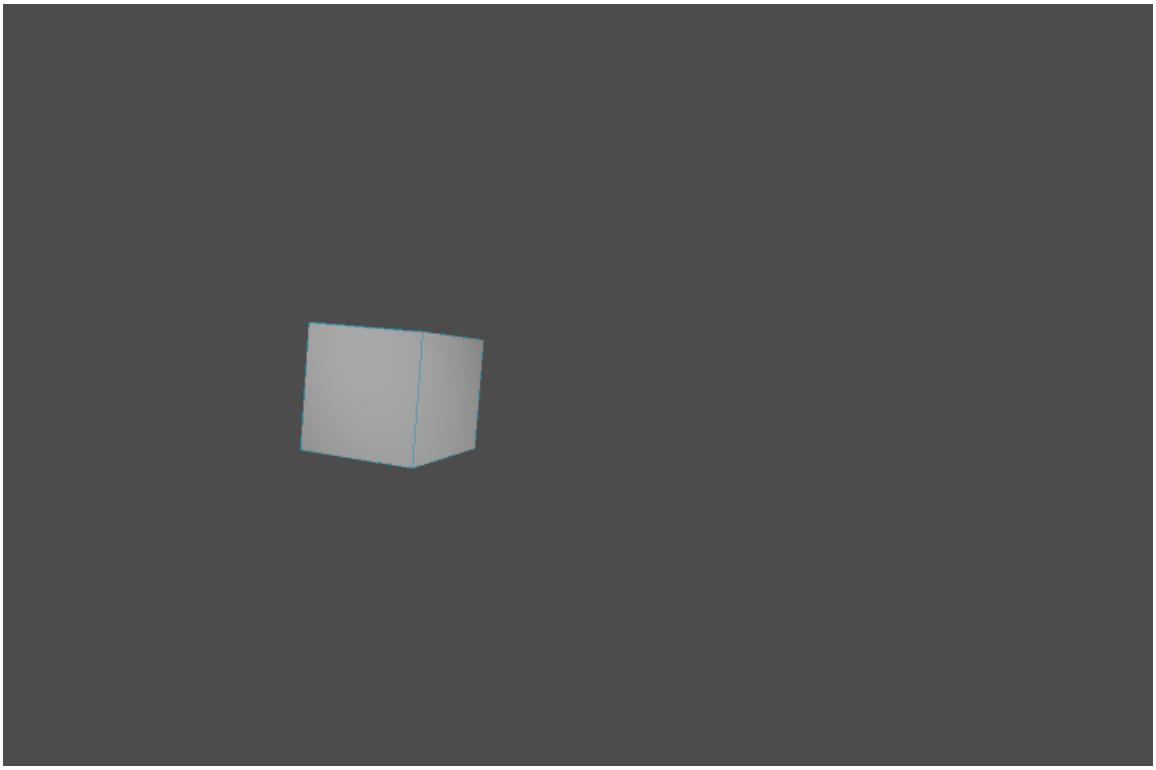
Aiemmin mainitussa osiossa 3D-mallinnus yleensä aloitetaan ohjelmiston tarjoamasta valmiista primitiivimuodosta, kuten kuutio. Mutta muotojen rakentuessa vertekseistä, jotka ovat vain pisteitä 3D-avaruudessa, voidaan muotoja rakentaa ja hallita ohjelmoinnin kautta. Kuvassa 2. oleva kuutio on tehty ohjelmoimalla, toteutus kuvassa 3.

```
53 func _construct_cube(wi, he, de, md):
54
55     # Kuution rakentamiseksi annetaan kulmille positiot,
56     # tässä tapauksessa leveys, korkeus ja syvyys. md on muodon data
57
58     var a := Vector3(0,he,0)
59     var b := Vector3(wi,he,0)
60     var c := Vector3(wi,0,0)
61     var d := Vector3(0,0,0)
62     var e := Vector3(0,he,de)
63     var f := Vector3(wi,he,de)
64     var g := Vector3(wi,0,de)
65     var h := Vector3(0,0,de)
66
67     md[ArrayMesh.ARRAY_VERTEX] = PackedVector3Array([a,b,c,d,e,f,g,h])
68
69     #Asetetaan kolmioiden pisteet
70     md[ArrayMesh.ARRAY_INDEX] = PackedInt32Array([
71         2,1,0, 0,3,2, #Kolmio kääntyy ulospäin kun mennään vastapäivään
72         4,5,6, 6,7,4,
73         3,0,4, 4,7,3,
74         5,1,2, 2,6,5,
75         4,0,1, 1,5,4,
76         2,3,7, 7,6,2
77     ])
```

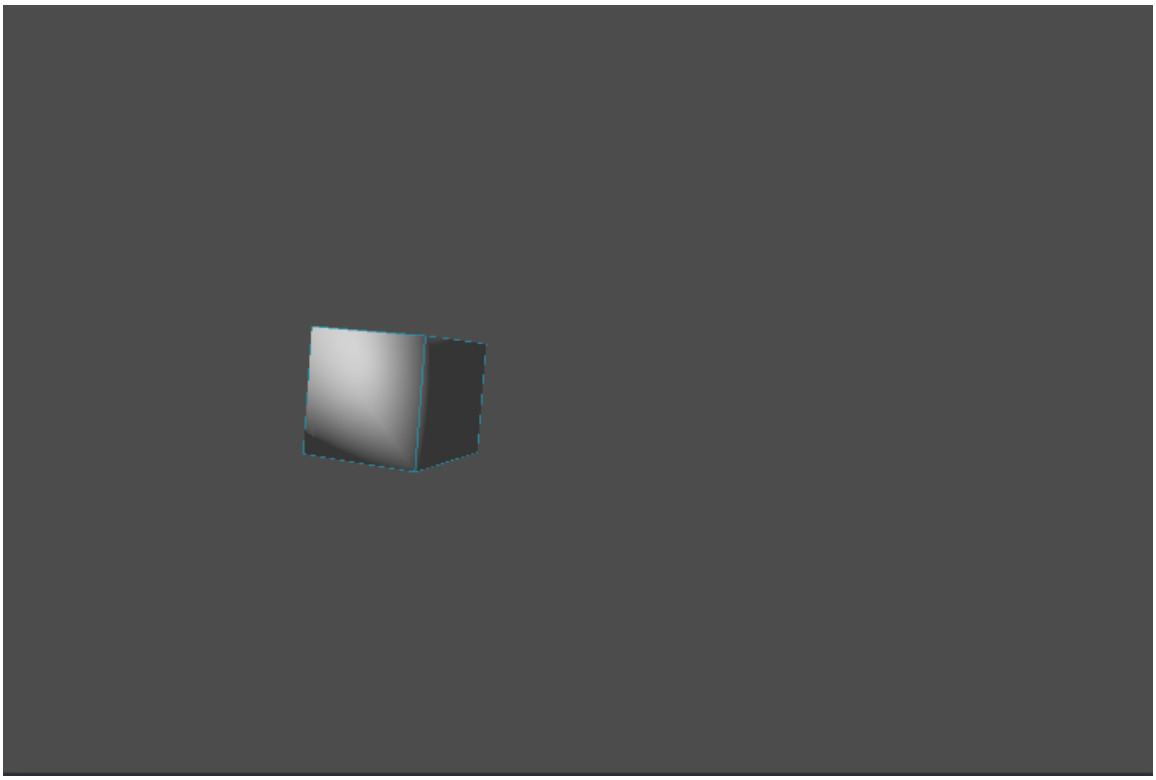
Kuva 3. Kuution rakentaminen ohjelmoimalla

2.2 Verteksidatan muokkaaminen

Verteksi dataa täydentämällä saadaan muoto käyttäytymään paremmin. Kuvassa 4. näkyy, miten valot eivät toimi oikein, vaan kuutio on tasaisen harmaa. Mutta jos vertekseille lasketaan normaalit, alkavat valot käyttäytyä oikein, kuten kuvassa 5. nähdään.



Kuva 4. Kuutio ilman lisäverteksidataa



Kuva 5. Kuutio, jolla on vertekseillä normaalidata

```
for i in range(mdt.get_face_count()):

    var a = mdt.get_face_vertex(i, 0)
    var b = mdt.get_face_vertex(i, 1)
    var c = mdt.get_face_vertex(i, 2)

    var ap = mdt.get_vertex(a)
    var bp = mdt.get_vertex(b)
    var cp = mdt.get_vertex(c)

    # Tällä valot käyttäytyy oikein, mennään kolmion verteksit käänteisessä järjestyksessä 210
    var n = (bp - cp).cross(ap - bp).normalized()

    mdt.set_vertex_normal(a, n + mdt.get_vertex_normal(a))
    mdt.set_vertex_normal(b, n + mdt.get_vertex_normal(b))
    mdt.set_vertex_normal(c, n + mdt.get_vertex_normal(c))

mesh.clear_surfaces()
mdt.commit_to_surface(mesh)
```

Kuva 6. Verteksien normaalidatan asettaminen ohjelmoimalla

Nämä esimerkit osoittavat muotojen olevan vain pohjimmiltaan verteksidataa ja näin muokattavissa. Tämä on tärkeä ymmärtää lopun opinnäytetyön kannalta.

3 Godot-pelimoottori

Ariel Manzur ja Juan Linietsky olivat jo pidemmän aikaa tehneet pelimoottoreita pelien tekemiseen, koska aikoinaan ei vielä yleistarkoitukseen olevia pelimoottoreita ollut. Tämä johti 2007 Godot-pelimoottorin luomiseen teknologian kehityksen myötä. He tahtoivat työkalun, jolla pystyi tekemään pelejä niin iPhonelle ja PSP:lle kuin myös PlayStation 3:lle. (80 Level, n.d.)

Godot-pelimoottorista tehtiin avoin helmikuussa 2014. Se toimii MIT-lisenssin alla, joka periaattessa antaa tehdä mitä vaan Godotilla, kaupallisesta käytöstä lähdekoodin kopiointiin, ilman korvausvelvoitteita tai muuta. (Github, n.d.)

Godotin kehitystä on hallinnoinut vuodesta 2022 Godot Foundation. Godot avoimen lähdekoodin projektina elää ulkopuolisista lahjoituksista. Se koordinoidusti käyttää toimintaan saatuja lahjoituksia Godot-pelimoottorin kehitykseen. Tämä muun muassa kattaa kehittäjien palkkaamista moottorin kehitykseen, laitteiston hankkimista kehittäjille ja matkakulujen hoitaminen alan tapahtumiin. (Godot foundation, n.d.)

3.1 Yleistä ja kehitys

Godot-pelimoottori toimii monilla alustoilla 2D- ja 3D-pelien kehitykseen, ja tavoitteena on helpon kehitysympäristön tarjoaminen käyttäjilleen. Pelejä voi julkaista eri alustoille, niin työpöytäympäristöihin, kuten Windows tai MacOS, kuin puhelimille ja vaikka selaimeen HTML5:llä.

Moottorin itsenäinen vapaan lähdekoodin kehitys tapahtuu yhteisön voimalla, mahdollistaen moottorin muokkaamisen omiin tarkoituksiin. Kaikki mitä Godotilla teet on omaisuuttasi. (Godot Docs, n.d.)

Yhteisö on vahvasti myös mukana ohjaamassa Godotin kehitystä. Kuka tahansa voi tarkastella ehdotettuja muutoksia ja kaikkien, myös pääkehittäjien, muutosehdotukset käyvät läpi tarkastelun ja keskustelun. Godotin kehityksen suhteen ydinajatuksena onkin, että moottori olisi yleismaallinen, kaikkien käyttäjien käytettävissä. (The Stack Overflow Podcast, 2023.)

Palkatut kehittäjät hoitavat yleensä haastellisempia kehityskohteita, jolloin yhteisö voi osallistua helpommin projektin kehitykseen. Nämä ovat yleensä ei niin korkealla prioriteettilistalla ja ei niin teknisesti haastavia. Näin Godot saa hyödynnettyä rajallisia resurssejaan.(Godot Engine, 2023.)

Avoimella lähdekoodilla on kuitenkin rajoituksia alustojen suhteen. Konsolien, kuten XBOX tai PlayStation, kehittäminen tapahtuu suljetuissa ympäristöissä ja vaatii salassapitosopimuksia. Tätä varten 2022 Godotin pääkehittäjät perustivat W4 Gamesin. W4 toimii kaupallisena entiteettinä, jonka kautta Godot pystyy tarjoamaan myös ominaisuuksia, jotka eivät ole itsessään avointa lähdekoodia. (W4 Games, 2022.)

Itse Godotia voi käyttää vaivattomasti perinteisen Windows-ympäristön lisäksi Linuxissa, selaimessa tai vaikka Android-puhelimessa.

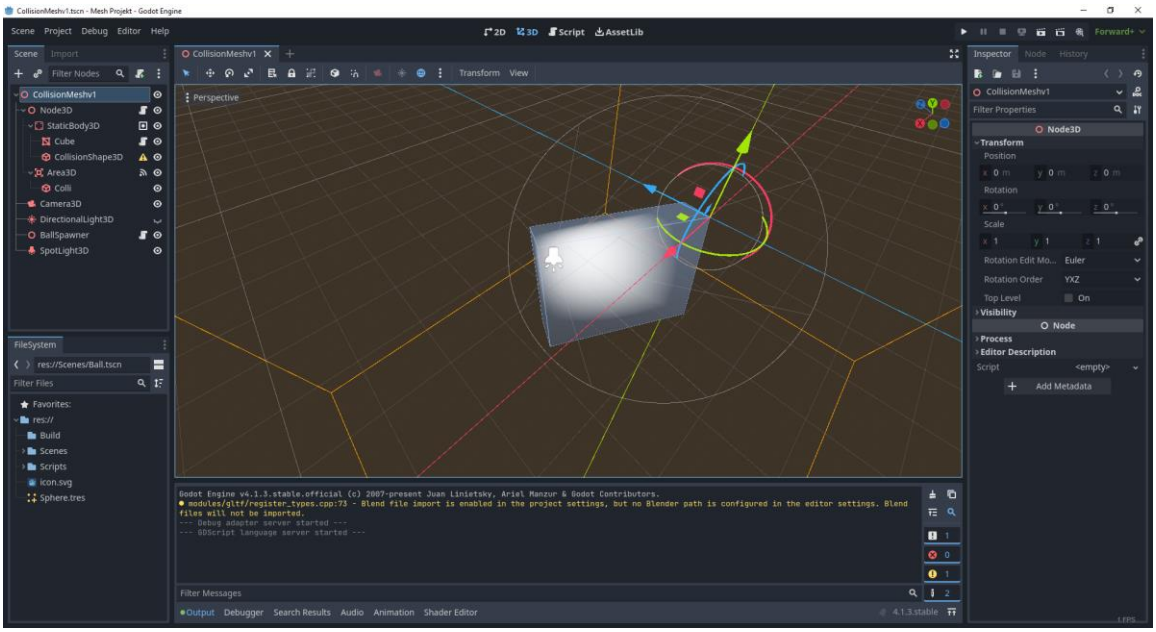
Virallisesti Godot tukee ohjelmointikielinä C#, C++ ja moottoria varten kehitettyä GDScript. Lisäksi pelimoottori on helposti laajennettavissa GDExtensionilla tukemaan muitakin kieliä, kuten Rust tai Swift. (Godot Docs, n.d.)

3.2 Godotin ominaisuudet

Godotissa voi näkymää tarkastella 2D- ja 3D-tasoilla. Lisäksi Godotista löytyy moottorin sisäinen koodieditori ja omaisuuskirjasto. Koodieditori tukee GDScriptiä, ja sitä varten löytyy myös tuki syntaksille ja virheenjäljittäjällä virheiden testaaminen. Kehitysympäristö myös tarjoaa pelimoottorin dokumentaation.

Godot perustuu solmuihin, joka on sen perusluokka. Kaikki objektit ja ympäristöt ovat erilaisia solmuja. Pelimoottorin käyttö tapahtuu luomalla erityyppisiä solmuja ja niille lapsisolmuja. Tämä muodostaa näkymälle puumaisen rakenteen. Näitä näkymiä voi ilmentää sitten toisten näkymien alle. Godotissa näin näkymät onkin pelattava hahmo, pelattava taso tai vaikka käyttöliittymä.

Kuvassa 7. vasemmalla ylhäällä on editorinäkymän rakenne. Sen alla on projektin tiedostot. Keskellä ylhäällä voi vaihtaa 2D, 3D ja tekstieditorin välillä. Keskellä kuvassa näkymäportti. Alhaalla löytyy moottorin ilmoitukset ja notifikaatiot. Oikealla voi tarkastella valitun solmun ominaisuuksia, kuten kuvassa näkyvää Transformia, joka sisältää sijainnin, kierron ja skaalan.



Kuva 7. Editorinäkymä

4 Meshin muuntaminen ohjelman ajon aikana

Meshin muokkamiseen ohjelman suorittamisen aikana voi olla eri syitä, mutta yleensä syynä on realismin lisääminen. Näitä voi olla esimerkiksi törmäyksen tapahtumien vaurioiden esittäminen, kappaleiden litistyminen jäädessään jonkun painavan alle tai vaikka lumessa kahlaaminen.

Kuva 8. Wreckfest (2018) ja Rise of the Tomb Raider(2015) Vasemmassa näkyy auton rungon lyttääntymistä törmäyksestä, oikealla pelihahmon liikkumisen jäljet lumessa. (THQ Nordic, n.d.)(Oliver, 2015.)

Vaikka työssä aiemmin muotoa ja sen rakennetta käsiteltiin, on asiaan muutamia lähestymistapoja. Kyseessä on pohjimmiltaan peliobjektin muokkaaminen lennosta. Seuraavaksi nostan eri vaihtoehtoja asian ratkaisemiseksi.

4.1 Meshin vektorien kontrollointi

Johannes Rajala piti vuonna 2008 seminaaripuheen Asembly-tapahtumassa aiheesta ”Dynamic Mesh Deformation for Car Games”. Esityksessään hän esitti seuraavia tapoja muodon käsittelyyn:

- massa-jousi-malli
    - Hooke’n lakiin perustuva, jossa pisteet ovat jousimaisesti toisiinsa liitoksissa. Myöhemmin mainittu Soft bodyn voi toteuttaa tällä tekniikalla.
- Vapaamuotoinen uudelleenmuotoilu

    - Muoto on säleikkökontrolli järjestelmän sisällä.

- Rajallinen elementtianalyysi

    - Muoto jaetaan rajallisiksi elementeiksi, materiaali arvot tallennetaan jokaiseen elementtiin ja näistä muodostuvat matriisit lasketaan yhteen.

- Muoto-vapaa metodi

    - Ei välitetä verteksien yhteyksistä, kontrolloidaan yksittäisiä verteksejä

Rajala nostaa näistä tavoista seuraavia yhtänäisyyksiä:

1. Ajatus kontrolloida partikkeleita/verteksejä.
2. Fysiikkasimulaatio operoi kontrolloitavissa partikkeleissa.
3. Käytetään matemaattisia lausekkeita muodon datan liikutteluun

Tämän jälkeen hän nostaa ongelmat eri metodeissa:

- massa-jousi-malli

    - Monimutkaisen muodon jousiverkoston rakentaminen

- Vapaamuotoinen uudelleenmuotoilu

    - Luotu säleikkö on rajallinen muodossaan

- Rajallinen elementtianalyysi

    - Laskenta

    - Vaatii esiprosessointia

- Muoto-vapaa metodi

    - Mahdolliset tulkinnat

    - Kasvatettaessa ratkaistavaa ongelmaan, laskenta ja muisti vaatimukset nousevat merkittävästi

Rajala valitsee parhaaksi metodiksi Muoto-vapaan metodin. Syiksi hän antaa seuraavia:

- Tehokas matemaattinenlauseke(Voi tehdä näytönohjaimessa tai usealla säikeellä)
- Joustavuus
- Vakaus
- Nopea esi-prosessointi(Solujen konstruktointi)
- Muodon säilyvyys mietitty

Lopuksi hän mainitsee, että tulevaisuudessa ala näyttäisi liikkuvan kohti rajallisen elementin analyysimetodia.

(Rajala, 2008.)

Tässä onkin hyvä huomioida, että esitys on vuodelta 2008. Tutkiessani aihetta, ovat tekniikat edelleen päteviä, mutta tietokoneiden tehot ovat moninkertaistuneet. Tässä Rajala olikin oikeassa tulevaisuuden suunnan suhteen, mutta mielestäni soft body-tekniikka on tekniikka, joka on tämän päivän juttu.

Keskityn verteksien manipulointiin käytännön osuudessa.

4.2 Soft-body muoto

Soft-body eroaa perinteisestä muodosta, että Soft body koostuu pisteista massaa. Sen muoto muodostuu näistä pisteistä. Esimerkiksi narun simuloinniksi pisteet on yhdistetty pareina. (Gold, n.d.)

Voikin ajatella, että soft-bodyt koostuvat molekyyleistä. Näin objektia voi muokata tarkemmin, mutta on laskennallisesti raskaampaa. Yllä esitetyn *Wreckfestin* autot on tehty soft bodeina.

4.3 Heightfield displacement/Tesselation

Pinnanmuodon kompleksisuutta muodossa voi nostaa lisäämällä polygoneja, mutta tämä on muistin kannalta raskasta. Tesselaatio, auttaa tässä, vieden tehoja näytönohjaimelta tarvittaessa. Displacement map, joka antaa datan verteksien liikuttamiseen, muuttaa tasaisen muodon epätasaiseksi maastoksi. (Flick, 2017.)

4.4 Laplacian-pinnanmuotoilu

Tavallisesti verteksit esitetään karteesisessa koordinaatistossa. Laplacien-tekniikassa käytetään erilaista koordinaatistoa geometrian käsittelemiseksi. Laplacian-koordinaatistolla säilytetään luontainen geometria mahdollisuuksien mukaan samalla mahdollistaen hyödyllisiä pinnan käsittelyn operaatioita. Näitä on vapaamuotoinen uudelleenmuotoilu, pinnoitteen siirtäminen ja muodon siirtäminen. (Sorkine ym., 2004, 1-2.)

5 Käytännön toteutus

Kuten aiemmin työssä on mainittu, on muodon muokkaaminen ajon aikana verteksien kontrollointia. Seuraavaksi esitetään, miten toteutin tämän Godot-pelimoottorissa.

```
305 func _deform_mesh(hitLoc, hitVel : Vector3):
306
307     if hitVel.length() < minDamage:
308         return
309
310     var mdt = MeshDataTool.new()
311     mdt.create_from_surface($StaticBody3D/Cube.mesh, 0)
312     var vertices = []
313
314     for i in mdt.get_vertex_count():
315         vertices.append(mdt.get_vertex(i))
316
317     var localHit = to_local(hitLoc)
```

Kuva 9. Uudelleenmuotoilu-toteutuksen alkuosa

```
319     for i in vertices.size():
320
321         var distFromColl = to_global(vertices[i]).distance_to(hitLoc)
322         var distFromOrig = originalVertices[i].distance_to(vertices[i])
323
324         if distFromColl < deformRadius && distFromOrig < maxDeform:
325             var falloff = 1 - (distFromColl / deformRadius) * damageFalloff
326
327             var deformLoc = localHit + to_local(hitVel).normalized() * falloff
328
329             var diffVec : Vector3 =  deformLoc - vertices[i]
330             diffVec.clamp(Vector3.ZERO,diffVec.normalized() * maxDeform)
331             vertices[i] += (diffVec * damageMultiplier)
332
333             mdt.set_vertex(i, vertices[i])
334
335
336     meshi.clear_surfaces()
337     mdt.commit_to_surface(meshi)
338     $StaticBody3D/Cube._normal_madness()
339     $StaticBody3D/Cube._update_collision_shape()
```

Kuva 10. Uudelleenmuotoilu-toteutuksen loppuosa

5.1 Alun käsittely

Funktio ottaa argumentteina osumakohdan ja osumanopeuden. Godotissa törmäyksistä dataa pystyvät antamaan Rigidbody ja Characterbody. Tässä esimerkissä törmääjä on Rigidbody.

Aluksi tarkastetaan osuman voiman riittävyys vertaamalla osumanopeuden pituutta vaadittuun minimiin. Tämän jälkeen luodaan MeshDataTool. Tämä on Godotin sisäinen luokka muotojen käsittelyyn. Täytetään tämä käytössä olevan kuution muodon datalla. Luodaan myös taulukko valmiiksi.

Käydään läpi meshidatan verteksit ja syötetään ne luotuun taulukkoon. Tallennataan muuttujaan osumakohta paikallisessa avaruudessa. Aletaan käymään läpi taulukon verteksejä.

Ensiksi otetaan etäisyys osumakohdan ja verteksin välillä ja etäisyys verteksin alkuperäisestä sijainnista. Tarkastetaan, ovatko nämä arvot raja-arvojen sisällä. Jos näin on, aloitetaan verteksien käsittely.

5.2 Verteksien käsittely

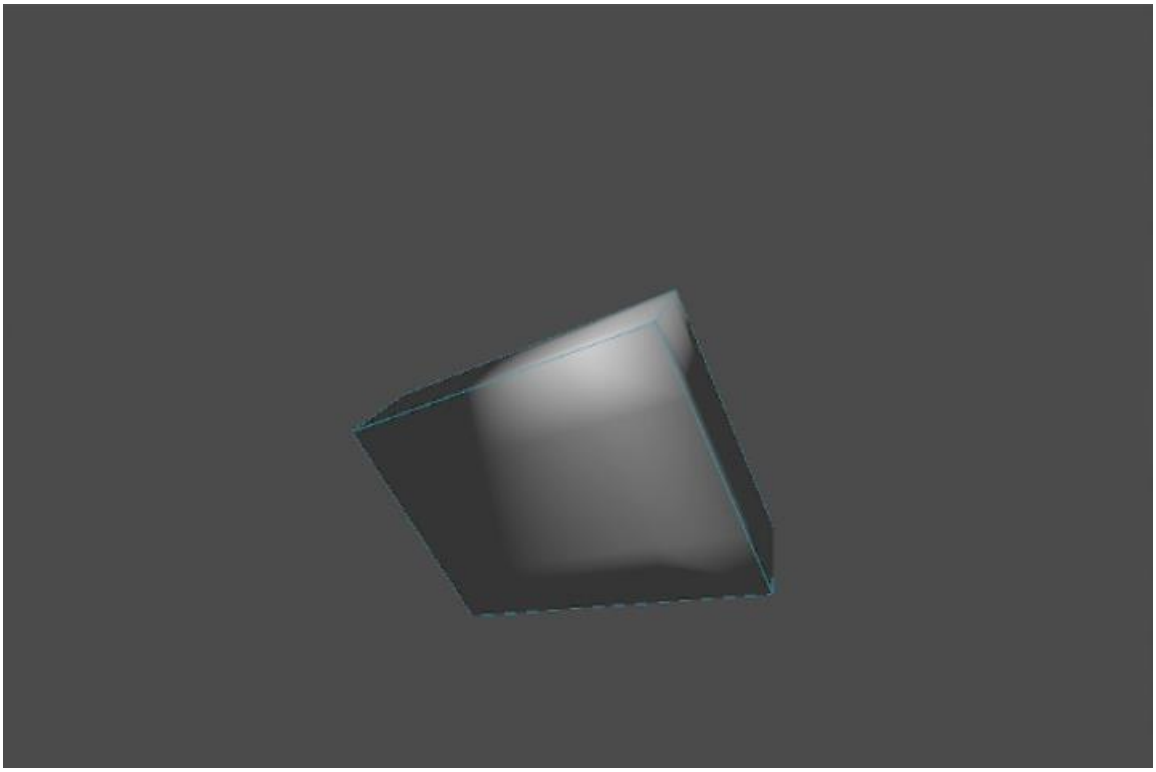
Lasketaan ensin uudelleen muotoiluetäisyys. Käytetään uudelleenmuotoilu kohdesijainnin laskemiseen ottamalla osumakohta ja lisämäällä osumanopeuden suunta paikallisessa avaruudessa kerrottuna lasketulla etäisyydellä.

Otetaan tästä erotus vektorin uudelleenmuotoilusijainnin ja vektorin tämän hetkisen sijainnin välillä. Tästä saatu vektori rajataan matemaattisen clamp()-funktion arvolla nollan ja maksimi uudelleenmuotoilun välille. Tämä lisätään verteksin tämän hetkiseen sijaintiin käyttäen mukana vahinkokerrointa efektin säätelyyn. Lopuksi päivitetään aikaisemmin luodun MeshDataToolin verteksien sijainnin uusilla sijainneilla.

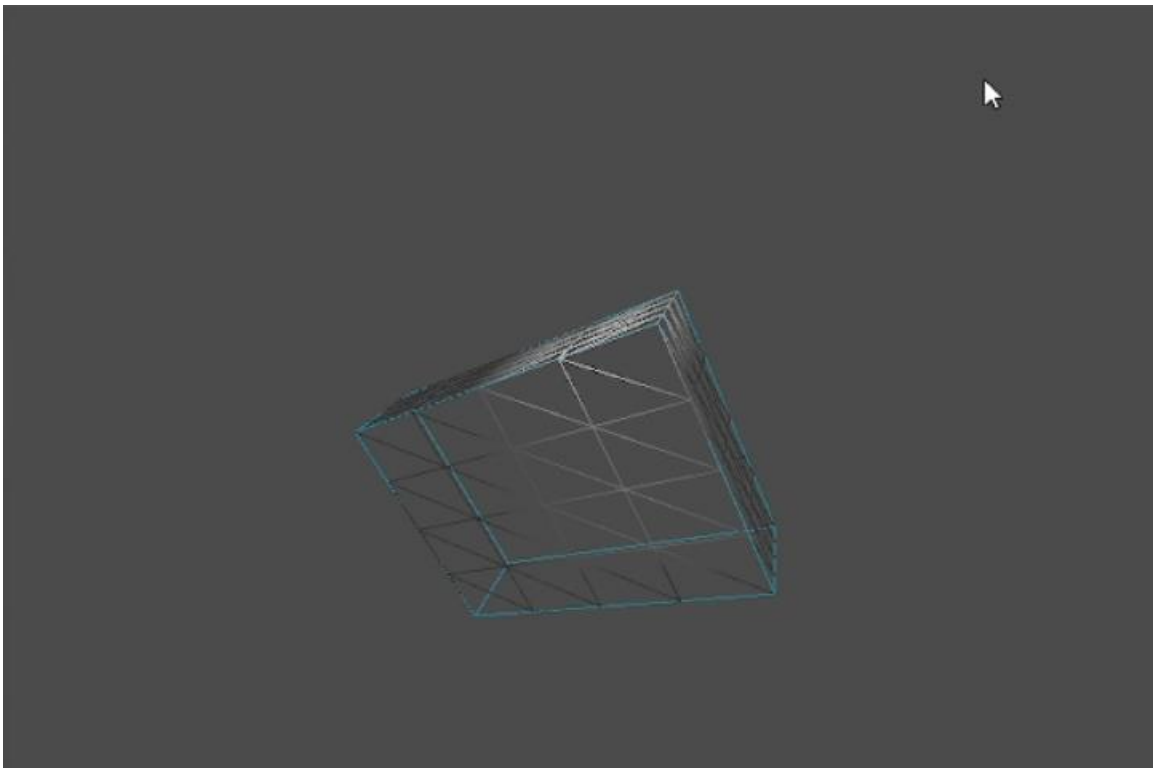
Itse muodon päivittämiseksi ensin tyhjätään tämän hetkiset pinnat, ettei tule päällekkäisyyksiä uuden muodon kanssa. Sen jälkeen ajetaan MeshDataToolin data muodolle. Tämän jälkeen vain päivitetään normaalit ja törmäysmuodot.

5.3 Toteutuksen toiminnallisuus havainnekuvina

Kuvissa 11 ja 12 näemme lähtötilanteen, meillä on suorakulmiomuoto. Muodon polygoneja on osioitu, jotta osuman efekti olisi tarkempi.

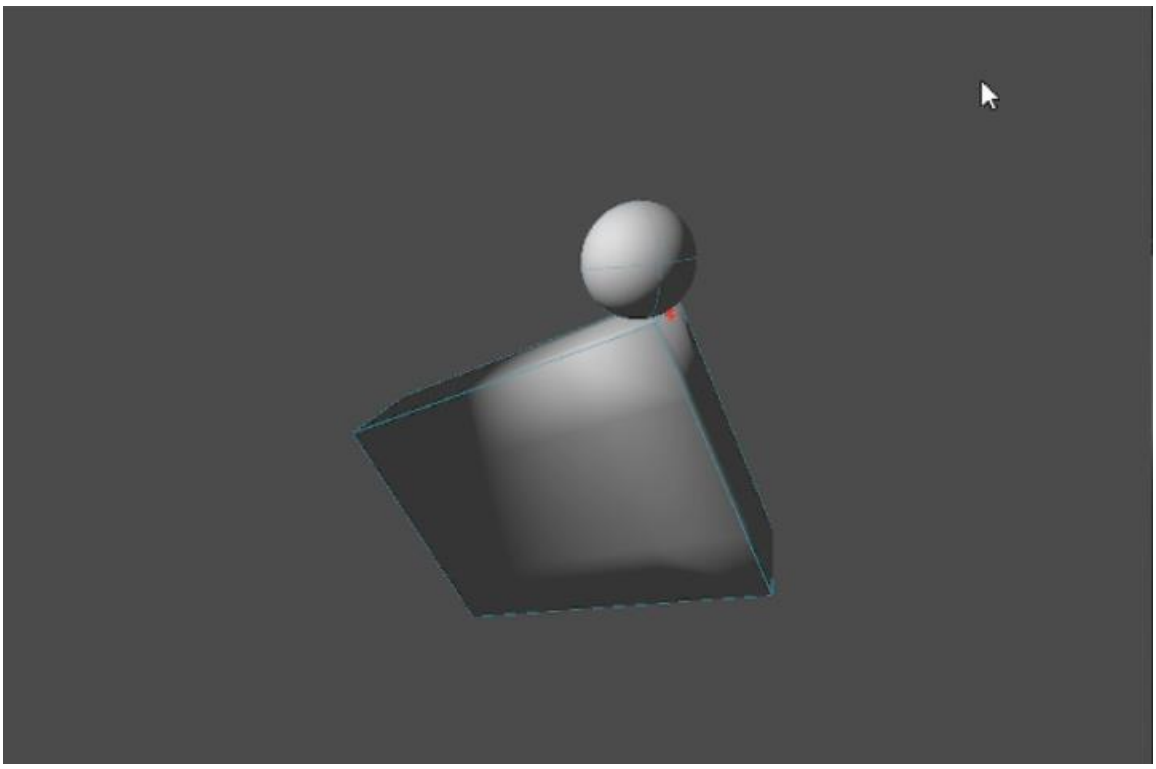


Kuva 11. Oletusnäkymä suorakulmiosta



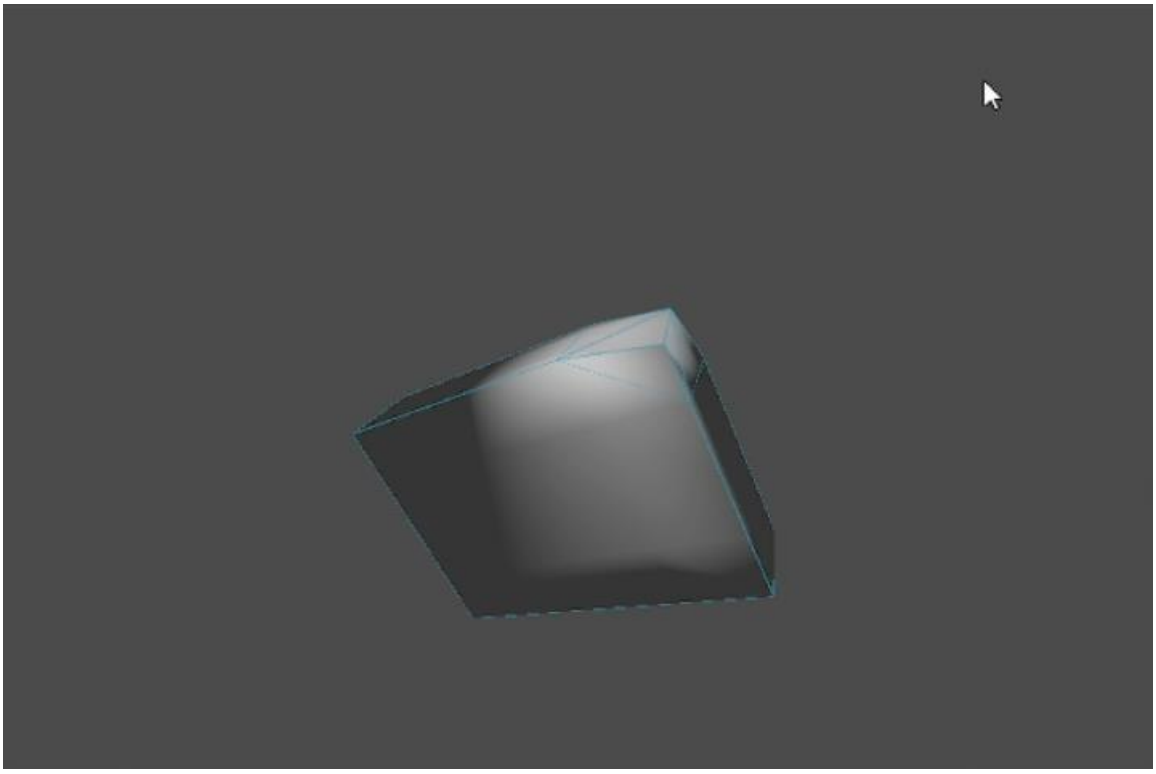
Kuva 12. Rautalankamallinäkymä

Kuvassa 13 näkyvä pallo on Rigidbody, joka törmää suorakulmion kanssa. Kuvassa näkyvä punainen piste on korostuksena näyttämässä törmäyskohdan.

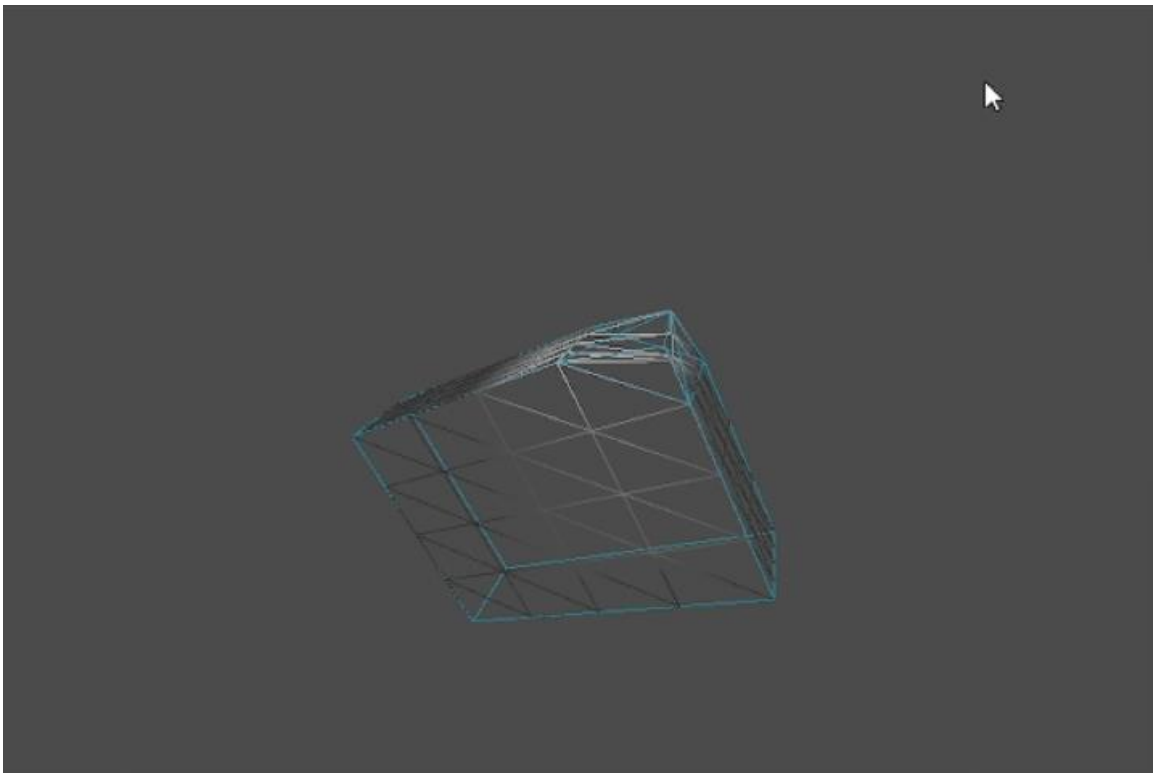


Kuva 13. Törmäystapahtuma pelimoottorissa

Kuvissa 14 ja 15 voi nähdä törmäksen vaikutuksen. Kuvassa 14 näkee kulmauksen luhistumisen myötä valon käyttäytymisen muuttuneen. Kuvassa 15 näkee vaikutuksen paremmin, verteksit ovat luhistuneet sisäänpäin osuman suunnan mukaisesti.



Kuva 14. Törmäysvaikutus perusnäkymässä



Kuva 15. Törmäysvaikutus rautalankamallina

5.4 Kommentteja toteutuksesta

Sain koko toteutuksen mahtumaan yhteen funktioon ja 35 riviin koodia. Tästä näkee, että moderneilla työkaluilla, kuten Godotilla, ei varsinaista käsityötä tarvitse merkittävästi, vaan esimerkiksi laskentaan liittyvät toimenpiteet voi jättää työkalun vastuulle.

Tästä voi huomata, että toteutus perustuun kaikkien verteksien läpikäymiseen. Yksinkertaisissa muodoissa ei tämä ole ongelma, mutta monimutkaisissa muodoissa muodon rajaaminen soluihin on kannattavaa. Testeissä en huomannut suorituskyvyssä laskua ennen kuin puhuttiin tuhansista vertekseistä, joten muun pelinkehityksen tapaan, on hyvä testata suorituskykyä, ennen toteutuksen monimutkaistamista.

Totetuksessa käytetään jonkin verran niin sanottuja ”taikalukuja”. Kuten koodissa rivillä 325 muuttujat ”deformRadius” ja ”damageFalloff” on vain minun asettamia liukulukuja. Paremman toteutuksen kannalta olisi hyvä laskea fysiikan avulla vaikutussuunta. Törmäykseen fysiikassa vaikuttaa kiihtyvyyden lisäksi massa. Jo tätä tietoa hyödyntämällä voitaisiin saada realistisempi törmäyksen vaikutuksen laskeminen, mutta halusin pitää tältäosin toteuksen yksinkertaisena ymmärrättävyyden ja ohjelmoinnin kannalta.

Lisäksi rivillä 339 tapahtuva törmäysmuodon päivittäminen on yksinkertainen.

```
199 func _update_collision_shape():
200     coll = $"../CollisionShape3D"
201     coll.shape = null
202     coll.make_convex_from_siblings()
```

Kuva 16. Koodin pätkä törmäysmuodon käsittelystä

Otetaan vain viittaus törmäysmuotoon, nollataan se ja käytetään Godotista löytyvää funktiota ”make_convex_from_siblings()”. Tämä pyrkii luomaan yksinkertaisen törmäysmuodon sille asetetun muodon mukaan. Tästä saatu lopputulos on laskennallisesti tehokas, mutta yksinkertainen. Tarkemman törmäysmuodon saamiseksi voitaisiin käyttää päivitetyn muodon verteksejä, mutta tämä vaatisi testaamista varsinaisen toimisen kannalta.

Vielä lopuksi ohjelmointikielen valinnasta. Toteutuksessa käytetty GDScript on Godotin oma, Pythoniin perustuva koodikieli. Laskentanopeudessa se häviää kuitenkin pelimoottorin tukemalle C#:lle. Godot mahdollistaa myös GDExtensioneiden avulla C++:san kirjoittamisen ilman moottorin kääntämistä. Koska uudelleenmuotoilu on laskennallisesti vaativaa, on toisen ohjelmointikielen valinta aiheellista jatkokehittäessä, viimeistään tuotannossa.

6 Yhteenveto

Opinnäytetyöhön tietoa etsiessäni mietin, miksi muodon uudelleenmuotoilu on jäänyt taka-alalle peleissä. Pääasiassa jalan- tai renkaanjäljet ovat, mitä pelaaja näkee. Toinen on elokuvamaisten osuuksien aikana, jolloin pelaajalle ei ole kontrollia. Törmäsinkin Jalopnik artikkeliin ”Game Developer Explains Why New Racing Games Suck at Car Damage”. (Ismail, 2023.)

Artikkelissa haastatellaan anonyymiksi jäävää autopelien kehittäjää. Haastattelussa nostetaan syiksi työn määrä sekä autonvalmistajien hyväksyntä ominaisuudelle. Pelien vuosien myötä kasvanut osuus markkinointikanavana on näkynyt autonvalmistajien vaatimuksiin, että vahinko saa korkeintaan olla vähäistä, että auto pysyy edustuskunnossa.

Tämä on linjassa, kun kysyin autopelejä pelaavalta kaveriltani, että miten nyky autopeleissä vahinkoa tehdään. Hän vain totesi törmäyksissä lähinnä vauhdin hidastuvan ja ehkä ajovalojen rikkoutuvan. Haastattelu tosin myös nostaa esille, että harva haluaa ajaa hajonneella autolla, joten pelisuunnittelulla on varmasti osansa asiaan.

Ymmärrys muodosta ja sen ominaisuuksien tutkiminen oli myös ohjelmointitaustaiselle hyvä oppimiskokemus. Grafiikka voi tuntua ohjelmoijalle kaukaiselta, mutta kuten työ osoitti, voi sitä tehdä vaikka ihan ohjelmoimalla.

Godot on Suomen pelinkehityspiireissä lähinnä nimitasolla tunnettu asia, mutta maailmanlaajuisesti se on suosiossa nouseva työkalu. Tähänkin työhön se antoi yllättävän hyvät lähtökohdat ja pystyin käytännön osuudessa keskittymään itse toteutukseen ilman matalan tason ohjelmointia.

Uudelleenmuotoilun pystyi pitämään yksinkertaisena vaikkakin toteutustapoja on useita. Opinnäytetyön jatkokehityskohde voisikin olla toteutuksen kehittäminen pidemmälle. Lähteiden iän puolesta myös lähivuosien pelien tutkiminen nykyaikaisten tekniikoiden selvittämisen kannalta voisi antaa työlle lisäarvoa.

Olen kuitenkin tyytyväinen monipuolisen aiheen kompaktiin lopputulokseen. Työ osoitti, että jo yksinkertaisella toteutuksella voi tuoda lisärealismia peleihin.

Lähteet

80 Level. (N.d), Godot 2.0: Talking with the Creator, Saatavilla: 8.12.2023 https://80.lv/articles/godot2-interview/

Clay, J. (7.11.2023). Clay John: The Future of Rendering in Godot[video]. Godot Engine. Youtube Saatavilla: 8.12.2023 https://www.youtube.com/watch?v=MW3IFMvDTCY

Flick, J. (24.12.2017), Surface Displacement, Saatavilla: 8.12.2023 https://catlikecoding.com/unity/tutorials/advanced-rendering/surface-displacement/

Github. (N.d), godotengine/godot, Viitattu: 8.12.2023https://github.com/godotengine/godot

Godot Docs. (N.d), Introduction, Saatavilla: 8.12.2023 https://docs.godotengine.org/en/stable/about/introduction.html

Godot Docs. (N.d), List of features, Saatavilla: 8.12.2023 https://docs.godotengine.org/en/stable/about/list_of_features.html

Godot Foundation. (N.d), The Godot Foundation, Saatavilla: 8.12.2023 https://godot.foundation/

Gold, S. (N.d), An introduction to soft-body physics, Saatavilla: 8.12.2023 https://stephengold.github.io/Minie/minie/minie-library-tutorials/softbody.html

Gregory, J. (2014). Game Engine Archiitecture. Taylor ja Francis Group.

Ismaiil, A. (2.2.2023), Game Developer Explains Why New Racing Games Suck at Car Damage, Saatavilla: 11.12.2023 https://jalopnik.com/why-car-damage-sucks-in-racing-games-developer-explains-1850060288

Kuva 8. Oliver, M. (24.10.2015) Dev Blog: Snow Tech and Houdini Simulations, Saatavilla: 8.12.2023 https://tombraider.tumblr.com/post/131825841425/dev-blog-snow-tech-and-houdini-simulations-mike

Kuva 8. THQ Nordic. (N.d), Wreckfest [kuva], Viitattu: 8.12.2023 https://order.wreckfestgame.com/

Laplacian Surface Editing, Olga Sorkine Daniel Cohen-Or Yaron Lipman Marc Alexa, 2004

MDN Web Docs. (N.d), Explaining basic 3D theory, Saatavilla: 8.12.2023 https://developer.mozilla.org/en-US/docs/Games/Techniques/3D_on_the_web/Basic_theory

Rajala, J. (2008), Dynamic Mesh Deformation for Car Games [video]. AssemblyTV. Youtube. Saatavilla: 8.12.2023 https://www.youtube.com/watch?v=VNS-0cJlpIs

Sorkine, O., Cohen-Or, D., Lipman, Y., Alexa, M., Rössl, C., Seidel, H-P. (2004) Laplacian Surface Editing

The Stack Overflow Podcast, (28.2.2023), The open-source game engine you've been waiting for: Godot (Ep. 543) [podcast], Viitattu:8.12.2023 https://stackoverflow.blog/2023/02/28/the-open-source-game-engine-youve-been-waiting-for-godot-ep-542/

W4 Games, (9.8.2022). Hello World: W4 Games formed to strengthen Godot ecosystem, Saatavilla: 8.12.2023 https://w4games.com/2022/08/09/hello-world-w4-games/

Liitteet: