Osaamista
ja oivallusta
tulevaisuuden
tekemiseen

Leo Tamminen

Luurankoanimaatiojärjestelmä 3D-verkoille

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tieto- ja viestintätekniikka

Insinöörityö

19.5.2021

Tekijä Otsikko Sivumäärä Aika	Leo Tamminen Luurankoanimaatiojärjestelmä 3D-verkoille 35 sivua 19.5.2021
Tutkinto	Insinööri (AMK)
Tutkinto-ohjelma	Tieto- ja viestintätekniikka
Ammatillinen pääaine	Pelisovellukset
Ohjaaja	Lehtori Miikka Mäki-Uuro

Modernit videopelit käyttävät lähes poikkeuksetta osana pelitaiteen välineistöään erilaisia animaatiotekniikoita ja -teknologioita. Yksi näistä teknologioista on luurankoanimaatio, jossa pelihahmoa ohjataan eräänlaisella luurankoa muistuttavalla rakenteella.

Insinöörityön tarkoituksena oli perehtyä luurankoanimaatioon ja tuottaa toimiva luurankoanimaatiojärjestelmä. Järjestelmän tuli tukea erillisessä mallinnusohjelmassa luotuja verkkoja, luurankoja ja animaatioita. Järjestelmä toteutettiin työn alla olevaan pelimoottoriin. Pelimoottori on kirjoitettu C++-ohjelmointikielellä ja siinä on käytetty Vulkan-grafiikkaohjelmointirajapintaa, ja näitä käytettiin myös animaatiojärjestelmän kirjoittamiseen.

Animaatiojärjestelmän osat toteutettiin tarpeen määräämässä järjestyksessä: ensimmäisenä toteutettiin luuranko eli verkkojen vääntäminen, koska ilman sitä ei verkkoa voida animoida. Toisena toteutettiin itse animaatio, käyttäen yksinkertaisia ohjelmoimalla luotuja animaatioita. Viimeisenä tehtiin animaatioiden lukeminen tiedostosta.

Insinöörityö perustuu enimmäkseen tekijän omaan ohjelmointiosaamiseen, jota on kerrytetty vuosien intensiivisellä harjoittelulla. Animaatiojärjestelmän teoria pohjautuu lisäksi löyhästi yhteen esimerkkitoteutukseen, joka on tehty eri teknologioilla ja jonka laajuus on merkittävästi suppeampi.

Työssä saatiin toteutettua sellainen järjestelmä, jossa kyetään animoimaan satoja malleja samanaikaisesti useilla eri animaatioilla.

Avainsanat	animaatio, videopelit, pelimoottori, Vulkan

Author	Leo Tamminen
Title	Skeletal Animation System for 3D-Meshes
Number of Pages	35 pages
Date	19 May 2021
Degree	Bachelor of Engineering
Degree Programme	Information and Communications Technology
Professional Major	Game Applications
Instructor	Miikka Mäki-Uuro, Senior Lecturer

Most modern video games use different kinds of animation techniques and technologies as part of their game art arsenal. One of these technologies is skeletal animation, where characters are controlled with a skeleton like structure.

The subject for the thesis was to study the technology and produce a working example for such system. System will need to support skeletons and animations authored in an external modeling software. System is implemented to an in-progress game engine. The game engine and therefore also the animation system are written with C++-programming language and they both use Vulkan graphics API.

Different parts of animation system were created in the order of necessity. First was the skeleton i.e. deforming of the mesh. Without it no animation could not be shown. Next was animation itself, using crude programmed animations to work with. Loading the animations from asset files was the last step.

The foundation of the thesis lies mostly on the authors own expertise on programming, which has been rigorously honed over many years. The theory behind of animation system is derived from an example implementation, which has been made with different technologies, and the scope of which was substantially smaller.

The final system is capable of animating and rendering hundreds of models simultaneously using multiple different animations.

Keywords	Animation, Video Games, Game Engine, Vulkan

Sisällys

1 Johdanto

Animaatio on keskeinen osa modernia videopeliä. Se ei ainoastaan lisää immersiota, pelimaailman uskottavuutta, vaan myös luo kokonaan uusia mahdollisuuksia ilmaista asioita, esimerkiksi ihmishahmon kasvojen ilmeillä. Animaatio videopeleissä ammentaa ilmaisukeinoja elokuvasta ja sitä kautta myös jo muinaisesta teatterista ja on näin ollen osa vuosituhansia kestänyttä taideilmaisun jatkumoa.

Toisin kuin elokuvissa ja teatterissa, simuloiduissa videopeleissä tarvitaan laaja valikoima ohjelmointitaidon ja matematiikan työvälineitä. Animaatiot luodaan siihen erikoistuneilla työkaluilla ja ohjelmistoilla, ne tallennetaan tietokoneen kiintolevylle erilaisina tiedostoina ja lopuksi tulkitaan merkityksellisesti valmiissa videopelissä. Tyypillisesti kaikki nämä vaiheet ovat useiden eri tiimien aikaansaannoksia.

Tämän insinöörityön tarkoituksena on toteuttaa kehityskelpoinen pohja luurankoanimaatiojärjestemää varten. Järjestelmän tulee kyetä asettamaan pelihahmoja erilaisiin asentoihin, toistamaan asentoja muuttavaa animaatiota sekä lukemaan animaatioita ja luurankoja ulkoisessa mallinnusohjelmassa tehdyistä tiedostoista.

Animaatiojärjestelmä toteutetaan erääseen työn alla olevaan pelimoottoriin. Pelimoottori projektina on olemassa sen tekijöiden intohimosta oppia ja ymmärtää interaktiiviseen ilmaisuun liittyviä teknologioita mahdollisimman syvällisesti. Animaatiojärjestelmä ammentaa samasta intohimosta. Pelimoottori ei ole millään mittapuulla valmis, ja se vaikuttaa myös työn etenemiseen, sillä tarvittavia tukijärjestelmiä saattaa tulla toteutettavaksi.

Järjestelmä on raportissa jaettu kolmeen osaan, luurankoon, animaatioon ja tiedostojen lukemiseen, ja näitä käsitellään omissa luvuissaan. Luvussa 3 käydään läpi luuranko: mitä sillä tarkoitetaan, miten sitä voidaan käyttää animaatiossa ja miten se toteutettiin järjestelmään. Luvussa 4 esitellään animaatioita ja niiden toistamiseen käytettäviä toimintoja. Luvussa 5 tutustutaan järjestelmässä käytettyyn tiedostoformaattiin ja siihen, miten siitä saadaan luettua sekä luurangon että animaation informaatiot järjestelmän käyttöön.

2 Taustaselvitys

Useissa kaupallisissa ja avoimissa pelimoottoreissa on sisäänrakennettuna valmis yleinen animaatiojärjestelmä. Tästä esimerkkejä ovat Unity, Unreal Engine, Godot ja OGRE. Monissa näistä pystyy ulkoisessa mallinnusohjelmassa luodun animaation tuomisen lisäksi myös luomaan animaatiota suoraan pelimoottorin omassa editorissa.

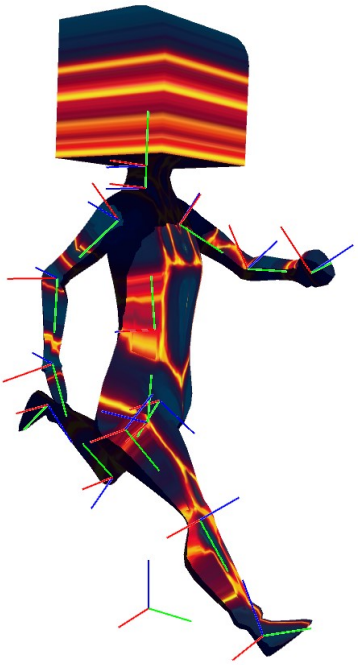
Videopeleissä ja vastaavissa simulaatioissa voidaan käyttää erilaisia animaatioteknologioita. Kaksiulotteisissa peleissä voidaan käyttää esimerkiksi sprite-animaatiota, jossa animaatio on tehty kuvankäsittelyohjelmassa sarjaksi kuvia, joita esitetään peräkkäin yksi toisensa jälkeen. Peleissä, joissa hahmot on mallinnettu kaksi- tai kolmiulotteisiksi verkoiksi, voidaan käyttää luurankoanimaatiota. Siinä on itse verkon (engl. *mesh*) lisäksi mallinnettu luurankorakenne, jonka mukaan animoitavaa verkkoa voidaan vääntää (engl. *deform*) haluttuun asentoon. Verkkoja on myös mahdollista animoida yksi verkon piste kerrallaan, jolloin animaatiota ei rajoita ennalta määrätty luuranko.

Monissa järjestelmissä voidaan hahmon liikkeen lisäksi myös animoida käytännössä mitä ohjelman tai pelin ominaisuutta hyvänsä. Esimerkiksi valaistuksen kulmaa ja väriä voidaan animoida luomaan illuusio päivän ja yön vaihtelusta.

2.1 Luurankoanimaatiojärjestelmä ja sen vaatimukset

Insinöörityön tarkoituksena oli toteuttaa luurankoanimaatiojärjestelmä kolmiulotteisille verkoille. Luurankoanimaatiossa on ennalta määritetty luuranko, joka vääntää animoitavaa verkkoa haluttuun asentoon. Luuranko tarkoittaa tässä luista koostuvaa hierarkkista puurakennetta. Hierarkkisen järjestyksen lisäksi luista tunnetaan esimerkiksi niiden sijainti ja asento eli transformaatio luurangon avaruudessa.

Luurankoa animoidaan kääntämällä, siirtämällä ja skaalaamalla luita, toisin sanoen päivittämällä tietoa niiden transformaatiosta. Kuvassa 1 on luita kääntämällä saatu hahmo juoksevaan asentoon.



Kuva 1. Väännetty esimerkkimalli, jonka luiden lokaalin avaruuden akselit on piirretty nivelien kohdalle.

Toteutetussa järjestelmässä tuli pystyä animoimaan yhtä tai useampaa hahmoa. Animaatioita tuli pystyä toistamaan useita erilaisia, ja niitä täytyi pystyä sekoittamaan, niin että voidaan sulavasti vaihtaa esimerkiksi kävelyanimaatiosta juoksuanimaatioon.

Animoitavia verkkoja, luurankoja ja animaatioita tulee pystyä luomaan ulkoisessa mallinnusohjelmassa ja tuomaan pelimoottoriin jollain olemassa olevalla tiedostotyypillä.

Järjestelmän tuli olla vähintään tyydyttävällä tasolla tehokas. Sen täytyi myös olla joustava, jotta sitä voidaan tulevaisuudessa pelimoottoriprojektin edetessä laajentaa esiin tulevien tarpeiden mukaan.

Toteutettujen ominaisuuksien havainnollistamiseksi ja testaamiseksi tuli toteuttaa jonkinlainen ohjain, jolla testattavaa mallia ja sen animaatioita voidaan käyttää pelimoottorissa. Ohjain voi toimia joko pelaajan syötteellä tai tekoälyllä.

2.2 Käytetyt teknologiat

Pelimoottori on ohjelmoitu C++-kielellä, ja se on käännetty clang++-kääntäjällä [10]. Ohjelman koodi käyttää C++17-standardia [5] ja lisäksi joitain clang++-kääntäjän laajennuksia, joten se ei välttämättä käänny muilla kääntäjillä. Pelimoottori on toteutettu Windows-käyttöjärjestelmälle.

Animaatiojärjestelmä rakennettiin pelimoottorissa valmiina olevan Vulkan-grafiikkaohjelmointirajapintaan perustuvan renderöijän päälle. Luurankoanimaation toteuttamiseen valittiin verkon vääntäminen piste-shaderissa. Kaikki shader-ohjelmat kirjoitetaan Vulkan-ohjelmissa GLSL-shader-ohjelmointikielellä.

Grafiikkaohjelmointirajapinnat mahdollistavat kommunikoinnin varsinaisen peliohjelman ja tietokoneen grafiikkasuorittimen välillä. Grafiikkasuoritin on tietokoneen erikoistunut laite, joka pystyy valtavaan määrään laskutoimituksia sekunnissa, ja näin ollen myös piirtämään näytölle valtavan määrän asioita kussakin pelin päivityssilmukassa. Vulkan on yksi tällainen grafiikkaohjelmointirajapinta, jota voidaan käyttää useiden eri valmistajien grafiikkasuorittimien kanssa. Se ei itsessään ole varsinainen ohjelma tai koodikirjasto, vaan pelkkä ohjelmointirajapinnan määritelmä jonka mukaan grafiikkasuorittimien valmistajat toteuttavat lopullisen kirjaston.

Vulkan on määritelty C-ohjelmointikielelle, mutta monet tahot ovat toteuttaneet myös muunkielisiä versioita. Pelimoottorissa käytetään alkuperäistä C-kielistä versiota; se on yhteensopiva myös C++-kielen kanssa. Ohjelmoitaessa Vulkania, pääosa ohjelmasta kirjoitetaan samalla kielellä kuin muukin ohjelma, tässä tapauksessa C++. Tässä osassa ohjataan grafiikkasuorittimen resursseja, kuten verkoille ja tekstuureille varattavaa muistia, sekä asetetaan kulloinkin piirrettävien mallien tiedot grafiikkasuorittimen saataville oikeaan aikaan ja paikkaan.

Varsinainen piirtäminen tapahtuu shader-ohjelmissa, jotka kirjoitetaan GLSL-kielellä ja suoritetaan grafiikkasuorittimella. Shader-ohjelmat ovat sellaisia ohjelmia, jotka käännettäessä optimoidaan nimenomaan grafiikkasuorittimen arkkitehtuurille. Shader-ohjelmia on muutamia erilaisia, joista yleisimmät ovat vertex- eli piste-shader sekä fragment- eli sirpale-shader. Piste-shader siirtää piirrettävän verkon pisteet niiden mallinnetusta sijainnista niiden lopulliseen sijaintiin näytöllä. Tässä kohtaa myös verkon animoitu asento

otetaan huomioon. Sirpale-shader laskee värit näytön pikseleille sen mukaan, mihin verkon pisteet on piste-shaderissa siirretty. Se toimii samalla tavoin riippumatta siitä, onko mallia animoitu vai ei.

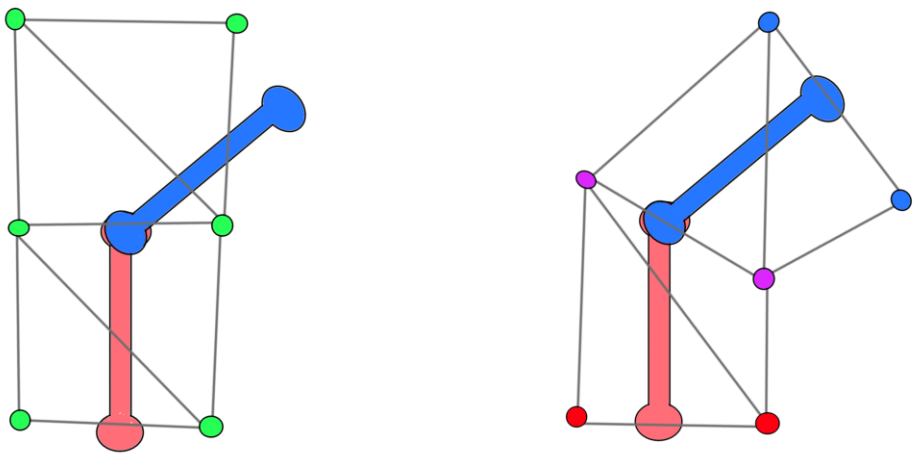
3 Luuranko

Animaatiojärjestelmän ensimmäinen kahdesta isosta tärkeästä osasta on luuranko (engl. *skeleton*). Luuranko mahdollistaa animoitavan verkon asettamisen haluttuun asentoon. Se koostuu yhdestä tai useammasta luusta, joista kukin kattaa muun muassa sellaista informaatiota kuten alkuperäinen ja nykyinen sijainti sekä asema luurangon hierarkiassa.

Luuranko tehdään mallinnusohjelmassa samaan tapaan kuin itse verkko. Luurangolle määritellään yksi juuriluu ja sen alle hierarkkisesti kaikki muut luut. Tällä tavoin esitettynä esimerkiksi olkavarsiluuta seuraa kyynärvarsiluu, jota seuraa ranneluu ja niin edelleen. Luilla ei tarvitse välttämättä olla oikeaa anatomista vastinetta; anatomiaa monesti yksinkertaistetaan rajusti. Hierarkiassa luuta edeltävää luuta sanotaan sen vanhemmaksi ja sen omassa puun haarassa sen jälkeen tulevia luita sen lapsiksi.

Verkko täytyy aina nylkeä (engl. *skinning*) luurangolle, mikä tarkoittaa sitä, että verkon pisteisiin tallennetaan tieto niistä luista, jotka vaikuttavat siihen. Esimerkiksi pään alueella sijaitseviin verkon pisteisiin kirjataan pääluu, ja niihin, jotka ovat lähempänä kaulaa, voidaan kirjata myös kaulaluu. Kullekin pisteelle määrätään kullekin siihen vaikuttavalle luulle painoarvot, joiden mukaan voidaan sekoittaa useamman luun asentoa sen pisteen siirtämiseen. Tämä on hyödyllistä orgaanisten asioiden, kuten ihmisten animoinnissa, erityisesti nivelten kohdalla, jossa vaatteet ja iho venyvät kummankin viereisen luun suuntiin.

Kuvassa 2 näytetään painotettujen pisteiden siirtyminen luiden mukaan. Vihreiden pisteiden painoarvo punaiselle ja siniselle luulle on 0, ja ne eivät liiku luiden mukana ollenkaan. Sinisten ja punaisten pisteiden painoarvot ovat 1 vastaavan värisille luille, ja ne liikkuvat pelkästään niiden luiden mukana. Nivelen kohdalla sijaitsevat violetit pisteet on painotettu sekä siniselle että punaiselle luulle, ja molemmat otetaan huomioon pisteitä piirrettäessä.



Kuva 2. Pisteiden siirtyminen painotetusti eri luiden mukaan.

Samaa luurankoa voidaan käyttää usealle eri verkolle, ja samaa verkkoa voidaan väänttää usealla eri luurangolla. Kukin verkko täytyy kuitenkin nylkeä aina erikseen kullekin luurangolle.

3.1 Luurangon esittäminen järjestelmässä

Luuranko esitetään ohjelman muistissa **AnimatedBone**-tyyppisenä taulukkona, kuten nähdään esimerkkikoodista 1. Kuhunkin luuhun on tallennettu sekä sen nykyinen että oletustransformaatio **Transform3D**-tyyppisinä kenttinä. **Transform3D**-tyyppi sisältää tiedon sijainnista, asennosta ja mittakaavasta.

```
// C++
struct AnimatedBone
{
      // Tila
      Transform3D     boneSpaceTransform;

      // Ominaisuudet
      Transform3D     boneSpaceDefaultTransform;
      m44             inverseBindMatrix;
      s32             parentIndex;
};

struct AnimatedSkeleton
{
      Array<AnimatedBone> bones;
};
```

Esimerkkikoodi 1. Animoitu luuranko (**AnimatedSkeleton**) on yksinkertaisesti taulukollinen animoituja luita.

Luiden tilat on esitetty erityisesti nimenomaan luiden omassa avaruudessa (engl. *bone space*), jolloin yksittäisen luun siirtäminen ja kääntäminen on intuitiivista; sitä varten ei tarvitse ensin laskea vanhempiluiden sijainteja ja asentoja. Esimerkiksi rannetta kuvaava luu tuntee oman sijaintinsa suhteessa kyynärvartta kuvaavaan luuhun. Vanhemmat otetaan huomioon, ja niiden transformaatiot lisätään mallia piirrettäessä.

Luun ominaisuuksia kuvaavat kentät **boneSpaceDefaultTransform** ja **inverseBindMatrix** kertovat luun alkuperäisen sijainnin. Ensimmäistä käytetään luuta animoitaessa, jos animaatiolla ei ole sopivaa kanavaa luulle; tästä lisää animaatioita käsittelevässä luvussa 4. Jälkimmäistä käytetään verkon pisteiden lopullisen siirtymän laskemiseen mallin avaruudessa. Viimeinen kenttä **parentIndex** kuvaa luun sijaintia hierarkiassa, tarkemmin sanottuna se on indeksi **AnimatedSkeleton**-tyypin **bones**-taulukkoon siihen kohtaan, jossa luun vanhempi on tallennettuna.

AnimatedSkeleton tyyppiä käytetään paljon, ja usein. Kunkin luurangon **bones**-taulukko käydään useaan kertaan läpi kussakin päivityssilmukassa, ja määrä kerrotaan vielä animoitavien mallien määrällä. Kaikkia **AnimatedBone**n kenttiä ei kuitenkaan käytetä kaikissa taulukon läpikäynneissä. Tätä on mahdollista optimoida tulevaisuudessa esimerkiksi erottamalla luiden **inverseBindMatrix**-kentät erilliseksi taulukoksi **bones**-taulukon rinnalle animoituun luurankoon.

3.2 Luurangolle määrättyjä sääntöjä

Pelimoottorissa luurangon esittämiselle on määrätty joitain sääntöjä, eli rajoituksia, joilla optimoidaan muistinkäyttöä ja vähennetään turhia ajonaikaisia vertailuja. Toteuttamalla tietotyyppien muistinasettelu hyvin ja poistamalla ylimääräiset kentät voidaan tehdä paljon, mutta nämä säännöt kuvaavat myös sellaisia asioita, jotka liittyvät datan varsinaiseen sisältöön ja joita ei siten voida esittää suoraan C++-kielen ominaisuuksilla. Näiden sääntöjen toteutumista vahditaan luurankoa luettaessa tiedostosta tai luurankoa ohjelmallisesti luotaessa. Kehitysvaiheessa voidaan ohjelman suoritus keskeyttää, jos koh-

dataan sääntöjen vastainen luuranko. Myöhemmin, mikäli julkaistavaan peliin tulee ominaisuus, jolla pelaaja voi tuoda omia luurankoja, joudutaan joko muokkaamaan luurankoa peliin sopivaksi tai hylkäämään luuranko kokonaan.

Luurangon luut on järjestettävä niin, että kunkin luun vanhempi tulee aina taulukossa ennen sitä itseään. Kaikilla muilla kuin juuriluulla tulee olla jokin toinen luu vanhempana, eli irrallisia luita ei suvaita puurakenteessa. Tällöin juuriluu on implisiittisesti aina taulukon ensimmäinen, eikä luita läpikäydessä tarvitse testata, onko luu juuriluu vai onko sillä olemassa vanhempi. Lisäksi luita läpikäydessä silmukassa tiedetään koko ajan, että luun vanhemman tila on jo valmiiksi päivitetty, koska se on tullut käsitellyksi taulukossa aikaisemmin.

Juuriluun avaruuden tulee olla sama kuin animoitavan mallin avaruus, toisin sanoen niiden origot ovat samassa pisteessä ja ne osoittavat samaan suuntaan. Tämä selkeyttää animoitujen mallien asentojen laskemista.

Luurangon luiden määrä on rajattu kolmeenkymmeneenkahteen. Tämä helpottaa luille ohjelman joissain vaiheissa tarvittavan muistin varaamista, kun enimmäismäärä tiedetään etukäteen. Jo enimmäismäärä on täsmälleen kolmekymmentäkaksi, voidaan eri luita valita helposti maskaamalla niitä 32-bittisellä kokonaisluvulla. Tätä toimintoa ei toteutettu, sillä sille ei ole toistaiseksi olemassa käyttötapausta. Enimmäismäärä voisi olla myös esimerkiksi 64, mikä mahdollistaa samat toimenpiteet 64-bittisellä kokonaisluvulla, mutta toistaiseksi 32 luuta on ollut riittävä. Mikäli maskaamista ei haluta tukea, voi enimmäismäärä olla mitä hyvänsä.

3.3 Verkon vääntäminen luurangon avulla

Luurangon tarkoitus on vääntää tai muovata (engl. *deform*) tiettyä verkkoa haluttuun asentoon. Vääntäminen tapahtuu ennen animoidun mallin piirtämistä laskemalla kunkin luun lopullinen asennon muutos luurangon avaruudessa. Tämä lasketaan esimerkkikoodin 2 **update_animated_renderer**-funktiossa.

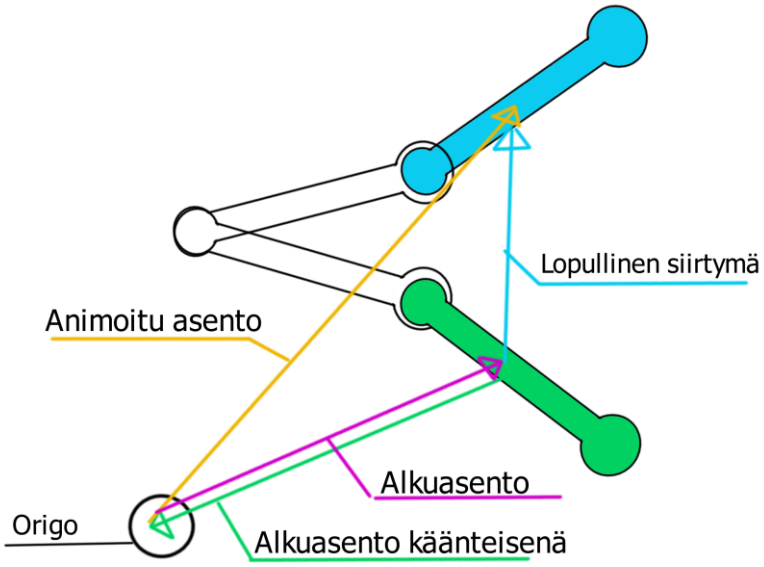
```
// C++
void update_animated_renderer(  m44 * boneTransformMatrices,
                                Array<AnimatedBone> const & skeletonBones)
{
```

```
        s32 boneCount = skeletonBones.count();

        boneTransformMatrices [0]
                  = transform_matrix(skeletonBones[0].boneSpaceTransform);

        for (s32 i = 1; i < boneCount; ++i)
        {
              s32 parentIndex = skeletonBones[i].parent;
              boneTransformMatrices [i]
                    = boneTransformMatrices [parentIndex]
                    * transform_matrix(skeletonBones[i].boneSpaceTransform);
        }

        for (s32 i = 0; i < boneCount; ++i)
        {
              boneTransformMatrices [i]
                    = boneTransformMatrices [i]
                    * skeletonBones[i].inverseBindMatrix;
        }
}
```

Esimerkkikoodi 2. Luurangon asennon päivittäminen animoidun piirtäjän luumatriisitaulukkoon.

Funktiossa lasketaan ensimmäisenä juuriluun transformaatio; tämä tehdään silmukan ulkopuolella, jotta silmukassa ei tarvitse iteraatiolla testata, onko kyseessä juuriluu. Muut luut lasketaan silmukassa ja niissä otetaan vanhempiluut huomioon. Luille määrättyjen sääntöjen mukaan jokainen luu tulee silmukassa vasta oman vanhempansa jälkeen, joten vanhemman transformaatio voidaan lukea **boneTransformMatrices**-taulukosta, johon lopputulokset myös tallennetaan.

Ensimmäisen silmukan jälkeen **boneTransformMatrices**-taulukossa on kunkin luun sijainti koko luurangon avaruudessa. Piirtämistä varten tarvitaan luun siirtymä sen alkuperäisestä sijainnista luurangon avaruudessa sen nykyiseen sijaintiin samassa avaruudessa (ks. kuva 3).



Kuva 3. Luun siirtäminen animoidun asennon ja alkuasennon avulla. ”Alkuasento käänteisenä”-nuoli kuvaa luun **inverseBindMatrix**ia.

Hakemalla alkuperäisen ja lopullisen sijainnin erotus saadaan haluttu transformaatio. Transformaatiomatriisien laskusääntöjen mukaan tämä lasketaan kertomalla loppusijainnin transformaatio alkusijainnin käänteistransformaatiolla.

3.4 Väännetyn verkon piirtäminen grafiikkaohjelmointirajapinnassa

Väännetyn verkon piirtäminen Vulkan-grafiikkaohjelmointirajapinnassa vastaa pitkälti tavallisen verkon piirtämistä. Yksinkertaistettuna prosessi on jokseenkin seuraavanlainen:

1. Piirrettävän verkon pisteet ja indeksit sisältävä puskuri aktivoidaan.
2. Piirrettävän verkon muulle informaatiolle, kuten mallin transformaatiolle, varataan oikea määrä muistia yleiskäyttöpuskurista ja alue aktivoidaan myös.
3. Samat vaiheet toistetaan kullekin toissijaiselle piirtokohteelle, kuten varjoille.
4. Grafiikkasuoritin suorittaa piste-shaderin, joka laskee verkon pisteen projisoidun sijainnin näytöllä, sekä sirpale-shaderin, joka laskee värin kullekin pikselille sen mukaan, mikä verkon piste tai pinta osuu sen kohdalle.

Väännettyjä verkkoja piirrettäessä eroa animoimattomaan malliin tulee kahdessa vaiheessa. Vaiheessa 2 täytyy muistia varata myös luurangon luiden transformaatiotaulukkoa varten. Vaiheessa 4 piste-shader laskee myös pisteeseen vaikuttavien luiden transformaatioiden painotetun summan ja siirtää pistettä myös sen mukaan. Muut vaiheet ovat identtisiä tavallisen verkon piirtämisen kanssa.

Datan kopioimiseen keskussuorittimen ja grafiikkasuorittimien muistien välillä tarvitaan tietotyypit sekä C++- että GLSL-ohjelmiin. Nämä on esitetty esimerkkikoodissa 3.

```
// C++
struct ModelUniformBuffer
{
      alignas(16) m44 localToWorld;
      alignas(16) f32 isAnimated;
      alignas(16) m44 bonesToLocal [32];
};

// GLSL
layout(set = 2, binding = 0) uniform ModelUniformData
{
      mat4 localToWorld;
      float isAnimated;
      mat4 bonesToLocal [32];
} model;
```

Esimerkkikoodi 3. C++- ja GLSL-kieliset rakenteet, joilla mallin ja luurangon tiedot kopioidaan ohjelmasta grafiikkasuorittimelle. Kummallakin on kielestä riippuvat lisämääritteet, joilla varmistetaan rakenteiden identtinen esitysmuoto tietokoneen muistissa.

Pelimoottorissa 4 x 4 -transformaatiomatriiseja kuvataan **m44**-tyypillä. GLSL-kielessä, jota käytetään shader-ohjelmien kirjoittamiseen grafiikkasuorittimelle, on sisäänrakennettuna **mat4**-tyyppi, joka kuvaa samaa rakennetta. Skalaarityyppi **f32** on 32-bittinen liukuluku, GLSL:ssä vastaava tyyppi on **float** (C++-kielessä on myös **float**-tyyppi, mutta pelimoottorissa näille käytetään tyyppialiasta).

Kommunikoitaessa eri ohjelmointikielien välillä on tärkeää ottaa huomioon datan täsmällinen asettelu, koska eri kielten kääntäjät eivät välttämättä kohdista (engl. *align, alignment*) muistia samalla tavalla. GLSL- ja C++-kielissä on erilaiset oletussäännöt muuttujien kohdistuksille. Kohdistukset saattavat lisäksi vaihdella keskussuorittimen ja grafiikkasuorittimien, sekä ajurien välillä, joten on tarpeellista kohdistaa muuttujat eksplisiittisesti. Pelimoottoria ei toistaiseksi ole käytetty kehitysympäristön ulkopuolella, joten tässä tapauksessa on ollut riittävää kohdistaa C++-tyypin **ModelUniformBuffer**in kentät **alignas**-avainsanalla. Ilmaisu **alignas(16)** esimerkiksi tarkoittaa, että muuttujan voi asettaa

0, 16, 32, 48 jne. tavun päähän tyypin alusta, ja tarvittaessa muuttujien väliin voidaan lisätä tyhjiä tavuja niin, että ehto toteutuu. Tämä tulee uudelleen tarkastelun kohteeksi, kun pelimoottoria aletaan testata kehitysympäristön ulkopuolella.

Malleja piirrettäessä grafiikkasuorittimelta varataan kullekin mallille yhden **ModelUniformBuffer**in tarvitsema tila (esimerkkikoodi 4), ja siihen kopioidaan päivityssilmukassa laskettu mallin yleinen transformaatiomatriisi **localToWorld**-kenttään sekä luurangon luiden transformaatiomatriisit **bonesToLocal**-kentän alkioihin. Luurangossa ei välttämättä ole kolmeakymmentäkahta luuta, vaikka **bonesToLocal**-taulukkoon on varattu niille tila. Tällä ei ole väliä, sillä verkon pisteisiin tallennetut luuindeksit eivät viittaa ylimääräisten luiden alkioihin. Sen sijaan ylimäärä luita on haitallista ja aiheuttaa ohjelman hallitun kaatumisen.

```
// C++
...

// Tarvittavan muistin määrän laskeminen ja varaaminen
u64 uniformBufferOffset = context->currentUniformBufferOffset;
u64 uniformBufferSize
     = get_aligned_uniform_buffer_size(context, sizeof(ModelUniformBuffer));

context->currentUniformBufferOffset += uniformBufferSize;


// Grafiikkasuorittimen muistin kuvaaminen ohjelmaan
ModelUniformBuffer * pBuffer;

vkMapMemory(    context->device,
                context->modelUniformBuffer.memory,
                uniformBufferOffset,
                uniformBufferSize,
                0,
                (void**)&pBuffer);


// Mallin ja luurangon informaation kopioiminen
pBuffer->localToWorld = transform;
pBuffer->isAnimated  = bonesCount > 0 ? 0 : 1;
copy_memory(pBuffer->bonesToLocal, bones, sizeof(m44) * bonesCount);

// Muistin kuvauksen purkaminen
vkUnmapMemory(context->device, context->modelUniformBuffer.memory);

...
```

Esimerkkikoodi 4. **Context** on osoitin rakenteeseen, johon on tallennettu kaikki piirtämiseen Vulkanissa tarvittavat asiat. **Context**iin varatusta grafiikkasuorittimen kanssa yhteensopivasta muistista varataan edelleen pienempi osa kullekin piirrettävälle mallille, ja tiedot kopioidaan siihen.

Puskureissa käytettävä muisti on oikeastaan varattu grafiikkasuorittimelta jo ohjelman käynnistyessä, ja tässä vaiheessa siitä varataan pienempiä lohkoja yksittäisten mallien käyttöön. Varatun muistin kohdalta kuvataan oikea määrä muistia **vkMapMemory**-funktiolla. Tähän voidaan suoraan asettaa arvot **localToWorld**- ja **isAnimated**-kenttiin. Koska C++-kielessä taulukoiden asettaminen ei ole mahdollista, viimeinen kenttä **bonesToLocal** kopioidaan **copy_memory**-funktiolla.

Pisteen lopullinen sijainti näytöllä tai kohdetekstuurissa lasketaan piste-shaderissa. Piste-shaderiin syötetään **in**-avainsanalla merkityt muuttujat yleiskäyttöpuskurista yksi piste kerrallaan. Esimerkkikoodin 5 piste-shaderin **main**-funktio suoritetaan kerran kullekin piirrettävän mallin pisteelle, ja sijainti näytöllä lasketaan siinä. Piirrettäessä tavallista animoimatonta verkkoa lasketaan pisteen sijainti siirtämällä **inPosition**-vektoria suoraan kameran ja mallin transformaation mukaan ja sen normaalivektori kääntämällä **inNormal**-vektoria mallin transformaation mukaan. Tällöin ne piirtyvät sen mukaan, kuin ne on mallinnusohjelmassa asetettu.

```
// GLSL
in vec3 inPosition;
in vec3 inNormal;
in uvec4 inBoneIndices;
in vec4 inBoneWeights;

void main()
{
      mat4 poseMatrix =
            inBoneWeights[0] * model.bonesToLocal[inBoneIndices[0]] +
            inBoneWeights[1] * model.bonesToLocal[inBoneIndices[1]] +
            inBoneWeights[2] * model.bonesToLocal[inBoneIndices[2]] +
            inBoneWeights[3] * model.bonesToLocal[inBoneIndices[3]];

      vec4 posePosition = poseMatrix * vec4(inPosition, 1);
      vec4 poseNormal = poseMatrix * vec4(inNormal, 0);

      // lisää laskuja, joissa posePositionia ja poseNormalia käytetään
      // pisteen lopullisen sijannin ja normaalivektorin laskemiseen
}
```

Esimerkkikoodi 5. Luurangon asennon verkon pisteisiin aiheuttaman siirtymän laskenta piste-shaderissa.

Animaation aiheuttama vääntyminen otetaan huomioon rakentamalla kullekin pisteelle ylimääräinen transformaatio, tässä **poseMatrix**, jolla piste siirretään ja käännetään oikein. Transformaatio rakennetaan laskemalla painotettu summa yleiskäyttöpuskuriin syötetyistä animoitujen luiden transformaatioista. Valittavien luiden indeksit luetaan **inBoneIndices**-vektorin elementeistä ja niiden painoarvot **inBoneWeights**-vektorista.

Kummatkin on määritelty alun perin mallinnusohjelmassa, ja ne on verkkoa luettaessa tallennettu verkon grafiikkasuorittimen esitykseen.

Sekä **posePosition**ia että **poseNormal**ia käsitellään homogeenisina koordinaatteina, eli kolmiulotteisessa avaruudessa niihin lisätään neljäs komponentti, joka on sijaintia kuvaaville vektoreille kuten **posePosition** 1 ja suuntaa kuvaaville vektoreille kuten **poseNormal** 0.

Tästä eteenpäin **posePosition** ja **poseNormal** korvaavat tavallisten mallien piirtämisessä käytettävät **inPosition**- ja **inNormal**-muuttujat.

4 Animaatio

Toinen kahdesta tärkeästä osasta animaatiojärjestelmässä on itse animaatio. Yleisesti voidaan animoida mitä tahansa pelin sisältöä, mutta kuten insinöörityön nimi jo paljastaa, tässä työssä välitetään ainoastaan kolmiulotteisten verkkojen luurankoanimaatioista. Järjestelmä rakentuu luvussa 3 kuvatun luurankokonseptin päälle; animaatiot ohjaavat aina yhden luurangon luiden siirtymiä ja kääntymiä luurangon omassa avaruudessa.

Animaatioita voi jäsennellä eri tavoilla. Voidaan esimerkiksi määrätä yhdelle aikajanalle avainkehyksiä, jotka sisältävät yhden kokonaisen asennon koko luurangolle, eli yhden siirtymän ja yhden kääntymän jokaiselle luurangon luulle. Toinen tapa on määrätä kullekin luulle omat erilliset aikajanansa, jolloin animoidaan yhtä luuta kerrallaan, ja animaatiot yhdessä tuottavat hahmon kokonaisen animaation. Ensimmäisen tavan etu on tarve vähemmälle kirjanpidolle, kun taas jälkimmäisen tavan etu on joustavuus ja mahdollisuus vähempään muistinkäyttöön, kun voidaan jättää paikallaan olevien luiden avainkehykset kirjaamatta. Animaatiojärjestelmä on toteutettu tässä työssä jälkimmäistä mallia mukaillen.

4.1 Animaation esittäminen ohjelmassa

Animaatioita jaoteltiin pelimoottorissa niiden avainkehysten interpolaatiotavan ja animoitavan kohteen mukaan. Animaatiojärjestelmä tukee kahta interpolaatiotapaa, porrastettu

(engl. *step*) ja lineaarinen (engl. *linear*). Porrastettu animaatio näyttää aina viimeisimmän ohitetun avainkehyksen. Lineaarisessa interpolaatiossa lasketaan animaation ajasta suhteellinen etäisyys edelliseen ja seuraavaan avainkehykseen ja lasketaan sillä painotettu summa näiden arvoista.

Avainkehykset tallennetaan animaatiokanavia kuvaaviin tyyppeihin (ks. esimerkkikoodi 6). Järjestelmä tukee siirtymän (engl. *translation*) ja kääntymän (engl. *rotation*) animointia. Kumpaakin kuvaamaan on omat samankaltaiset tyypit **TranslationChannel** ja **RotationChannel**. Niihin on tallennettu **targetIndex,** joka kertoo kohteena olevan luun indeksin luurangon luutaulukossa. Kanavatyyppien kenttä **times** kertoo avainkehysten ajankohdat animaation aikajanalla. Siitä voidaan myös lukea avainkehysten lukumäärä. Avainkehysten varsinaisia arvoja eli siirtymiä ja kääntymiä kuvaavat taulukot **translations** ja **rotations**. Siirtymää kuvataan järjestelmässä kolmiulotteisena vektorina **v3**, ja kääntymää kvaterniona **quaternion**. Nämä ovat ne arvot, joiden mukaan animoitavia luita siirretään ja käännetään animaation edetessä. Viimeisenä niihin on tallennettu interpolointitapa, **interpolationMode**.

```
struct TranslationChannel
{
      s32                   targetIndex;
      Array<f32>            times;
      Array<v3>             translations;
      InterpolationMode     interpolationMode;
};

struct RotationChannel
{
      s32                   targetIndex;
      Array<f32>            times;
      Array<quaternion>     rotations;
      InterpolationMode     interpolationMode;
};

struct Animation
{
      Array<TranslationChannel> translationChannels;
      Array<RotationChannel>    rotationChannels;
      f32                       duration;
};
```

Esimerkkikoodi 6. Animaatioiden tallentamiseen käytettävät tietotyypit.

Animaatio itse koostuu useasta animaatiokanavasta, jotka on tallennettu kohdetyypin mukaan eri taulukoihin. Animaation kokonaiskesto, aika nollasta kaikkien animaation kanavien myöhäisimpään avainkehykseen, on tallennettuna erikseen; sen mukaan animaation aika nollataan ja avainkehysten toistaminen aloitetaan alusta.

Animointia varten täytyy luurangon ja animaatioiden välille luoda yhteys. Tämä tehdään järjestelmässä esimerkkikoodin 7 mukaisella **SkeletonAnimator**-tietotyypillä. Luuranko on tallennettuna sen **skeleton**-kenttään suoraan objektina. Animaatiot yhdistetään osoitintaulukkona, sillä animaatiot on tallennettu ohjelman muistiin yhden kerran globaaliin taulukkoon, ja useampi malli voi käyttää niitä samaan aikaan. Animaation painot **weights**-kentässä ovat niin ikään taulukossa, jossa kukin alkio vastaa yhtä edellisen taulukon animaatiota. Kummankin lukumäärä kerrotaan **animationCount**-kentässä.

```
// C++
struct SkeletonAnimator
{
      AnimatedSkeleton     skeleton;

      s32                  animationCount;
      Animation const **   animations;
      f32 *                weights;

      f32                  animationTime;
};
```

Esimerkkikoodi 7. **SkeletonAnimator**-tyyppi kokoaa yhteen luurangon ja animaatiot.

Animaatioita päivitetään edistämällä animaation aikaa; sitä kuvaa **animationTime**. Animaation aika on yhteinen kaikille samaan aikaan toistettaville animaatioille, mutta kukin animaatio rajoittaa oman lokaalin aikansa oman kestonsa mukaan. Tämä mahdollistaa animaatioiden joustavamman toistamisen, sillä niiden yksilöllisiä aikoja ei tarvitse erikseen ylläpitää. Toisaalta yhteisen ajan nollaamiseen animaation päästessä loppuun ja alkaessa taas alusta ei ole yksiselitteistä sopivaa aikaa. Tämä ratkaistiin nollaamalla aika aina 60 sekunnin välein, mutta se saattaa aiheuttaa artefakteja, jos jonkin animaation aika ei mene tasan siihen. Ratkaisua tähän ongelmaan pohditaan tulevaisuudessa lisää.

4.2 Animaation toistaminen ja luurangon päivittäminen

Animaation toistaminen on järjestelmässä hyvin pitkälle sama asia kuin luurangon päivittäminen, sillä animaation nykyinen tila on tallennettuna nimenomaan luurankoon. Molemmat tehdään tämän vuoksi samassa **update_skeleton_animator**-funktiossa.

Animaation tilan edistäminen järjestelmässä tapahtuu yksinkertaisesti edistämällä animaation aikaa. Animaatioiden aika on tallennettuna yhteisesti **SkeletonAnimator**-tyypin muuttujaan, jota kasvatetaan tiettyyn pisteeseen asti ja sitten nollataan ja jatketaan taas alusta. Nollaus tehdään, jotta animaatioiden interpoloinnissa käytettävä aika ei saavuttaisi sellaista suuruutta, jossa liukulukujen esitystarkkuus aiheuttaisi ongelmia.

Nykyisen tilan laskeminen päivitetystä ajasta alkaa järjestämällä animaatioiden kanavat yksittäisten luiden mukaiseen järjestykseen. Esimerkkikoodissa 8 **ChannelInfo**-tyyppiin on tallennettu osoitin kanavaan, koko animaation kesto, animaation paino sekä nykyinen lokaali aika. Nykyinen aika löydetään laskemalla liukulukujakojäännös animaatioiden yhteisestä nykyisestä ajasta ja animaation kestosta. Osoitin kanavaan on tallennettu **void**-tyyppisenä osoittimena, jolloin voidaan tallentaa samaan kenttään minkätyyppisiä animaatiokanavia hyvänsä.

```
// C++
struct ChannelInfo
{
      void const * channel;
      f32 duration;
      f32 weight;
      f32 time;
};

Array<Array<ChannelInfo>> translationChannelInfos;

for (s32 i = 0; i < animationCount; ++i)
{
      f32 duration    = animations[i]->duration;
      f32 time        = modulo(animator.animationTime, duration);
      f32 weight      = weights[i];

      for (auto const & channel : animations[i]->translationChannels)
      {
            translationChannelInfos[channel.targetIndex]
                  .push({&channel, duration, weight, time});
      }
}
```

Esimerkkikoodi 8. Animaatioiden lajittelu ja järjestäminen luurangon luiden mukaan. **TranslationChannelInfos** on kaksiulotteinen taulukko, jossa on jokaiselle luulle taulukko siihen vaikuttavista siirtymäanimaatiokanavista. Sama lajittelu tehdään myös kääntymisanimaatioille.

Lajitellut ja luiden mukaan järjestetyt animaatiot käydään läpi luu kerrallaan esimerkkikoodissa 9. Luuhun vaikuttavat animaatiokanavat, ne, joiden painoarvo on enemmän kuin 0 ja joiden **targetIndex** vastaa luuta, näytteistetään animaation ajan mukaan ja interpoloidaan kanavan interpolointitavan mukaan. Interpoloidut arvot lisätään painotetusti kokonaissiirtymään tai kääntymään. Samalla pidetään kirjaa luuhun vaikuttavien kanavien kokonaispainoarvoista. Kanavien läpikäynnin jälkeen luun asentoon lisätään painotetusti sen oletusasentoa, mikäli kokonaispainoarvo jää alle yhden. Täyttä painoarvoa vastaava kokonaisasento tallennetaan luun **boneSpaceTransform**-kenttään.

```
// C++
v3 totalTranslation  = {};
f32 totalAppliedWeight    = 0;

for(ChannelInfo const & channelInfo : translationChannelInfos[boneIndex])
{
      TranslationChannel const & channel
      = *reinterpret_cast<TranslationChannel const *>(channelInfo.channel);

      v3 translation = {};

      if (channel.interpolationMode == INTERPOLATION_MODE_STEP)
      {
            s32 previousKeyframeIndex
                  = previous_keyframe(channel.times, channelInfo.time);
            translation = channel.translations[previousKeyframeIndex];
      }
      else
      {
            // Muut interpolointitavat
      }

      totalAppliedWeight   += channelInfo.weight;
      totalTranslation     += translation * channelInfo.weight;
}

f32 defaultPoseWeight = 1.0f - totalAppliedWeight;
totalTranslation
      += defaultPoseWeight
      * skeleton.bones[boneIndex].boneSpaceDefaultTransform.position;

skeleton.bones[boneIndex].boneSpaceTransform.position = totalTranslation;
```

Esimerkkikoodi 9. Siirtymäanimaation näytteistys ja interpolointi yhdelle luulle kaikille siihen vaikuttaville kanaville. Tämä koodi suoritetaan silmukassa, kerran kullekin luulle.

Interpoloimalla avainkehysten välillä saadaan aikaan sulavaliikkeistä animaatiota ilman, että jokaisen kehyksen asentoa tarvitsee mallintaa erikseen. Kuvassa 4 on kahden

avainkehyksen osoittamat asennot ja niiden välistä interpoloitu välitila. Näin voidaan esimerkiksi juoksuanimaatio esittää yhteensä neljällä avainkehyksellä, joista tässä on vain kaksi.

Kuva 4. Kahden avainkehyksen välistä on interpoloitu uusi asento, jota ei ole tarvinnut animaation mallinnusohjelmassa erikseen luoda.

Interpolointi auttaa animaation tekijää siinä, että kaikkia asentoja ei tarvitse erikseen asettaa, vaan ainoastaan esimerkiksi tärkeimmät ääriasennot. Interpolointi mahdollistaa animaation toistamisen usealla eri kuvataajuudella. Lisäksi animaatiota voidaan ohjelmallisesti muokata helpommin, kun avainkehyksiä on vähemmän. Tätä voitaisiin hyödyntää esimerkiksi hahmon jalkojen asettamisessa portaalle käytettäessä tavallista kävelyanimaatiota.

4.3 Animaation interpolointi

Animaatiota kuvataan sarjana avainkehyksiä, joiden varsinaiset arvot ovat siirtymän animoinnissa kolmiulotteisia vektoreita ja kääntymisen animoinnissa neljästä komponentista koostuvia kvaternioita. Kustakin animaatiokanavasta saadaan animaation tämänhetkistä aikaa ja avainkehysten aikoja vertailemalla laskettua edellinen ja seuraava avainkehys sekä suhteellinen aika, joka kertoo, miten kaukana suhteellisesti ollaan kummastakin. Suhteellinen aika on aina nollan ja yhden välillä; nolla tarkoittaa, että ollaan

täsmälleen edellisen avainkehyksen kohdalla, ja yksi tarkoittaa, että ollaan täsmälleen seuraavan avainkehyksen kohdalla. Muut arvot kuvaavat eri välitiloja.

Kanavista luetaan edellistä ja seuraavaa avainkehystä vastaavat arvot, ja niiden välillä interpoloidaan kanavan interpolaatiotavan mukaan. Porrastettu interpolaatio tarkoittaa, että näytetään aina viimeisimmän ohitetun, toisin sanoen edellisen, avainkehyksen arvo. Lineaarisessa interpolaatiossa lasketaan avainkehysten painotettu summa kaavan 1 mukaan.

$$v = p * (1 - t) + n * t \quad (1)$$

Kaavassa *v* on interpoloitu arvo, *p* on edellisen avainkehyksen arvo, *n* on seuraavan avainkehyksen arvo ja *t* on suhteellinen aika edellisen ja seuraavan avainkehyksen aikojen välillä. Interpoloitaessa vektoreita voidaan käyttää täsmälleen tätä kaavaa, niin että *v*, *p* ja *n* ovat vektorityyppisiä muuttujia ja *t* on skalaariarvo.

Kvaternioiden interpolointi ei onnistu suoraan tällä tavoin niitä koskevien laskusääntöjen vuoksi, mutta logiikka on lopulta sama. Kvaterniot kuvaavat kolmiulotteista kääntymää eräällä tavalla neliulotteisella yksikkövektorilla. Interpolointia varten erotetaan kahden kvaternion erotuksesta kulma ja akseli ja muodostetaan lopputulos interpoloimalla erotuksen kulmaa suhteellisen ajan mukaan.

Kahden tai useamman animaation välillä interpoloidaan samalla tavalla kuin avainkehysten välissä. Interpoloinnin *t* arvona käytetään animaatioille annettuja painoarvoja. Näin voidaan sulavasti vaihtaa yhdestä animaatiosta toiseen tai sekoittaa esimerkiksi kävely ja kyyristymisanimaatiota, jolloin saadaan aikaan kyyryssä kävelevä animaatio, kuten kuvasta 5 nähdään.

Kuva 5. Kävely- ja kyyristymisanimaatioiden välillä interpolointi.

Sekoitettavia animaatioita mallinnettaessa on kuitenkin pidettävä mielessä, että eripituiset animaatiot eivät välttämättä sekoitu mielekkäällä tavalla. Niiden aikoja voitaisiin toki skaalata yhteensopivan pituisiksi, mutta tällaista ominaisuutta ei toteutettu järjestelmään. Kuvassa näkyvän kävelyanimaation pituus on tasan yksi sekunti, ja kyyristymisanimaatio on oikeastaan vain yksi kyyristymisasentoa kuvaava avainkehys; sen kesto on nolla sekuntia. Näin ollen kyyristymisanimaation kesto menee tasan kävelyanimaation kestoon ja animaatiot sekoittuvat siististi.

4.4 Animaation ohjaaminen

Animaation ohjaamisen tarpeet ja rajoitteet riippuvat suuresti varsinaisen pelin tai muun sovelluksen tarpeesta ja jopa yksittäisen animoitavalla mallilla esitettävän asian ominaisuuksista; ohjaako hahmoa pelaaja vai tekoäly, kuvaako hahmo ihmistä vai eläintä ja niin edelleen. Tämän vuoksi ei ole mielekästä, ainakaan pelimoottorin elinkaaren tässä vaiheessa, yrittää rakentaa kaiken kattavaa yleistä järjestelmää animaation ohjaamiseen. Sen sijaan animaation toistamisen funktiot tehtiin monimutkaisista rakenteista riippumattomiksi ja niitä voi käyttää monen erilaisen konkreettisen ohjaimen kanssa.

Ohjelmaan toteutettiin kuitenkin yksi ihmisenkaltaista hahmoa ohjaava ohjain. **CharacterMotor**-tyyppi eli hahmomoottori pitää kirjaa hahmon liikkeistä. Sen päivitysfunktiossa sille syötetään suuntavektori ja totuusarvot hyppäämistä tai kyyristymistä varten. Suuntavektorin ja hahmomoottorin nykyisen nopeuden ja muiden arvojen perusteella funktio laskee uuden suunnan ja nopeuden. Näiden perusteella voidaan määrätä animaatiojärjestelmän toistettavaksi sopiviksi katsotut animaatiot. Esimerkkikoodin 10 **weights**-taulukko on sama, josta aliluvussa 4.1 luettiin animaatioiden painoarvot niiden sekoitusta varten. Painoarvojen laskemisen monimutkaisuus kasvaa nopeasti sen mukaan, mitä enemmän animaatioita on valittavissa. Apufunktio **override_weight** lisää yhden animaation painoarvoa ja vähentää muiden, niin että kokonaispaino on edelleen täsmälleen 1 eli sata prosenttia.

```
if (speed < 0.00001f)
{
      weights[IDLE]   = 1 * (1 - crouchPercent);
      weights[CROUCH] = 1 * crouchPercent;
}
else if (speed < motor.walkSpeed)
{
      f32 t = speed / motor.walkSpeed;

      weights[IDLE]   = (1 - t) * (1 - crouchPercent);
      weights[CROUCH] = (1 - t) * crouchPercent;
      weights[WALK]   = t;

      f32 crouchValue = crouchPercent;
      override_weight(CROUCH, crouchValue * crouchOverridePowerForAnimation);
}
else
{
      f32 t = (speed - motor.walkSpeed) / (motor.runSpeed - motor.walkSpeed);

      weights[WALK]   = 1 - t;
      weights[RUN]    = t;

      f32 crouchValue = crouchPercent;
      override_weight(CROUCH, crouchValue * crouchOverridePowerForAnimation);
}
```

Esimerkkikoodi 10. Animaatioiden painoarvojen laskemista hahmomoottorin päivitysfunktiossa.

Hahmomoottori ei tässä toteutuksessa välitä, mistä suuntavektori ja muut syötteet tulevat; pelissä on yksi pelaajan ohjaama ja satoja ohjelman ohjaamia hahmoja, jotka käyttävät samaa **CharacterMotor**-tyyppiä ja sen samaa päivitysfunktiota.

4.5 Animaatioiden yhdistäminen maskaamalla

Animaatioita voidaan myös yhdistää toisella tavalla: maskaamalla. Kahta tai useampaa animaatiota ei sekoiteta keskenään, vaan kustakin valitaan maskin avulla tietty valikoima luita, joita animoidaan. Tällöin voidaan esimerkiksi käyttää jaloille samaa juoksuanimaatiota koko ajan ja vaihtaa käsien animaatiota sen mukaan, kantaako hahmo jotain esinettä vai ei.

Animaatiojärjestelmään ei toteutettu tällaista ominaisuutta. Se otettiin kuitenkin huomioon valintoja tehdessä, ja nykyinen järjestelmä ei tarvitse suuria muutoksia sen toteuttamiselle.

5 Luurangon ja animaation lukeminen tiedostosta

Verkkojen, luurankojen ja animaatioiden tuottamista varten on olemassa lukuisia erilaisia mallinnusohjelmia. Niitä ovat muun muassa Blender sekä Autodeskin Maya ja 3ds Max. Ohjelmissa on valtavat määrät eri työkaluja, joilla sisällön muodostavan monimutkaisen datan tuottaminen on tehty helpoksi ja jokseenkin tosimaailmaa vastaavaksi, esimerkiksi veistotyökalujen avulla (engl. *scultping*).

Ohjelmat esittävät kuitenkin omat datansa yleensä jonkinlaisessa omassa sisäisessä formaatissa, joka on joko suljettu tai niin monimutkainen, että sen lukeminen muualla ei ole järkevää. Tätä varten on olemassa joitakin yleisiä formaatteja, joiden avulla voidaan siirtää malleja mallinnusohjelmista peleihin. Tällaisia formaatteja ovat esimerkiksi OBJ, COLLADA, FBX ja glTF. Osalla on ne pois rajaavia puutteita: OBJ ei pysty tallentamaan animaatioita, ja FBX on Autodeskin suljettu formaatti, jonka käyttämiseen tarvitaan erillinen kirjasto Autodeskilta. COLLADA ja glTF ovat avoimia formaatteja, ja ne pystyvät tallentamaan animaatioita. COLLADA on kuitenkin kokonaan tekstipohjainen formaatti, kun taas glTF pystyy tallentamaan suuren osan datasta binäärimuodossa, joka on helppo ja nopea kopioida ohjelman muistiin.

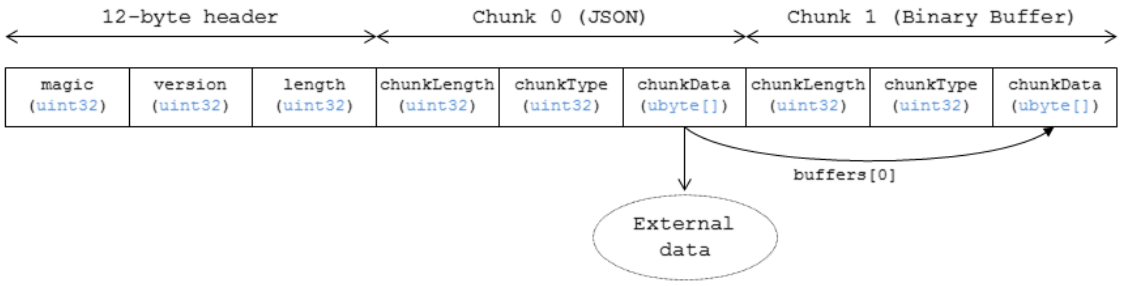
Pelimoottoriin valittiin käyttöön glTF-formaatti, ja erityisesti sen binäärimuotoinen malli glb.

5.1 glTF-tiedostoformaatti

glTF on Khronos Groupin kehittämä alustariippumaton tiedostoformaatti muun muassa 3D-allien ja animaatioiden siirtämiseen mallinnussovelluksesta pelimoottoriin [3]. Vakiomuotoinen glTF koostuu JSON (Javascript Object Notation) -muotoisesta, .gltf-päätteisestä tiedostosta, erillisestä .bin-päätteisestä datatiedostosta sekä erillisistä JPG- tai PNG-muotoisista kuvatiedostoista. Lisäksi glTF-spesifikaatio määrittää binäärimuotoisen yhdistetyn .glb-päätteisen tiedostoformaatin, jossa JSON- ja dataosuudet on tallennettu samaan tiedostoon. Pelimoottoriin toteutettiin lukija .glb-tiedostoille. Kuvien lataaminen on valmiina pelimoottorissa, eikä se kuulu animaatiojärjestelmän laajuuteen.

Binäärimuotoinen tiedosto rakentuu kolmesta osasta: otsakkeesta (engl. *header*), JSON-osuudesta eli JSON-lohkosta sekä binääriosuudesta eli binäärilohkosta.

Otsake koostuu kolmesta kokonaisluvusta, jotka kertovat tiedoston tyypin, version ja tiedoston kokonaispituuden (ks. kuva 6). Otsake sijaitsee aivan tiedoston alussa, joten luvut voidaan lukea muuntamalla tiedoston ensimmäiset tavut kokonaisluvuiksi.



Kuva 6. Binäärimuotoisen glTF-tiedoston rakenne [3].

Määritelmän mukaan JSON-lohko sijaitsee aina välittömästi otsakkeen perässä. Se alkaa kahdella kokonaisluvulla, jotka kertovat lohkon pituuden ja tyypin. Lohkon dataosuus voidaan muuttaa UTF-8-tyyppiseksi merkkijonoksi, jota voidaan jäsentää ja tulkita millä tahansa JSON-jäsentimellä. Binäärilohko alkaa niin ikään välittömästi JSON-lohkon jälkeen. Myös se alkaa informaatiolla lohkon pituudesta ja tyypistä.

JSON-formaatti on tekstimuotoinen ja ihmisen luettavissa. Se koostuu kokoelmasta avaimia ja niitä vastavia arvoja. Avaimet ovat aina merkkijonoja, ja arvot voivat olla merkkijonoja, totuusarvoja, kokonais- tai desimaalilukuja tai näistä koostuvia objekteja tai taulukoita. Objektit voivat sisältää taulukoita ja taulukot objekteja, ja näin ne voivat muodostaa vapaamuotoisen hierarkkisen rakenteen. Hierarkian uloimpana, tai ylimpänä tai alimpana, riippuu siitä, miten päin maailmaa katselee, on aina yksi nimeämätön objekti, joka sisältää kaikki muut, kuten nähdään esimerkkikoodista 11.

```
// JSON
{
    "asset": {
        "generator": "Khronos glTF Blender I/O v0.9.36",
        "version": "2.0"
    },

     // Kaikki muut tiedoston asiat
}
```

Esimerkkikoodi 11. glTF:n JSON-lohkon ensimmäinen kenttä on **asset**-niminen objekti, joka kertoo, mikä ohjelma on tallentanut tiedoston (**generator**-kenttä) ja mikä on tiedostomuodon versio.

Animaation ja erityisesti tämän pelimoottorin kannalta mielenkiintoiset kentät glTF-tiedoston JSON-lohkossa ovat seuraavat:

- Solmut (engl. nodes), joiden avulla voidaan tunnistaa tiedoston sisältämiä objekteja muun muassa nimen perusteella sekä rakentaa luurangon tarvitsema hierarkia.
- Verkot (engl. meshes), nylkemiset (engl. skins) ja animaatiot (engl. animations), joiden käyttöä ohjelmassa on kuvattu edellä. Pelimoottori käyttää nylkemisen sijaan luuranko-konseptia, mutta molemmat kuvaavat samaa asiaa.
- Ohjaimet (engl. accessors), puskurinäkymät (engl. buffer views) ja puskurit (engl. buffers), joiden avulla luodaan yhteyksiä JSON-lohkon objekteista binäärilohkon vastaaviin osuuksiin.

Muissa kentissä voidaan esittää muun muassa kameran ja materiaalien ominaisuuksia, tai sovelluskohtaista mukautettua varsinaisen glTF-määritelmän ulkopuolelle jäävää informaatiota.

5.2 glTF-tiedoston lukeminen ohjelmassa

Tiedosto luetaan aluksi kokonaisuudessaan ohjelman muistiin esimerkkikoodissa 12 esitettyyn **GltfFile**-tyyppiseen muuttujaan. Koko tiedoston binääriesitys on tallennettuna **memory**-taulukkoon. Lukijassa käytetään JSON-lohkon jäsentämiseen Tencent-yhtiön avoimen lähdekoodin RapidJson-kirjastoa [11], ja sen avulla glTF:n JSON-lohko saadaan muutettua ohjelmassa käytettävään muotoon **json**-kenttään. Binärilohkoa ei tarvitse kopioida uudestaan, vaan tiedostosta luetaan sen alkukohta ja tallennetaan **binaryChunkOffset**-kenttään.

```
struct GltfFile
{
      Array<byte>           memory;
      rapidjson::Document   json;
      u64                   binaryChunkOffset;

      byte const * binary() const { return memory.data() + binaryChunkOffset; }
};
```

Esimerkkikoodi 12. GltfFile-tyyppi kuvaa valmiiksi jäsennettyä glTF-tiedostoa, jota voidaan jakaa referenssinä sitä tarvitseville funktioille.

Kullekin tiedostolle riittää, että se luetaan kerran. Tämän jälkeen siitä voidaan eri kohdissa ohjelmaa lukea sen sisältämiä eri asioita. Esimerkiksi hahmon verkko, luuranko ja animaatiot voivat kaikki olla tallennettuna samaan tiedostoon, ja voidaan käyttää samaa kerran luettua **GltfFile**a niiden lukemiseen eri vaiheissa. Samassa tiedostossa voi myös olla useita eri verkkoja, ja niitä voidaan samaan tapaan lukea niitä toisistaan riippumatta.

Nykyisessä toteutuksessa alkuperäinen tiedostokahva avataan ja sen sisältö kopioidaan kokonaan ohjelman muistiin. Binärilohkon kopioinnin voisi kuitenkin välttää ja tallentaa **GltfFile**-rakenteeseen pelkän tiedostokahvan, josta voisi suoraan lukea tarvittavat dataosuudet. Se olisi optimointia.

Dataosuuteen käsiksi pääseminen vaatii ohjainten, puskurinäkymien ja puskurien yhdistämistä. Verkot, nylkemiset ja animaatiot viittaavat aina johonkin ohjaimeen. Ohjain puolestaan viittaa puskurinäkymään, joka viittaa lopulta itse puskuriin.

Tämä monimutkainen rakenne mahdollistaa lomitetun esitysmuodon esimerkiksi verkon pisteille ja niiden ominaisuuksille. Pelimoottorin verkkojen pisteet on esitetty nimenomaan lomitetusti, mutta tätä ominaisuutta ei hyödynnetä, sillä kehityksessä käytettävissä olevissa glTF-tiedostoissa pisteitä ei ole lomitettu.

5.3 Verkon lukeminen

Verkot esitetään tiedoston JSON-lohkossa ”meshes” (suom. *verkot*) -taulukossa. Verkolla voi olla nimi, josta sen voi tunnistaa, mutta tässä toteutuksessa verkot tunnistetaan taulukosta solmusta luetun indeksin mukaan. Verkolla on yksi tai useampia primitiivejä, jotka oikeastaan ovat verkkoja itsekin ja voivat muodostaa usean verkon ryhmän. Tällaista esitystapaa ei tueta pelimoottorissa, joten tiedostosta luetaan vain taulukon ensimmäinen primitiivi. On huomattava, että tässä kontekstissa primitiivi tarkoittaa eri asiaa kuin verkon varsinaisen piirtämisen yhteydessä; tässä se on kokonainen verkko, piirtämisen yhteydessä esimerkiksi pelkkä yksittäinen verkon kolmio.

Primitiivin ”attributes”-taulukko (suom. *ominaisuudet*) kertoo, mitä tietoa verkon pisteistä on tiedostoon tallennettu. Kunkin ominaisuuden arvo kertoo indeksin glTF-ohjaimeen, jolla löydetään ominaisuutta vastaava sijainti binäärilohkosta. Esimerkkikoodissa 13 primitiivillä on ominaisuuksina sijainti, normaalivektori, tekstuurikoordinaatit, pisteeseen vaikuttavien luiden indeksit sekä näitä vastaavat painoarvot.

```
"meshes": [
        {
            "name": "cube_head",
            "primitives": [
                {
                    "attributes": {
                        "POSITION": 0,
                        "NORMAL": 1,
                        "TEXCOORD_0": 2,
                        "JOINTS_0": 3,
                        "WEIGHTS_0": 4
                    },
                    "indices": 5
                }
            ]
        }
    ],
```

Esimerkkikoodi 13. Verkon esittäminen glTF-tiedoston JSON-lohkossa ”meshes”-taulukossa.

Primitiivin ominaisuuksia vastaavat ohjaimet löytyvät JSON-lohkosta ohjaimia kuvaavasta taulukosta. Ohjain kertoo, mistä puskurinäkymästä data löytyy ja kuinka monta kappaletta arvoja siinä on. Esimerkkikoodissa 14 käytetty funktio **get_buffer_start** hakee ohjaimen perusteella oikean puskurinäkymän ja sen kautta osoittimen oikeaan kohtaan varsinaisessa binääripuskurissa. Osoittimien avulla kopioidaan muistista pisteiden ominaisuudet.

```
// C++
s32 positionAccessor      = attributes["POSITION"].GetInt();
s32 positionCount         = accessors[positionAccessor]["count"].GetInt();
v3 const * positions      = get_buffer_start<v3>(file, positionAccessor);
// Haetaan osoittimet myös muille ominaisuuksille, kuten normaalivektoreille

s32 vertexCount = positionCount;
Array<Vertex> vertices;

for (s32 vertexIndex = 0; vertexIndex < vertexCount; ++vertexIndex)
{
     vertices[vertexIndex].position  = positions[vertexIndex];
     // Kopioidaan myös pisteiden muut ominaisuudet
}
```

Esimerkkikoodi 14. Verkon pisteiden sijaintien kopioiminen tiedostosta taulukkoon, johon pisteet tallennetaan. Samat toiminnot tehdään myös muille ominaisuuksille, kuten normaalivektoreille.

Verkolla on pisteiden ominaisuuksien tapaan myös ohjain indekseille, jotka määräväät verkon pisteiden muodostamat kolmiot ja siten myös lopulta verkon tarkan muodon. Indeksejä ei ole lomitettu tiedoston eikä pelimoottorin esityksissä, joten ne voidaan kopioida suoraan muistiin.

5.4 Luurangon lukeminen tiedostosta

Luurangon esittämiseen tarvitaan sen hierarkiaa kuvaava puurakenne sekä tieto kunkin luun sijainnista. Luuranko kuvataan glTF-tiedostossa niin, että kukin luu on oma solmunsa (engl. *node*) JSON-lohkon solmutaulukossa ja niissä on listattuna niiden omat lapsiluut. Esimerkkikoodissa 15 näytetään, kuinka luurangon muodostavat eli siihen kuuluvat luut on lueteltu ”skins”-taulukon (suom. *nylkemiset*, vastaa konseptuaalisesti luurankoa) alkioiden ”joints”-taulukossa (suom. *nivelet*, vastaa konseptuaalisesti luita).

```
// JSON
"skins": [
    {
        "inverseBindMatrices": 6,
```

```
        "joints": [ 16, 15, 8, 1, 0, 4, 3, 2, 7, 6, 5, 11, 10,
                    9, 14, 13, 12, 17, 18, 19, 20 ],
        "name": "Armature"
    }
]
```

Esimerkkikoodi 15. Luuranko esitettynä glTF-tiedostossa. glTF käyttää nylkemiskonseptia (engl. *skins*), siinä missä pelimoottori käyttää luurankoa ja niveliä (engl. *joints*) luiden kohdalla. Lopulta kyse on kuitenkin samasta informaatiosta.

Toisin kuin toteutetussa animaatiojärjestelmässä, glTF-kuvaa puurakennetta niin, että kukin luu tuntee omat lapsensa. Järjestelmässä sen sijaan valittiin tallentaa kuhunkin luuhun tieto sen vanhemmasta, koska tällöin kukin luu tallentaa täsmälleen yhden toisen luun indeksin. Luurankoa luettaessa täytyy tämä tieto hakea käymällä glTF:n solmutaulukkoa läpi, kunnes löydetään jonkin luun lapsista se luu, jonka vanhempaa etsitään.

Verkon vääntämistä varten luut tarvitsevat myös käänteiskiinnitysmatriisin (engl. *inverse bind matrix*). Ne olisi mahdollista laskea luun oletussijainnista, mutta glTF-tiedostossa ne ovat valmiiksi tallennettuna mallinnusohjelman laskemana, joten ne voidaan suoraan kopioida.

5.5 Animaation lukeminen

Animaatiot ovat glTF:ssä esitettynä samankaltaisesti kuin animaatiojärjestelmässäkin; itse asiassa glTF:n esitystapa toimi inspiraation lähteenä järjestelmää kirjoitettaessa.

Yhdessä tiedostossa voi olla useita animaatioita, ja ne tunnistetaan lukiessa nimen perusteella. Esimerkkikoodista 16 nähdään, että kussakin animaatiossa on taulukollinen kanavia (engl. *channels*) ja toinen taulukollinen näytteistimiä (engl. *samplers*). Näytteistimissä on ”input”-kenttä, joka kertoo ohjaimen animaatiokanavan avainkehysten aikoihin ja ”output”-kenttä, joka kertoo ohjaimen varsinaisiin avainkehysten arvoihin. Näytteistin myös kertoo kanavan interpolaatiotavan. Kukin kanava puolestaan viittaa yhteen näytteistimeen sekä kertoo animoitavan solmun indeksin, joka joudutaan taas hakemaan solmu-taulukon kautta, ja sen, animoiko kanava sijaintia, asentoa vai mittakaavaa.

```
// JSON
// Kanava
{
      "sampler": 0,
      "target": {
```

```
            "node": 16,
            "path": "rotation"
      }
},

// Näytteistin
{
      "input": 7,
      "interpolation": "STEP",
      "output": 8
},
```

Esimerkkikoodi 16. Animaation esitys JSON-lohkossa. Sekä kanavia että näytteistimiä varten on omat taulukkonsa kullekin animaatiolle.

Koska animoitavan luun indeksi joudutaan hakemaan vaivalloisesti solmutaulukosta, ja niin tehdessä täytyy tuntea animaatio nimen lisäksi myös luurangon nimi, päädyttiin lukijassa rajoittamaan luurankojen määrä yhteen luurankoon tiedostoa kohden. Tämä on tietysti ratkaistavissa, mutta siihen palataan tulevaisuudessa, jos sille koskaan tulee oikeaa tarvetta.

Animaatiokanavaa ohjelmassa kuvaavat **TranslationChannel**- ja **RotationChannel**-tietotyypit tallentavat avainkehysten ajan arvot samalla tavalla kuin glTF-formaatti, joten ne voidaan kopioida suoraan tiedostorakenteen **binary**-jäsenfunktion palauttamasta osoittimesta. Esimerkkikoodissa 17 käytetään pelimoottorin muistinhallintaan käytettävän **Array**-luokkamallin allokointifunktiota, johon syötetään muistinvaraajan (engl. *allocator*) lisäksi osoittimet glTF-puskuriin avainkehysten aikojen alku- ja loppukohtiin.

```
float const * timesStart  = get_buffer_start<float>(file, inputAccessor);
float const * timesEnd    = timesStart + keyframeCount;
Array<f32> times = allocate_array<float>(allocator, timesStart, timesEnd);
```

Esimerkkikoodi 17. Avainkehysten aikojen kopioiminen **times**-taulukkoon.

Kunkin animaatiokanavan input-ohjaimessa on lisäksi kerrottu sen pienin ja suurin arvo eli kanavan ensimmäisen ja viimeisen avainkehyksen ajankohta. Animaation kesto määritetään sen kaikkien kanavien kaikista aikaisimman ja kaikista myöhäisimmän ajankohdan mukaan.

6 Työn tulosten arviointi

Insinöörityössä toteutetun animaatiojärjestelmän luvussa 2.1 nimetyt tavoitteet saavutettiin onnistuneesti. Monilta osin järjestelmä jopa ylitti siihen kohdistuneet odotukset. Esimerkiksi yhden hahmon animoinnin sijaan pystytään järjestelmässä animoimaan satoja hahmoja samanaikaisesti. Järjestelmän toteuttaminen kuitenkin kasvatti sitä tietomäärää, jonka perusteella sitä voidaan arvioida, ja siinä on paljon asioita, joita voidaan tulevaisuudessa toteuttaa vielä paremmin.

Yleisesti tietokoneohjelmissa toteutettujen asioiden hyvyyttä voidaan mitata ainakin kahdella akselilla. Ensimmäinen on ohjelman ajonaikainen suorituskyky, joka johtaa suoraan parempaan asiakaskokemukseen. Toinen on ohjelman muodostavan koodin helppokäyttöisyys, joka johtaa nopeampaan ohjelmistokehitykseen eli todennäköisesti suorituskykyisempään ohjelmaan ja siitä edelleen parempaan asiakaskokemukseen.

6.1 Järjestelmän tehokkuudesta

Pelimoottorissa, johon animaatiojärjestelmä kirjoitettiin, on ajanmittaustoiminto, jolla voidaan suurpiirteisesti mitata joihinkin pelin toimintoihin kuluvia aikoja. Näiden aikojen perusteella voidaan muodosta joitain mielipiteitä onnistumisesta sekä jatkossa tutkimalla ja kokeilemalla selvittää, millaiset tietorakenteet ja -tyypit antavat parhaat suoritusajat.

Yksi järjestelmän tavoitteista oli pystyä animoimaan vähintään yhtä hahmoa. Järjestelmä saatiin kuitenkin toteutettua sellaiseksi, että sillä on helppo animoida kuinka monta hahmoa hyvänsä. Tässä avuksi on ollut sellainen ohjelmoinnin lähtökohta, että kaikkia asioita on aina monta ja lukumäärä yksi on vain erikoistapaus monesta. Taulukosta 1 nähdään joitain mitattuja aikoja, jotka kuvaavat järjestelmän kapasiteettia. Lukuja tarkastellessa on kuitenkin otettava huomioon muutama asia. Ensinnäkin pelimoottoria ei ollut järjestelmän kehityksen aikana ollut mahdollista kääntää käyttäen clang++-kääntäjän optimointiasetuksia, eli kaikki ajat mitattiin ohjelman hitaammasta testausversiosta. Toisaalta hahmoja ohjaavat tekoälyrakenteet ovat merkillepantavan yksinkertaisia, joten niihin kuluva aika ei vastaa todellista tilannetta. Lisäksi pelimoottorissa ei ole kattavaa profilointiominaisuutta, joten ei voida tietää, mihin nimenomaiseen toimenpiteeseen aikaa todellisuudessa kuluu.

Taulukko 1. Animoitavien mallien määrän vaikutus päivityssilmukkaan kuluvaan aikaan.

Mallien määrä, kpl	Kuvaan kuluva aika, ms
50	9
100	13
200	22
300	30
400	44

Videopelien yhteydessä monesti mainitaan yhdeksi tehokkuutta kuvaavaksi luvuksi kuvataajuus (engl. *frame rate*) tai kuvaan kuluva aika (engl. *frame time*). Monesti ilmoitetaan kuvataajuus, mutta se on vain kuvaan kuluvan ajan käänteisluku, joten jälkimmäistä on mielekkäämpi vertailla. Videopeleille tyypillisiä kuvataajuusodotuksia ovat 60 tai 30 kuvaa sekunnissa, eli 16 tai 33 millisekuntia kuvaa kohti. Mitatuista ajoista nähdään, että järjestelmällä voidaan sen nykyisessä tilassa animoida vähintään sataa hahmoa erittäin hyvällä kuvaan kuluvalla ajalla. Tyydyttävään kuvaan kuluvaan aikaan päästään edelleen jopa kolmellasadalla mallilla.

On keinoja, joilla järjestelmää voidaan nopeuttaa tulevaisuudessa lisää. Funktioiden ja tietotyyppien data-asettelua voidaan parantaa, mutta tämä vaatii parempaa profilointityökalua pelimoottoriin, jotta voidaan tehdä oikeita valintoja. Järjestelmä voidaan myös säikeistää varsin helposti. Tällä hetkellä se toimii yhdellä säikeellä, mutta monet sen osat ja eri vaiheet on suunniteltu niin, että ne voidaan suorittaa rinnakkaisissa silmukoissa eri säikeillä.

Järjestelmän käyttämää dataa on pyritty järjestämään mahdollisimman aikaisessa vaiheessa ohjelman suoritusta, jotta vältytään turhilta ajonaikaisilta testauksilta jonkin datan ominaisuuden perusteella. Esimerkiksi **Animation**-tietotyypissä on siirtymä- ja kääntymäanimaatiokanavat tallennettu eri taulukoihin. Kumpikin taulukko voidaan käydä läpi omassa silmukassaan ilman tarvetta ajonaikaiselle testaukselle kanavan tyypistä. Toisaalta kanavatyyppejä kuvaavissa **TranslationChannel-** (siirtymäkanava) ja **RotationChannel**-tyypeissä (kääntymäkanava) on tallennettuna interpolaatiotapa, joka testataan turhaan joka kerta, kun kanavaa näytteistetään. Tämä voidaan korjata tulevaisuudessa esimerkiksi järjestämällä taulukko animaatiota lukiessa ja tallentamalla erikseen eri interpolaatiotapojen alkuindeksit.

6.2 Ohjelmointiominaisuuksista

Järjestelmään liittyvät funktiot ja tyypit pyrittiin pitämään yksinkertaisina. Funktioiden osalta tämä tarkoittaa, että niiden kutsuissa on suosittu C++-kielen perustyyppejä, kuten osoittimia ja kokonais- tai liukulukuja monimutkaisempien rakenteiden sijaan. Tämä mahdollistaa niiden käytön helposti monessa erilaisessa kontekstissa, joissa informaatio on mahdollisesti tallennettuna erilaisilla tavoilla, samalla kuitenkaan lisäämättä turhaa yleisrasitetta. Tyyppien osalta tämä tarkoittaa, että ne pidettiin pelkkinä yksinkertaisina tietotyyppeinä (*engl. "plain old data", POD*), jotka voidaan alustaa yksinkertaisesti nollaamalla niille varattu muistialue.

Animaatioiden ohjaaminen riippuu kaikissa tapauksissa aina konkreettisesta animaation kohteesta. Tämän takia funktioiden ja tyyppien yksinkertaisuus on tärkeää, jotta myöhempänä aikana kirjoitettavalle koodille ei pakoteta turhia rajoitteita.

Nykyisellään järjestelmä ei suoraan taivu ei-luurankotyyppiselle animaatiolle, sillä animoitava rakenne **AnimatedBone** ja siihen liittyvät funktiot käyttävät sen kenttää **inverseBindMatrix**-animoidun siirtymän laskemiseen. Pelkkään transformaatioon perustuvissa animaatioissa, esimerkiksi tasohyppelypeleistä tuttujen alustojen liike, ei käytetä sitä, eikä niitä kuvattavissa tietotyypeissä ole mielekästä sitä tallentaa. Animaation päivitysfunktio **update_skeleton_animator** ei myöskään tarvitse sitä, vaan sitä käytetään ainoastaan väännetyn verkon piirtämiseen. Näin ollen kentän voisi erottaa luun rakenteesta erilliseksi taulukoksi luita ympäröivään **AnimatedSkeleton**-rakenteeseen ja päivitysfunktion muuttaa vastaanottamaan vain taulukon luiden tai muiden transformaatioita.

6.3 Tiedostoista

Tiedoston lukeminen valituista glTF-tiedostoista onnistui tyydyttävästi. Yhtäältä järjestelmä on käytännöllinen, koska tiedosto voidaan lukea kerran ohjelman muistiin ja sitä voidaan referoida aina tarvittaessa uudestaan sekä siitä voidaan ladata eri malleja tai animaatioita tarvitsematta avata itse tiedostoa aina uudelleen. Toisaalta tiedostoa kuvaavaan **GltfFile**-tyyppiin luettavan binäärilohkon kopioiminen olisi vältettävissä. Tiedostoa luettaessa voitaisiin vain avata kahva tiedostoon ja välivaiheen kopioimisen sijaan

kopioida vain tarvittava osa suoraan oikeaan paikkaan ohjelman muistissa. Lisäksi verkkojen kohdalla voitaisiin välttää vielä toinenkin kopiointi, jossa ladattu verkko kopioidaan grafiikkasuorittimen muistiin. Myös kolmannen osapuolen JSON-lohkoa tulkitseva RapidJson-kirjasto kopioi tiedoston sisällön yhteen kertaan.

Nämä parannukset vaativat kuitenkin muita muutoksia pelimoottoriin eivätkä siten kuulu varsinaisen animaatiojärjestelmän laajuuteen.

Yhden luurangon lukemiseen kuluu aikaa noin 0,2 ms ja kuuden animaation lukemiseen noin 1,8 ms. Nämä toimenpiteet tehdään kerran pelin alkaessa tai esimerkiksi uudelle alueelle siirryttäessä, eivätkä ne lisää kuvaan kuluvaa aikaa, joten lopputulos on silti varsin kohtuullinen.

7 Loppukatsaus

Animaatiojärjestelmän voidaan todeta onnistuneen; sille asetetut tavoitteet täyttyivät. Koko työnkulku 3D-verkon, luurangon ja animaation lukemisesta tiedostosta aina niiden piirtämiseen pikseleiksi näytölle asti toimii luotettavasti ja tyydyttävällä tehokkuudella.

Animaatioita voi toistaa paljon ja joustavasti, ja niitä älykkäästi sekoittamalla saadaan niistä vähällä työmäärällä paljon irti. Merkittävänä puutteena verrattuna joihinkin muihin animaatiojärjestelmiin voidaan mainita toistettavan animaation valinta kullekin luulle yksilöllisesti; tälle ei toisaalta ole pelimoottorissa ollut käyttötarkoitusta vielä.

Tiedoston lukemiseen käytettävät aliohjelmat toimivat oikein, mikä on riittävä vaatimus pelimoottorin elinkaaren vaihe huomioon ottaen. Myöhemmässä vaiheessa pelissä käytettävät mallit ja animaatiot tallennetaan kollektiivisesti johonkin tietokantaa muistuttavaan rakenteeseen, jolloin tämä osa järjestelmää tulee kirjoitettavaksi uudelleen.

Tulevaisuudessa järjestelmää päivitetään sitä mukaa, kuin pelimoottorin ja varsinaisen pelin muut osat edistyvät ja järjestelmälle tulee uusia ja konkreettisempia vaatimuksia. Järjestelmää myös testataan muun kehityksen ohessa, ja siitä varmasti löytyviä puutteita ja virheitä paikataan sitä mukaa, kuin niitä tulee esiin.

Lähteet

1 OpenGL Skeletal Animation Tutorial #1 - #4. 2017. Verkkoaineisto. Wimble, Karl (nimim. ThinMatrix). https://youtu.be/f3Cr8Yx3GGA. 17.1.2017. Katsottu 9.5.2020.

2 Khronos Vulkan Registry. 2020. Verkkoainesto. The Khronos Group. https://www.khronos.org/registry/vulkan/. Luettu 9.5.2020.

3 glTF 2.0 Specification. 2017. Verkkoaineisto. The Khronos Group. https://github.com/KhronosGroup/glTF/tree/master/specification/2.0. 9.6.2017. Luettu 9.5.2020.

4 The OpenGL Shading Language. 2017. Verkkoaineisto. Kessenich, John. https://www.khronos.org/registry/OpenGL/specs/gl/GLSLangSpec.4.50.pdf. 9.5.2017. Luettu 9.5.2020.

5 C++17. 2020. Verkkoaineisto. cppreference.com. https://en.cppreference.com/w/cpp/17. 17.2.2020. Luettu 9.5.2020.

6 Animation. Verkkoaineisto. Unity Technologies. https://docs.unity3d.com/Manual/AnimationSection.html. 7.5.2020. Luettu 9.5.2020.

7 Animation System Overview. Verkkoaineisto. Epic Games. https://docs.unrealengine.com/en-US/Engine/Animation/Overview/index.html. 2020. Luettu 9.5.2020.

8 Linietsky Juan, Manzur Ariel and the Godot community. 2020. Animations. Verkkoaineisto. https://docs.godotengine.org/en/stable/getting_started/step_by_step/animations.html. Luettu 9.5.2020.

9 Animation. Verkkoaineisto. OGRE. https://ogrecave.github.io/ogre/api/1.12/_animation.html. Luettu 9.5.2020.

10 Clang: a C language family frontend for LLVM. Verkkoaineisto. LLVM Foundation. https://clang.llvm.org/. Luettu 9.5.2020.

11 RapidJSON. 2015. Verkkoaineisto. Tencent. https://rapidjson.org/. Luettu 10.5. 2020.