3D-visualisointi

Projektityön dokumentti

Joonatan Korpela, Tietotekniikka 2022 - 100551572

Sisältö

1 Yleiskuvaus

Kurssin aikana on luotu 3D-visualisointiohjelma. Ohjelmassa käyttäjä liikuttaa pelaajaa kolmiulotteisessa maailmassa, joka koostuu ruudukossa olevista seinistä. Seinät muodostavat huoneita ja käytäviä, joita käyttäjä pystyy tarkastelemaan liikkumalla maailmassa.

Ohjelmassa ohjataan pelaajaa, joka voi liikkua ja katsoa ympärilleen kaikissa eri suunnissa. Maailmaa visualisoidaan tekemällä 3D-perspektiiviprojektio maailmassa liikkuvan pelaajan kohdalla ja renderöimällä tulos käyttäjän näytölle. Ohjelmassa yritetään luoda aavemainen ympäristö vilkkuvilla kattolampuilla ja kummittelevalla taustamusiikilla. Jotta maailmasta saisi mahdollisimman realistisen, ohjelma toteuttaa myös valaistusmoottorin kattolamppujen valon heijastumisen laskentaa varten.

Aihe on mahdollista toteuttaa sekä keskivaikealla että vaikealla vaikeustasolla. Työ on toteutettuu vaikealla vaikeustasolla, sillä toteutettujen ominaisuuksien vaatimustaso on selvästi korkeampi. Ohjelma esimerkiksi ei rajoita pelaajan liikettä tai katsesuuntaa ja sen renderöinnissä on toteutettu ylimääräisinä ominaisuuksina tekstuurimäppäys sekä valaistuksen laskenta. Toteutuksessa on myös valittu monimutkaisempi OpenGL-renderöintirajapinta, jonka tarjoama suorituskyky mahdollistaa renderöintimoottorin raskaan laskennan. Projektissa ei myöskään esiinny merkittäviä bugeja ja se on pitkälle viimeistelty.

2 Käyttöohje

2.1 Ohjelman käynnistäminen

Ohjelmaa on kehitetty IntelliJ:n kehitysympäristössä, jossa sen voi kääntää ja käynnistää lataamalla SBT:n riippuvuudet ja luomalla ajokonfiguraation *src/main/scala/main.scala* -tiedostolle. Projektin voi viedä JAR-tiedostoksi, jolloin ohjelman voi käynnistää joko tuplaklikkaamalla tiedostoa tai käyttämällä komentoa *"java -jar ohjelma.jar*". Käyttäjällä täytyy kuitenkin olla asennettua riittävän ajantasainen Javan ajoympäristö (JRE), mielellään versio 17 tai uudempi. Ohjelman viemiseen voi myös käyttää työkaluja kuten launch4j, joka paketoi javaohjelman ympäristöstä riippumattomaan ajotiedostoon.[1] Tällöin käyttäjän ei tarvitsisi pitää huolta siitä, että hänellä on käytössä oikea JRE.

2.2 Tarjolla olevat komennot

Ohjelma on samaan aikaan sekä 3D-visualisointidemo että peli, jolla voi viihdyttää itseään. Käyttäjällä on valittavissaan kaikenkaikkiaan neljä eri komentoa: pelaajan sijainnin muuttaminen, katsesuunnan muuttaminen, juokseminen sekä ohjelman sulkeminen. Pelaajan sijaintia pystyy muuttamaan käyttämällä joko nuolinäppäimiä tai w, a, s ja d -nappeja vastaavassa muodossa. Pelaajan katsesuuntaa muutetaan liikuttamalla hiirtä samaan suuntaan kuin mihin katsesuuntaa toivotaan muutettavan. Hiiren pystyy lukitsemaan ohjelmaan tai vapauttamaan ohjelmasta käyttämällä hiiren vasemmanpuoleista näppäintä. Pelaaja pystyy juoksemaan pitämällä vasemmanpuoleista shift-näppäintä pohjassa ja ohjelmasta voi positua escape-näppäimellä.

Kaikki ohjauskomennot saa ohjelman sisällä näkyviin painamalla F1-nappia.

3 Ohjelman rakenne

Ohjelma voidaan jakaa kahteen keskeiseen osioon: ensimmäinen osio vastaa maailman tilasta ja sen muuttamisesta ja toinen osio vastaa maailman tilan visualisoimisesta. Erittely tehdään jakamalla projektin koodi kahteen pakkaukseen, *logic* ja *graphics*. Koodi, mitä ei voida mielekkäästi luokitella kumpaakaan osioon, sijoitetaan joko *default*-pakkaukseen tai omaan pakkaukseensa.

3.1 Maailman logiikka

Maailman tila ja sen ylläpitämiseen liittyvä logiikka löytyy *logic*-pakkauksesta. Tämä muodostaa itsenäisen kokonaisuuden, jolla ei ole riippuvuuksia mihinkään toiseen ohjelman osaan. Tälläinen rakenne mahdollistaa *logic*-paketin tehokkan yksikkötestauksen, koska silloin testatessa ei tarvitse ottaa huomioon muita ohjelman osia.

3.1.1 Peliolennot

Maailman tila koostuu peliolentojen sekä seinien tiloista. Peliolentoja ovat kaikki maailman olennot, joilla on oma tilansa ja jotka toteuttavat jotakin logiikka. Jokainen peliolento perii *GameObject*-luokkaa, jossa määritellään kaikille peliolennoille yhteiset kentät ja metodit. Kaikilla peliolennoilla on sijainti ja tieto siitä, onko peliolento kuollut. *GameObject*-luokassa määritellään myös alaluokille määritettäväksi *tick*-metodi, jota luokan käyttäjä kutsuu päivittääkseen peliolentojen tilaa. Peliolentoja ovat muun muassa pelaaja, kuutiot, viholliset sekä valonlähteet.

3.1.2 Pelaaja

Pelaajan sijainti, katsesuunta sekä sitä muuttava koodi löytyy *GameObject*-luokkaa perivästä *Player*-luokasta. Se perii myös *logic*-pakkauksen *EventListener*-piirreluokkaa, jonka avulla se toteuttaa käyttäjän syötetteen kuuntelemiseen tarvittavan rajapinnan. Piirreluokka on luotu, jotta käyttäjän syötteen sieppaaminen on voitu ulkoistaa jollekin toiselle ohjelman osalle. Player-luokkassa on seuraavat keskeiset metodit:

- *move()* laskee pelaajan uuden sijainnin nykyisen sijainnin ja nopeuden perusteella. Se myös tarkistaa, onko uusi sijainti pelimaailman rajojen sallima ennen pelaajan sijainnin päivittämistä.
- *getNormalizedVelocity()* laskee pelaajan normalisoidun nopeuden tällä hetkellä painettujen näppäinten ja pelaajan suunnan perusteella. Se palauttaa suuntaa kuvaavan kolmiulotteisen yksikkövektorin.
- *captureInput()* sieppaa käyttäjän syötteen ja päivittää pelaajan liikenopeuden ja -suunnan sen mukaisesti.

3.1.3 Muut peliolennot

Maailmassa olevia pyöriviä kuutioita edustaa *Cube*-luokka, joka on myös *GameObject*-luokan aliluokka. Sillä on ominaisuuksia, jotka määrittelevät sen alkuasennon, koon, pyörimisen ja

kelluvan käyttäytymisen. Luokan toteuttamassa *tick*-metodissa päivitetään kuution sijaintia ja kiertymää.

Light-luokka edustaa yhtä maailman kattolamppua ja se on *GameObject*-luokan aliluokka. Sen ominaisuudet määrittävät sen sijainnin, kirkkauden ja sen, välkkyykö se. Jos valo on asetettu välkkymään, sen toteuttamassaan *tick*-metodissa päivitetään valon tilaa kutsumalla *updateFlicker*-metodia, joka muuttaa valon kirkkautta sattumanvaraisesti.

Maailmassa liikkuvia demonihahmoja edustaa *GameObject*-luokkaa perivä *Demon*-luokka. Demonin ominaisuuksia ovat sen koko, liikkumisnopeus ja suunta, joka määritetään pelaajan ja demonin sijaintien välisen kulman avulla. Sille asetetut kynnysarvot määrittävät, milloin demoni hyökkää pelaajaa kohti ja milloin se säikäyttää pelaajaa. Jos demoni on tarpeeksi lähellä pelaajaa, sen liikkumista päivitetään *move*-metodissa, joka myös tarkistaa, voiko demoni siirtyä uuteen sijaintiin. Demonin läheisyydessä toistetaan sille asetettua ääniraitaa, jonka voimakkuus päivittyy *updateSound*-metodissa demonin ja pelaajan välisen etäisyyden funktiona.

3.1.4 Seinät

Seinien tila sekä niiden hallinnointiin liittyvä koodi löytyy *Stage*-luokasta, joka lataa ja jäsentelee pelimaailman määrittävän YAML-tiedoston. Maailmatiedosto sisältää tiedon pelaajan syntymäpisteestä, maailman asettelusta sekä peliolentojen sijainneista. *Stage*-luokka määrittelee rajapinnan, jonka avulla muut luokat voivat käyttää maailmaa koskevia tietoja. Luokka käyttää sisäisesti SnakeYAML-kirjastoa karttatiedoston lukemiseen.[2]

Stage-luokassa on useita yksityisiä apumetodeja. *generateWalls*-metodi luo maailman seinät käymällä läpi maailmatiedoston ja tarkistamalla, onko tarkasteltavassa ruudussa seinää. Seinä luodaan kahden ruudun välille vain, jos viereisessä ruudussa ei ole seinää. *findObjects*()-metodi etsii maailmassa olevia peliolentoja käymällä läpi maailman sijainnit ja tarkistamalla, onko siinä annettuja tunnisteita vastaavaa objektia. Tätä apumetodia käyttävät *findDemons*, *findCubes* ja *findLights* -metodit.

Luokan keskeisiä julkisia metodeja on muun muassa *getAllAvailablePositions*, joka palauttaa listan kaikista ruuduista, joissa ei ole seinää, sekä *canBeInPosition*, joka tarkistaa voiko tietyn kokoisen neliön sijoittaa annettuun maailman sijaintiin niin, että se ei ole päällekkäin minkään seinän kanssa.

3.1.5 Maailman hallinnointi

Edellisten luokkien lisäksi on vielä *World*-luokka, joka yhdistää kaikki *logic*-pakkauksen luokat käyttämällä niiden metodeja sopivalla tavalla. *World*-luokassa tallennetaan osoittimet kaikkiin maailman peliolentoihin sekä *Stage*-olioon. Luokan *tick*-metodissa päivitetään sekä maailman tilaa että kaikkien maailman peliolentojen tilaa niiden omien *tick*-metodiensa avulla. *World*-luokan käyttäjän tulee kutsua *tick*-metodia säännöllisin väliajoin.

Teknisessä suunnitelmassa ehdotettiin, että kaikki maailmaan liittyvät metodit ja tietorakenteet voisi periaatteessa laittaa *World*-luokkaan, koska maailmalla on niin vähän tilaa. Lopullisessa ohjelmassa, tämä ei olisi kuitenkaan ollut hyvä ratkaisu, sillä maailmaan toteutettiin lopulta enemmän toiminnallisuutta, kuin mitä oli suunniteltu. Jakamalla toiminnallisuutta omiin luokkiinsa parannetaan projektin koodin luettavuutta ja laajennettavuutta.

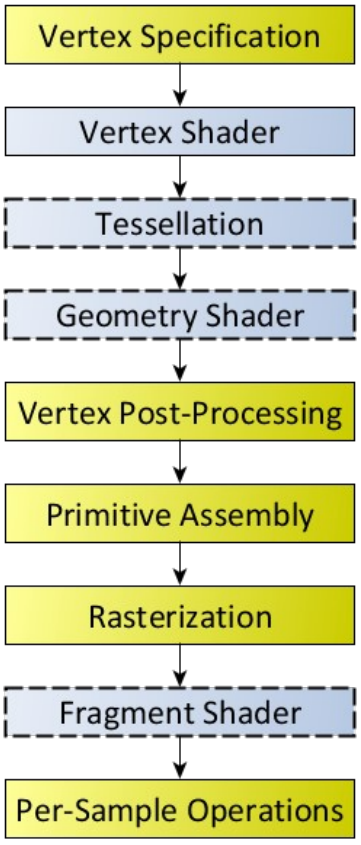
3.2 Visualisointi

Maailman visualisoimiseen liittyvä koodi löytyy *graphics*-pakkauksesta. Tällä ohjelman osalla on suora riippuvuus logic-pakkaukseen, sillä sen luokista löytyy visualisoitavan maailman tila.

3.2.1 Valitun teknologian kuvaus

Maailman visualisoinnissa käytetään LWJGL-kirjastoa (Light Weight Java Game Library) ja sen GLFW- (Graphics Library Framework) sekä OpenGL-rajapintoja (Open Graphics Library) toteuttamaan tehokas 3d-renderöintimoottori.

OpenGL:n renderöintiputki koostuu monesta osasta, jotka kootaan yhteen prosessorissa tapahtuvassa sovellusvaiheessa. Siinä toteutetaan monet renderöinnin kannalta tärkeät algoritmit ja kootaan myöhempien vaiheiden käyttämä data.[3] Sovellusvaiheessa on muun muassa konfiguroitava OpenGL:n renderöintiasetukset, luotava tarvittavat VBO:t (Vertex Buffer Object) sekä ladattava näytönohjaimelle verteksi- ja fragmenttivarjostimet. Renderöintiputken myöhempiin vaiheisiin sisältyy muun muassa verteksi- ja fragmenttiprosessointi, joiden aikana suoritetaan ladatut varjostimet.[4]



Kuva 1: OpenGL-rajapinnan grafiikkaputki[4]

3.2.2 Ikkuna

Window-luokka toteuttaa GLFW-kirjaston rajapinnan ja hallinnoi muun muassa ohjelman ikkunaa ja vastaa sen virkistämisestä. Luokalla on useita yksityisiä funktioita, kuten *initGLFW* ja *createWindow*, joita kutsutaan ikkunan luomiseksi, sekä *keyCallback*, *cursorPosCallback*, *mouseButtonCallback* ja *frameBufferSizeCallback*, jotka käsittelevät erilaisia ajon aikana ilmeneviä tapahtumia.

Window-luokka sisältää kaksi muuttuvaa kokoelmaa, *keyListeners* ja *cursorListeners*, joihin voidaan lisätä tapahtumakuuntelijoita julkisen *addEventListener*-metodin avulla. Tapahtumakuuntelijat ovat *logic*-pakkauksen *EventListener*-piiriluokan toteuttavia objekteja. Kuuntelijoita kutsutaan, kun sitä vastaava näppäin- tai kursoritapahtuma ilmenee.

Window-luokan julkisia metodeja on muun muassa swapBuffer, shouldClose, pollEvents, wasResized, getAspectRatio ja destroy. Näiden avulla voidaan virkistää näyttöä, tarkistaa, pitäisikö ikkuna sulkea, kysellä ikkunan tapahtumia, tarkistaa, onko ikkunan kokoa muutettu, saada ikkunan kuvasuhde sekä tuhota ikkuna.

3.2.3 Varjostinohjelmat ja tekstuuriobjektit

ShaderProgram-luokka edustaa yhtä OpenGL-varjostinohjelmaa. Se ottaa parametrinä verteksi- ja fragmenttivarjostimen nimet, ja se lataa varjostimien lähdekoodin nimiä vastaavista varjostintiedostoista. Luokka tarjoaa metodeja uniform-muuttujien asettamiseen ja huolehtii muun muassa varjostinohjelman kääntämisestä, linkittämisestä ja validoinnista. Luokassa on myös metodeja varjostinohjelman sitomiseen, irrottamiseen sekä sen tuhoamiseen, kun sitä ei enää tarvita.

OpenGL-tekstuuriobjektien hallinnoimista varten on määritelty *Texture*-luokka. Sillä on yksityinen metodi *loadTexture*, joka lataa annettua nimeä vastaavan kuvatiedoston */resources/textures*-hakemistosta, purkaa sen PNGDecoder-kirjaston avulla ja luo siitä tekstuuriobjektin. Tekstuuriobjektin osoitin tallennetaan *textureHandle*-muuttujaan, jonka avulla siihen voidaan viitata myöhemmin. Luokalla on myös julkiset metodit *getAspectRatio*, joka palauttaa tekstuurin kuvasuhteen, sekä *bind* että *unbind*, joka joko sitoo tai irrottaa tekstuuriobjektin OpenGL-kontekstista.

3.2.4 Apuluokka renderöinnille

OpenGL tarjoaa itsessään hyvin matalan tason rajapinnan, joka on monesti kehittäjälle vaikeakäyttöinen ja kömpelö. Renderöintirajapinnan kapselointia varten on *RenderingHelper*-luokka, joka käsittelee matalan tason OpenGL-renderöintioperaatioita.

Rakentajametodissaan luokka alustaa OpenGL-ympäristön ja luo varjostinohjelmia 2D- ja 3D-renderöintiä varten. *createQuadrilateralVertices*-metodi luo ja lataa nelikulmion verteksitiedot VBO:hon, joka on sidottu VAO:hon (Vertex Array Object). VAO:hon tallennetaan verteksiattribuuttien konfiguraatio, joka tässä tapauksessa sisältää kunkin verteksin sijainnin ja tekstuurikoordinaatit.

RenderingHelper-luokka tarjoaa neljä korkeamman tason metodia renderöintiä varten: *drawTexture3D*, *drawColor3D*, *drawTexture2D*, *drawColor2D*. Kolmiulotteiset piirtometodit renderöivät model-matriisin määrittämät verteksit käyttäen 3D-perspektiiviprojektia, mutta

kaksiulotteiset piirtometodit ei tee ollenkaan perspektiiviprojektiota. *drawColor*-metodit täyttävät piirretyn nelikulmion annetulla värilla ja *drawTexture*-metodit kuvaavat annetun tekstuurin nelikulmiolle.

Luokka tarjoaa myös rajapinnan valaistuksen hallinnoimiselle. Pistevaloja voi määrittää *setPointLights*-metodilla, joka ottaa parametrinaan listan pistevaloja, jotka koostuvat niiden sijainnista ja kirkkaudesta. Taustavalo asetetaan *setAmbientLightBrightness*-metodin avulla.

3.2.4 Maailman renderöinti

Itse maailman renderöinti tapahtuu *Renderer*-luokassa, joka käyttää edellä kuvattuja luokkia sopivalla tavalla. *Renderer*-luokan rakentaja ottaa parametrina *Game*-olion sekä *Window*-olion, joiden avulla saadaan visualisoitavan maailman sekä visualisointiin käytettävän ikkunan tiedot.

Luokassa määritellään visualisointia varten renderöintietäisyys, kameran sijainti, kameran suunta sekä matriisit seinän, lattian ja peliolennon muodoille. *Renderer*-luokka myös hallinnoi piirrettävien kappaleiden tekstuureita. Luokan tärkein metodi on *render*, joka ottaa argumenttina *GameState*-enumeraatio-olion ja renderöi maailman sen mukaisesti. Riippuen *GameState*-olion tilasta se kutsuu yhtä kolmesta metodista: *renderGame*, *renderMenu* tai *renderTransition*.

renderGame-metodi piirtää itse maailman. Se laskee kameran sijainnin ja päivittää valaistuksen, minkä jälkeen se piirtää lattian ja katon, seinät sekä peliolennot tuossa järjestyksessä. *renderMenu*-metodi piirtää valikkonäkymän ja *renderTransition*-metodi piirtää siirtymän valikon ja pelin välillä.

Rendererillä on myös useita apumetodeja. Keskeisimpiä näistä ovat *drawGameObject*, joka piirtää annetun peliolennon, *visibleWalls*, joka määrittää, mitkä seinät ovat näkyvissä pelaajan sijainnin ja katsesuunnan perusteella sekä *createModelMatrix*, joka luo matriisin peliobjektin sijainnin ja kierron perusteella.

3.2.5 Varjostimet

Vastoin kuin muu grafiikkakoodi varjostimissa käytetty glsl-koodi (OpenGL Shading Language) ei sijaitse *graphics*-pakkauksessa vaan se ladataan omista glsl-tiedostoista *src/main/glsl*-kansiosta (vrt. scala-koodi *src/main/scala*-kansiossa). Tämä mahdollistaa muun muassa IntelliJ-pluginien käyttämisen varjostimien kehityksessä ja se myös parantaa grafiikkakoodin laajennettavuutta.

Tässä projektissa on toteutettu ohjelmoitava grafiikkaputki (programmable pipeline), jolla on mahdollista toteuttaa toiminnallisuuksia, jotka eivät ole perinteisesti mahdollisia kiinteällä grafiikkaputkella (fixed pipeline). Putken varjoistinohjelmissa on käytetty melko tyypillistä pohjaa, joka suorittaa 3D-perspektiiviprojektion matriisilaskennan avulla.

Yksi näistä erityisistä toiminnallisuuksista on fragmenttikohtainen valaistus (per-fragment shading). Tämän toteutus perustuu siihen, että verteksivarjostimella lasketaan kappaleen view space -kordinaattijärjestelmässä olevat kordinaatit, joka interpoloidaan lineaarisesti fragmenttivarjostimelle. Siellä interpoloituja kordinaatteja käytetään sulavan valaistuksen laskemiseen. Taustavalon sekä 10 pelaajaa lähimmän pistevalojen tiedot annetaan varjostimille uniform-muuttujien avulla.

Verteksikohtaiseen valaistukseen (per-vertex shading) verrattuna per-fragment valaistus on merkittävästi sulavampaa mutta laskentatehon kannalta raskaampaa. Tässä projektissa valinnalla on kuitenkin enemmän väliä, koska visualisoitavissa kappaleissa on hyvin vähän verteksejä, jolloin valaistus näyttäisi selvästi karkealta. Muita ohjelmoitavan grafiikkaputken mahdollistamia toiminnallisuuksia on muun muassa näytön tärisyttäminen siihen mukautettujen uniform-muuttujien avulla sekä kappaleiden kirkkauden laskeminen sen etäisyyden funktiona.

3.2.6 Erityishuomiot toteutuksessa

Perinteisesti koko staattinen maailma olisi tallennettu yhteen VBO:hon, jossa jokaisen verteksin normaalivektorit olisi mukana verteksidatassa. VBO-data olisi voitu melko helposti generoida karttatiedoston perusteella, sillä se sisältää kaiken tiedon, mitä VBO-dataan tarvitaan. Ohjelmassa luodaan kuitenkin vain yksi VBO, joka sisältää yhden seinän neljä verteksiä ja niiden tekstuurikordinaatit. Sen normaalivektori annetaan siis varjostinohjelmille uniform-muuttujan avulla, mikä on tässä tapauksessa mahdollista, sillä ohjelmassa piirretään vain tasaisia pintoja. Tämä on ratkaisuna merkittävästi hitaampi kuin yhden VBO:n ratkaisu, sillä maailman piirtämiseen tarvitaan useampi järjestelmää kuormittava draw-kutsu. Valitulla ratkaisulla saadaan kuitenkin suorituskyvyn kustannuksella määritettyä kehittäjälle helpompi renderöintirajapinta, jossa ei tarvitse huomioida näitä optimointikysymyksiä.

Maailman visualisointikoodia voidaan myös ajaa niin usein kuin mahdollista tavallisessa *while*-silmukassa ilman, että maailmaa joudutaan renderöimään ylimääräisiä kertoja. Tämä voidaan tehdä käyttämällä GLFW:n *glfwSwapInterval*-motida, jolla saadaan *glfwSwapBuffers*-metodi odottamaan näytön virkistystaajuutta vastaavan ajan ennen kuin se päivittää ruudun.[5]

3.3 Äänimoottori

Kaikki peliäänien toistamiseen liittyvä koodi sijaitsee *audio*-pakkauksessa, jossa on kaksi luokkaa: *AudioPlayer* ja *Sound*. Pakkauksen luokat tarjoavat helppokäyttöisen rajapinnan äänien toisamiseen.

Sound on enumeraatioluokka, jonka arvot edustavat yksittäistä ääniraitaa. Enumeraatioluokassa on yksityinen metodi *loadAudio*, joka lataa äänitiedoston enumeraatioarvolle annetun nimen perusteella. Tämä metodi käyttää javan Sound API:ta äänitiedoston lukemiseen ja palauttaa *Clip*-olion, jota voidaan käyttää äänen toistamiseen. *Sound*-enumeraatio määrittelee myös useita suojattuja metodeja (protected method) ääniraidan toistamiseen, pysäyttämiseen, silmukointiin ja äänenvoimakkuuden säätämiseen.

AudioPlayer on staattinen luokka (Scalassa object-avainsanalla määritelty luokka), joka tarjoaa korkeamman tason rajapinnan *Sound*-enumeraatioluokassa määriteltyjen ääniraitojen toistamiseen ja hallintaan. Sen metodit kapseloivat *Sound*-luokan metodit siepaten mahdolliset poikkeukset, jotka ilmenevät niitä kutsuttaessa. *AudioPlayer*-luokalla on myös *playRandom*-metodi, joka ottaa vastaan luettelon *Sound*-olioita ja valitsee niistä satunnaisen äänen soitettavaksi.

3.4 Pääsilmukka

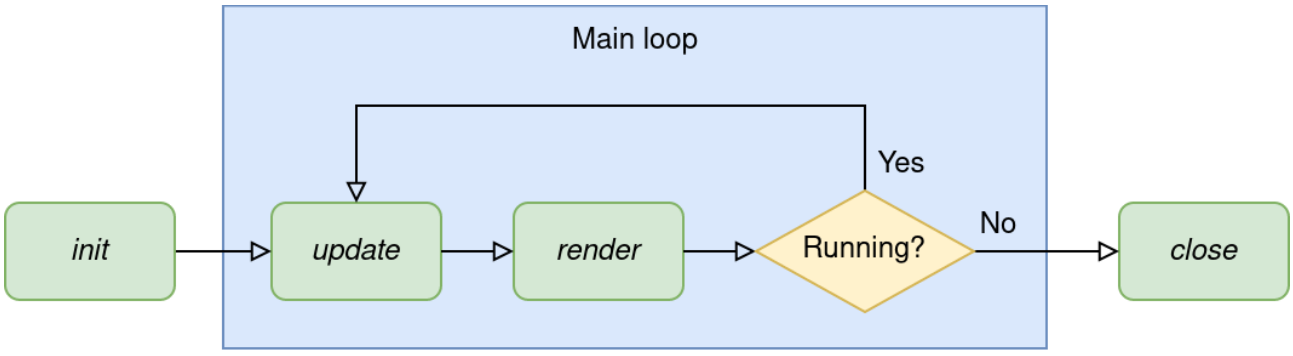
Koska *logic*- ja *graphics*-pakkausten riippuvuudet menevät vain yhteen suntaan, näitä ohjelman eri osia voidaan ajaa rinnakkain kahdella säikeellä. Ajamalla ohjelman osia rinnakkain voidaan parantaa ohjelman suorituskykyä, mikä on erityisen tärkeää tässä projektissa. Rinnakkaisohjelmoinnissa on kuitenkin otettava huomioon yhteisen muistin hallitsemiseen liittyvät haasteet.

3.4.1 Moottori

Jotta ohjelma käyttäytyy odotetusti, *World*-luokan *tick*-funktiota on kutsuttava säännöllisin väliajoin. Tämä voitaisiin toteuttaa esimerkiksi *java.util*-pakkauksen *Timer*-luokalla, mutta sen tarjoama rajapinta on kuitenkin hiukan puutteellinen. Maailman tilaa halutaan lähtökohtaisesti päivittää 60 kertaa sekunnissa, sillä se vastaa monien modernien näyttöjen virkistystaajuutta. *Timer*-luokalle ei kuitenkaan pysty riittävän tarkasti määrittämään tätä väliaikaa, sillä se ottaa kutsujen välisen ajan parametrikseen vain millisekunnin tarkkuudella.

Tämän ongelman ratkaisemiseksi toteutetaan oma *Engine*-luokka, joka kutsuu sille annettuja callback-funktiota säännöllisin väliajoin. Rajapinta tarjoaa myös mahdollisuuden kutsua funktiota niin usein kuin mahdollista. Ajan mittaaminen toteutetaan busy-waitingin avulla käyttäen *System*-luokan *nanoTime*-metodia. *Engine*-luokka pitää myös kirjaa siitä, kuinka monta kertaa mitäkin callback-funktiota kutsuttiin yhden sekunnin aikana. *Engine*-luokka ja sen tarjoama rajapinta sijaitsee *engine*-pakkauksessa.

Engine-luokan rajapinnan määrittää *GameInterface*-piirreluokka, jolla on neljä metodia: *init*, *update*, *render* ja *close*. *init*-metodia kutsutaan kerran ohjelman elinkaaren alussa ja se antaa parametrinaan *EngineInterface*-olion, jonka avulla rajapinnan toteuttava luokka voi hallinnoida moottoria. *update*- ja *render*-metodia kutsutaan jatkuvasti ohjelman elinkaaren aikana. Niiden kutsumistaajuus on määritelty *Engine*-luokkaa luotaessa. Ohjelman elinkaaren lopussa kutsutaan kerran *close*-metodia, ennen kuin ohjelman suoritus pysähtyy.



Kuva 2: Ohjelman elinkaarta kuvaava vuokaavio. Yksinkertaistamisen vuoksi kaaviossa on esitetty, että update- ja render-metodeja kutsutaan peräkkäin, mutta todellisuudessa niitä kutsutaan rinnakkain ja mahdollisesti jopa eri väliajoin.

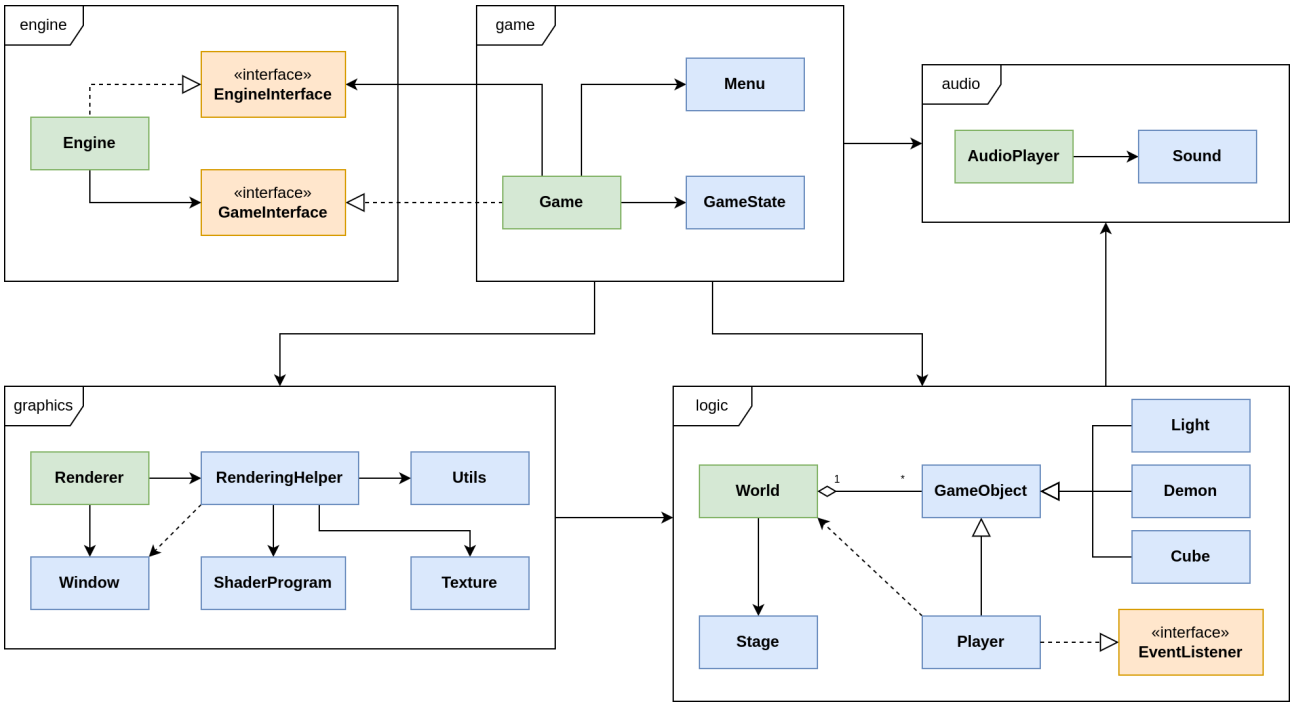
3.4.2 Game-luokka

GameInterface-piirreluokan toteuttaa *game*-pakkauksessa sijaitseva *Game*-luokka. Se hallinnoi ohjelman kaikista korkeimman tason tilaa ja logiikkaa ja toimii käytännössä koko ohjelman liimana. Luokka käyttää muun muassa *Window*-luokkaa, *World*-luokkaa, *Renderer*-luokkaa ja *Menu*-luokkaa ja luo niistä olioita. *GameInterface*-piirreluokan määrittämien metodien lisäksi, *Game*-luokalla on myös metodeja ohjelman siirtymien ja ajastimien käsittelyyn.

Pelin tilaa hallitaan *GameState*-enumeraation avulla, jolla voi olla kolme eri arvoa: Menu, Game ja MenuToGame. Nämä edustavat pelin eri tiloja, kuten päävalikossa olemista tai siirtymistä valikosta peliin. Valikon tilaa hallinnoi *Menu*-luokka, joka myös toteuttaa *logic*-pakkauksen *KeyListener*-piirreluokan. Sen tehtävänä on muun muassa aloittaa siirtymä valikosta peliin ja näyttää kätyttäjälle ohjeteksti.

3.4.3 Ohjelman käynnistyminen

Ohjelman main-funktio sijaitsee *default*-pakkauksessa main.scala-tiedostossa. main-funktiossa luodaan esiintymä *Game*- ja *Engine*-luokista, minkä jälkeen ohjelman moottori käynnistetään *Engine*-luokan *start*-metodin avulla.



Kuva 3: Luokkien ja pakkauksien välisiä suhteita kuvaava kaavio. Sininen väri kuvaa tavallista luokkaa tai enumeraatiota, orannsi väri kuvaa piirreluokkaa ja vihreä väri kuvaa pakkauksen tärkeintä luokkaa. Kaavioon piirrettyjä suhteita on merkittävästi yksinkertaistettu selkeyden parantamiseksi.

3.5 Vaihtoehtoisia rakenteita

Teknisessä suunnitelmassa ehdotettiin, että koska maailmassa ei ole mitään itsestään liikkuvia osia, ohjelman voisi toteuttaa kokonaan tapahtumapohjaiseksi. Tälläisen ohjelman tapahtumasilmukassa päivitettäisiin ja visualisoitaisiin maailman tilaa vain silloin, kun käyttäjä on antanut ohjelmalle jotakin syötettä. Tapahtumasilmukan etu olisi se, että maailmaa ei jouduta päivittelemään ja visualisoimaan turhaan, jolloin säästettäisiin järjestelmän resursseja. Rakenne ei olisi kuitenkaan tässä projektissa toiminut, sillä maailmaan lopulta toteutettiin itsestään päivittyviä elementtejä. Jos ohjelma olisi ollut täysin tapahtumapohjainen, muun muassa valot ja pyörivät kuutiot eivät olisi pystyneet päivittämään itseään.

Ohjelman renderöinnissä voitaisiin myös käyttää jotakin Javaan tai Scalaan sisäänrakennettua grafiikkakirjastoa, kuten Swingiä tai JavaFX:ää. Näiden ongelma on kuitenkin se, että iso osa renderöinnin laskennasta tapahtuu prosessorin puolella, mikä merkittävästi heikentää ohjelman suorituskykyä. Vaikka suorituskyky riittäisikin yksinkertaiseen 3D-renderöintiin, se rajoittaa ohjelman laajennettavuutta, koska suorituskyky ei riitä monimutkaisempien renderöintitehtävien suorittamiseen. Näitä kirjastoja ei myöskään ole suunniteltu 3D-renderöintiin, jolloin niiden tarjoama rajapinta ei aina ole siihen sopiva.

On olemassa monia 3D-pelimoottoreita, jotka sopisivat tähän projektiin erinomaisesti. Projektin vaatimuksena on kuitenkin se, että ohjelmointikielenä käytetään Scalaa ja että 3D-projektio on tehtävä itse. Siksi näitä kirjastoja ei ole harkittu.

4 Algoritmit

4.1 Renderöinti

4.1.1 Renderöintimenetelmät

Renderöintimenetelmiä on lähtökohtaisesti kolmea erilaisia: rasterointi (rasterization), säteensuuntaus (ray-casting) ja säteenseuranta (ray-tracing).[6] Tässä projektissa pääsääntöiseksi renderöintimenetelmäksi on valittu rasterointi, sillä se on laskennallisesti merkittävästi nopeampaa kuin muut vaihtoehdot.[7] Rasterointi perustuu siihen, että geometrisesti laksetaan miten jokin kolmiulotteinen kappale kuvautuu kaksiulotteiselle näytölle.

Toinen mahdollinen renderöintitekniikka olisi ollut säteensuuntaus. Se perustuu siihen, että virtuaalisesta kamerasta lähetetään säteitä kolmiulotteiseen maailmaan. Jokaiselle säteelle tunnistetaan, milloin se kohtaa jonkin esteen, ja sen perusteella määritetään sitä sädettä vastaavan pikselin väri. Tässä projektissa vaatimuksena oli piirtää vain seiniä, jolloin olisi riittänyt lähettää säteitä vain yhdellä tasolla ja määrittää sekä seinän korkeus että väri säteen kulkeman matkan perusteella. Tällä algoritmilla on kuitenkin se merkittävä rajoite, että se ei suoraan salli katsesuunnan muuttamista ylös tai alas.[8]

4.1.2 3D-perspektiiviprojektio

Maailmaa visualisoidaan tekemällä 3D-perspektiiviprojektio käyttäjän ohjaaman pelaajan kohdalla. 3D-projektio perustuu siihen, että voidaan matriisialgebran keinoin rakentaa MVP-matriisi (Model, View, Projection), joka kuvaa jokaisen pisteen kolmiulotteisessa avaruudessa johonkin käyttäjän kaksiulotteiselle näytölle. Tämä matriisi voidaan ladata näytönohjaimen muistiin varjostimille, jossa 3D-projektio suoritetaan laskemalla yksi matriisin kertolasku jokaiselle maailman verteksille v.

$$v_{final} = MVP \times v_{original}$$

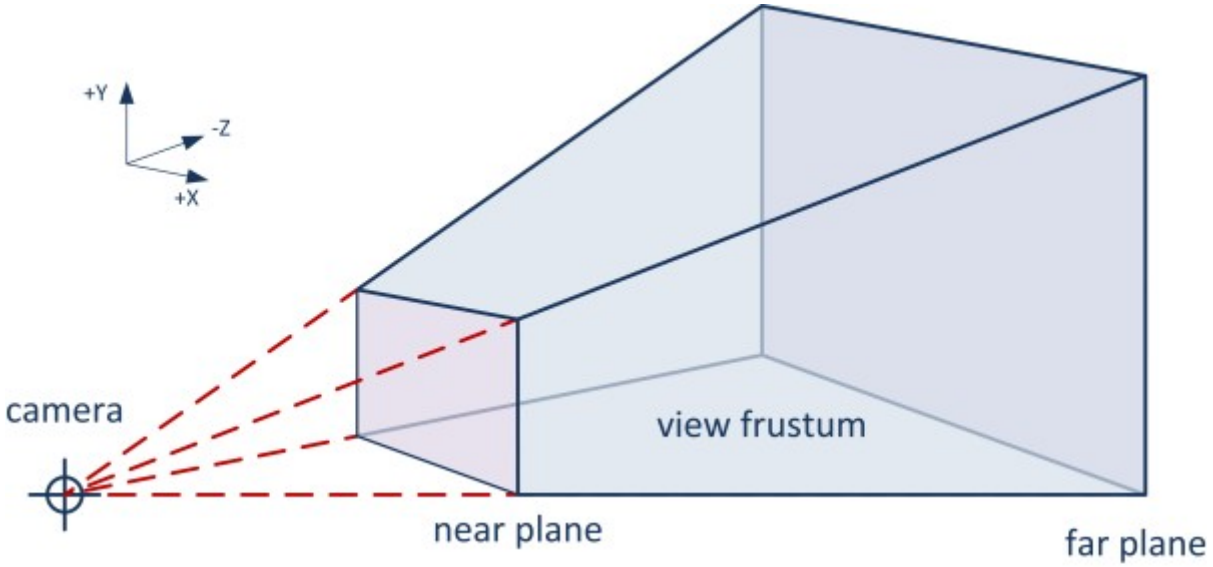
$$MVP = P \times V \times M$$

MVP-matriisi koostuu kolmesta osasta, joista ensimmäinen on model-matriisi. Se määrittää jonkin kappaleen asennon, koon ja sijainnin maailmassa, ja jokaisella maailman kappaleella on oma model-matriisinsa. Se muodostetaan ensin skaalaamalla kappaletta skaalausmatriisilla, sitten pyörittämällä kappaletta rotaatiomatriisilla ja lopuksi siirtämällä kappaletta translaatiomatriisilla.

$$M = T_{Translate} \times R_{Rotate} \times S_{Scale}$$

View-matriisi määrittää kameran sijainnin maailmassa. Jos kamera on esimerkiksi jossakin maailman pisteessä P ja se osoittaa vektorin v suuntaan, view-matriisi kuvaa kaikki maailman pisteet rotaation ja translaation avulla niin, että P kuvautuu origoon ja v kuvautuu osoittamaaan negatiivisen z-akselin suuntaan. Kaikki maailman kappaleet ikään kuin siirretään kameran kanssa origoon.

Lopuksi on projection-matriisi, joka kuvaa maailman 3D-pisteet 2D-tasolle. On olemassa monia erilaisia 3D-projektioita ja näille löytyy erilaisia matemaattisia matriisiesityksiä. Erilaiset 3D-projektiot kuvataan Wikipedian artikkelissa "3D projection".[9] Tässä projektissa käytetään perspektiiviprojektiota (vrt. ortograafinen projektio), koska ne ovat monissa tätä projektia vastaavissa peleissä yleisesti käytettyjä. Perspektiiviprojektio perustuu siihen, että näytölle kuvataan ne pisteet, jotka sijoittuvat jonkin katkaistun kartion (engl. frustum) sisälle.[10]



Kuva 4: Visualisaatio perspektiiviprojektiosta ja siinä käytettävästä frustumista.[10]

4.1.3 Valaistuksen laskeminen

Valaistukseen kuuluu useita komponentteja, jotka vaikuttavat renderöitävän kuvan lopulliseen ulkonäköön. Ympäristövalo (ambient light) on kaikista suunnista tulevaa valoa, joka simuloi epäsuoraa valaistusta eli valoa, joka on hajonnut ja heijastunut useita kertoja. Se on tasaista ja jatkuvaa valoa, jolla ei ole mitään tiettyä suuntaa, ja se antaa kuvalle perusvalaistuksen, vaikka suoraa valonlähdettä ei olisikaan. Se voidaan laskea kertomalla jokaisen pikselin väri C jollakin valon intensiteettiarvolla I.

$$C_{ambient} = C \cdot I_{ambient}$$

Hajavalo (diffuse light) on valoa, joka siroaa ja heijastuu kaikkiin suuntiin pinnasta, jota valaisee jokin suora valonlähde. Pinnan hajavalon voimakkuuden määrää pinnan normaalivektorin ja saapuvan valon suuntavektorin välinen kulma sekä pinnan heijastusominaisuudet. Se voidaan laskea kertomalla jokaisen pikselin väri C valonlähteen voimakkuudella I sekä normaalivektorin n ja suuntavektorin v pistetulolla. Jotta heijastus näkyisi vain pinnan toisella puolella, hajavalo on 0, jos pistetulo on negatiivinen.

$$C_{diffuse} = C \cdot I_{source} \cdot max\left(0 ; \hat{n} \cdot \hat{v}\right)$$

Peilivalo (specular light) on suorasta valonlähteestä aiheutuva korostus tai heijastus, joka saa aikaan kiiltävän pinnan vaikutelman. Peilivalo muistuttaa hyvin paljon hajavaloa, mutta vastoin kuin

hajavalo, peilivalo ei heijastu kaikkiin suuntiin. Se voidaan laskea samalla tavalla kuin hajavalo mutta korottamalla suuntavektorien pistetulo johonkin potenssiin n, joka kuvaa materiaalin heijastusominaisuutta.

$$C_{specular} = C \cdot I_{source} \cdot max\left(0; \hat{n} \cdot \hat{v}\right)^n$$

Pikselin lopullinen väriarvo saadaan laskemalla ympäristövalo, hajavalo ja peilivalo yhteen.

$$C_{final} = C_{ambient} + C_{diffuse} + C_{specular}$$

Lopullisessa ratkaisussa halutaan häivyttää kauempana oleiva pintoja, jotta voidaan rajoittaa käyttäjän katsekantamaa. Tämä saavutetaan kertomalla vielä lopullinen väri C jollakin mielivaltaisella pinnan etäisyydestä riippuvalla laskevalla funktiolla.

4.2 Seinät

4.2.1 Seinien määrittäminen

Karttatiedosto ladataan käyttäen SnakeYAML-kirjastoa ja maailman asettelua kuvaava merkkijono luetaan kirjaston luoman olion avulla. Merkkijono voidaan pilkkoa taulukoksi välilyöntien ja rivinvaihtojen kohdalla käyttämällä stringin *split*-metodia. Tavoitteena on tämän taulukon perusteella määrittää lista kaikista maailmassa olevista pystysyyntaisista ja vaakasuuntaisista seinistä. Tähän käytetään seuraavaa algoritmia:

1. Luo ja alusta korkeus + 1 kokoinen lista pystysuuntaisille seinille ja leveys + 1 kokoinen lista vaakasuuntaisille seinille.
2. Tarkista kunkin taulukon sijainnin (x, y) kohdalla, onko kyseisessä paikassa seinälaattaa.
3. Jos paikassa (x, y) on seinä, tarkista viereiset paikat (x+1, y), (x-1, y), (x, y+1) ja (x, y-1). Jokaisessa suunnassa, missä ei ole seinälaattaa, lisätään sitä sijaintia vastaava seinä joko pystysuuntaisten tai vaakasuuntaisten seinien listaan.
4. Toista vaiheet 2 ja 3 kaikille taulukon sijainneille.

4.2.2 Törmäyksen tunnistaminen

Pelaajan liikkuessa on aina tarkistettava, ettei se kulje seinän läpi. Tarkistus tehdään jokaisessa pelaajan kulkemassa ulottuvuudessa erikseen, jotta pelaajan liike ei kokonaan pysähtyisi sen törmättyä esteeseen. Törmäys tietyssä ulottuvuudessa voidaan tunnistaa vertaamalla pelaajan alkuperäistä sijaintia sen tulevaan sijaintiin. Mikäli pelaaja on siirtymässä kohtaan, jonka etäisyys lähimpään seinään on alle jonkin kynnysarvon, pelaaja on törmännyt seinään eikä tätä liikuteta kyseisessä ulottuvuudessa. Jos pelaajaa on riittävän kaukana lähimmästä seinästä, sitä siirretään tavallisesti. Tarkistus tarvitsee tehdä vain pelaajan lähellä oleville seinille.

5 Tietorakenteet

5.1 Matriisit ja vektorit

Koska ohjelmassa sovelletaan paljon matriisialgebraa, tarvitaan omat tietorakenteet vektorien ja matriisien kuvaamiseen. Tätä varten on valittu LWJGL mukana tuleva JOML-kirjasto (Java OpenGL Math Library). Se tarjoaa valmiit tietorakenteet erikokoisille ja erityyppisille vektoreille ja matriiseille. Nämä tietorakenteet tukevat lähes kaikkia matriisialgebran perusoperaatioita, kuten skalaarituloa ja matriisin kertolaskua. Kirjasto tarjoaa myös valmiit apufunktiot matriisien affiinikuvauksille ja erilaisille 3D-projektioille.

Tämä kirjasto on valittu ensisijaisesti siksi, että se on sisäänrakennettu LWJGL:n kanssa. Kirjastolle löytyy myös paljon dokumentaatiota OpenGL:n kanssa integroimiseen, mikä helpottaa kehitystä. Valmiiden kirjastojen tietorakenteet eivät aina ole yhtä joustavia kuin itsetehtyjen, mutta ne ovat hyvin testattuja ja yleisesti käytettyjä. Omien tietorakenteiden tekeminen olisi työlästä ja tarpeetonta.

5.2 Seinät

Seinät tallennetaan *Stage*-luokassa kahdessa kaksiulotteisessa listassa *Wall*-olioita. *Wall*-luokka tallentaa seinän sijainnin ja suunnan. Maailman ollessa n kertaa m kokoinen, lista on kooltaan n+1 kertaa m+1, koska sekä pysty- että vaakasuunnassa on seinille ruutujen määrä + 1 mahdollista paikkaa. Ensimmäinen listoista on pystysuuntaisille seinille ja toinen listoista puolestaan vaakasuuntaisille seinille. Jos jossakin kohdassa (x, y) ei ole seinää, sitä vastaavassa listan kohdassa ei ole arvoa.

Seinien tallentamiselle löytyy paljon vaihtoehtoisia tietorakenteita. Seinät voitaisiin tallentaa yksittäisenä listana *Wall*-olioita. Tämä voisi tietyissä tilanteissa viedä vähemmän muistia ja se myös helpommin mahdollistaisi esimerkiksi sen, että seinät voivat olla mielivaltaisessa asennossa. Tällainen ratkaisu tekisi kuitenkin törmäyksen tunnistamisesta monissa tapauksissa O(n)-pituisen operaation, kun taas edellä mainittu ratkaisu on O(1). Mikäli tulevaisuudessa halutaan esimerkiksi automaattisesti generoida uutta maailmaa, seinien lukumäärä voi kasvaa suureksi.

Seinät voisi myös tallentaa ruudukkona boolean-arvoja. Tämä olisi varmasti kaikista helpoin ratkaisu, sillä se on tietorakenteena hyvin yksinkertainen ja törmäyksentunnistukseen tarvittava algoritmikin olisi helppo. Se ei kuitenkaan sallisi sitä, että kahden vierekkäisen ruudun välissä olisi seinää, sillä tällä tietorakenteella seinä varaisi koko ruudun. Nykyisessä ohjelmassa karttatiedoston avulla ei ole mahdollista luoda tälläisiä ohuita seiniä, mutta se on kuitenkin ohjelman toteutuksen kannalta mahdollista valitun tietorakenteen ansiosta.

5.3 Pelaajan sijainti ja katsesuunta

Pelaajan sijainti tallennetaan *GameObject*-luokasta perityssä kentässä ja sen katsesuunta tallennetaan *Player*-luokassa. Pelaajan sijainti tallennetaan kolmiulotteisessa *float*-vektorissa, ja

pelaajan katsesuunta kaksiulotteisessä *float*-vektorissa. Pelaajan sijaintivektori voi saada mitä tahansa arvoja, mutta katsesuunta halutaan pitää jonkin rajan sisällä. Katsesuunnan vaakasuuntainen kulma voi saada arvoja 0 ja 2 pii välillä ja pystysuuntainen kulma voi saada arvoja -pii/2 ja pii/2 välillä.

Ohjelman suorituskyvyn kannalta ei ole merkittävää väliä käytetäänkö 64-bittistä *double*-tyyppiä vai 32-bittistä *float*-tyyppiä. Jälkimmäinen on kuitenkin monissa tapauksissa valittu, koska se on suoraan yhteensopiva OpenGL-rajapinnan kanssa. Ohjelmassa halutaan käyttää likulukuja, sillä diskreetit arvot eivät mahdollista pelaajan sulavaa liikkumista.

6 Tiedostot

6.1 Karttatiedosto

Karttatiedostossa käytetään YAML-formaattia. Formaattina voisi myös käyttää esimerkiksi JSONia, mutta YAML on tässä tapauksessa parempi, koska sillä pystyy esittämään monirivisiä merkkijonoja paremmin. Ohjelmaa varten ei tehdä omaa tiedostoformaattia, koska sille ei ole tarvetta.

Ohjelman kartta tallennetaan *resources/world.yml*-tiedostossa. Tiedostosta luetaan kaksi kenttää: *spawn*, joka tallentaa pelaajan syntymäsijainnin, sekä *stage*, joka tallentaa maailman kartan. Syntymäsijaintia varten määritellään kaksi kenttää *spawn*-kentän alle: *x* ja *y*, joista molemmat ovat kokonaislukuja. Kartta tallennetaan merkkijonona, jonka merkit kuvaavat sen sijainnissa olevaa laattaa. Merkit erotetaan toisistaan välilyönneillä ja rivinvaihdoilla, ja jokaisella merkkijonon rivillä on oltava yhtä monta laattaa. Ylimääräiset tyhjät merkit sivuutetaan. Formaatti sallii alla olevat merkit.

Merkki	Sen kuvaama laatta
X	Seinä
L	Lamppu
l	Vilkkuva lamppu
c	Pyörivä kuutio
d	Vihollinen
Kaikki muut	Tyhjä

6.2 Muut ohjelman tiedostot

Kaikki ohjelman ajon aikana ladattavat tekstuurit ovat png-binääritiedostoja ja äänitiedostot ovat wav-binääritiedostoja. Varjostimet ladataan glsl-tekstitedostoista.

7 Testaus

Projektin testaus koostuu automaattisesta yksikkötestauksesta sekä manuaalisesta järjestelmätestauksesta. Testaus eteni ohjelman toteutuksen aikana hyvin ja ohjelma läpäisee kaikki testit.

7.1 Yksikkötestaus

Yksikkötestaukseen käytetään ScalaTest-kirjastoa. Se on yksi suosituimmista Scalan testauskirjastoista, ja sille löytyy kattavaa dokumentaatiota ja mallikoodia. Se on minulle myös ennaltaan tuttu, sillä sitä käytettiin muun muassa Aalto-yliopiston Tietorakenteet ja algoritmit -kurssilla tehtävien testaamisessa.

Testaus kohdistuu ensisijaisesti *logic*-pakkaukseen ja sen luokkiin, sillä ne ovat ohjelman toimivuuden kannalta keskeisimpiä. Tärkeimpiä testitapauksia on muun muassa se, että pelaaja reagoi näppäinten painalluksiin ja että pelaaja ei esimerkiksi pysty kulkemaan esteiden läpi. Testejä voidaan kirjoittaa myös muille ohjelman osille, mikäli sille tulee tarvetta.

Jotta voidaan varmistaa, että projektin *main*-branchille ei päädy rikkinäistä koodia, yksikkötestit ajetaan automaattisesti projektin CI/CD-putkessa. Putki konfiguroidaan YAMLin avulla *.gitlab-ci.yml*-tiedostossa. Koodia pyritään pitämään siistinä Scalafmt-formatterin avulla.

7.2 Järjestelmätestaus

Järjestelmätestauksen tavoitteena on varmistaa, että ohjelma toimii kokonaisuudessaan odotusten mukaisesti. Testaus tehdään manuaalisesti ja siinä keskitytään erityisesti käyttöliittymän ja 3D-renderöinnin toimivuuteen. Nämä osat ovat luonteeltaan sellaisia, että ne voivat muuttua ohjelman kehityksen aikana, jolloin niille ei ole mielekästä kirjoittaa yksikkötestejä.

8 Analyysia ohjelmasta

8.1 Ohjelman tunnetut puutteet ja viat

Ohjelmassa ei ole mitään merkittäviä puutteita tai vikoja, mutta seuraavat asiat kannattaa ottaa huomioon.

- Ohjelma ei tunnista, milloin jonkin pisteen ja valonlähteen välissä on este. Tämän seurauksena kattolamput valaisevat myös seinien takana olevia pintoja, mikä ei vastaa todellisuutta. Tämän puutteen takia maailma pitää suunnitella niin, että mikään valonlähde ei ole liian lähellä mitään kulmaa, jonka takaa valo voisi heijastua.
- LWJGL ei heitä automaattisesti poikkeusta OpenGL:n virhetilanteissa. Ne tunnistetaan käyttämällä *glGetError*-funktiota ja tarkastelemalla sen paluuarvoa. Virheiden tunnistamiseksi on tehty *glCheck*-niminen apufunktio, joka suorittaa parametrina annetun funktion ja varmistaa, ettei se aiheuttanut virhettä OpenGL:n sisällä. Tätä apufunktiota joutuu kuitenkin käyttämään melkein koko ajan renderöintikoodissa, mikä tekee siitä vähemmän ylläpidettävää. Yksi mahdollinen ratkaisu olisi tehdä jonkinlainen esikääntäjä, joka tunnistaa OpenGL:n funktiot ja automaattisesti lisää virheentarkistuksen jokaiseen kohtaan.
- Ohjelma voi olla monessa eri tilassa (valikossa, pelissä jne.). Näiden hallinnointiin on käytetty vähän kömpelöä ratkaisua, joka ei välttämättä tue projektin laajennettavuutta.
- Pelin ääniä soittavaa koodia kutsutaan kaiken muun toiminnallisuuden seassa. Tämä ei ole välttämättä kovin laajennettava ratkaisu, sillä se ei noudata Separation of concerns -periaatetta. Eräs mahdollisesti parempi ratkaisu olisi, että kaikki äänien soittamiseen liittyvä toiminnallisuus toteutettaisiin omassa kokonaisuudessaan, mutta tämän toteuttaminen järkevästi ei ole ihan itsestäänselvää. Ajatus on, että *logic*-pakkaus voisi tarjota rajapinnan, jolla voi luoda tapahtumankuuntelijoita äänen soittamista varten. Tätä varten pitäisi myös laukausita niitä tapahtumia jotenkin *logic*-pakkauksesta käsin, mutta tämä pitäisi myös tehdä niin, että koodi säilyy selkeänä ja ylläpidettävänä.
- CI-putkessa äänitiedostojen lataaminen ei onnistu. Tämä saattaa toistua myös muissa ajoympäristöissä.
- Pelin käynnistettyä ei ole mahdollista palata takaisin valikkoon.
- Ohjelma on muutaman kerran kaatunut satunnaisesti, mutta ongelmaa on vaikea saada toistumaan.

8.2 Parhaat ja heikoimmat kohdat

Ohjelman kolme parasta kohtaa:

- Ohjelma ei rajoita pelaajan liike- tai katsesuuntaa / Liikkumen on sulavaa

- Valaistusmoottori
- Tekninen toteutus

Ohjelma kolme heikointa kohtaa:

- Valaistusmoottorissa esiintyy bugeja: se ei piirrä varjoa seinän taakse, vaikka mikään ei valaise sitä. Tämän puutteen takia maailma pitää suunnitella niin, että mikään valonlähde ei ole liian lähellä mitään kulmaa, jonka takaa valo voisi heijastua. Tämän voisi korjata esimerkiksi shadow mappingin avulla.[11]
- Ohjelman suorituskyky voi olla laitteen tehosta ja näytön resoluutiosta riippuen vähän heikko. Tämä johtuu siitä, että varjostimissa tehdään raskasta valaistuksen laskentaa, mitä ei ole erityisemmin optimoitu. Ohjelmassa on tehty pieniä renderöinnin optimointeja, kuten että pelaajan takana olevia seiniä ei piirretä ollenkaan. Renderöintiä voisi kuitenkin optimoida edelleen tunnistamalla, mitkä kappaleet ovat kokonaan piilossa, tai nopeuttamalla varjostimissa tehtävää laskentaa.
- Visualisointikoodi on monimutkaista ja sen jatkokehittäminen vaatii ohjelmoijalta OpenGL:n tuntemusta.

8.3 Poikkeamat suunnitelmasta, toteutunut työjärjestys ja aikataulu

Projekti toteutettiin kutakuinkin siinä järjestyksessä, mikä kuvattiin teknisessä suunnitelmassa. Etenemisjärjestys oli seuraava:

1. Projektin luominen ja git-repon konfigurointi
2. GLFW-rajapinta ja ikkunan luominen
3. OpenGL-rajapinta ja ensimmäisen kolmion piirtäminen
4. Alustavat testit, CI/CD sekä GitLab runnereiden konfigurointi
5. Varjostimet ja 3D-projektio
6. Törmäyksen tunnistaminen ja pelaajan liikkuminen
7. Konfiguraatiotiedoston lataaminen

Alkuperäinen suunnitelma toteutettiin kokonaan, mutta projektia laajennettiin merkittävästi senkin jälkeen. Projektin vaikeustason ja laajuuden nostamiseksi siihen lisättiin vielä seuraavat ominaisuudet tässä järjestyksessä:

1. Tekstuurimäppäys
2. Valaistusmoottori ja kattolamput
3. Pelaajaa seuraava vihollinen
4. Äänimoottori, taustamusiikki ja muut peliäänet
5. Pyörivä kuutio

Suunnitelmassa arvioitu ajankäyttö projektille osui oikein niiltä osin, mitä suunnitelmassa oli otettu huomioon. Perustoiminnallisuuden toteuttamiseen kului noin 20 tuntua, mutta lisäominaisuuksiin kului ainakin 30 tuntia lisää.

Projektin aikana päätettiin myös vaihtaa konfiguraatiotiedoston formaatiksi YAML. Syynä oli se, että maailma haluttiinkin mielummin esittää monirivisenä merkkijonona, mikä ei onnistu JSON:ssa. YAML on myös kauniimpi formaatti tällaiselle tiedostolle.

8.4 Kokonaisarvio lopputuloksesta

Ohjelmasta tuli lopulta erittäin laadukas ja toimiva kokonaisuus. Työssä ei ole oleellisia puutteita, mutta ohjelmaa voi edelleen jatkokehittää ja parantaa. Varjojen laskennan lisääminen on yksi selvä kehityskohde, mutta ohjelmaa voisi myös esimerkiksi pelillistää enemmän lisäämällä pelaajalle jokin tavoite. Ratkaisumenetelmien, tietorakenteiden tai luokkajaon valintaa ei olisi voinut tehdä merkittävästi paremmin, sillä ohjelman rakenne soveltuu muutosten ja laajennusten tekemiseen erinomaisesti.

Jos aloittaisin projektin nyt uudelleen, tekisin kaiken samalla tavalla.

8.5 Kirjallisuusviitteitä

Projektin kehityksessä käytettiin Scalan[12], LWJGL:n[13], SnakeYAML-kirjaston[2], SBT-rakennustyökalun[14], Scalafmt-formatterin[15], ScalaTest-kirjaston[16] ja GitlabCI:n[17] dokumentaatiota ja mallikoodia. Virheenkorjauksessa käytettiin Stackoverflow-nettisivua.[18]

9 Lähdeluettelo

1: Grzegorz Kowal, Launch4J, 2022, https://launch4j.sourceforge.net/
2: MvnRepository, SnakeYAML, 2023, https://mvnrepository.com/artifact/org.yaml/snakeyaml
3: Jari-Pekka Salonen, Unity-projektin siirtäminen uuteen renderöintiputkeen, 2020, https://www.theseus.fi/bitstream/handle/10024/353508/Salonen_Jari-Pekka.pdf
4: , Rendering Pipeline Overview, 2022, https://www.khronos.org/opengl/wiki/Rendering_Pipeline_Overview
5: GLFW, Context reference, 2022, https://www.glfw.org/docs/3.3/group__context.html
6: Wikipedia, Rendering (computer graphics), 2023, https://en.wikipedia.org/wiki/Rendering_(computer_graphics)
7: Wikipedia, Rasterisation, 2023, https://en.wikipedia.org/wiki/Rasterisation
8: xilefian, Ray Casting - Technique for detecting intersection of an object and a line in virtual space., 2019, https://www.giantbomb.com/ray-casting/3015-1517/
9: Wikipedia, 3D projection, 2023, https://en.wikipedia.org/wiki/3D_projection
10: Six Jonathan, Frustum Culling, 2021, https://learnopengl.com/Guest-Articles/2021/Scene/Frustum-Culling
11: Learn OpenGL, Shadow Mapping, 2016, https://learnopengl.com/Advanced-Lighting/Shadows/Shadow-Mapping
12: docs.scala-lang.org, Learn Scala, 2023, https://docs.scala-lang.org/
13: LWJGL, Lightweight Java Game Library, 2023, https://www.lwjgl.org/
14: scala-sbt.org, sbt Reference Manual, 2023, https://www.scala-sbt.org/1.x/docs/
15: scalameta.org, Scalafmt Installation, 2023, https://scalameta.org/scalafmt/docs/installation.html
16: scalatest.org, ScalaTest User Guide, 2023, https://www.scalatest.org/user_guide
17: GitLab, GitLab CI/CD, 2023, https://docs.gitlab.com/ee/ci/
18: Stack Overflow, Stack Overflow, 2023, https://stackoverflow.com/

10 Liitteet

1. Projektin lähdekoodi
2. Kuvakaappauksia ohjelmasta