Klaus Vesa

3D- JA PELIOHJELMOINTIA OPENGL-RAJAPINNALLA

Tietotekniikan koulutusohjelma
Ohjelmistotekniikan suuntautumisvaihtoehto
2011

samk
Satakunnan ammattikorkeakoulu
Satakunta University of Applied Sciences

3D- JA PELOHJELMOINTIA OPENGL-RAJAPINNALLA

Vesa, Klaus
Satakunnan ammattikorkeakoulu
Tietotekniikan koulutusohjelma
Marraskuu 2011
Ohjaaja: DI Niemi, Juha
Sivumäärä: 42
Liitteitä: 0

Asiasanat: OpenGL, 3D-grafiikka, peliohjelmointi

Opinnäytetyön tavoitteena oli tutustua 3D-maailmaan sekä toteuttaa tämän pohjalta yksinkertainen kolmiulotteinen sovellus. Grafiikkatoiminnallisuuksien hyödyntäminen oli tarkoitus toteuttaa nimenomaan vuorovaikutteisten rakenteiden kautta, joihin esimerkiksi videopelit perustuvat.

Päätyminen juuri grafiikka- ja peliohjelmoinnin aihealueeseen pohjasi allekirjoittaneen hieman ehkä yllättäenkin heränneestä kiinnostuksesta pelien arkkitehtuuria ja rakenteita kohtaan. Aihekokonaisuus vaikutti jo heti alusta alkaen erittäin mielenkiintoisesta, joten omaehtoisen peli- ja 3D-ohjelmointiopiskelun pohjalta luotu opinnäytetyö tuntui tätä vasten loogiselta jatkumolta.

Opinnäytetyön pohjana käytetty henkilökohtainen 3D-projekti kasvoi lopulta ajateltua suuremmaksi. Syynä lienee jatkuvasti kumpuilleet uudet ideat sekä halu päästä kokeilemaan niiden toimivuutta käytännössä. Paisunut sisältö pakotti lopulta hylkäämään suunnitelmat projektin yksityiskohtaisesta dokumentoinnista.

Kompromissiratkaisuna opinnäytetyö päätettiin rakentaa kahdesta osasta. Ensimmäinen osio keskittyisi 3D-maailman käsitteisiin ja toimintaan yleisemmällä tasolla. Grafiikkarajapintana käytetty OpenGL tarjoaisi selkeän liittymän perustoiminnallisuuksien havainnollistamiseksi. Toisessa osiossa perehdyttäisiin lähemmin itse toteutettuun sovellukseen. Tarkastelu jouduttaisi tosin rajoittamaan lähinnä luokkien rakenteisiin ja vastuisiin johtuen juuri projektin liiaksi kasvaneista mittasuhteista.

3D- AND GAME PROGRAMMING WITH OPENGL-INTERFACE

Vesa, Klaus
Satakunnan ammattikorkeakoulu, Satakunta University of Applied Sciences
Degree Programme in Information Technology
November 2011
Supervisor: Msc(eng) Niemi, Juha
Number of pages: 42
Appendices: 0

Keywords: OpenGL, 3D-graphics, game programming

The purpose of this thesis was to get familiar with the basic concepts of three dimensional programming. The idea was also to produce a simple interactive application which would utilise graphical functionalities in a video-game-like approach.

The personal interest for graphics and game programming has been developing for quite some time now and the decision to use it as a topic for a thesis was not very difficult to make. The challenges provided by the very subject itself have proved to be both interesting and rewarding.

The actual 3D-application project, that the thesis is based on, proved to be too large for detailed documenting. The reasons behind the uncontrolled growth can be traced down to curiosity and urge to try out one's never-ending personal ideas.

As a result, the thesis itself was divided in to two halves. The first half would concentrate on describing the basic principals of 3D-graphical utilities by using OpenGL-interface. The second half is dedicated to the accomplished 3D-project itself. Due to it's relatively large size the focus is concentrated on the basic structure and behaviour of the classes built.

SISÄLLYS

TERMILUETTELO

OpenGL	Open Graphics Library. Silicon Graphicsin kehittämä grafiikantuottamisrajapinta.
DirectX	Microsoftin multimediatuotannossa käytetty ohjelmointirajapintaperhe, johon lukeutuu muun muassa grafiikantuotannossa suosittu Direct3D.
Win32 API	Microsoftin ikkunatoiminnallisuuksiin keskittynyt ohjelmointirajapinta.
.NET Framework	Microsoftin ikkunatoiminnallisuuksiin keskittynyt ohjelmointirajapinta.
Polygoni	Vähintään kolmesta pisteestä muodostuva monikulmio (eng. polygon), joita käytetään kolmiulotteisen maailman rakentamisessa.
Verteksi	Muodostaa polygonin kulmat. Verteksit (eng. vertex) ovat kolmiulotteisen maailman perusta.
Tekstuuri	Useimmiten bittikarttakuvasta luotu pintakuvio. Tekstuureilla (eng. texture) elävöitetään polygonien pintoja 3D-grafiikkaa tuotettaessa.
Md2	3D-mallien rakentamiseen käytetty vanhahtava tiedostotyyppi. Sisältää kaiken olennaisen datan mallin piirtämiseksi ruudulle.

1 JOHDANTO

Suomalaisen peliteollisuuden esiinmarssi on myllertänyt perinteistä IT-alan kenttää esittelemällä uuden ja melko tuntemattoman aluevaltauksen. Peleissä tai niiden tuotannossa ei sinänsä ole mitään uutta, mutta suomalaisen teollisuuden haarana sitä ei perinteisesti olla totuttu näkemään. Ala on kuitenkin tullut jäädäkseen, ja kasvavien pelitalojen vauhdikas nousu onkin muodostamassa Suomeen Pohjoismaiden suurimman peliteollisuuden keskittymän. (Suomen peliala... 2011; Sijoitusenkelit katsastivat... 2011)

Pelit eivät kuitenkaan tee itse itseään, eikä osaajien tarve tulevaisuudessa ole ainakaan pienenemään päin. Kokeneiden virtuoosien tarve on jatkuva, mutta kasvavasta työvyyhdistä selvitäkseen on myös nuoremman ja kokemattomamman sukupolven potentiaali valjastettava hyötykäyttöön hyvissä ajoin. (Työvoimapula... 2011)

Pelialasta kiinnostuneen tie lopulliseen tuotantotiimiin ei kuitenkaan ole täysin kivetön. Koulutusta tarjoaa vain kourallinen opinahjoja, ja monet peliteollisuudesta kiinnostuneet päätyvätkin itseopiskelun tielle pelialalle pyrkiessään. Epävarmuutta lisää suomalaisten pelitalojen sirpaleinen kenttä sekä epämääräiset kasvu- ja rekrytointifilosofiat. Alalle pyrkijän onkin jo alusta asti asetettava rima hyvinkin korkealle ja väsymättä kehitettävä omaa osaamistaan. (Hannula 2007)

Tämä opinnäytetyö käsittelee peli- ja 3D-ohjelmointia juuri aloittelijan näkökulmasta ilman alakohtaista koulutusta. Pohjana on käytetty laajahkoa henkilökohtaista *Project3D*-projektia, jossa tutustutaan peliohjelmoinnin ja grafiikan tuottamisen saloihin paljolti käytännön ja kokeilun kautta. Tekeleessä sovelletaan niin koulussa opittua perusohjelmointiosaamista, kuin sekalaisista lähteistä ammennettua peliohjelmointitietoisuutta. Alkuosan dokumentoitu tarkastelu keskitetään lähinnä peruskäsitteistöön ja yksinkertaistettuihin tekniikoihin. Lopussa perehdytään tarkemmin toteutetun ohjelmointiprojektin toiminnallisuuteen ja luokkarakenteisiin.

2 POHJA JA TYOKALUT

Vaikka tarkastelu pyritään pitämään mahdollisimman yleisellä tasolla, on esimerkiksi 3D-grafiikkaa tuotettaessa sukellettava pintaa syvemmälle rajapintojen yksityiskohtaisempiin toimintoihin. Tarkoituksen on luoda hyvä pohja ja perusymmärrys kolmiulotteisen maailman käsitteistöstä ja toiminnallisuudesta.

Grafiikan tuottamisen lisäksi olennaisia teemoja ovat käyttäjän ja ohjelman välinen kommunikaatio sekä ikkunoiden varaaminen käyttöjärjestelmältä tarvittaviin piirtorutiineihin. Käyttöliittymäikkuna ja sen oikea käsittely onkin perusedellytys mitä tahansa graafiseen visuaalisuuteen nojaavia sovelluksia luotaessa.

Ohjelman ja koodin jaottelu olio-ohjelmoinnin näkökulmasta tulee myös keräämään osan huomiosta. Selkeä ja hyvin jäsennelty koodi kapseloituine toimintoineen on perusedellytys vähänkään laajemmalle kokonaisuudelle. Konkreettisimmin tämä tulee näkymään setvittäessä itse malliprojektia raportin loppupuoliskolla.

2.1 C++

Kielivanhus jaksaa yhä. Syntymästään saakka kiistanalainen kehitystyökalu on kerännyt matkallaan niin risut kuin ruusut, mutta oli itse kielestä ja sen kommervenkeista mitä mieltä tahansa, niin totuutta on hankala kiistää; C++ porskuttaa edelleen ja on niittänyt mainetta erityisesti peliohjelmoinnissa.

Kielivalinta malliprojektin raapustelulle onkin ollut itsestäänselvyys oikeastaan jo alusta alkaen. Päätökseen on ollut vahvasti vaikuttamassa ohjelmoinnin ensiaskeleiden ottaminen nimenomaan C/C++-perspektiivistä. Ensikieli on se oikea kieli, jonka perustalle kaikki muu ohjelmointiosaaminen on myöhemmin rakentunut.

Valintaan on vaikuttanut toki myös käytännöllisemmät syyt. Esimerkiksi hyvin monet peli- ja grafiikkaohjelmointia käsittelevät oppimateriaalit noudattelevat melkeinpä poikkeuksetta C-perheen syntaksia. Täten myöskään projektin toteutusta tukevan informaation tai esimerkkimateriaalin hankinta erinäisistä lähteistä ei ole päässyt muodostumaan ylitsepääsemättömäksi ongelmaksi. Valintakriteereihin lisättäköön vielä C++:n lähes saumaton rajapintayhteensopivuus.

Myös suorituskyvylliset tekijät liputtavat kutakuinkin yksimielisesti valitun kielen puolesta. C++-kääntäjä puskee ulos puhdasta ja välittömästi suoritettavaa konekieltä, joka takaa maksimaalisen ajonopeuden niin perusprosessorin kuin grafiikkasuorittimenkin osalta. Monet virtuaalikoneet ja niihin nojautuvat kielet ovat toki vuosien saatossa nostaneet profiiliaan suorituskykyään kasvattamalla, mutta siitä huolimatta C++ tuntuu tässä yhteydessä luonnollisimmalta vaihtoehdolta. (C++ referenssisivusto)

Kritiikkiä C++ kerää oikeastaan samoista syistä kuin kehuja. Monipuolinen toiminnallisuus ja syntaksi altistaa myös laadullisesti heikompaan koodiin. Onkin sanottu, että C++:lla on todella helppoa kirjoittaa huonoa koodia. Allekirjoittaneen havaintojen perusteella esimerkiksi Java todella tuntuu tarjoavan selkeämpiä sääntöjä ja standardeja. C++ taas vaikuttaa selvästi pirstaleisemmalta tarjoten laadukkaan koodin mielipiteitä välillä vähän turhankin leveältä rintamalta. (Wikipedia C++)

2.2 Win32 API

Graafisen sovelluksen tarvitsema ikkunaympäristö sekä siihen liittyvä toiminnallisuus toteutetaan käyttöjärjestelmän omalla käyttöliittymärajapinnalla. Windows API:n tarjoama suoraviivainen ja kevyt C-kielinen kirjasto kattaa monipuolisia funktioperheitä käyttöliittymäympäristön luomiseen ja ylläpitoon. Piirtoa ja visuaalisuutta hyödyntävien sovellusten onkin käytännössä aina tukeuduttava ikkunatoiminnallisuuteen sulavan toteutuksen takaamiseksi.

Ikkunoiden lisäksi tarjoaa rajapinta myös tapahtumakäsittelijän, joka mahdollistaa muun muassa käyttäjän syötteiden pähkäilyn. Win32-rajapintaa hyödyntävän kuunte-

lijaproseduurin toteutus onkin pyrittävä sulauttamaan mahdollisimman saumattomasti muun kokonaisuuden yhteyteen. Haasteita asettaa lähinnä olioajatteluun perustuva projektin rakenne, jonka sääntöihin C-kieleen nojaava tapahtumien käsittely pitää sulavasti taivuttaa.

Käyttöliittymän toteutukseen löytyy toki muitakin kirjastoja. Päätökseen Win32 API:n käyttöönotosta liittyi ennen kaikkea useiden oppimateriaalien ikkunatoiminnallisuuden rakentuvan juuri kyseisen rajapinnan ympärille. Varteenotettavin vaihtoehto nimenomaan pelituotannossa lienee Microsoftin DirectX-perhe, joka tarjoaa työkalut käyttöliittymän ylläpidon lisäksi niin grafiikka- kuin äänipuolenkin kommervenkkeihin. Kelpona kandinaattina pidettäköön niin ikään Microsoftin paimentamaa .NET-rajapintaa, joka on suosittu valinta monien muiden ikkunasovellusten perustaksi. (Microsoft MSDN kehittäjäsivusto)

2.3 OpenGL

Pääasiassa 3D-grafiikan piirtotoiminnallisuuksiin keskittyvä rajapinta, joka on hyvin suosittu muun muassa eri tyyppisten simulaattoreiden, CAD-ohjelmistojen ja videopelien tuotannossa. Nimensä mukaisesti *(Open Graphics Library*) se keskittyy vain ja ainoastaan grafiikkarutiineihin jättäen muun toiminnallisuuden, kuten ikkunoiden ja syötteiden käsittelyn, muiden rajapintojen hoidettavaksi. (Shreiner 2010, 2)

Alun perin Silicon Graphicsin vuonna 1992 kehittämä ja sittemmin Khronos Groupille siirtynyt OpenGL spesifikaatio on ehtinyt 4.2-versioonsa (vuosi 2011). Perusidea ja toiminnot ovat uudistuksista huolimatta pysyneet samoina, eikä piirtorutiinien suoritus ole vuosien saatossa juurikaan muuttunut. Esimerkiksi opinnäytetyön malliprojekti perustuu versioihin 1.2 ja 1.3 taaten kaikki tarvittavat työkalut yksinkertaisen 3D-sovelluksen toteuttamiseen. Uudet ja kehittyneemmät versiot tarjoavat lähinnä viriteltyjä lisäominaissuksia ja laajennuksia.

OpenGL on toteutettu C-kielellä ja täten sen funktiot tottelevat samaa syntaksia. Useimmat komennot ovat hyvin suoraviivaisia ja lyhyitä, joilla tilakoneen tavoin toimiva kokonaisuus asetetaan oikeaan asentoon ennen varsinaisia piirtotoimintoja.

Ohjelmoijan kasaama vyyhti asetetaan lopuksi ”liukuhihnalle” *(eng. rendering pipeline)*, jossa vaihe vaiheelta lasketaan tilakoneen mukaisilla parametreilla näytölle piirrettävä lopputulos. (Shreiner 2010, 10-11)

Komentojen pelkistetty luonne pakotta kehittäjän syventymään ajoittain hyvinkin tarkasti toimintojen yksityiskohtiin. OpenGL ei juuri tarjoa kaikenkattavia tai kokonaisuuksia alleen kätkeviä rutiineja, vaan ohjelmoijan on itse kyettävä pienistä palikoista rakentamaan haluamansa lopputulos. Toisaalta tämä on varmasti myös edesauttanut perustoiminnallisuuden pysymistä lähes muuttumattomana vuodesta toiseen.

OpenGL:n suosio takaa myös verrattain hyvän oppimateriaalin saatavuuden. Aktiiviset ammattilaiset ja harrastelijat ovatkin luoneet pussillisen varteenotettavia internetlähteitä, joista osa kattaa kokonaisia tutoriaalisarjoja. Konkreettisempaa kosketusta kaipaavat löytävät etsimänsä OpenGL-kirjallisuutta notkuvista kirjahyllyistä. Opuksista suosituin lienee Khronos Groupin masinoima virallinen OpenGL Programming Guide, josta julkaistaan uusi versio aina spesifikaation päivityksen yhteydessä. Hienouksistaan huolimatta järkälemäinen kirja sisältää tarvittavat tiedot myös hapuilevia ensiaskelia ottaville aloittelijoille.

3 PELIEN PERUSRAKENNE

Monimuotoisuudestaan huolimatta pelien ja myös muiden grafiikkaa hyödyntävien ohjelmien toteutus nojaa hyvin samankaltaiseen logiikkaan. Ohjelman käynnistyttyä toteutetaan vaadittavat alustukset ja tarkistukset, ja itse suoritus toteutetaan vaatimusten mukaisessa toistorakenteessa.

Ikkunoiden ja tapahtumankäsittelijöiden luonti, sekä grafiikkarajapinnan sitominen käyttäjäikkunaan ovat tavallisimpia käynnistystoimenpiteitä. Alustusoperaatiot käsittävät usein muitakin ohjelman kannalta kriittisiä peruskonfigurointeja.

Olennaisin peruspalikka lienee kuitenkin ohjelman suoritusta ylläpitävä toistorakenne. Pelisilmukaksi *(eng. game loop)* videopelituotannossa ristitty hyvinkin yksinkertainen *while*-toistorakenne koko on sovelluksen sydän. Elinkaarensa alusta loppuun pelisilmukka huolehtii tilannepäivityksistä, jotka peruskokoonpanossa kattavat niin tapahtumankäsittelyn kuin grafiikkarutiinein suoritetun ruudunpäivityksen.

Pelisilmukan voi siis karkeasti jakaa kahteen osaan. Ensimmäinen liittyy ulkopuolisin tapahtumiin ja on käytännössä aina liitoksissa käyttöliittymäikkunan tapahtumankäsittelijään. Käyttöjärjestelmä kuuntelee aktiiviseen peli-ikkunaan liittyviä tapahtumia, kuten käyttäjän syötteitä, ja lähettää ne viestijonoon odottamaan käsittelyvuoroa. Pelisilmukassa sijaitseva tarkistusrutiini tutkii viestijonon ja lähettää kaikki uudet viestit tapahtumankäsittelijälle, joka viestien sisällön perusteella ryhtyy tarvittaviin toimenpiteisiin.

Toinen osa huolehtii itse pelitilanteen päivittämisestä. Tämä on käytännössä aina kytköksissä kuluneeseen aikaan. Esimerkiksi 16,67 millisekunnin välein päivittyvä pelitilanne tarkoittaa 60 kuvaa sekunnissa *(eng. frames per second, fps)* näyttöpuskurille kopioivaa rutiinia. Seuraava yksinkertaistettu koodinpätkä havainnollistaa pelin perusrakennetta.

```
#include <tarvittavatKirjastot>

int main ()
{
  ikkunanLuonti ();
  muutAlustukset ();

  while (peliPyorii) {
    if (viestijonossaViesteja ()) {
      viestiTapahtumankasittelijalle ();
    }
    else if (kulunutAika() >= paivitystiheys ()) {
      paivitaPelitilanne ();
      piirraPelitilanne ();
      nayttopuskurinPaivitys ();
    }
  }

  ikkunanTuhoaminen ();
  muutVapautukset ();

  return 0;
}
```

Esimerkki koostuu alustuksista, viestien ja tapahtumien käsittelystä, pelitilanteen päivityksestä sekä resurssien vapautuksesta ohjelman päättyessä. Pelitilanteen päivitys on edelleen jaettu kahteen osioon, joista ensimmäisessä toteutetaan loogisten parametrien asetus, kuten esimerkiksi pelihahmon uusi sijainti käyttäjän näppäinpainalluksiin perustuen, ja toisessa hoidetaan piirtorutiinit käyttäen edellä päivitettyjä parametreja. Esimerkki on pelkistetty versio perustuen lähinnä kirjoittajan omiin kokemuksiin, eikä missään nimessä ole ainoa tai paras ratkaisu.

3.1 Peruspiirtoa vertekseillä

OpenGL mahdollistaa niin kaksi- kuin kolmiulotteiset piirtorutiinit. 3D-toiminnallisuus lienee kuitenkin se suurin hyödynnettävä hienous.

Kolmiulotteinen maailma on täysin abstrakti. Mitään syvyyskomponenttia ei todellisuudessa ole; lopputuloshan piirtyy aina pikseleinä ruudulle kaksiulotteisessa koordinaatiossa. 3D luo silkan illuusion syvyydestä laskemalla piirrettäville pikseleille oikean väriarvon ja sijainnin. Näillä laskelmilla ei ohjelmoijan kuitenkaan tarvitse päätään vaivata, sillä grafiikkarajapinta, tässä tapauksessa OpenGL, huolehtii kolmiulot-

teisuusefektin vaatimista rasterointilaskelmista *(eng. rasterization)*. Ohjelmoija voi täysillä keskittyä geometrian lakeja tottelevan 3D-maailman rakentamiseen.

Kolmiulotteinen kuvio rakentuu vähintään kolmesta pisteestä avaruudessa. Näiden pisteiden väliin piirretään viiva ja mahdollisesti täytetään sisälle jäävä osa. Kuviota kutsutaan polygoniksi *(eng. polygon)* eli monikulmioksi. Pisteitä, joista polygoni muodostuu, kutsutaan vertekseiksi *(eng. vertex)*. Yleisin polygonimuoto on kolmio, joka siis muodostuu kolmesta verteksistä. Kolmio on käytännössä pienin rakennuspalikka, joista monimutkaisimmatkin 3D-mallit muodostuvat. Seuraavassa periaatteellisessa esimerkissä piirretään yksinkertainen kolmio kolmiulotteiseen avaruuteen.

```
#include <tarvittavatKirjastot>

int main ()
{
  ikkunanLuonti ();

  while (piirretaan) {
    glClearColor (0.0, 0.0, 0.0, 0.0);
    glClear (GL_COLOR_BUFFER_BIT);

    glBegin (GL_TRIANGLES);
      glVertex3f (-1.0,  1.0, -3.0);
      glVertex3f (-1.0, -1.0, -3.0);
      glVertex3f ( 1.0, -1.0, -3.0);
    glEnd ();

    nayttopuskurinPaivitys ();
  }

  ikkunanTuhoaminen ();

  return 0;
}
```

Koodiesimerkissä ei luoda varsinaista pelisilmukkaa edellisen esimerkin tapaan, vaan tyydytään selvittämään yksinkertaista piirtorutiinia. Kaikki gl-alkuiset funktiot ovat aitoja OpenGL funktioita, joita täydentää periaatteellista toimintaa kuvaavat funktiot ja muuttujat.

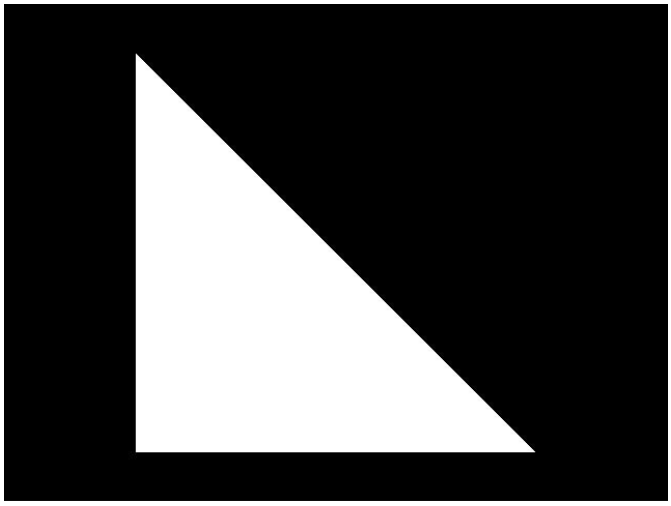
Toistorakenteen alussa tyhjennetään ikkuna mustaksi. Funktio *glClearColor* määrää nimensä mukaisesti ruuduntyhjennysvärin asettamalla halutun väriarvon komponenteittain. Perinteisesti punaista, vihreää, sinistä sekä läpikuultavuuden määräävää alp-

ha-komponentia on käsitelty arvovälillä 0...255. OpenGL sen sijaan lähestyy väriarvoja useimmiten liukulukuasteikolla välillä 0.0...1.0 säilyttäen suhteen kuitenkin samana. Itse ruuduntyhjennys toteutetaan *glClear*-funktiolla parametrinaan väriarvojen tyhjennyksestä kertova lippu. Kyseistä funktiota on siis mahdollista käyttää myös muihin puhdistustoimenpiteisiin parametreina annettujen lippujen mukaisesti.

Varsinainen polygonin piirto suoritetaan aina *glBegin*- ja *glEnd*-funktioiden välissä. Parametrina annettava symbolinen vakio *GL_TRIANGLES* -kertoo piirrettävän polygonin muodostuvan kolmesta verteksistä. Itse verteksit luodaan antamalla jokaiselle sijainti xyz-avaruudessa. Verteksifunktion nimen lopussa oleva *3f* tarkoittaa avaruuskoordinaattien muodostuvan kolmesta float-liukuluvusta. Piirtämisen jälkeen päivitetään näyttöpuskuri, jolloin kuva saadaan siirrettyä grafiikkakortilta näytölle.

3D-avaruus noudattaa ns. vasemman käden koordinaatistoa, jolloin syvyydestä vastaava z-akseli osoittaa näytöstä käyttäjään päin. Syvälle näytön ”sisään” päästäkseen on siis matkattava z-akselia negatiiviseen suuntaan kuten *-3.0* jokaisen verteksin z-parametrina indikoi. Lähempänä ollessaan kolmio näyttäisi tietysti isommalta ja syvyyden ollessa *0.0* yksikköä se piirtyisi tarkalleen näytön tasalle.

Seuraava kuva esittelee edellä esitetyn koodiin perustuvan kolmionmuotisen peruspolygonin piirron kolmen yksikön syvyyteen vakioväriasetuksilla.



Kuva 1. Peruspolygoni kolmen yksikön syvyydessä

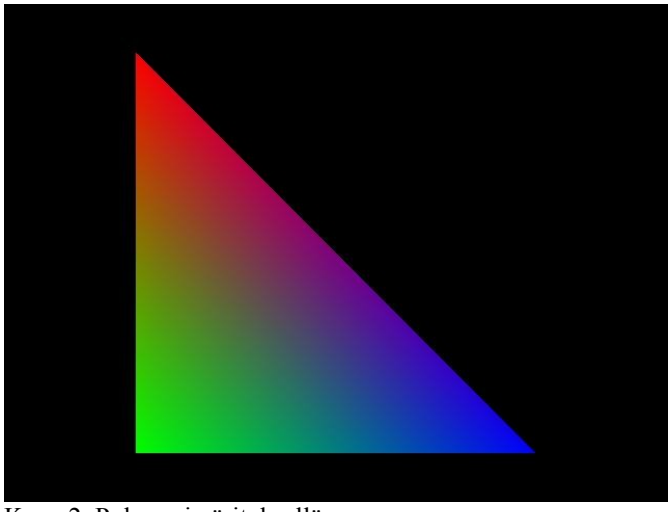
3.2 Väritys

Edellisen esimerkin valkoinen kolmio käyttää vakioväriasetusta. Kolmiulotteisten kappaleiden tai mallien piirrossa on kuitenkin mahdollista hyödyntää värityskomentoja pintojen elävöittämiseksi.

Perusväritys lasketaan verteksikohtaisesti. Jos jokainen kolmion kulma saa piirron yhteydessä eri väriarvon, kolmion kokonaisväritys muodostuu kirjavaksi väriyhdistelmäksi. Seuraava koodiesimerkki havainnollistaa värityksen käyttöä aiemmin esitetyn peruspolygonin piirron yhteydessä. Esimerkissä käsittelyyn on otettu ainoastaan silmukkarakenne muun toiminnallisuuden pysyessä muuttumattomana.

```
while (piirretaan) {
  glClearColor (0.0, 0.0, 0.0, 0.0);
  glClear (GL_COLOR_BUFFER_BIT);

  glBegin (GL_TRIANGLES);
    glColor3f (1.0, 0.0, 0.0); glVertex3f (-1.0,  1.0, -3.0);
    glColor3f (0.0, 1.0, 0.0); glVertex3f (-1.0, -1.0, -3.0);
    glColor3f (0.0, 0.0, 1.0); glVertex3f ( 1.0, -1.0, -3.0);
  glEnd ();

  nayttopuskurinPaivitys ();
}
```

Väriarvot asetetaan *glColor3f*-funktiolla. Jokaiselle verteksille asetettu erillinen väriarvo saa aikaan seuraavan kuvan kaltaisen väriyhdistelmän. Yhden väriarvon käyttöön taas riittäisi yksi funktion kutsu ennen verteksien piirtoa.



Kuva 2. Polygoni värityksellä

3.3 Tekstuurit

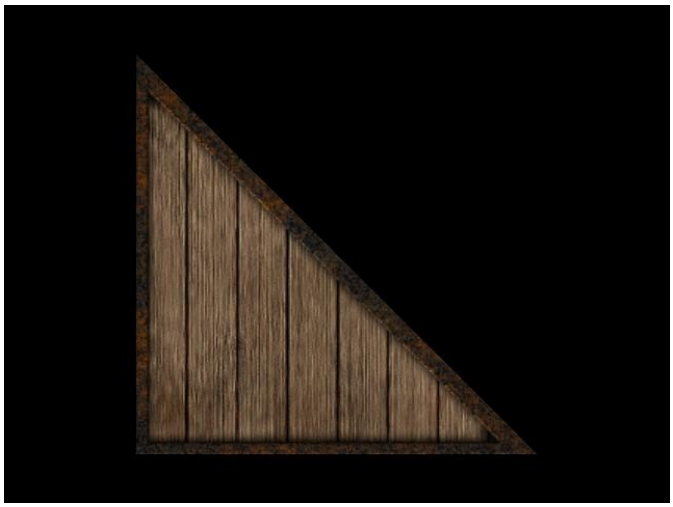
Pintojen väritystä luotaessa ei suinkaan tarvitse tyytyä tasaisiin perusväreihin. Erityyppisillä tekstuureilla on mahdollista elävöittää 3D-maailmaa monipuolisemmalla pintakoristelulla. Tekstuurin voidaankin ajatella olevan polygonin päälle liimattu kuva. Se luodaan lähes aina erillisestä kuvatiedostosta, josta poimitaan itse kuvadata pikseli-informaatioineen. Tätä pikselidataa hyödyntämällä OpenGL luo piirrettäviin polygoneihin yhteensopivan tekstuurin.

Kaksiulotteisesta kuvadatasta muodostuvaa tekstuuria käsitellään niin ikään 2D-periaatteella. Kahta komponenttia nimitetään *s*- ja *t*-koordinaateiksi. Monesta muusta rajapinnasta poiketen, OpenGL katsoo origon sijaitsevan aina vasemmalla alhaalla. Esimerkiksi Windows olettaa (0,0) pisteen löytyvän aina vasemmalta ylhäältä. Joka tapauksessa on tekstuurikoordinaatteja hyödyntämällä mahdollista sitoa tekstuuri tai osa tekstuurista verteksien kautta piirrettävään polygoniin. Seuraava periaatteellinen esimerkki lisää edellä piirrettyyn valkoiseen kolmioon bmp-kuvatiedostosta luodun tekstuurin.

```
while (piirretaan) {
  glClearColor (0.0, 0.0, 0.0, 0.0);
  glClear (GL_COLOR_BUFFER_BIT);

  glEnable (GL_TEXTURE_2D);
  glBindTexture (GL_TEXTURE_2D, tekstuurinTunnus);

  glBegin (GL_TRIANGLES);
    glTexCoord2f (0.0, 1.0); glVertex3f (-1.0,  1.0, -3.0);
    glTexCoord2f (0.0, 0.0); glVertex3f (-1.0, -1.0, -3.0);
    glTexCoord2f (1.0, 0.0); glVertex3f ( 1.0, -1.0, -3.0);
  glEnd ();

  nayttopuskurinPaivitys ();
}
```

Käsittelyssä on vain tekstuurin liitos piirtämisen yhteydessä. Itse tekstuurin luonti tapahtuu esimerkin ulkopuolella. Uutta koodissa on tekstuurien käytön salliminen *glEnable*-funktiolla, sekä halutun tekstuurin valinta käyttäen *glBindTexture*-funktiota. Toisena parametrina toimiva kokonaisluku, *tekstuurinTunnus,* annetaan jokaiselle OpenGL-tekstuurille luonnin yhteydessä.

Piirtoa suoritettaessa *s*- ja *t*-koordinaatit sitovat tekstuurin polygonin muodostaviin vertekseihin *glTexCoord3f*-funktiolla. Esimerkiksi ensimmäisellä rivillä annettavat (0.0, 1.0) koordinaatit sitovat tekstuurin vasemman yläkulman vasemmassa yläkulmassa sijaitsevaan verteksiin. Tekstuuri ei siis käytä samoja 3D-koordinaatteja, vaan omia *s*- ja *t*-koordinaatteja sidottaessa kuvaa tai sen osaa polygoniin. Esimerkkikoodin tuottama polygoni tekstuureineen näyttäisi seuraavan kuvan kaltaiselta.



Kuva 3. Polygoni tekstuurilla

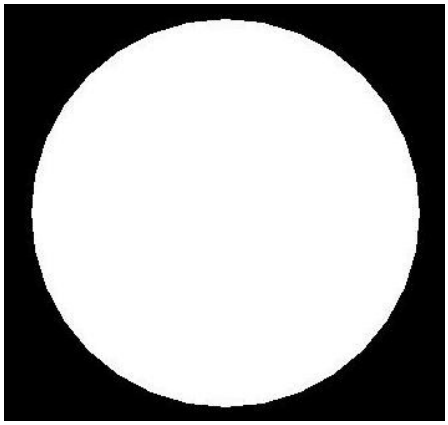
Itse tekstuuri on muodostettu seuraavan kaltaisesta 24-bittisestä bmp-tiedostosta. Kuva on siis ikään kuin saksittu vasemmasta yläkulmasta oikeaan alakulmaan ja liimattu polygoniin.

Kuva 4. Tekstuurin muodostava bmp-kuva

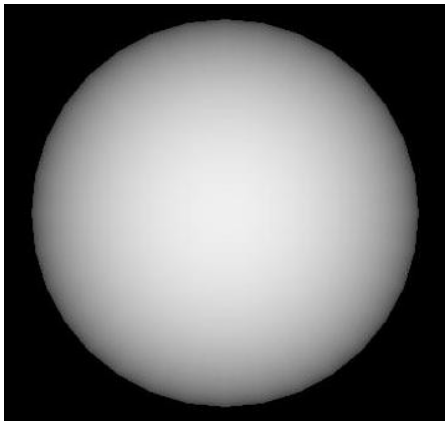
Tekstuurin ovat verteksien rinnalla hyvin olennainen osa 3D-grafiikan tuottamista. Ongelmaksi nousee kuitenkin piirrettävän lopputuloksen laatu. Kolmiulotteisessa maailmassa objekteja katsellaan aina eri kulmista ja syvyyksistä, jolloin tekstuuri ei koskaan ole alkuperäisen kuvan kaltainen, vaan uudelleen laskettu tulkinta katselukulmasta ja syvyydestä riippuen. Tämä laskenta on vaativaa ja aikaa vievää. Videopeleissä huomaa usein, kuinka esimerkiksi päähahmon tekstuurit ovat hyvin tarkkoja ja huolella tehtyjä, mutta peliympäristön taustatekstuurit ovat selvästi yksinkertaisempia ja suttuisempia johtuen juuri laskentakuormituksen priorisoinnista ja kohdentamisista pelikokemuksen kannalta olennaisimpiin objekteihin.

3.4 Valaistus ja normaalit

Eräs hyvin keskeinen tekijä kolmiulotteisen maailman rakentamisessa on valo ja sen käyttäytyminen eri pinnoilla. Valaistuksella ei tässä yhteydessä ole oikeastaan mitään tekemistä näkyvyyden kanssa vaan sitä käytetään työkaluna syvyysvaikutelmaa luotaessa. Seuraava esimerkki valaisee asiaa yksinkertaisen pallonpiirron yhteydessä.



Kuva 5. Pallo ilman valaistusta

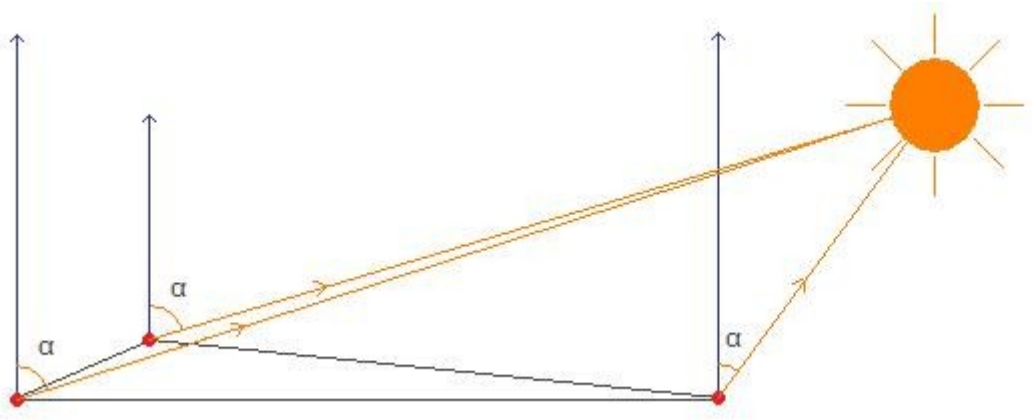


Kuva 6. Pallo valaistuksella

Molemmissa kuvissa on täysin sama ja yhtä monesta polygonista rakennettu valkoinen pallo. Kuitenkin vain toisessa on otettu huomioon valaistus ja valonlähde. Tällöin verteksien värilaskelmissa polygonin pintaan kohtisuorasti paistava valo antaa kirkkaamman väriarvon, kuin kulmassa olevalle pinnalle. Tässä tapauksessa valonlähde on siis ruudusta ”ulkona” katsojan ja näytön välissä valaisten pallon etupintaa

saaden aikaan illuusion pallon pyöreydestä. Oikeanlaisella ja huolellisesti suunnitellulla valaistuksella on mahdollista saada selkeästi eloa ja syvyyttä 3D-maailmaan.

Valaistus ei kuitenkaan toimi oikein pelkällä verteksi- tai tekstuuri-informaatiolla. Valonlähde ei yksinkertaistettuna ole muuta kuin pelkkä piste avaruudessa, jonka sijainnin suhteen lasketaan verteksin värisävy *(eng. shade)*. Jotta laskelmat onnistuisivat, tarvitsee verteksi sijaintitietonsa lisäksi yksikkövektorin osoittamaan etupuoltaan. Käytännössä yhden polygonin jokainen yksikkövektori osoittaa samaan suuntaan muodostaen polygonin tason vastaisen normaalin. Nämä polygonin pinnalta kohtisuoraan sojottavat vektorit siis määrittelevät polygonin etupuolen valaistuslaskelmien yhteydessä. Seuraava kuva havainnollistaa normaalilaskelmien ideaa.



Kuva 7. Normaalien idea valolaskelmissa

Kuvaan piirretty kolmion muotoinen polygoni muodostuu kolmesta punaisella värillä merkitystä vertkeksistä. Kolmion kulmista kohtisuoraan ylöspäin osoittavat vektorit kuvaavat vertekseille asetettuja normaaleja. Normaalivektorin ja verteksi-valonlähdevektorin välinen kulma määrää kyseisen verteksin värisävyn. Sävykerroin syntyy pistetulosta eli kahden yksikkövektorin välisen kulman kosinista. Toisin sanottuna mitä suurempi kulma sen tummempi väriarvo. Kuvan tapauksessa oikeanpuoleisen kulman ympäristö näyttäisi jonkin verran kirkkaammalta kuin vasemmanpuoleisten kulmien.

Seuraava koodi pohjaa valkoiseen peruspolygoniesimerkkiin. Lisänä koodissa ovat valonlähde ja normaalit.

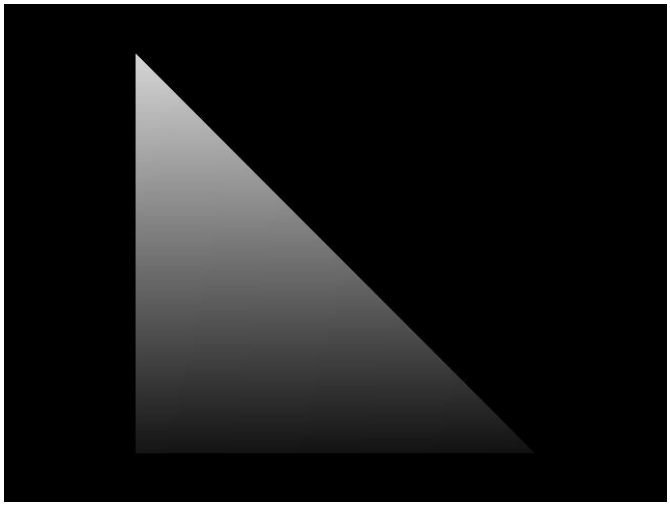
```
while (piirretaan) {
  glClearColor (0.0, 0.0, 0.0, 0.0);
  glClear (GL_COLOR_BUFFER_BIT);

  glLightfv (GL_LIGHT0, GL_POSITION, sijaintiVektori);
  glEnable (GL_LIGHT0);
  glEnable (GL_LIGHTING);

  glBegin (GL_TRIANGLES);
    glNormal3f (0.0, 0.0, 1.0);
    glVertex3f (-1.0,  1.0, -3.0);
    glVertex3f (-1.0, -1.0, -3.0);
    glVertex3f ( 1.0, -1.0, -3.0);
  glEnd ();

  nayttopuskurinPaivitys ();
}
```

Valonlähteen perusasetukset hoidetaan *glLightfv*-funktiolla. Parametrina annettu *GL_LIGHT0*-vakio asettaa käsittelyn kohteeksi ensimmäisen OpenGL:n monista valonlähteistä. Toisena parametrina toimiva *GL_POSITON*-vakio taas kertoo asetusten kohdistuvan nimenomaan valonlähteen sijaintiin avaruudessa. Kolmantena parametrina annetaan itse sijaintiedot taulukkomuodossa. Funktion päätteenä oleva *fv* tarkoittaakin datan koostuvan nimenomaan liukulukutaulukosta *(float vector)*.

Muissa valoasetuksissa haluttu valonlähde sallitaan ja asetetaan aktiiviseksi *glEnable*-funktioilla. Valaistus on siis yleisesti sallittava vielä erikseen käyttäen *GL_LIGHTING*-vakioa.

Piirron yhteydessä kaikille vertekseille asetetaan samaan suuntaan osoittava normaali *glNormal3f*-funktiolla. Jokaiselle verteksille on toki mahdollista asettaa erillinen normaali, mutta polygonin pinnan ollessa tasainen yksi funktiokutsu riittää kaikille vertekseille. Normaalivektori osoittaa siis ”ruudusta ulos” katsojaan päin.

Koodiesimerkin mukainen kuva osoittaa väriarvojen käyttäytymisen valonlähteen sijaitessa lähellä kolmion vasenta yläkulmaa.



Kuva 8. Peruspolygoni ja valaistus

Valon paistaessa kohtisuoraan ylempään verteksiin on väriarvo selvästi kirkkaampi. Alempien vertksien tummuus johtuu nimenomaan suuremmasta kulmasta normaalien ja valonlähteen välillä. Polygonin pinnan suhteen kulma on tällöin siis jyrkempi.

Kolmio värittyy kokonaisuutena kohtalaisen sulavasti verteksejen normaaleista ja valaistuksesta riippuen. Tämän saa aikaan väriarvojen sulava sävytys *(eng. smooth shading)*. Vaihtoehtoinen tapa on käyttää tasaista sävytystä *(eng. flat shading)*, jolloin yhden polygonin väriarvo muodostuu vain yhdestä yhtenäisestä väristä yksittäisten verteksien valolaskelmista huolimatta.

Valot, värit ja tekstuurit saattavat välillä käyttäytyä konstikkaasti. Lopulliset värilaskelmat muodostuvat aina useista eri parametreista, eivätkä väriarvot välttämättä vastaakaan omia laskelmia. Jos valotoiminnallisuuteen haluaa saada hyvän otteen, on uhrattava aikaa aihealueen syvempään tutkiskeluun.

3.5 Pyöritystä matriisein

Syvyyskomponentti on kolmiulotteisen maailman ehkä olennaisin piirre. Tämä tarkoittaa käytännössä mahdollisuutta tarkastella objekteja eri suunnilta ja etäisyyksiltä kuitenkaan rikkomatta oikeita mittasuhteita.

Yksinkertaisia siirtoja ja pyörityksiä on mahdollista tehdä suhteellisen helposti suoraviivaisilla komennoilla tuntematta juurikaan yksityiskohtia toiminnallisuuden takana. Seuraavassa esimerkissä hyödynnetään aikaisempaa laatikkotekstuurikoodia täydentämällä sitä siirto- ja pyöritystoiminnoilla.

```
while (piirretaan) {
  glClearColor (0.0, 0.0, 0.0, 0.0);
  glClear (GL_COLOR_BUFFER_BIT);

  glEnable  (GL_TEXTURE2D);
  glBindTexture (GL_TEXTURE2D, tekstuurinTunnus);

  glLoadIdentity ();
  glTranslatef (0.0, 0.0, -3.0);
  glRotatef (kulma, 0.0, 1.0, 0.0);

  glBegin (GL_TRIANGLES);
    glTexCoord2f (0.0, 1.0); glVertex3f (-1.0,  1.0, 0.0);
    glTexCoord2f (0.0, 0.0); glVertex3f (-1.0, -1.0, 0.0);
    glTexCoord2f (1.0, 0.0); glVertex3f ( 1.0, -1.0, 0.0);
  glEnd ();

  nayttopuskurinPaivitys ();
}
```

Vain muutamalla lisärivillä saadaan taas ihmeitä aikaiseksi. Ensimmäisenä suurennuslasin alle otetaan *glLoadIdentity*-funktio. Nimensä mukaisesti se lataa identiteettimatriisin. Käytännössä tämä tarkoittaa siirtojen ja pyöritysten nollausta ja paluuta lähtöasetelmiin hieman samaan tyyliin kuin ruuduntyhjennyksen yhteydessä. Itse matriiseihin sekä niiden ideaan OpenGL:n osana palataan tuota pikaa.

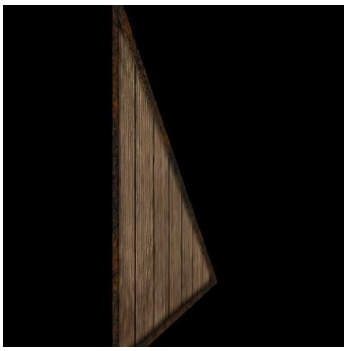
Seuraavana käsittelyyn pääsee *glTranslatef*-funktio, jonka avulla voi ”siirtää” piirrettävän kohteen uuteen sijaintiin avaruudessa ennen varsinaista piirtotoimenpiteitä. Parametreista z-komponentille on annettu arvo *-3.0*, joten kohde piirretään kolme yksikköä ruudun ”sisään.” Vastaavasti verteksien z-komponentti piirron yhteydessä on nollattu, sillä muuten kokonaissyvyydeksi tulisi *-6.0*.

Itse polygonin pyöritys tapahtuu *glRotatef*-funktiolla. Ensimmäisenä parametrina annetaan haluttu kääntökulma asteina. Seuraavilla kolmella parametrilla asetetaan yksikkövektori, joka osoittaa pyörimisakselin suunnan. Esimerkiksi tässä tapauksessa vektori osoittaa suoraan ylöspäin, joten pyöriminen tapahtuu sivuttain y-akselin ympäri.

Seuraava kuvasarja havainnollistaa esimerkkikoodin mukaista tilannetta kolmella eri pyörityskulmalla.


Kuva 9. Kierto 10°


Kuva 10. Kierto 45°


Kuva 11. Kierto 70°

Kierto on toteutettu asettamalla *Rotatef*-funktion *kulma*-parametriksi joko *10*, *45* tai *70* astetta.

Kiertosuunta on helppo selvittää kotikonstein asettamalla ensiksi vasemman käden sormet koordinaattiakseleiden suuntaisesti. Peukalo sojottaa ylöspäin y-akselin suuntaan, etusormi oikealle x-akselin suuntaan ja koukistettu keskisormi itseään päin z-akselin suuntaan. Seuraavaksi tartutaan oikealla kädellä akselia vastaavasta vasemman käden sormesta peukalon osoittaessa kyseisen akselin positiiviseen suuntaan. Oikean käden sormien kiertyminen sormen ympäri vastaa positiivista kiertosuuntaa sitä vastaavan akselin ympäri.

Edellä esitetty esimerkki on hyvin pelkistetty katsaus suhteellisen vaativaan aihealueeseen. Pyöritysten kanssa touhuaminen on kuitenkin hyvin olennainen osa 3D-maailman rakennusta ja erityisen haastavaa suurten vuorovaikutteisten kokonaisuuksien yhteydessä. Ymmärtäminen vaatii hyvää hahmotuskykyä, ja oiva tapa oppia on-

kin yhdistelmä kantapään kautta kokeilua sekä teorian sulattelua sopivan kokoisina paloina.

Matriisit ovat OpenGL:n tapa pitää itsensä ajan tasalla avaruuden pyörityksistä ja liikehdinnöistä. Käytännössä matriisit ovat vain säilytyspaikkoja parametreille, joita käytetään rasterointilaskelmissa lopullisen ruudunpiirron yhteydessä. Dataa säilytetään matriisimuodossa oikeastaan vain tiettyjen laskutoimitusten ja muun käsittelyn helpottamiseksi. Peruspiirrossa matriisien toimintaan ei juuri tarvitse syventyä, mutta muutama asia on kuitenkin hyvä tietää.

Projektiomatriisin *(eng. projection matrix)* ehkä tärkein tehtävä on vastata ruudulle oikein piirtyvistä kokosuhteista. Esimerkiksi edellisissä koodiesimerkeissä käytetyt avaruuskoordinaattien arvot ovat täysin suhteellisia, eivätkä vastaa konkreettisia ruudun pikseleitä kaksiulotteisen maailman tavoin. Näin ollen kokosuhteiden määrittely on aina toteutettava erikseen. Projektiomatriisi ylläpitää myös piirtoetäisyyden rajoja.

Hankalasti suomennettava modelview-matriisi *(eng. modelview matrix)* vastaa katselukulmista ja etäisyyksistä. Edellisen pyöritysesimerkin kaltaiset etäisyys- ja kiertokulmamuutokset vaikuttavat nimenomaan modelview-matriisin arvoihin. Yksinkertaisen piirron yhteydessä matriisin toiminnallisuus ja käyttö jää kutakuinkin näkymättömäksi.

OpenGL-termistössä puhutaan usein kahdesta eri koordinaatiosta. Absoluuttiset maailmakoordinaatit *(eng. world coordinates)* kertovat sijainnin täsmälleen ruudun suhteen origon sijaitessa näytön keskellä. Jos esimerkiksi kohde sijaitsee positiivisella puolella z-akselia, ruudulle piirtyminen ei tällöin onnistu kohteen sijaitessa ”kameran takana.” Sivuttaisliike noudattaa samaa ideaa. Näkökentän leveys ja korkeus riippuu muun muassa projektiomatriisin asetuksista.

Suhteellinen objektikoordinaatisto *(eng. object coordinates)* taas liikkuu ja pyörii absoluuttiseen koordinaatistoon nähden. Tämä mahdollistaa esimerkiksi ”kameran” liikkumisen piirretyssä ympäristössä. Nimensä mukaisesti objektikoordinaatteja käytetään useimmiten objektien piirtämiseen tiettyyn sijaintiin pelimaailmassa. Model-

view-matriisin avulla kyetään laskemaan kohteen uusi sijainti ja katselukulma absoluuttisessa koordinaatistossa.

Kaiken kaikkiaan matriisien kanssa touhuilu on jokseenkin konstikasta ja vaatii monessa tapauksessa syvällistä paneutumista aiheeseen. Alkuun pääsee toki vähemmälläkin, mutta matriisien toiminnallisuuden yksityiskohtaisempi ymmärrys ja hyödyntäminen vaatii hyvän matemaattisen peruspohjan.

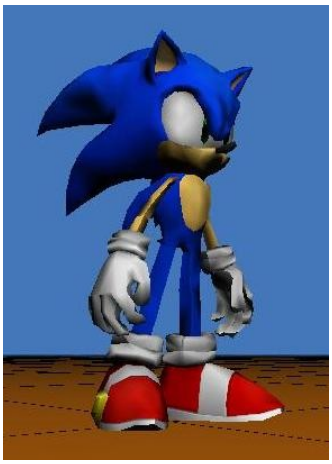
3.6 3D-mallit

Edelliset esimerkit luovat perustan jo kohtalaisen vaativillekin operaatioille. Kysehän on pohjimmiltaan verteksien ja tekstuurien asettelusta oikeaan aikaan oikeaan paikkaan, sekä valaistuksen ja matriisien vaatimusten mukaisesta manipuloinnista. Itse verteksit, jotka siis muodostuvat 3D-avaruuden kolmesta komponentista, säilötään useimmiten valmiisiin taulukoihin tai vektoreihin helpottaen niiden käyttöä polygoneja muodostettaessa. Myös tekstuurien koordinaatit löytyvät taulukosta, jotka polygonia luotaessa liitetään niitä vastaavin vertekseihin.

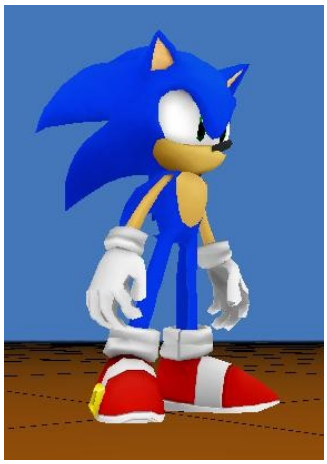
3D-mallit lienevät yleisin tapa hyödyntää taulukoitua verteksidataa. Tarkasti ennalta määrätyistä pisteistä muodostuva polygonikokonaisuus voi olla hyvinkin monimutkainen ja näyttävä totellen kuitenkin täysin samoja komentoja ja lakeja, kuin yksinkertaisimmatkin peruspolygonit. Staattisen verteksiverkon sijaan moni 3D-malli koostuu useasta erillisestä kuvakehyksestä *(eng. frame)*, joita oikea-aikaisesti peräkkäin piirtämällä luodaan haluttu animaatio. Jokainen kuvakehys koostuu siis omasta verteksijonosta odottaen omaa piirtovuoroaan.

3D-mallien data on tallennettu vakiotyyppisiin tiedostoihin, joidenka rakenteen ymmärtämällä voi verteksien ja tekstuurien tiedot ladata muistiin itse piirtodatan luomiseksi. Mallitiedostot ovat usein verrattain monimutkaisia ja vaativatkin kärsivällistä perehtymistä ennen käyttöönottoa.

Seuraavissa esimerkkikuvissa komeilee vanhahtavaan md2-tiedostomuotoon perustuva 3D-malli. Toinen kuvista näyttää mallin ilman valaistusta havainnollistaakseen valolaskelmien merkitystä pintojen syvyydelle ja muotojen pyöreydelle.



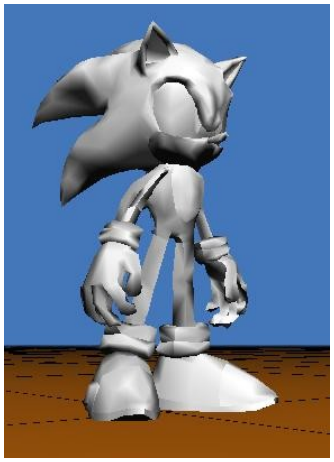
Kuva 12. 3D-malli valaistuksella



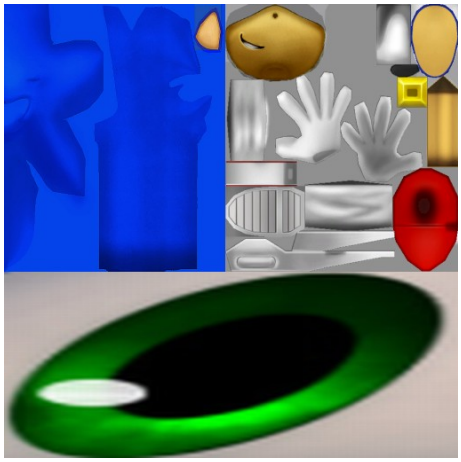
Kuva 13. 3D-malli ilman valaistusta

Tekstuurien luonti 3D-mallille seuraa täysin samoja periaatteita, kuin aikaisemmissa esimerkeissä. Itse kuvadata tekstuurille muodostetaan vaatimusten mukaisesta tiedostosta, joka koodiin toiminnallisuudesta riippuen voi olla esimerkiksi bmp-kuva. Piirron yhteydessä jokaiseen verteksiin liitetään md2-tiedoston ohjeistuksen mukaiset tekstuurikoordinaatit.

Seuraavat kuvat esittelevät edellisen 3D-mallin ilman tekstuureja sekä tekstuurien luomiseen käytetyn 24-bittisen bmp-kuvatiedoston.



Kuva 14. 3D-malli ilman tekstuureja



Kuva 15. 3D-mallin pintakuvioinnissa käytetty tekstuurikuva

Mallien luonti toteutetaan käytännössä aina erillisellä piirto-ohjelmalla. Erityyppisiin tiedostomuotoihin tallennettu data liitetään osaksi rakennettavan sovelluksen resurssitiedostoja, josta se on helposti ladattavissa osaksi ohjelmaa ja sen visuaalista toiminnallisuutta. Nykypäivän 3D-mallit ovat erittäin korkeatasoisia työllistäen juuri ulkoasusuunnitteluun ja animaatioihin erikoistuneita kehittäjiä. (Mallien lähde: Jeck Jims)

3.7 Jaottelu ja luokat

Olio-ohjelmoinnin tarjoamaa toiminnallisuuden kapselointia kannattaa pyrkiä noudattamaan mahdollisimman pitkälle. Selkeä jako vähänkään monimutkaisemmissa projekteissa helpottaa huomattavasti kokonaisuuksien hallintaa ja mahdollistaa saumattomamman jatkokehityksen. Mitä itsenäisempiä osioita kykenee rakentamaan sitä kivuttomampaa niiden hyödyntäminen todennäköisesti tulee olemaan.

Hyvän jaottelun toteutus ei kuitenkaan ole täysin ongelmatonta. Ensinnäkin sen suunnittelu vaatii runsaasti aikaa ja kärsivällisyyttä. Molemmista on valitettavan usein huutava pula. Suoraviivaiset ja nopeat ratkaisut ovat monesti huomattavasti houkuttelevampia, kuin alituinen veivaaminen luokkasuhteista ja toiminnallisista näkyvyyksistä.

Toinen kompastuskivi, joka koskee erityisesti hybridiluoteista C++-kieltä, löytyy käytettävistä rajapinnoista. Opinnäytetyön pohjana käytetyssä *Project3D:ssä* sekä käyttöliittymätoiminnallisuuden tarjoava Win32 API että grafiikkarutiineista huolehtiva OpenGL ovat puhtaasti proseduraalisen C-kielen tuotoksia. Näiden toimintojen sovittaminen olioajatteluun ja kapselointiin vaatii helposti kompromissin jos toisenkin.

Esiin on syytä nostaa myös suorituskyvylliset tekijät. Luokkarakenteita ja -suhteita suunniteltaessa on toimivan jaottelun lisäksi hyvä ottaa huomioon rakenteiden raskaus ja kuormitus. Esimerkiksi huolimaton muodostinten viljely saattaa huomattavasti lisätä laskentataakkaa, vaikka itse koodi ja sen toiminnallinen toteutus muuttuisikin entistä selkeämmäksi.

3.8 Matemaattista pohdintaa

Kolmiulotteisen maailman rakentaminen ja käsittely ilman geometrian ja algebran hyvää perusosaamista voi monessa tapauksessa muodostua ylitsepääsemättömäksi haasteeksi. Peruspiirto onnistuu vielä hatarammallakin laskentataustalla, mutta vaativamman toiminallisuuden yhteydessä edellyttää jo vankkaa matemaattista pohjaa.

Vektorilaskenta ja avaruuskoordinaatiston käsite luo perustan kaikelle 3D-toiminnallisuudelle. Peruslaskutoimitusten lisäksi on syytä hallita niin piste- kuin ristitulojen käyttö ja soveltaminen. Vastaan tulee väistämättä tarve myös sini-, kosini- ja tangenttifunktioille, joten trigonometrian tietotaito on syytä olla tiukasti hyppysissä.

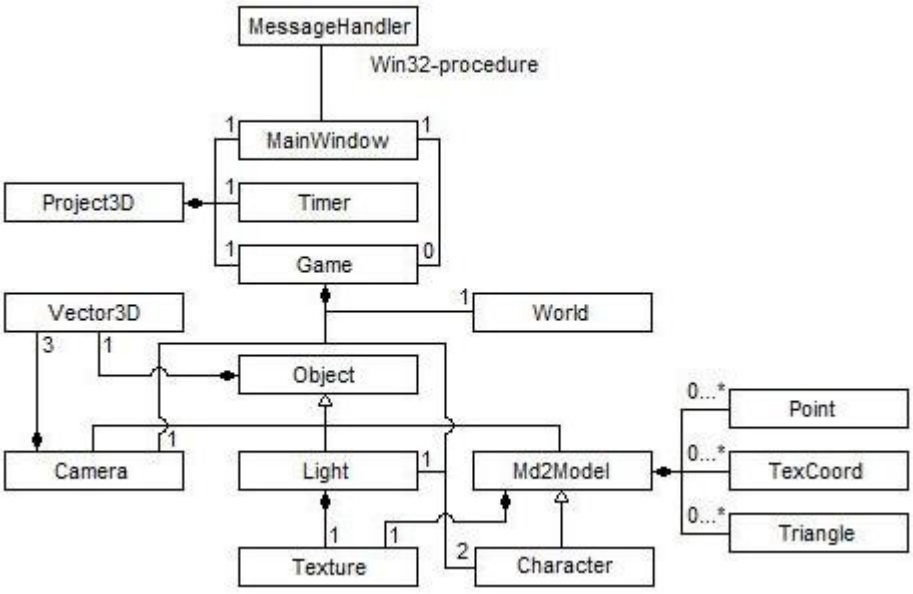
Matriisien toiminta ja niihin liittyvät laskutoimitukset on hyvä osata. Moni OpenGL:n toiminnallisuus liittyy matriisien välisiin laskutoimituksiin. Vaikka tämä toiminta on ohjelmoijalle useimmiten näkymätöntä, moni tilanne vaatii suoraa vuorovaikutusta matriisien kanssa edellyttäen laskusääntöjen hallintaa.

4 PROJECT3D

Lähempi tarkastelu itse pohjamateriaalina toimivaan lyhyeen 3D-demoon pyrkii valaisemaan edellä käsiteltyjä aihekokonaisuuksia käytännön toteutuksesta käsin.

Toteutettu sovellus on yksinkertainen kolmiulotteinen maailma, jossa käyttäjä kykenee ohjaamaan md2-tiedostojen pohjalta mallinnettuja objekteja. Ohjelma hyödyntää muun muassa valaistusta, 3D-mallien animointia sekä matriisien kontrolloinnilla toteutettua kameran ja pelihahmojen sulavaa liikehdintää.

Tarkastelu toteutetaan tutkimalla luokkien tarjoamia toimintoja ja palveluita. Aikaisemmin kuvatut haasteet selkeästä toiminnallisuuden jaottelusta, sekä kompromisseihin taipumisesta olosuhteiden pakosta ovat jatkuvasti läsnä. Moni asia kaipaisikin hiomista tai jopa kokonaisten rakenteiden uudelleensuunnittelua. Aikataululliset tekijät ovat kuitenkin osaltaan pakottaneet moniin hyvinkin karkeisiin ratkaisuihin. Itse luokkajakoa on selvitetty seuraavassa kuvassa.



Kuva 16. Project3D:n luokkajako

4.1 Tapahtumien käsittely

Win32-rajapinta asettaa tiettyjä ehtoja kirjoitettavan ohjelman rakenteesta. Tapahtumien käsittely, kuten esimerkiksi käyttäjän syötteiden seuranta, on toteutettava vaatimusten mukaisella proseduurilla. Käytännössä tämä tarkoittaa muusta luokkahierarkiasta erillään olevaa funktiota, joka vastaanottaa ja käsittelee käyttöjärjestelmän viestijonoon lähettämät viestit. Itse viestien sisällöstä riippuva lopullinen tapahtumankäsittely ja toiminnallisuus toteutetaan kutsumalla käyttöliittymäikkunasta vastaavan MainWindow-luokan operaatioita.

4.2 Project3D-luokka

Toimii ylimmällä tasolla huolehtien ohjelman perusalustuksista ja pelisilmukan pyörittämisestä. Toteuttaa myös asianmukaiset purkutoimenpiteet ohjelman päättyessä.

Itse olio koostuu käyttöliittymäikkunasta vastaavasta MainWindow-oliosta, pelitapahtumista vastaavasta Game-oliosta sekä ajanseurantarutiineista vastaavasta Timer-oliosta. Alustusoperaation tehtävänä onkin huolehtia jokaisen osaolion valmistelusta varsinaiseen pelisilmukan ajoon. Itse silmukka on jaettu alussa esitetyn esimerkin mukaan viestien kuunteluun ja pelitilanteen päivittämiseen.

Kuuntelutoiminnallisuuden yhteydessä nojataan vahvasti Win32-proseduuriin. Käyttöjärjestelmän havaitessa peli-ikkunaan kohdennetun tapahtuman, kuten käyttäjän näppäinsyötteen, asettaa se kyseistä tapahtumaa vastaavan viestin viestijonoon, joka puolestaan tarkastetaan pelisilmukan jokaisella kierroksella. Viestin osuessa kohdalle se lähetetään tapahtumankäsittelijälle, joka päättä jatkotoimenpiteistä kutsumalla MainWindow-olion operaatioita.

Pelitilanteen päivittäminen on taas paljolti riippuvainen kuluneesta ajasta. Päivitys pyritään pitämään mahdollisimman lähellä näyttölaitteen virkistystaajuutta. Kriittiseksi tekijäksi nousee päivitysjaksojen yhdenmukaisuuden puute aiheuttaen pelitilanteen nykimistä. Sovellukseen on asetettu mahdollisuus vaakasynkronoinnin käyt-

töönottoon tarkoituksenaan vähentää tökkivää ruudunpäivitystä. Tämä toisin ei toimi kaikkien näytönohjainkorttien yhteydessä.

Ohjelman päättyessä on tärkeää vapauttaa kaikki tarvittavat resurssit. Tämä on toteutettu erillisellä lopetusoperaatiolla. Erityisesti käyttöliittymäikkunan tuhoaminen on suoritettava huolella sujuvan lopetuksen takaamiseksi.

4.3 Timer-luokka

Luo edellytykset ajan tarkkaan mittaamiseen. Win32-rajapinnan tarjoaman korkean resoluution laskurin avulla on mahdollista toteuttaa pelitilanteen päivitysten hallinnassa vaaditut aikaparametrilaskelmat.

Alustuksissa selvitetään laitteiston ja käyttöjärjestelmän tarjoama ajanottoresoluutio. Näin kellopulssien lukumäärällä toteutettu ajanotto kyetään muuttamaan millisekunneiksi. Itse ajanoton alkaessa nollataan aloitusaika. Aloituksesta kulunut aika saadaan selville erillisellä operaatiolla ja viimeksi tiedusteltu aikaleima on mahdollista asettaa uudeksi aloitusajaksi. Tämän kaltaista rutiinia tarvitaan esimerkiksi kontrolloitaessa pelisilmukan päivitysnopeutta.

4.4 MainWindow-luokka

Project3D-olion osaolio, jonka vastuulla on käyttöliittymäikkunan toteutus toimintoineen. Win32-rajapinnalla toteutetun ikkunanluonnin yhteydessä olennaisena osana on OpenGL:n sitominen ikkunan piirtoalueelle *(eng. client area)*. Käytännössä tämä sallii ikkunaan kohdistettujen piirtorutiinien suorittamisen OpenGL-funktioilla.

Ikkunarutiinien lisäksi MainWindow-oliolla on tärkeä rooli käyttäjän syötteiden vastaanotossa. Win32-rajapinnan edellyttämä tapahtumankäsittelijä kutsuu käyttöjärjestelmän lähettämän tapahtumaviestin sisällön perusteella sitä vastaavaa MainWindow-olion operaatiota. Käytännössä syötetapahtumat tallennetaan luokkamuuttujiin myöhempää käyttöä varten pelitilanteen päivitysten yhteydessä.

Toiminnallisuus kattaa myös tekstin kirjoittamisen ruudulle vakiofontilla. Kirjainten koko sovitettaan ruudulle sopivaksi ottamalla huomioon sen hetkisen ikkunan koko. Itse kirjoittaminen toteutetaan antamalla ikkunakoordinaatit ja tulostettava teksti Game-olion toteuttamien piirtorutiinien yhteydessä .

Mahdollisuus vaakasynkronoinnin käyttöön saattaa helpottaa ruudunpäivityksen yhteydessä havaittavaa nykimistä. Osa näytönohjainkorteista ei toiminnallisuutta kuitenkaan tue koodissa ajatellulla tavalla tehden liikkeestä mahdollisesti entistä pätkivämpää.

Luokka on kokonaisuudessaan kohtalaisen iso ja kattaa toiminnallisuuksia, joidenka parempi kapselointi ja jäsentely erillisiin luokkiin olisi selkeästi nykyistä parempi ratkaisu.

4.5 Game-luokka

Vastaa itse pelistä ja sen sisällöstä. Olio koostuu joukosta osaolioita, joidenka toiminnallisuudesta koko päivitettävä pelitilanne rakentuu. Näihin lukeutuu muun muassa käyttäjän kontrolloimat pelihahmot.

Alustusoperaatioiden päätehtävänä onkin osaolioiden valmistelut asianmukaisilla parametreilla. Nämä käsittävät muun muassa sijaintitietoja sekä toimintaan ja animointiin liittyviä yksityiskohtia. Myös OpenGL-asetukset viritetään perustilaan omalla valmisteluoperaatiolla. Oletusparametreja muuttamalla saadaan pelitilanteen päivitysten yhteydessä erityyppisiä piirtotoiminnallisuuksia.

Itse pelitilanteen päivitys on jaettu kahteen erilliseen osioon. Loogisten päivitysten yhteydessä kulunutta aikaa ja käyttäjän syötteitä käytetään peliobjektien parametrien päivittämiseen. Toinen osio vastaa itse pelitilanteen piirrosta perustuen aiemmin tehtyihin loogisiin päivityksiin. Myös tekstin kirjoitus ruudulle toteutetaan pelitilanteen piirron yhteydessä. Tähän tosin käytetään kahvana saadun MainWindow-olion kirjoitusoperaatiota.

Ehkä tärkein yksittäisistä peliluokan osasista on Camera-olio. Tämän avulla pelitilanne saadaan piirtymään oikeasta kulmasta ja oikealta etäisyydeltä. Luokka siis manipuloi OpenGL-matriiseja käyttäjän hiirenliikkeiden mukaan luoden illuusion avaruudessa liikkuvasta kamerasta.

World-olion tehtävänä on piirtää yksinkertainen peliympäristö täyttämään tyhjää avaruutta. Game-olion piirtopäivitysten yhteydessä kutsuttu operaatio toteuttaa käytännössä lattian ja sumuefektin piirron.

Pelihahmot ovat toiminnallisuuden suola. Toteutukseen on käytetty Character-luokkaa, joka taas periytyy Md2Model-luokasta. Pelihahmot liikkuvat käyttäjän syötteiden mukaan ominaisuuksiensa rajoissa, ja Camera-olion päivittämällä perspektiivillä kyetään seuraamaan valittua pelihahmoa.

Pelialueen valaistus on toteutettu Light-oliolla. Loogista valonlähdettä on täydennetty piirtämällä tuikkiva tekstuuri sen absoluuttiseen sijaintiin. Seuraava kuva esittää peliympäristöä hahmoineen ja valonlähteineen.

Kuva 17. Project3D:n ympäristö

4.6 Object-luokka

Yliluokka kaikille pelitilanteen muodostaville olioille. Poikkeuksena tästä on World-olio, jonka pääaisallisena tarkoituksena on nopean ja yksinkertaisen peliympäristön luonti.

Alkuperäisen tarkoituksen mukainen yhteisten toimintojen ja ominaisuuksien niputus Object-luokkaan ei ole täysin toteutunut. Käytännössä vain sijainti avaruudessa sekä ohjelmamoduulin fyysinen sijainti tiedostopolkuna ovat riittävän yleisluontoisia sopiakseen yliluokan vastuulle. Rakenteiden muuttaminen ja kehittäminen saattaisi tosin tuoda lisäarvoa periytymistä tukeville suunnittelulähtökohdille.

Ominaisuuksista sijainti lienee kohtalaisen itsestään selvä. Ohjelmamoduulin hakemistopolku sen sijaan kaivannee hieman avaamista. Kyseistä ominaisuutta tarvitaan resurssitiedostoja etsittäessä peliobjektien luonnin yhteydessä. Käytännössä tieto ohjelmamoduulin, eli ajettavan exe-tiedoston, sijainnista sallii sovelluksen ajamisen muualtakin, kuin pelkästään työhakemistosta.

4.7 Md2Model-luokka

Käytetään pelitilanteen osana toimivan olion rakentamiseen 3D-mallista sekä toteuttamaan samaisen olion piirtorutiinit. Käytännössä tämä tarkoittaa olion luomista ladatuista md2- ja tekstuuriresurssitiedostoista sekä rakennetun 3D-mallin piirtämistä ruudulle päivitettyjen parametrien mukaisella tavalla.

Md2-mallitiedoston lataus ja käsittely on jokseenkin mutkikasta ja vaatii huolellista perehtymistä. Tärkeintä on ymmärtää tiedoston rakenne, jotta kaikki tarvittava informaatio saadaan oikeassa muodossa muistiin mahdollistaen virheettömät piirtorutiinisuoritukset. Kuten jo aikaisemmin on todettu md2-tiedosto koostuu lähinnä verteksien sijaintitiedoista avaruudessa sekä niihin liitetyistä tekstuurikoordinaateista. Käytännössä malli koostuu useammasta kuvakehyksestä, joita oikeanaikaisesti peräkkäin piirtämällä luodaan vaatimusten mukainen animaatio.

3D-mallin latauksen ja rakentamisen ohella olio siis huolehtii myös piirtotoiminnallisuuden toteutuksesta, joka niin ikään nojaa verrattain monimutkaiseen toteutukseen. Tarkoituksena on piirtää kulloinkin piirtovuorossa olevaa kuvakehystä vastaava verteksijono ruudulle tekstuureineen. Toteutus tapahtuu muodostamalla aina kolmesta verteksistä kolmion muotoisia polygoneja, jotka yhdistettynä tekstuuri- ja normaaliinformaatioon piirretään ruudulle. Perusperiaatteeltaan lopullinen piirto-operaatio noudattaa samoja sääntöjä, kuin alussa esitetyt yksinkertaiset polygonienpiirtoesimerkit.

4.8 Character-luokka

Laajentaa Md2Model-luokkaa lisäämällä pelihahmokohtaisia operaatioita 3D-malliin. Käyttäjän syötteistä riippuen kyseinen pelihahmo voi joko seisoa, juosta tai hypätä. Tilakoneena toteutettu toimintojen ajaminen käsitellään pelitilanteen loogisten päivitysten yhteydessä, jolloin kutsuttavan operaation lopputulos riippuu käyttäjän napinpainallusten lisäksi myös kuluneesta ajasta.

Jokaiseen pelihahmon ulkoiseen toimintoon ja tilaan liittyy sitä vastaava animaatio. Sulavuus ja oikea-aikaisuus muodostuvat visuaalisen annin kannalta ratkaisevaksi. Animointi asetetaan jokaiselle toiminnolle erikseen osoittamalla sille näytettävä kuvakehyssarja sekä animointinopeuteen liittyvä aikaparametri.

Md2-mallien pohjalta luodut animaatiot perustuvat avainkuvakehyksiin *(eng. key frame)*. Käytännössä tämä tarkoittaa jokaisen ruudulle piirrettävän kuvakehyksen olevan laskelma kahden md2-mallitiedostosta ladatun kuvakehyksen väliltä. Päivitysajalla määrätään, kuinka kauan aikaa kuluu kahden avainkuvakehyksen piirron välillä. Animointia voi siis nopeuttaa lyhentämällä päivitysaikaa tai vastaavasti hidastaa kasvattamalla sitä. Itse laskelmat verteksien asemasta kuvakehysten välillä toteutetaan yliluokkana toimivan Md2Model-luokan piirtorutiineissa.

Character-luokassa toteutetaan hahmotoimintojen ja -animaatioiden lisäksi myös itse liikkuminen eli sijainnin muutos 3D-avaruudessa. Operaatio on kokonaisuudessaan kohtalaisen monimutkainen. Sijaintia päivitettäessä on otettava huomioon kameran

katselukulma, hahmon kulkusuunta sekä käyttäjän syötteet. Pelihahmon liikesuunta avaruudessa on riippuvainen kamerakulmasta, joten esimerkiksi eteenpäin liikuttaessa hahmo kääntää selkänsä ja liikkuu ”poispäin” ruudusta.

Kokonaisuutena hahmon tilojen sekä niihin sidottujen toimintojen toteutus vaatisi hieman selkeämpää ja säännönmukaisempaa toteutusta. Erityisesti pelitilanteen päivittämisen yhteydessä tapahtuva tilojen manipulointi kaipaisi tiettyjen rakenteiden uudistamista, joka myös osaltaan helpottaisi mahdollisten lisätoimintojen suunnittelua ja toteutusta.

4.9 Light-luokka

Vastaa valaistukseen liittyvästä kokonaisuudesta, joka on jaettu loogisen valonlähteen asettamiseen sekä visuaalista valonlähdettä kuvaavaan tekstuurin piirtämiseen. OpenGL sisältää sarjan valmiita valonlähteitä, joita hyödyntämällä lasketaan pintojen väriarvoja valaistuksen ollessa aktiivinen.

Light-olion alustuksissa otetaan käyttöön ensimmäinen vapaa looginen valonlähde. Valon värin voi myös vaihtaa oletuksena toimivasta valkoisesta haluttuun väriarvoon. Alustuksiin liittyy myös valonlähdettä kuvaavan tekstuurin lataus. Jotta muiden väriarvojen käyttö kohdistuisi oikein myös valotekstuuriin, on mukaan ladattava myös tekstuuria vastaava maski.

Valonlähdettä päivitetään piirtämisen yhteydessä. Looginen sijainnin lisäksi on asetettava valotekstuuri samoihin koordinaatteihin. Jotta visuaalinen illuusio staattisesta ja pyöreästä valonlähteestä toteutuisi littanan 2D-kuvan sijaan, on suoritettava muutamia matriiseihin liittyviä toimenpiteitä. Näin valotekstuuri piirtyy aina samansuuntaisesti katsojaan päin kamerakulmasta riippumatta. Itse piirtotapahtuma toteutetaan piirtämälle ensin maski valotekstuurista, jonka jälkeen haluttuihin väriarvoihin asetettu varsinainen kuva on mahdollista piirtää virheettömänä ruudulle.

4.10 Camera-luokka

On vastuussa pelitilanteen näyttämisestä oikeasta perspektiivistä. Käytännössä tämä tarkoittaa jatkuvaa modelview-matriisin päivitystä käyttäjän syötteiden mukaisesti.

Kamera asetetaan seuraamaan aina tiettyä hahmoa peliavaruudessa. Pelitilanteen loogisten päivitysten yhteydessä toteutetut sijaintipäivitykset riippuvat siis pelihahmon liikkeistä, joihin Camera-luokka pääsee käsiksi saamallaan pelihahmokahvalla. Laskelmat kameran lopullisen sijainnin ja katselukulmaan määrittämiseen ovat jossain määrin monimutkaisia ja vaativat hyvää hahmotuskykyä sekä trigonometrian hallintaa.

Pelitilanteen piirtopäivitysten yhteydessä toteutetaan kaksi erillistä toimintoa. Ensimmäisessä asetetaan kamera oikealle paikalle peliavaruuteen. Varsinainen matriisin manipulointi tapahtuu siis piirtojen yhteydessä käyttäen loogisissa päivityksissä laskettuja parametreja.

Toinen huolehtii näkökentän rajojen selvittämisestä. Tämä mahdollistaa pelihahmojen ja muiden peliobjektien tarkemman piirtokontrollin. Objektien piirtorutiinit jätetään piirtopäivitysten yhteydessä suorittamatta, jos kyseistä kohdetta ei näy kameran silloisesta perspektiivistä. Kuormitus vähenee huomattavasti erityisesti paljon piirrettäviä peliobjekteja sisältäviä piirtorutiineja suoritettaessa.

4.11 Texture-luokka

Luo ja ylläpitää pelitilanteen piirrossa tarvittavia tekstuureja. Jokainen luokan instanssi on vastuussa aina yhdestä tekstuurista. Yhteys tekstuureja käyttäviin olioihin on toteutettu koostumussuhteella.

Tekstuurin pystyy muodostamaan joko bmp-, pcx- tai tga-kuvasta käyttäen sitä vastaavaa kuvanlatausoperaatiota. Ladatuista kuvista erotetaan itse kuvadata ja asetetaan oikeaan muotoon OpenGL-tekstuurin luontia varten. Lopputulosta käytetään tekstuurikoordinaattien avulla piirrettävien polygonien pinnoissa.

4.12 World-luokka

Vastaa peliympäristöön liittyvistä tapahtumista ja piirroista. Luokka on tehty nopeasti tarkoituksenaan luoda jonkin sortin miljöö tyhjän avaruuden sijaan ja toiminnot ovat kohtalaisen karkeasti ja suoraviivaisesti toteutettuja.

Toiminnallisuus on jaettu käytännössä kahteen osioon. Rajojen tarkastuksen tehtävänä on pitää pelihahmot rajatulla alueella. Tarkastelu toteutetaan loogisten päivitysten yhteydessä. Piirtotoiminnoissa keskitytään taas itse ympäristön luontiin. Ruudullisesta lattiasta ja sumuefektistä muodostuva piirtokokonaisuus luo toiminnallisuuden kannalta riittävästi visuaalista höystettä.

4.13 Vector3D-luokka

Alun perin luokan tarkoituksena oli tarjota helppo käyttöliittymä vektorien käsittelyyn ja laskentaan. Kolmesta avaruuden komponentista muodostuvia vektoreita olisi ylikuormitettujen operaattoreidensa ansiosta helppo käsitellä. Operaattoreiden hyödyntäminen myös lyhentäisi tarvittavan koodin määrää.

Ongelmaksi kuitenkin osoittautuivat suorituskyvylliset tekijät. Vector3D-luokan instanssien käyttö laskuoperaatioissa itsessään aiheutti monessa tapauksessa ylimääräisten muodostinten kutsumista. Kriittisissä toiminnoissa, kuten esimerkiksi piirtorutiineja suoritettaessa, muodostimet aiheuttivat selvää kuormitusta ja suorituskyvyn laskua pelisilmukan kierrosnopeuksissa mitattuna.

Malliprojektin kaltaisen yksinekertaisen ohjelman pyöritys ei nykykoneille kynnykseksi tietenkään nouse. Lopullinen päätös kuormitettujen operaattoreiden hylkäämisestä johtuikin lähinnä yleisistä suunnittelukriteereistä, joihin lukeutuu selvien pullonkaulojen välttäminen aina suunnitteluaikataulun sen salliessa.

Avaruusvektorilaskennat niitä vaativissa operaatioissa suoritetaan nykyisessä ohjelmaversiossa pääosin komponenteittain vakiotietotyyppejä käyttäen, joka onkin suorituskyvyllisistä lähtökohdista osoittautunut paremmaksi ratkaisuksi, mutta on toisaalta

myös kasvattanut koodin määrää. Vector3D on edelleen käytössä vähemmän kriittisissä toiminnoissa, joissa suorituskyvyn maksimointi ei ole ensisijaisen tärkeää. Käytännössä koko luokkaa on riisuttu aputietueeksi, jolla korvataan komponenttitaulukko tai erilliset muuttujat. Sinänsä Vector3D-luokka nykyisessä kokoonpanossa on ehkä jokseenkin turha ja lähinnä jäänne aikaisemmista versioista.

4.14 Point-luokka

Jokainen tämän luokan instanssi sisältää polygonin piirtoon tarvittavan pisteen verteksi- ja normaali-informaation. Verteksin muodostuessa paikkavektorista ja normaalin suuntavektorista on molemmat säilötty kolme komponenttia sisältävään Vector3D-olioon. Md2-mallin latauksen yhteydessä Point-olioista muodostetaan taulukko, jonka sisältöä käytetään rakennettaessa kuvakehyksiä ruudulle.

4.15 TexCoord-luokka

Luokan instanssit säilövät polygonien piirron yhteydessä tarvittavat tekstuurikoordinaatit. Verteksi- ja normaali-informaation tavoin ne ladataan taulukkoon md2-pohjaista 3D-mallia luotaessa. Piirtorutiinien yhteydessä nämä liitetään niitä vastaaviin pisteisiin lopullisia polygoneja muodostettaessa.

4.16 Triangle-luokka

Käytetään itse ruudulle piirrettävien polygonien rakentamiseen hyödyntäen Point- ja TexCoord-olioita. Myös Triangle-luokan instansseille löytyy oma taulukko, johon informaatio ladataan md2-tiedostosta. Itse data muodostuu indekseistä piste- ja tekstuurikoordinaattitaulukoiden alkioihin. Triangle-luokka siis toimii tietynlaisena muottina piirrettävälle polygonille, johon verteksit, normaalit ja tekstuurikoordinaatit asetellaan indeksoidiuista taulukoista.

5 YHTEENVETO

Kokonaisuudessaan projektin kanssa pakertaminen on ollut erittäin opettavaista ja silmiä avaava. Testipenkkiin on joutunut niin koulussa opittu perusohjelmointiosaaminen kuin itsehankitun peli- ja grafiikkaohjelmointitiedon käytännön soveltaminen.

Ehkä suurin haaste on liittynyt kokonaisuuteen ja sen hallintaan. Kovin suuria projekteja ei ennen ole tullut vastaan, joten eri palasten järkevä jäsentely ja kontrollointi on ollut yllättävän aikaa vievää. Vaikka monet toiminnallisuudet perustuvatkin erinäisiin oppimateriaaleihin, ovat monet niissä esiintyvät esimerkit hyvin suoraviivaisesti toteutettuja eivätkä sellaisenaan ole soveltuneet ohjelman kokonaisuuteen. Materiaalit sisältävät myös paljon projektin kannalta täysin epäolennaista ja hyödytöntä rönsyilyä, joten rusinoiden tarkka tunnistaminen muusta pullamassasta on ollut ensiarvoisen tärkeää.

OpenGL:n perustuntemus jo projektin alkuvaiheissa on helpottanut monien rutiinien kehittämistä ja toteuttamista. Toiset toiminnot ovat taas vaatineet suurempaa soveltamista ja erityisesti monet avaruudelliseen laskemiseen liittyvät toiminnot, kuten kameran ja hahmojen oikeanlainen liikehdintä, ovat vaatineet miettimishetken jos toisenkin.

Toteutetusta 3D-sovelluksesta löytyy vielä runsaasti parannettavaa. Ensimmäisenä työn alle joutaa luokkien ja niiden välisten toiminnallisuuksien rakenteet ja suhteet. Moni asia on tehty suhteellisen nopeasti ja suoraviivaisesti ajan säästämiseksi, mutta pureutumalla syvemmin yksityiskohtiin olisi mahdollista toteuttaa yleiskäyttöisempiä ja selkeämpiä kirjastoja mahdollista jatkokehitystä varten. Erityisesti joidenkin luokkien suuri koko sekä useat jäykillä koostumussuhteilla toteutetut yhteydet pitäisi vasaroida uuteen uskoon. Myös tilakonerakenteet vaatisivat kasvojenkohotusta selkeämmän käytön ja helpomman laajentamisen nimissä.

Kosmeettisen lähdekoodikirurgian lisäksi projektilla riittää laajennettavuutta myös muihin ulottuvuuksiin. Toimintojen kattaminen muun muassa törmäyksen tunnistukseen ja fysiikkamoottoriin löytyvät kehityslistan kärjestä. Myös jonkinlaisen kenttäeditorin kehitys vaikuttaisi mielenkiintoiselta haasteelta. Tähän jos lisätään vielä tekoälyrutiineja ja jonkinmoinen peli-idea, alkavat ainekset yksinkertaiselle pelille olemaan kasassa.

Suurin yksittäinen puute lienee kuitenkin poissaolollaan loistava virheidenkäsittely. Alkuperäisen suunnitelman mukainen poikkeusten hallintaan keskittyvä toiminnallisuus oli tarkoitus suunnitella viimeisenä osakokonaisuutena. Aikataulutekijät tosin pakottivat jälleen kerran oikaisemaan nurmikon poikki jättäen virheiden kanssa pähkäilyn tulevaisuuden päänvaivaksi.

Grafiikkapuolelta OpenGL tarjonnee vielä runsaasti uutta opeteltavaa. Visuaalisen kerronnan katalyytiksi on tosin syytä harkita myös muitakin työkaluja, kuten esimerkiksi Microsoftin suosittua Direct3D-rajapintaa. Laaja-alaisen kokemuksen hankkiminen on aina omiaan tukemaan tulevaisuuden kehitystavoitteita.

Kokonaisuutena projektiin voi olla kohtalaisen tyytyväinen ottaen huomioon allekirjoittaneen verrattain vähäinen ohjelmointikokemus ja vielä alhaisempi grafiikka- ja pelipuolen osaaminen. Suuntavektori sojottaakin tästä eteenpäin jyrkästi yläviiston, ja toivon mukaan uudet oppimiskokemukset tulevat innostamaan entistä määrätietoisempaan tähtien tavoitteluun.

LÄHTEET

C++ referenssisivusto. C++ kuvaus. Viitattu 23.11.2011. Saatavissa: http://www.cplusplus.com/info/description/

Hannula, T 2007. Peliala on Suomessa itseoppineiden valtakunta. Helsingin Sanomat 11.11.2007. Viitattu 23.11.2011. Saatavissa: http://www.hs.fi/talous/artikkeli/Peliala+on+Suomessa+itseoppineiden+valtakunta/1135231745067

Jeck Jims. Vapaaseen käyttöön ladattavia 3D-malleja. Viitattu 25.11.2011. Saatavissa: http://mb.srb2.org/showthread.php?t=34800/

Microsoft MSDN kehittäjäsivusto. .NET kuvaus. Viitattu 24.11.2011. Saatavissa: http://msdn.microsoft.com/library/zw4w595w.aspx

Shreiner, D 2010. OpenGL Programming Guide. 7. Painos. Pearson Education: Boston.

Sijoitusenkelit katsastivat pelialaa Suomessa. 2011. Yle uutiset 4.11.2011. Viitattu 23.11.2011. Saatavissa: http://yle.fi/uutiset/talous_ja_politiikka/2011/11/sijoitusenkelit_katsastivat_pelialaa_suomessa_3002508.html

Suomen peliala kasvamassa Pohjoismaiden suurimmaksi. 2011. Talouselämä 28.6.2011. Viitattu 23.11.2011. Saatavissa: http://www.talouselama.fi/uutiset/suomen+peliala+kasvamassa+pohjoismaiden+suurimmaksi/a648435

Työvoimapula jarruttaa pelialan kasvua. 2011. Kaleva 12.4.2011. Viitattu 23.11.2011. Saatavissa: http://www.kaleva.fi/uutiset/tyovoimapula-jarruttaa-pelialan-kasvua/895863

Wikipedia. C++. Viitattu 23.11.2011. Saatavissa: http://en.wikipedia.org/wiki/C%2B%2B