UPPSALA
UNIVERSITET

Rapport för Spelprogrammering III

3D-game med OpenGL

Johannes Westberg
Spelprogrammering III, 5SD048
Uppsala Universitet
Jerry Jonsson

Abstrakt

Introduktion: I kursen *Spelprogrammering III* i *Uppsala Universitet* ingick en uppgift att programmera ett grafiskt 3D-demo i *C++* med biblioteken *OpenGL 3.3+* eller *DirectX 11*. Rapporten beskriver processen att implementera *scene management*, *lighting*, *frustum culing*, *font rendering* och *GUI*. I arbetet användes *OpenGL*.

Metod: Förutom biblioteket *OpenGL* användes öppna bibliotek för att underlätta arbetet, bland andra *GLM*, *FreeType* och *stb_image*. Objektorienterad programmering användes för att kapsla in funktioner från *OpenGL*. *Visual Studio 2015* användes under kodningen.

Resultat: En *scene graph*-modell skapades där grafiska objekt i en 3D-scen innehöll underliggande objekt som följde med i transformationer. Ljussättning gjordes med *ambience* och *directional lighting*. *Fustum culling* implementerades vilket resulterade i snabbare rendering än före implementationen. *Font rendering* och *GUI* implementerades. Dock uppstod problem med *GUI*-knappar eftersom programmet inte tog hänsyn till relationen mellan pixelkoordinater och *OpenGL*'s rendering vid storleksändring på spelfönstret.

Diskussion och Slutsats: Resultatet visade att det fungerade att implementera de krav som uppgiften ställde. Dock innehöll vissa av kraven stora koncept utan tydliga begränsningar. Genom egna tolkningar av olika källor gick det skapa ett slutresultat som förhoppningsvis skulle ha potential i professionella sammanhang.

1 Introduktion

1.1 Bakgrund

Den här rapporten beskriver arbetsprocessen med att programmera ett påhittat 3D-spel eller grafiskt demo i språket *C++* tillsammans med *OpenGL* (version `3.3` eller högre) eller *DirectX 11*. Arbetet ingick i kursen *Spelprogrammering III* på *Uppsala Universitet*. Den utsedda arbetstiden från start till deadline var cirka fem veckor. I det här arbetet användes biblioteket *OpenGL*.

1.2 Syfte, begränsning och frågeställning

Rapportens syfte är att beskriva arbetsprocessen, dess resultat och sedan reflektera kring vad som gick bra, eventuellt mindre bra och vad som hade kunnat förbättrats. I rapporten beskrivs hur och varför olika moment gjordes. Även referenser diskuteras, det vill säga andras lösningar på specifika problem som uppstod som ibland användes i det här arbetet.

Rapporten beskriver inte hur *OpenGL* fungerar, utan förutsätter att läsaren vet vad som avses med vertriser, shaders, texturobjekt, osv. Det förutsätts att läsaren har allmän kunskap om vanliga programmatiska termer på engelska. I texten skrivs dessa i kursivt format. Likadant gäller för matematiska begrepp som finns i texten, exempelvis vektorer och punktprodukter.

Rapporten har som mål att besvara följande frågor: Hur går det att implementera *scene management, lighting*, *frustum culling, font rendering* och *GUI* med *OpenGL*?

2 Metod

2.1 OpenGL

OpenGL var ett öppet bibliotek som fungerade som ett gränssnitt till datorns grafikkort. *OpenGL* innehöll funktioner till grafikkortet för att hantera data och rendera data på datorskärmen. I arbetet inkluderades filen `glew.h`, vilken innehöll *OpenGL*'s funktioner skrivna i språket *C*.

2.2 Visual Studio

Till kodningen användes *Visual Studio 2015,* vilket var en programutvecklingsmiljö utvecklat av *Microsoft* för skapande av applikationer och mjukvaror. Programmet stödde flera programmeringsspråk, inkluderat *C++*. *Visual Studio* erbjöd verktyg för felsökning i program samt *IntelliSense* som hjälpte till att upptäcka syntaxfel och gav namnförslag på bland annat variabler och funktioner under kodningen.

2.3 Objektorienterad programmering

Objektorienterad programmering handlar ofta om att skapa dataklasser som objekt med oftast inkapslade egenskaper och beteenden. Klasserna använder privata medlemsvariabler som innefattar objektets attribut. Klasserna har metoder för att modifiera dess tillstånd eller komma åt medlemsvariablerna på ett kontrollerat sätt för att skapa beteenden. I det här arbetet användes objektorienterad metodik för att packa in *OpenGL*'s lösa funktioner i klasser.

2.4 Övriga bibliotek

För att underlätta arbetet användes *GLM* för matematiska funktioner och datatyper. *GLM*, *OpenGL Mathematics*, var ett öppet *C++* bibliotek av *G-Truc Creation*. *GLM* innehöll funktioner och data för matematik som byggde ut *OpenGL*. Biblioteket använde *namespace* `glm`.

FreeType var ett öppet bibliotek i språket *C* specifikt för att ladda in textfonter, bland annat i filformatet `.ttf`. *FreeType* användes för att kunna klara font rendering i uppgiftens kriterier.

Stb_image var ett öppet bibliotek för inladdning av bildfiler i, bland annat i formaten *JPG* och *PNG*. Syftet att använda *stb_image* var för att underlätta att ladda in bildfiler i ett format för att sedan kunna skicka vidare datan till grafikkortet med *OpenGL*.

Standard C++ Library användes för flera saker i programmet, men främst för att underlätta hantering av datalistor med hjälp av `std::vector` och `std::map`.

3 Resultat

3.1 Scene management

Scene management gjordes genom att använda en *scene graph*-metod som fanns på internet. Objekt i scener representerades med rekursiva datatypen `SceneObject`. `SceneObject` innehöll en `Transform`-typ för position, rotation och skalning, en `Mesh` och `Texture` för visuell representation och en lista med `SceneObject`. Rekursionen i `SceneObject` slutade när det var en tom lista med scenobjekt. En hel scen gjordes med datatypen `Scene` som innehöll en lista av scenobjekt.

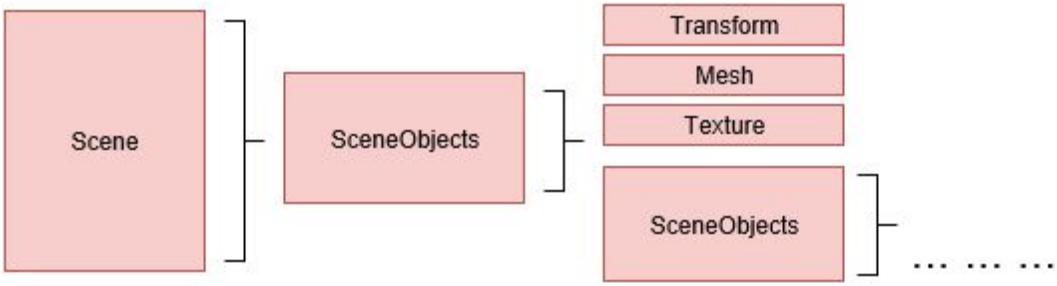

Bild 1: Bilden visar en förenklad modell på projektets scene graph.

Algoritmen för att rendera alla objekt i en scen liknades på följande sätt:

```
DrawSceneObjects(worldTransformation, sceneObjects):
      For each object in sceneObjects:
            localTransformation =
                  object.transformation * worldTransformation.
            Draw object.mesh with localTransformation.
            DrawSceneObjects(localTransformation, object.sceneObjects).
```

I sista raden från ovan sker det rekursion med `DrawSceneObjects`.

3.2 Lighting

Två olika typer av ljus användes i projektet: *ambient lighting* och *directional lighting*. *Ambient lighting* belyste alla ytor lika på varje objekt i scenen med en specifik färg och styrka. Detta oberoende objektets position, rotation eller omkringliggande objekt. *Directional lighting* belyste alla ytor på varje objekt i scenen med en given riktning med specifik färgstyrka. Ljusets riktning avgjorde belysningen beroende på hur ytor var vinklade.

Funktionen för att beräkna belysningen på ytor tog in en pixelfärg tillsammans med en normal och ljusinställningarna. Funktionen returnerade en ny uträknad pixelfärg. Funktionen beskrivs nedan:

```
(lightAmbience + pixelBrightness * lightDiffusion) * pixelColor
```

där `pixelBrightness` beräknades med punktprodukten mellan ljusets negativa riktning och pixelns normal:

```
pixelBrightness = (-lightDirection) ˚ pixelNormal
```

där `lightDiffusion` och `lightAmbience` är *RGB*-värden som pixelfärgen. `lightDiffusion` är färgen i *directional lighting*.

I projektet användes en `Mesh`-klass som representerade en mängd vertriser för att rendera komplexa 3D-objekt. Vertriserna innehöll bland annat normaler för att beräkna belysningen från ljuset. Normalen bestämde den riktning som en yta pekade mot. Ytor i den här kontexten handlade om trianglar som skapades mellan vertriserna.

I projektet gjordes ljusberäkningen i en *fragment shader*. *OpenGL* interpolerade normalvärdena för ytorna i *fragment shader*.

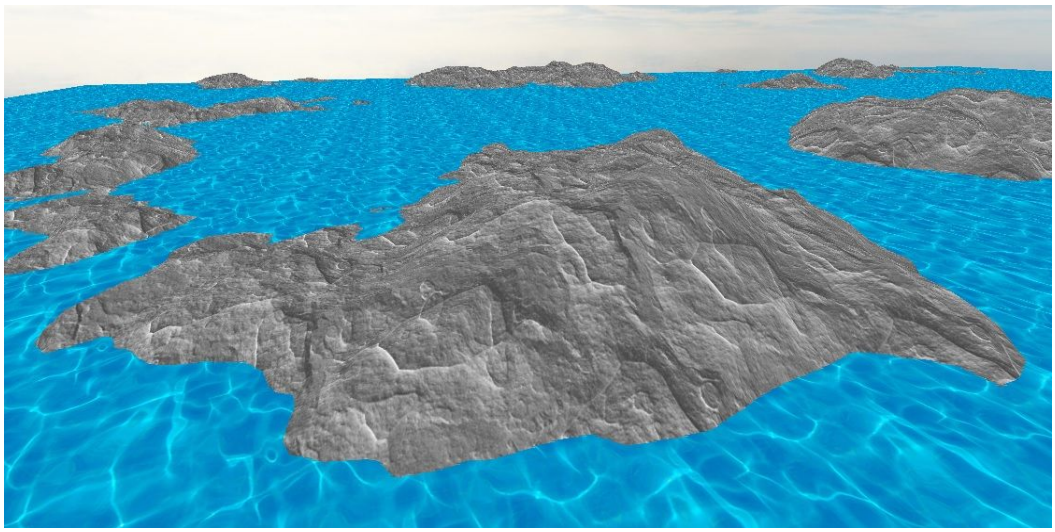

Bild 2: Före implementationen av ljus.

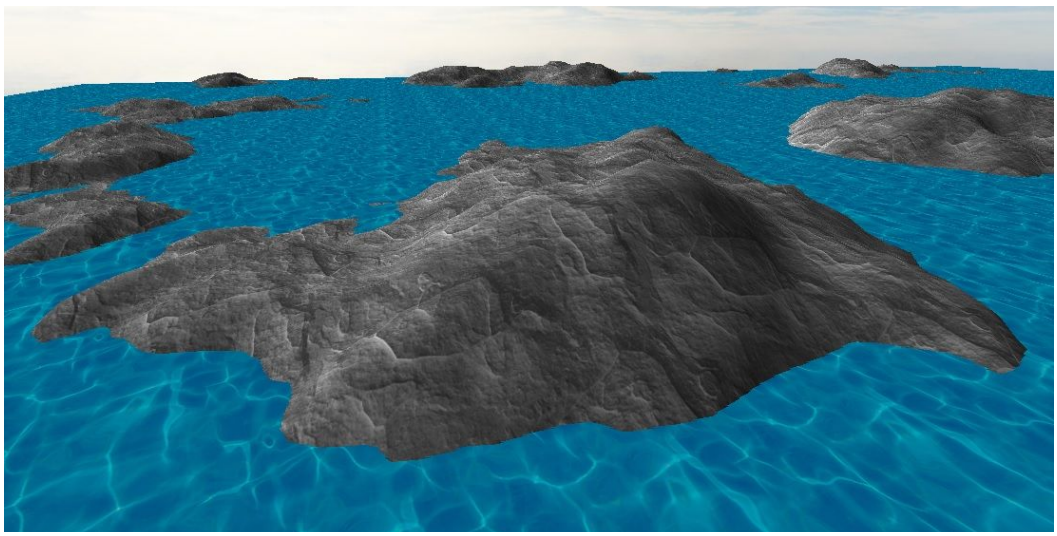

Bild 3: Efter implementationen av ljus.

3.3 Frustum culling

Frustum culling gjordes för att grafikkortet enbart skulle behöva rendera objekt i scenen som syntes på datorskärmen. För att objekt i scenen skulle renderas med *OpenGL* krävdes det att objektet fanns innanför kamerans *frustum*.

Frustum skapades med ett *near plane*, ett *far plane* samt planer ovantill, undertill och vänster och höger. Varje plan bestod av en punkt i 3D-rymden och en normal som pekade in mot frustumet

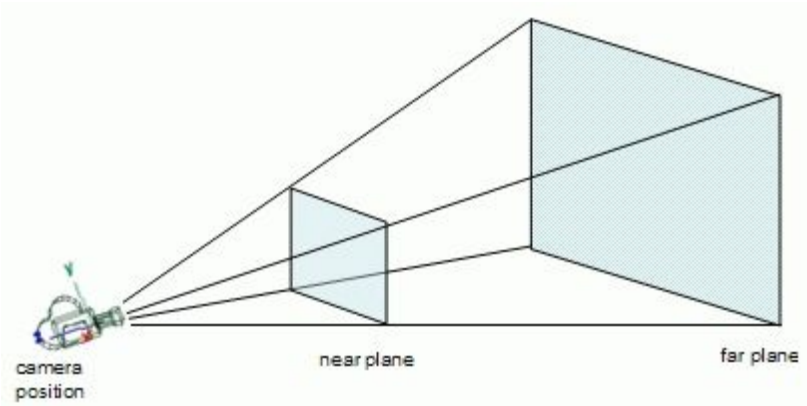

Bild 4: En grafisk beskrivning av ett frustum.

Punktprodukten användes för att kontrollera vilken sida av varje plan som objekt befann sig på. Punktprodukten användes mellan planets normal och vektorn mellan objektets position och planets position. Om resultatet från punktprodukten var positivt betydde det att objektet låg på rätt sida av planet. Denna uträkning gjordes likadant för varje plan. Positiv produkt från alla plan innebar således att objektet fanns innanför *frustumet.*

När ett objekt i scenen inte låg innanför *frustumet* uteslöts också dess underliggande objekt.

Programmets *frame rate* varierade efter att *frustum culling* hade implementerats. När kameran pekade mot en stor mängd objekt minskade antalet *frames* per sekund. Grafikkortet behövde rendera flera objekt. När kameran vändes bort från hopen av objekt ökade antalet *frames* per sekund. Detta var något som kunde uppfattas visuellt. Utan *frustum culling* blev antalet *frames* per sekund mindre.

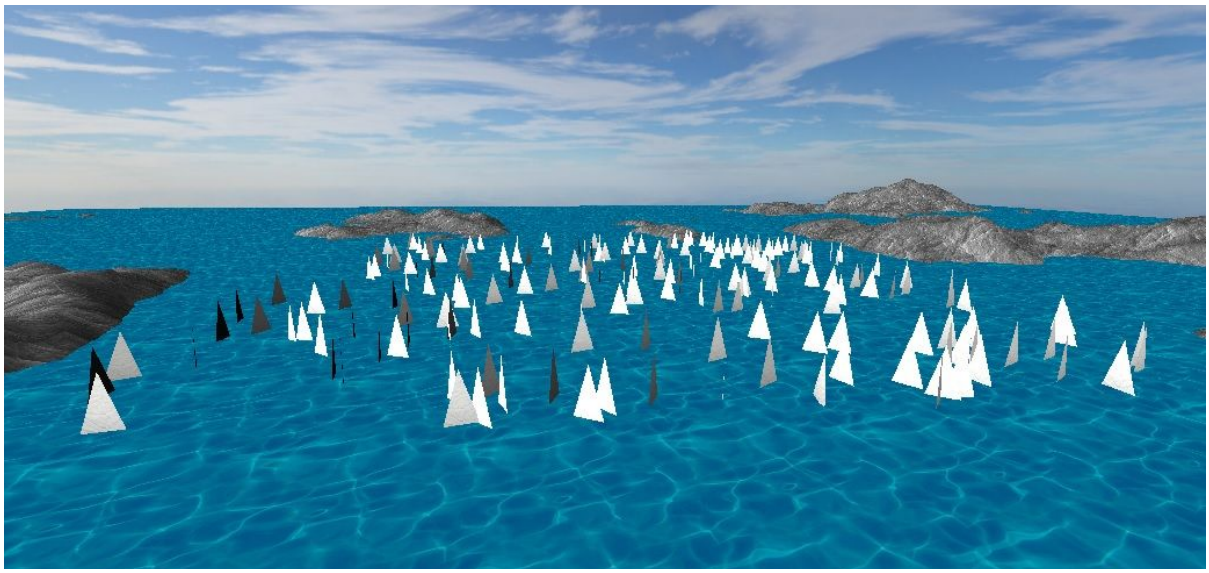

Bild 5: Med kameran riktad mot segelbåtarna märktes det att antal frames per sekund minskades.

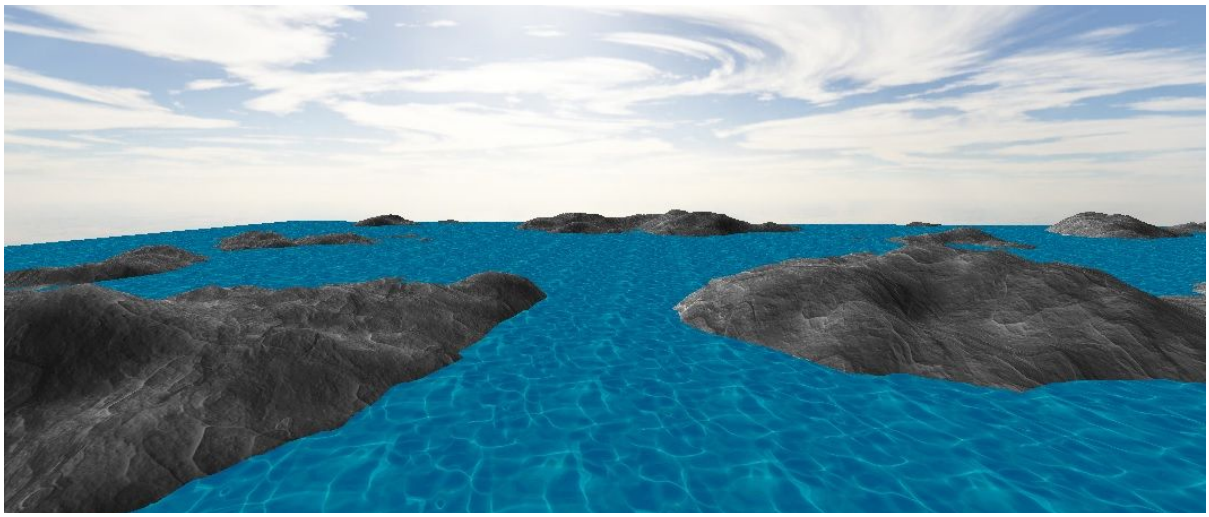

Bild 6: Antal frames per sekund ökade när färre objekt fanns framför kameran.

3.4 Font rendering

En `Font`-klass skapades för att innehålla inladdade fonter för att renderas som text på skärmen. Varje bokstav eller tecken i en specifik font sparades så att de hade ett eget buffertobjekt och ett texturobjekt i *OpenGL*. Tecknen representerades med datatypen `Glyph`, vilken innehöll informationen för ett tecken i en font. När text skulle renderas matchades varje tecken i ett `string`-värde med `Glyph`'s i `Font`-klassen.

FreeType användes i `Font`-klassens konstruktor för att ladda in `.ttf`-filer. Den inladdade datan användes sedan för att skapa buffertobjekt och texturobjekt för varje tecken i fonten. Informationen för objekten sparades som `Glyph`'s i en `std::map`. Nyckeln i `std::map` var av typen `char` för att kunna matcha enskilda tecken från texter i form av `string`-värden.

Bild 7: Renderad text med olika fonter. Texten till vänster använder fonten Arial och de övriga använder kursiv Courier.

3.5 GUI

GUI gjordes genom att skapa klasser som ärvde från en gemensam basklass. Med en basklass kunde alla ärvande klasser behandlas lika. Basklassen kallades för `GUIElement` och innehöll listor med objekt för rendering, bland annat `Text`-objekt och *sprite*-object. En `Rect`-typ användes för att bestämma koordinaterna på spelskärmen där *GUI*-elementet befann sig. En `bool`-variabel användes för att bestämma om *GUI*-elementet skulle renderas eller inte. Syftet med `bool`-variabeln var för att man skulle kunna gömma exempelvis menyer om de inte var aktuella. `GUIElement` innehöll virtuella metoder för uppdatering och input-hantering. Med metoderna kunde ärvande klasser ha olika beteenden.

Från basklassen `GUIElement` skapades klasserna `GUIImage`, `GUIText` och `GUIButton`. `GUIImage` representerade en bild på spelskärmen. `GUIText` representerade en text på spelskärmen. `GUIButton` representerade en bild och en text på spelskärmen. `GUIButton` kunde ta emot input när användaren klickade med muspekaren och anropade en bestämd metod i en annan del av programmet då muspekaren fanns innanför *GUI*-elementets koordinater.

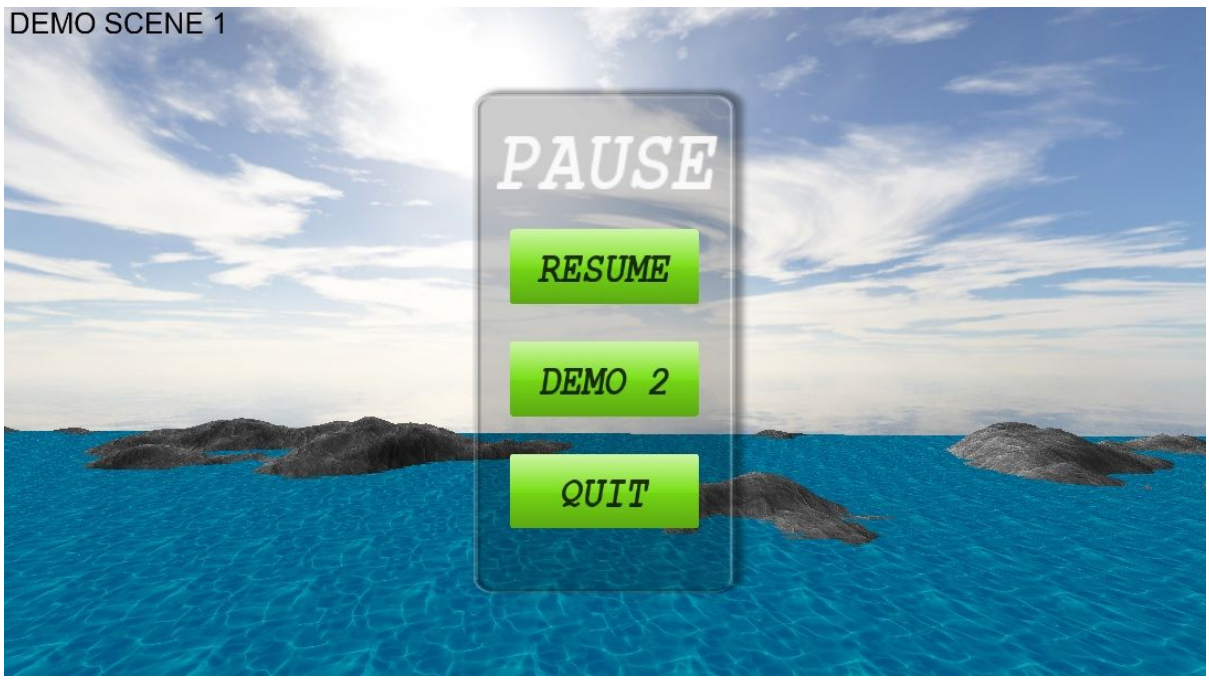

Bild 8: Tillsammans bildar flera GUI-element en interaktiv meny för användaren.

4 Diskussion

4.1 Scene management

Jag vill inledningsvis diskutera valet av metod. *Scene graph*-metoden användes för att fylla kravet *scene management* i uppgiften. Uppgiften innehöll inte någon konkret beskrivning på vad *scene management* handlade om. Information om *scene graph* hittades under sökningen på *scene management* på internet.

Scene graph-modellen i det här projektet byggdes på *Unity3D*'s sätt att hantera spelobjekt och byggdes på tolkning av prototypkod som hade getts av handledare. I *Unity* kunde spelobjekt ha så kallade *child objects* vilket liknade sättet som `SceneObject` gjordes i det här projektet. I prototypkoden som hade getts av handledare fanns en mängd kod utan några definitioner av klassmetoder. I koden fanns bland annat klassen `SceneNode`. `SceneNode` var ett begrepp som inte hade lärts ut i kursen vilket gjorde det svårt att veta vad prototypkoden skulle göra. Jag antog senare att `SceneNode` syftade till noder i en slags grafmodell, förmodligen till en *scene graph*-modell.

I koden gick det att skapa scenobjekt på ett sätt som gjorde det lätt att få överblick över en scen och alla dess objekt utan att behöva en specialiserad mjukvara. På följande förenklade sätt kunde det ungefär se ut i koden:

```
sceneObjects =
      new World(
            new Terrain(),
            new Water(
                  new SailingBoat(),
                  new SailingBoat(),
                  new SailingBoat(),
                  ...
```

Jag upplevde att kodexemplet ovan liknade *Unity3D*'s hierarkifönster för spelobjekt. Enligt min erfarenhet ett vanligt sätt att beskriva hierarkistrukturer i scener. Se nedan en bild på liknande scen gjord i *Unity3D*:

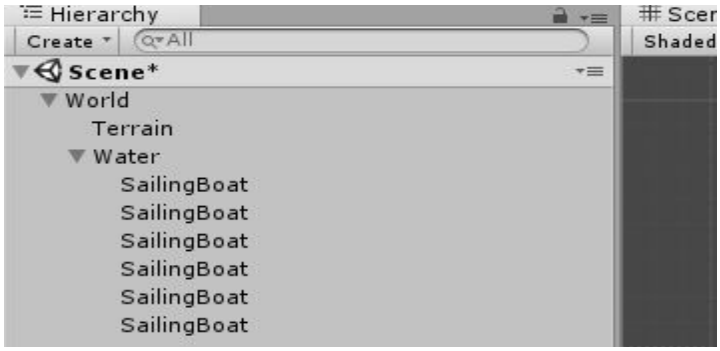

Bild 9: Spelmotorn Unity3D använder en stair-struktur för att visa hierarki av spelobjekt.

Enligt *Universal Principles of Design* kallas detta sätt att presentera en hierarki för en *stair structure*. Författarna skrev att *stair structure* är ett effektivt sätt att representera komplexa hierarkier. Nackdelen är att den kan vara svår att söka igenom och vara missvisande för relationer. Författarna menade att en *stair structure* är bra för hierarkier som regelbundet behöver gå genom förändringar. I det här projektet fanns inte de kraven eftersom scenerna inte hade särskilt komplexa hierarkier av objekt. Däremot upplevde jag att koden blev tydlig och lättförståelig. Fördelarna, och eventuellt nackdelarna, hade nog märkts tydligare i ett större projekt.

Rekursiva metoder fungerade bra ihop med rekursiva datatyper. Det märktes särskilt med metoden `DrawSceneObjects`, som användes för att rendera alla scenobjekt i en scen. När metoden anropades med översta scenobjekten i hierarkin och med en nollställd transformation ackumulerades transformationen för underliggande scenobjekt. Metoden var enligt min uppfattning lätt att förstå.

I projektet gick det bara att använda lokala transformationer för scenobjekten. Det betydde att typen `SceneObject` inte kunde hitta sin transformation oberoende av sina överliggande objekt. För att ha möjliggjort det hade förmodligen `SceneObject` behövt fler variabler och metoder samt ändringar i rekursiva metoder som `DrawSceneObjects`. Det var dock inget krav i detta arbete från min uppfattning.

4.2 Lighting

Uppgiftens kriterier var ospecifika med tanke på att begreppet *lighting* kunde innebära så mycket. I det här projektet användes en ganska simpel ljussättning om man jämför med många moderna spel idag. *Lighting* hade kunnat utvecklats mycket mer, exempelvis implementation av skuggor. I syfte att begränsa arbetstiden upplevde jag resultatet som tillräckligt.

I projektet fungerade *lighting* bara för en typ av vertriser. Det var de vertriser som tillhörde `Mesh`-klassen. Med tanke på att uppgiften handlade om att lära sig *OpenGL* så antog jag att det var tillåtet. I en professionell miljö hade man kanske velat ha flera variationer på objekt som kunde reagera på ljussättning. Bland annat i spelmotorn *Unity3D* fanns *mesh*-klasser som kunde fungera med olika *shaders* med olika uppbyggnader av vertrisdata. I det här projektet använde jag endast en typ av vertris som fungerade för ljusberäkningarna.

I prestandamässig synvinkel så hade det nog varit bättre att beräkna belysningen i en *vertex shader*. Det hade inneburit att grafikkortet enbart hade behövt göra beräkningar för vertriser istället för varje pixel. Nackdelen att göra det för varje vertris var att ljussättningen kunde få uttryck som inte upplevdes så detaljerat.

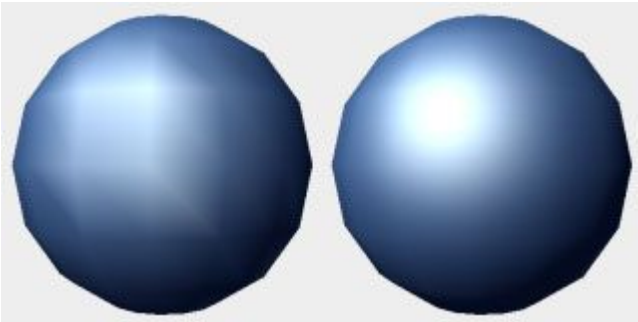

Bild 10: Bilden visar skillnaden på att beräkna belysning i vertex shader och i fragment shader. Bollen till vänster beräknar belysning i vertex shader. Bollen till höger beräknar belysning i fragment shader.

OpenGL hade som standard att interpolera värdena mellan vertriserna till *fragment shader*. Genom *fragment shader* gick det att få ut normalen för varje enskild pixel.

Självklart bör kriterierna i projektet avgöra om prestandan är viktigast eller inte. Exempelvis i spel som behöver optimeras för olika anledningen så vore det kanske bäst att välja ljusberäkningarna för varje vertris istället för varje pixel. I det här projektet var inte prestandan något krav. Därför valde jag att använda det visuellt snyggare alternativet.

4.3 Frustum culling

Det fanns flera sätt att få tag på ett frustum ur kamerans information. I det här projektet användes ett geometriskt sätt att beräkna ut frustument. Vinklar beräknades ur kamerans *field of view* och *aspect ratio* för att få tag på normalerna på frustumets horisontella och vertikala planer. Metoden inspirerades från en beskrivning på *lighthouse3d.com*. Enligt beskrivningar av *lighthouse3D* gick det att använda en metod de kallade för *clip space appraoch.* Eftersom geometriska formler var bekanta för mig sedan tidigare valde jag att inte försöka använda *clip space*. Det hade kanske varit användbart att lära sig den informationen men för uppgiften kändes det dock onödigt. Jag förstår ändå att det finns olika sätt att beräkna fram frustum.

Det hade möjligtvis gått att effektivisera beräkningen för frustum. I projektet användes geometriska beräkningar för varje *frame*. Om beräkningen kunde ha sparats på något sätt till nästa beräkning hade nog prestandan ökat lite, men kanske bara minimalt. Jag hittade ingen information om sätt att göra det. För projekt i framtiden kan det dock vara värt att skapa djupare kunskap om sätt att förbättra prestandan.

I projektet uteslöts rendering av objekt om inte någon av dess överliggande objekt fanns innanför kamerans frustum. Detta gjordes i antagandet att prestandan skulle förbättras med att färre geometriska beräkningar behövdes.

Ingen information hittades på hur mycket *frustum culling* gör prestandamässig skillnad. Det märktes dock förbättring i antalet *frames* per sekund utan att behöva göra tester. Samtidigt handlade uppgiften bara om att implementera fungerande *frustum culling* så vidare research hade nog varit onödigt för uppgiften. *Frustum culling* var dock en beprövad metod som användes i professionella sammanhang av bland annat *Unity3D*.

4.4 Font rendering

Det hade nog varit bättre att spara ett enskilt texturobjekt för alla tecken från en font. I projektet använde varje tecken istället egna texturobjekt. När texter renderades behövde *OpenGL* binda om texturobjekt för varje enskilt tecken för samma font. Om alla tecken från en font hade samlats i ett gemensamt texturobjekt hade inte *OpenGL* behövt binda om texturobjekt om en text med samma font hade renderats. Jag hittade ingen information om hur kostsamt texturbindning var, varken för *CPU*:n eller för *GPU*:n. Om jag spekulerar att det ska renderas en text på 500 tecken så skulle 500 texturbindningar behövts göras. Förmodligen märks det om renderingen ska göra för varje *frame* som i detta projekt och många digitala spel.

Bild 11: Bilden visar ungefär hur ett texturobjekt hade sett ut om alla tecken i en font delade på samma texturobjekt.

Varje tecken i Font innehöll egna buffertobjekt vilket ledde till samma problem som med texturobjekten. När texter renderades behövde *OpenGL* binda om buffertobjekten. En optimal lösning hade varit att rendera tecken med samma vertriser. Vertriserna hade i så fall behövt ändra koordinater så att de matchade med rätt tecken i ett texturobjekt för en font.

Ett alternativ till font-renderingen hade varit att använda en dataklass, i stil med objektorienterad programmering, som hanterade all textrendering. Klassen hade exempelvis

kunnat ha namnet `TextRenderer`. `TextRenderer` skulle ha en grupp vertriser som bildade en rektangel. Proceduren för textrendering hade då börjat med att binda ett texturobjekt med en särskild font. Sedan skulle texten itereras genom tecken för tecken. För varje iteration skulle koordinaterna för vertriserna ändras så att de matchade rätt tecken i texturobjektet. *Learnopengl.com* använde en liknande modell i en beskrivning på att rendera *sprites* med *OpenGL*.

4.5 GUI

`GUIButton` visade fel position på skärmen när spelfönstret ändrades i storlek. I det fallet kunde användarens muspekare befinna sig bortom knappen men ändå agera med den. Problemet låg i att *OpenGL* stretchade ut bilden när storleken ändrades. `GUIButton` tog inte hänsyn till det och fortsatte att använda pixelkoordinaterna för att veta om muspekaren fanns innanför knappens koordinater. Eftersom bilden stretchades ut hamnade knappen visuellt längre ner och mer till höger. Då överensstämde inte pixelkoordinaterna med dess renderade position.

I projektet gjordes en interaktiv meny för användaren. På skärmen såg menyn ut att vara ett enhetligt objekt men i praktiken var de separerade från varandra. Om placeringen på ramen hade ändrats hade inte knapparna innanför följt med. Det var inget problem eftersom menyn innehöll få antal *GUI*-element och gick igenom med få ändringar. Det hade förmodligen varit arbetsammare i ett projekt som behövt många designändringar.

En *scene graph*-modell hade nog gått att använda till *GUI*. En *GUI*-meny hade kunnat bestå av ett *GUI*-element som representerade en ram och innehålla flera GUI-knappar som underliggande *GUI*-element. Förflyttning av ramen hade då flyttat knapparna också.

5 Slutsats

Scene management var ett stort koncept och kunde innebära en mängd olika saker, bland annat modellen *scene graph*. Genom enbart kod gick det att skapa hierarkier av objekt på ett tydligt sätt utan att behöva specialiserade grafiska mjukvaror. *Stair*-strukturer var ett vanligt och beprövat sätt att representera hierarkier inom *scene graph*-modeller, speciellt i projekt som behövde genomgå kontinuerliga förändringar. Rekursiva funktioner eller metoder fungerade bra med att iterera genom rekursiva datatyper. I stora projekt kan det dock vara nödvändigt att lägga extra tid på att utveckla *scene graph*, exempelvis att implementera globala transformationer för scenobjekt förutom enbart lokala transformationer

Lightning var ett annat stort koncept som gick att utveckla nästan obegränsat. Oavsett tidsbegränsning gick det däremot att implementera effektfull ljussättning med *OpenGL* utan att det behövde vara allt för komplicerat.

Ljusberäkningar gjordes effektivare och snabbare i *vertex shader* än i *fragment shader*. Samtidigt gav *fragment shader* mer detaljerad belysning. Valet bör göras efter projektets krav på prestanda och presentation.

Textrendering går förmodligen att optimeras om alla tecken för en font sparades i ett gemensamt texturobjekt än om varje tecken skulle ha egna texturobjekt. Detsamma skulle gälla med buffertobjekt för vertriser. Ett effektivt sätt hade kanske varit att använda en specifik dataklass för att hantera textrendering.

Det fanns flera sätt att beräkna *frustum* , bland annat genom geometriska funktioner och *clip space*. *Frustum culling* gjorde märkbar skillnad visuellt efter implementation, och eftersom *frustum culling* användes i professionella sammanhang är det min reflektion att det har varit en särskilt värdefull lärdom.

Om *GUI*-knappar skapas kan det vara nödvändigt att ta hänsyn till relationen mellan pixelkoordinater och var *OpenGL* proportionellt renderar objekt. I projekt med flera designändringar för *GUI* kan det vara till fördel att använda en *scene graph*-modell för *GUI*-system.

Det här arbetet har visat att det går att implementera *scene management*, *lighting*, *frustum culling*, *font rendering* och *GUI*. Kraven i uppgiften innefattade stora koncept som hade kunnat bli hur stort som helst utan några begränsningar. Resultatet visade dock att med begränsad tid och implementationer gick det att skapa ett 3D-program med *OpenGL* som säkert skulle vara potentiellt i professionella sammanhang.

Referenser

Khronos Group. OpenGL. https://www.opengl.org/ (2016-09-25).

Uppsala Universitet. Spelprogrammering III. http://www.uu.se/ (2016-09-25).

Microsoft. Visual Studio 2015. https://www.visualstudio.com/ (2016-09-25).

Andrew P. Black. Portland State University. *Object-oriented programming: Some history, and challenges for the next fifty years*. 2012.

G-Truc Creation. OpenGL Mathematics (GLM). http://www.g-truc.net/ (2016-09-25).

The FreeType Project. FreeType. https://www.freetype.org/ (2016-09-25).

stb. *stb_image*. https://github.com/nothings/stb (2016-09-25).

Unity. https://unity3d.com/ (2016-09-25).

Bar-Zeev, Avi. *Scenegraphs: Past, Present, and Future*. http://www.realityprime.com/blog/2007/06/scenegraphs-past-present-and-future/ (2016-09-25).

Lidwell, Holden and Butler. *Universal Principles of Design*. 2003.

Lighthouse3D.com. http://www.lighthouse3d.com/tutorials/view-frustum-culling/ (2016-09-25).

de Vries, Joey. LearnOpenGL. *Rendering Sprites*. http://learnopengl.com/#!In-Practice/2D-Game/Rendering-Sprites (2016-09-25).