21 december 2018

Gymnasiearbete

Utveckling av grafikmotor i C++ med OpenGL

Teknikprogrammet - Hågesta Gymnasium

Läsåret 2017/2018

Av: Oliver Larsson

Handledare: Bo Höglund

0 Abstract

My interest in programming aswell as computer graphics naturally lead me to an urge of wanting to know more about how modern graphics is handled and displayed by our machines. I set out to create a rendering engine of my own as a way to better understand not only how, but also why engines are built the way they are. I encountered many problems along the way that have required me to dig deeper and understand some of the very core fundamentals of computing.

Innehåll

1 Bakgrund

Jag har varit intresserad av datorer och teknik sedan ung ålder, så valet av ämne för mitt gymnasiearbete föll sig ganska naturligt på mjukvaruutveckling. Däremot vilken typ av program jag ville tackla var lite knepigare att sätta i sten. Jag har sedan tidigare skapat program i ett antal olika miljöer och kan medge att jag föredrar att *inte* arbeta med webbsidor, så jag försökte att hitta något annat.

Eftersom datorspel även det varit ett av mina intressen bestämde jag mig för att skapa något som har med spel eller grafik att göra. Det är ett bra område att utveckla inom då man inte alltid behöver ett definierat problem att lösa med sitt program. Eftersom den tillgängliga tiden inte är oändlig kom jag fram till tre olika projektalternativ. Jag skulle kunna skapa ett tredimensionellt spel i en av de idag så populära grafikmotorerna som Unity eller Unreal Engine. Ett andra alternativ skulle vara att göra ett tvådimensionellt spel, inkluderat grafikmotorn. Men det alternativet som tilltalade mig mest och sedan kom att bli mitt arbete var att endast skapa en tredimensionell grafikmotor från grunden och att inte bygga något spel. Detta därför att jag inte känner någon riktig lust att göra ett spel, utan att jag är mer intresserad av att förstå hur motorn och bakgrundssystemen fungerar i de spel som finns idag.

1.1 Verktygen

Efter beslutet om vad måste jag också bestämma hur. Programmerare har en uppsjö av språk att välja mellan när de ställs inför ett problem som detta.

1.1.1 C++

Det kanske smartaste valet hade varit att skriva grafikmotorn i Java. Java är ett språk jag sedan tidigare programmerat mycket i och känner mig säker med. Men efter att funderat en stund kom jag fram till att det skulle vara en förlorad möjlighet att lära sig ett nytt språk om jag skrev i Java, så jag började leta efter ett alternativ.

Industristandarden bland professionella grafikmotorer har under en längre tid varit C++. C++ är ett språk som är snabbare än Java samt att det har större minnesfrihet. Det är ett språk som kanske tappat sin nödvändighet i vardagliga applikationer, men hastigheten som det erbjuder är mycket välkommen inom

den krävande grafikrenderingen. Därför valde jag C++, ett språk jag aldrig programmerat i förut, med målet att få nya och bredare kunskaper inom programmering.

1.1.2 OpenGL

För att direkt kunna använda mig av grafikkortets funktioner behöver jag ett grafikbibliotek som klarar av detta. Mitt val föll på OpenGL *(Open Graphics Library)*, men hade lika gärna kunnat falla på någon av konkurrenterna, Direct3D eller Vulkan. För att få tillgång till OpenGLs funktionaliteter behöver jag *(OpengGL Extension Wrangler)* och för att hantera fönstret och dess funktioner använder jag mig av GLFW *(Graphics Library Framework)*.

2 Syfte och frågeställning

Syftet med mitt arbete är att lära mig om och förstå hur tredimensionell grafik renderas och hanteras i dagens grafikmotorer samt att jag ska få en grundläggande kunskap inom programmeringsspråket C++.

2.1 Frågeställning

Hur utvecklas en tredimensionell grafikmotor där C++ och OpenGL används?

2.2 Kravspecifikation

- Grafikmotorn ska klara av att rendera tredimensionell grafik.
- Det ska finnas stöd för vatten.
- Motorn ska hantera ljus och skuggor.
- Modeller i Wavefront-format *(.obj)* skall gå att importera och rendera i motorn.

2.3 Avgränsning

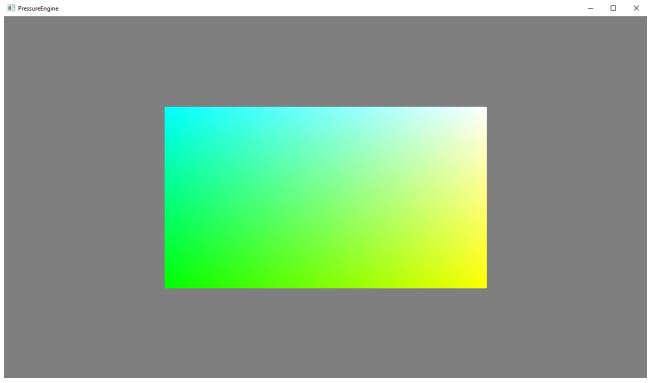
Jag begränsar mig till endast grafik, inget ljud kommer att finnas i grafikmotorn. Inget spel kommer att utvecklas i grafikmotorn, endast ett demo-landskap för att visa motorns funktionalitet. Animation är inte heller ett område jag kommer att titta närmare på.

Grafikmotorn ska ej vara lämpad för produktion, det vill säga att det inte är meningen att någon annan ska plocka upp motorn och använda den för att skapa ett spel eller annat. Den är tänkt att vara en inlärningsmöjlighet och undersökningsmiljö.

3 Genomförande

3.1 Första trianglarna

Första steget i skapandet av en grafikmotor är självfallet att skapa ett projekt att arbeta i. Jag använder mig av Microsofts Visual Studio som utvecklingsmiljö och ser till att både GLEW och GLFW finns importerade och att de länkas korrekt. Sedan skapar jag fönstret med hjälp av GLFW. Än finns inget i fönstret så jag undersöker om OpenGL fungerar som det ska genom att rendera två trianglar sida vid sida så att de formar en rektangel. Jag färgar sedan denna rektangel utifrån dess koordinat på fönstret och resultatet kan ses i figur 1.

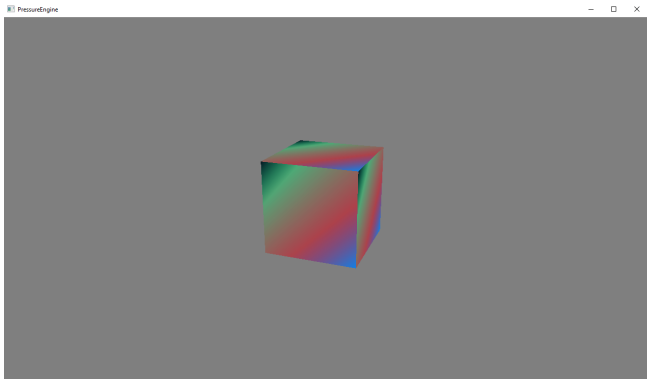


Figur 1: Första renderade trianglarna, 2017-08-28

3.2 Tredimensionalitet

Nästa steg i utvecklingsprocessen är att utöka motorn till att kunna rendera tredimensionalitet. Detta kräver ett antal olika nya funktioner och shaders är den viktigaste av dessa. En shader är ett program som körs på grafikkortet och använder grafikkortets minne. Det är i detta program vi beräknar var på skärmen en viss punkt ska visas och hur den ska se ut.

Det är i vår shader vi kan flytta, rotera och skala våra modeller. Detta görs med hjälp av matris-multiplikation, något grafikprocessorer är extremt snabba på. Jag använder mig av transformation-vy-projektion modellen där de koordinater jag matar in går igenom tre steg innan de visas på skärmen. Som kan ses i figur 2 lägger jag också in funktion för att använda texturer på modeller genom UV-koordinater. *(texturkoordinater)*

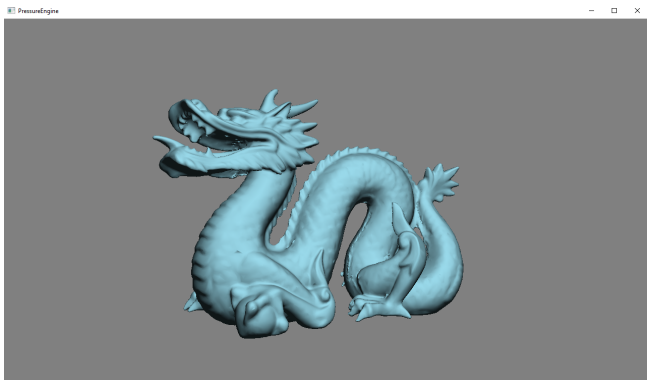


Figur 2: Tredimensionalitet, 2017-09-03

3.3 Wavefront modeller

När 3D är implementerat och det är möjligt att rendera enklare, hårdkodade modeller till skärmen är nästa steg att kunna ladda in större och mer komplicerade modeller från modellfiler. Det finns flera olika format, men Wavefront *(filändelsen .obj)* har en enkel struktur så det är det formatet motorn kommer att stödja.

Den första versionen av inläsningssystemet lyckades jag få igång utan större problem. Som stresstest av både denna inläsning och motorn i sin helhet använder jag Stanford-Dragon modellen som kan ses i figur 3, en modell som är 7,5 MB och 50 000 polygoner i storlek. Då märktes det att inläsningen skulle kräva lite arbete då denna modell krävde dryga 15 sekunder att läsa in.



Figur 3: Stanford-Dragon modellen och modellinläsning, 2017-09-07

Jag gick igenom ett antal iterationer innan jag blev nöjd, och jag lyckades sänka inläsningstiden för denna modell till under 1 sekund. Den förändring som gjorde störst skillnad var att först ladda in hela filen i minnet och sedan bearbeta den därifrån istället för att kontinuerligt läsa den från disk då den behövs. Det gick även att städa upp logiken och minnesallokeringen för *std::vector*.

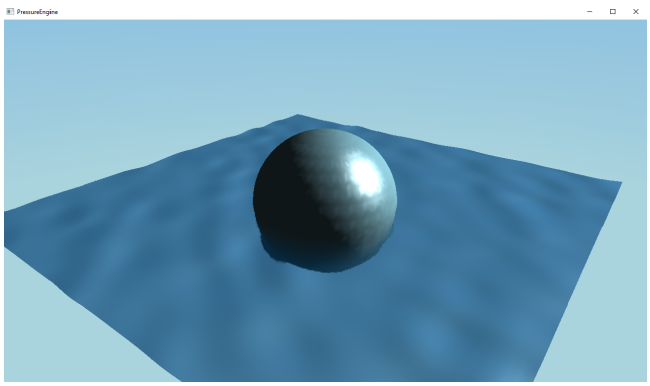
3.4 Ljussättning

Som nästa steg väljer jag att implementera ljussättning. Större delen av denna beräkning sker på grafikkortet. Efter att jag definierat koordinaterna för en ljuskälla kan jag beräkna en pixels ljusstyrka utifrån dot-produkten mellan ytans normal och den vektor som pekar mot källan från pixeln. Även spegelbelysning för blanka föremål går att beräkna på liknande sätt.

Jag implementerar också en skybox, eller himmelslåda, för att få en himmel. Den påverkas endast av rotation vilket betyder att kameran alltid kommer att befinna sig i det exakta centrumet av denna låda, så att kameran inte kan flyttas utanför himlen.

3.5 Vatten

Rendering av vatten är något som kräver ytterligare funktioner till min motor. Det finns ingen genomskinlighet i renderat vatten, utan det är ett plan som textureras med en blandning av en refraktionstextur och en reflexionstextur. Dessa texturer kommer från att man renderar scenen utan vattnet till så kallade framebuffers. Dessa kan sedan användas i shadern för att beräkna vattenplanets slutgiltiga textur.

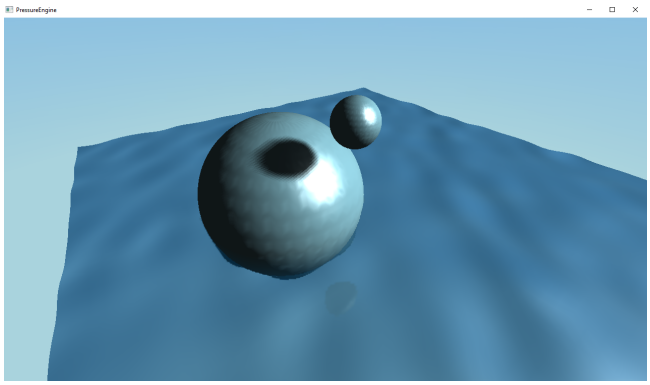


Figur 4: Ljus, skybox och vatten, 2017-09-17

3.6 Skuggor

Den sista av de stora funktionerna jag implementerade var föremåls förmåga att lägga skuggor på andra föremål. Det kräver att man använder sig av den depthbuffer (shadow-map) man får genom att rendera scenen från solens perspektiv genom en ortografisk vy. Det går då att beräkna om den finns något objekt mellan den pixel man beräknar och solen. Om detta är fallet ligger pixeln i skugga.

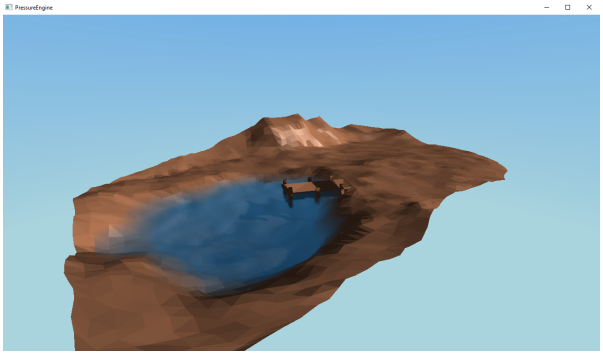
Jag hade under en längre tid problem med ett fenomen som kallas för shadow acne, eller skugg-acne. Detta problem uppstår då upplösningen på shadow-mapen inte är tillräckligt hög för att täcka alla pixlar på skärmen. Detta problem löste jag till slut genom att använda front-face culling vid rendering av min shadow-map. Då renderar jag endast de ytor vars normaler pekar bort från solen.



Figur 5: Det lilla klotet lägger en skugga på det större, 2017-11-07

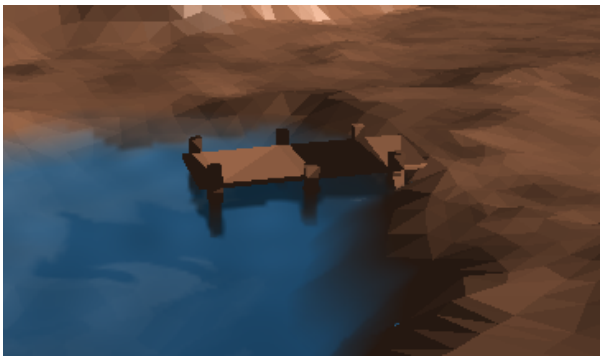
3.7 Demoscenen

För att visa upp grafikmotorn skapade jag en scen för användaren att titta runt i. De objekt som finns i denna scenen skapade jag i programmet blender. Samtidigt som jag modellerade denna scen gjorde jag en del omskrivningar av gammal kod jag kände att jag behövde förbättra samt att jag lade till några mindre funktioner, efterbehandling av bilden samt anti-aliasing med mera.



Figur 6: Demoscenen har börjat ta sin form, 2018-02-03

Under modelleringen av denna scen framkom flera buggar jag inte tidigare känt till. En av dessa var ett problem med hur motorn ljussätter ytor i modeller som angränsar en skarp vinkel. Ett exempel på detta kan ses i figur 6 på den lilla bryggan som ligger vid vattnet. En del av bryggans område är mycket mörkare än det som ligger precis bredvid (se figur 7), även om de borde vara samma färg. Det visade sig vara ett problem med hur motorn hanterade normalvektorer för skarpa hörn. Problemet kunde enkelt lösas med hjälp av en så kallad geometry-shader, vilket är en speciell typ av shader som kan användas vid behov.



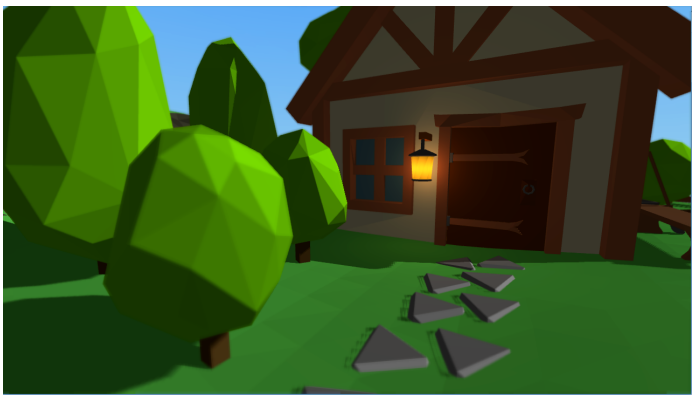
Figur 7: Problem med ljussättningen på ytor som angränsar skarpa hörn

4 Resultat

Resultatet av mitt arbete är en grafikmotor som, om än enkel, har gett mig goda grundläggande kunskaper inom grafikrendering. Det är en motor som går att hämta hem och köra på valfri windowsmaskin. Den kräver en dator med relativt bra prestanda för att köras smärtfritt, så de datorer skolan har lånat oss kan ha problem med den. Den ser inte riktigt ut som den ska när man kör den på en dator med grafikkrets byggd av Intel. Det beror på en begränsning i deras drivrutin.

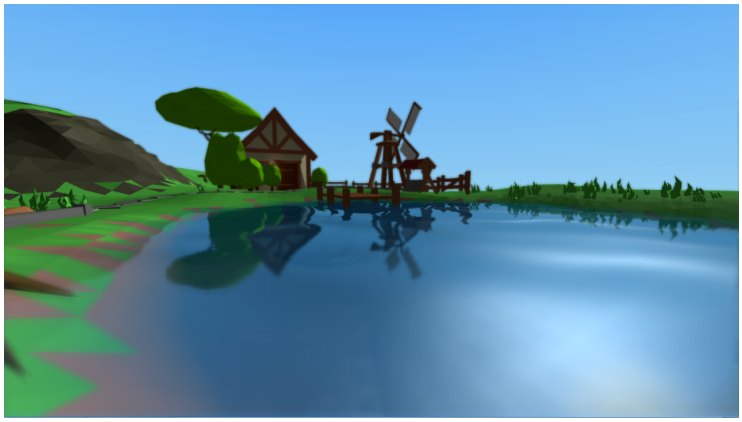
All källkod ligger öppet på GitHub.com under namnet PressureEngine. Där går det att läsa koden eller ladda ner den om man vill köra den själv. Projektet är en Visual Studio 17 solution och kräver windows SDK 10.0.16. (Windows 10 med Fall Creators Update)

Figur 8: Demoscenen är konstruerad, 2018-04-05

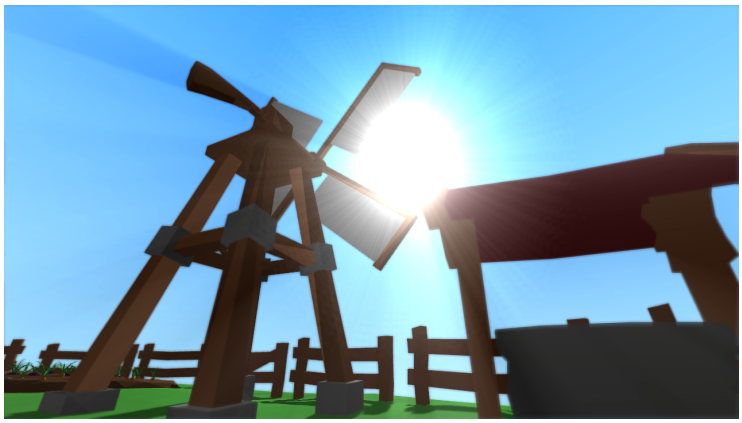


Figur 9: Lampan på husväggen lyser upp det skuggade området, 2018-04-05

Figur 10: Oskärpa tillämpas på objekt i bakgrunden och förgrunden, 2018-04-05



Figur 11: Reflexioner i dammen, 2018-04-05



Figur 12: Solens strålar bryter sig runt väderkvarnen, 2018-04-05

5 Diskussion och slutsats

Överlag anser jag att projektet blev en succé. Jag är nöjd med resultatet med tanke på de förkunskaper jag hade. Det finns ett antal moduler jag skulle byggt upp annorlunda om jag gjorde en ny grafikmotor, framför allt därför att jag lärt mig nya funktioner både inom C++ och OpenGL som skulle kunna göra de systemen bättre och snabbare.

Det finns också ett antal system jag skulle vilja bygga in för att utöka och förbättra min motor. Ett exempel på detta är att bygga ett sofistikerat materialhanteringssystem. Det skulle ge möjligheten för en och samma modell att ha flera olika ytor som alla kan ha olika reflektivitet eller andra egenskaper.

Men målet med arbetet var inte att producera en grafikmotor, utan att lära mig hur en sådan motor skulle kunna vara uppbyggd, och det tycker jag att jag fått en god förståelse för. Samtidigt har jag skaffat mig användbara och kraftfulla kunskaper inom både C++ och programmering i sin helhet.

6 Teoretisk förankring

6.1 Digital geometri

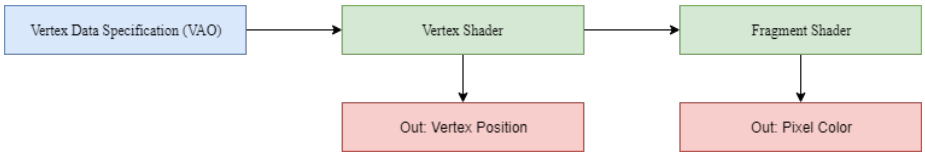
Det vanligaste sättet för grafikkort att rendera geometri är genom att använda sig av trianglar. Det betyder att alla former på skärmen är uppbyggda av trianglar i olika storlekar och former. Det betyder också att om man, som exempel, vill bygga en rektangel måste denna byggas upp av två trianglar som ligger sida vid sida. För att det ska gå att bestämma en triangels "framsida" specificeras de punkter den är uppbyggd av motsols runt centrum om man ser den framifrån.

6.2 Grafikkortets arbetsprocess

Grafikkortet arbetar med program precis som processorn gör. Dessa program kallas för shaders och skrivs för OpenGL i programmeringsspråket GLSL *(OpenGL Shading Language)*. Denna kod laddas som strängar till OpenGL som sedan kompilerar den och lägger programmen i grafikkortets minne. Shadern körs en gång för varje bild, eller frame, som ska visas på skärmen. Programmets input är den data specificerad i en VAO *(Vertex Array Object)* som innehåller data för att beskriva varje vertex på skärmen. Det kan vara data som position, texturkoordinater och normalvektor. Det går även att ladda in så kallade *uniforms*, variabler som är lika för alla vertex. Den enda output från dessa program är den bild som visas på skärmen, vilket gör dem svåra att debugga.

Den mest grundläggande shadern består av två delar, en vertex-shader och en fragment-shader. Vertex-shadern ligger först och den har som uppgift att bestämma punkternas position på skärmen. Den kör en gång för varje punkt som ska renderas, vilket blir ganska många gånger för varje bild. För en kvadrat blir det fyra gånger varje bild, en för varje hörn. Dess input är datan för den aktuella punkten i VAOn och dess output är huvudsakligen den slutgiltiga positionen för punkten, men också eventuell data som ska skickas vidare till nästa steg, fragment-shadern.

Fragment-shaderns uppgift är att bestämma färgen på den geometri vi specificerat i vertex-shadern. Den kör en gång för *varje* pixel den specificerade geometrin tar upp på skärmen. Programmets input är den data vi specifierat som output i vertex-shadern och dess output är ett RGBA värde, den färgen pixeln ska ha. Det är i fragment-shadern vi kan tillämpa texturer, skuggor och ljus med mera.

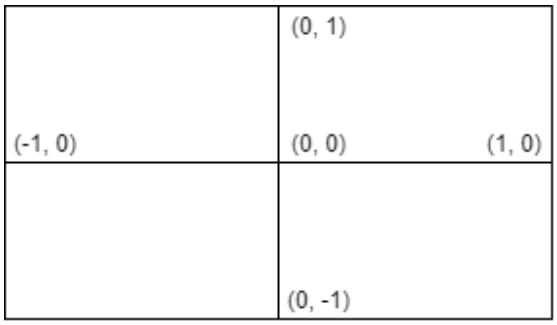


Figur 13: Grafikkortets pipeline

6.3 Transformation, Vy, Projektion

6.3.1 OpenGLs koordinatsystem

OpenGLs fönster har ett satt koordinatsystem som ej går att ändra på. Centrum på fönstret är (0, 0) och X-, Y- och Z-axlarna är placerade runt detta origo. Z-axeln pekar rakt ut ur skärmen med större värden närmare åskådaren.



Figur 14: OpenGLs skärmkoordinater

För att få illusionen av en kamera måste vi göra ett antal beräkningar och det är här TVP kommer in. Som exempel kanske vi vill flytta våran "kamera" åt höger, men eftersom detta inte går måste vi istället flytta hela världen åt vänster för att uppnå samma effekt.

6.3.2 Omvandlingen av koordinater

Alla tredimensionella modeller är definierade runt ett origo. Detta kallas att modellen är i sitt *model-space*. Det första vi måste göra är att omvandla dessa koordinater efter dess transformation. Detta gör man för att få modellens koordinater till *world-space* eller de koordinater som alla andra objekt går efter. Det är i transformationen man beräknar objektets position, rotation och storlek i förhållande till de andra objekten i världen.

Efter att koordinaterna befinner sig i *world-space* är det dags att förflytta världen efter den virtuella kamerans position. Det vill säga att det som är mitt i kameran ska ha koordinaterna (0, 0). Detta görs med en vy-matris och för koordinaterna till det som kallas för *camera-space*.

Sista steget är att omvandla den för nuvarande ortografiska vyn till en isometrisk. I en ortografisk vy ser alla objekt lika stora ut, oavsett hur långt de är från kameran, men vi vill att de objekt som är längre från kameran ska se mindre ut än de som är nära då det är så kameror fungerar i verkligheten. Då används en projektions-matris för att slutligen föra koordinaterna från *camera-space* till *screen-space*.

All denna konvertering av koordinater sker för varje punkt och varje bild i vertex-shadern. Det är matrismultiplikation mellan 4x4 matriser och fyrdimensionella vektorer som får detta att fungera så bra. Grafikprocessorer är extremt duktiga på denna typ av matematik och många moderna GPU:er kan göra en av dessa beräkningar under en enda cykel.

7 Källkodsanalys

Detta avsnitt har för avsikt att redogöra hur källkoden är strukturerad och uppbyggd för att ge en överblick över systemets olika moduler. Bör läsas med källkoden vid sidan.

Grafikmotorn är en DLL *(Dynamic Link Library)*. Det betyder att motorn i sig går inte att köra utan det behövs ett annat program för att använda den. Projektet *PressureEngine* är motorn och *PressureEngineViewer* är den implementation som använder sig av den.

PressureEngine/Src

Under *PressureEngine/Src* finns klassen PressureEngine. Det är den abstraktion där all kommunikation mellan motorn och dess implementering sker, den i sin tur anropar underliggande system vid behov.

Under mappen *Input* ligger de klasser som abstraherar GLFWs implementation av tangentbord och mus. Mappen *Services* är hem till klasserna FileStream och Properties. FileStream används vid läsning och skrivning av filer och Properties är den klass som sköter inläsning av filen pressure.properties.

Math är det matematikbibliotek som byggdes för motorn. Större delen av logiken för matriserna och vektorerna kommer från JOML[1]. Klassen *Math* består av statiska implementationer till funktioner som används i resten av motorn.

PessureEngine/Src/Graphics

MasterRenderer är den klassen som knyter ihop motorns renderingspipeline. Det är den som ser till att allt renderas i rätt ordning och på rätt sätt. Modellinläsningen sker i två steg, först läser *OBJLoader* in filen och staplar upp datan i vektorer som sedan skickas till *Loader* som bearbetar informationen och sätter in den i en *Models/RawModel*. (Vertex Array)

Många av de koncept som OpenGL introducerar är svåra att arbeta med om ingen abstrahering görs. I mappen *GLObjects* ligger dessa abstraheringar. *FrameBuffers* används till exempel till att rendera saker till texturer istället för till skärmen. Resterande klasser i mappen behövs för representeringen av 3D-modeller.

Data sparas i *VertexBuffers*, som i sin tur sedan sparas i *VertexArrays*. Som exempel kan en vertex buffer innehålla positionskoordinater, en annan texturkoordinater, och en tredje normalvektorer. Dessa sparas sedan i en vertex array som representerar den slutgiltiga modellen. *VertexBufferLayout* berättar för vertex arrayen vilket format dessa buffers har.

Mappen *Entities* innehåller de tre mest grundläggande objekten i grafikmotorn. *Camera* agerar som den virtuella kameran. Den går att flytta och rotera för att ge illusionen att en kamera faktiskt finns. *Entity* är de "fysiska" objekten i scenen och *Light* är ljuskällorna.

En entity är huvudsakligen uppbyggd av två delar, en *Models/TexturedModel* och en transformation angiven i position, rotation och skala. En *TexturedModel är en sammansättning av en Models/RawModel*, vilket är en representation av en tredimensionell modell utan textur, och en *Textures/ModelTexture*.

För att hantera texturer samt att läsa in bildfiler och ladda upp dem som texturer till grafikkortet använder jag biblioteket *FreeImage* och dess medföljande exempelimplementering som är modifierad efter motorns behov. Dessa texturer är sedan representerade i from av objekt av klassen *Textures/ModelTexture*.

Mappen *PostProcessing* omfattar de eftereffekter som kan appliceras på bilden. Det fungerar genom att först rendera scenen till en textur, och sedan bearbeta denna textur ungefär som i ett fotoredigeringsprogram. De funktioner som finns för nuvarande är kontrastförändring, skärpedjup och ljusspridning.

Renderingsenheter

Under mappen *PressureEngine/Src/Graphics* ligger även de renderingsenheter som har i uppgift att rita upp olika delar till skärmen. Dessa är uppbyggda på väldigt liknande vis men anpassade för det arbete de är tänkta att utföra. De enheter som finns i motorn är *EntityShaders, Guis, Particles, Shadows, Skybox och Water.*

Dessa enheter är ena moduler som anropas av *MasterRenderer* när de behövs. En enhet består i huvudsak av tre komponenter; en renderare, en shader och en shader-källkod. *Renderaren* agerar som kontrollklass för enheten, den skö-

ter också kommunikationen utifrån. *Shadern* är en klass som deriverar från *Shaders/Shader*. Den sköter shader-programmet som körs på grafikkortet, först genom att ladda upp och kompilera *Shader-källkoden* och sedan föra kommunikation mellan processor och shader.

8 Ordlista

buffer En yta i minnet där data kan sparas och modifieras. 8, 9, 20

c++ Programmeringsspråk utvecklat som en objekt-orienterad version av C med fokus på prestanda. 1–3, 5, 13

debugga Felsöka. 14

depthbuffer En framebuffer som endast innehåller avstånds-information. 9

framebuffer Ett alternativt mål för rendering istället för skärmen. Kan användas som textur. 8, 20

kompilera Omvandla källkod till maskinkod som datorn kan köra. 14

minnesallokering Skicka en förfrågan till datorns operativsystem om att programmet behöver mer minne, operativsystemet reserverar då detta minne åt processen. 8

normal Den vektor som pekar rakt ut ur en yta, den är vinkelrät mot ytans plan. En normal är oftast uttryckt som unit-vektor (Den har längden 1). 9, 10, 21

openGL En standard för kommunikation med grafikkortet. Anropen specificerade i standarden är implementerade av grafikkortstillverkarna i deras drivrutiner. 1, 2, 4, 5, 13, 14

ortografisk vy En vy där objekt längre från kameran inte blir mindre utan behåller sin storlek. 9

pixel En punkt på skärmen. 8, 9, 14

polygon En punkt eller "Hörn" i en modell, som exempel skulle en kub ha 8 polygoner (8 hörn). 7

rendera Att rendera något är att få det att visuellt representeras på skärmen. 5, 7–9, 14

shader Ett program som körs på grafikprocessorn. 6, 8, 10, 14

skybox En låda som är stor nog att övertygande föreställa en himmel när den färgsätts. 8

textur En bild som kan användas som för att färgsätta en 3D-modell. 6, 8, 14

texturera Använda en "textur" eller bild för att färgsätta en 3D-modell genom att "klistra" den på objektet. 8

texturkoordinater Koordinater som beskriver en texturs position på en 3D-modell. 6, 14, 21

transformation Ett samlingsnamn för position, rotation och storlek. 15

vertex Ungefär samma som polygon, men ett vertex kan innehålla mer data än en polygon. T.ex. normal och texturkoordinater. 14

9 Källförteckning

[1] *JOML-CI/JOML*, *GitHub*, *2018-03-21*
https://github.com/JOML-CI/JOML

[2] *TheChernoProject*, *Youtube*, *2018-04-03*
https://www.youtube.com/user/TheChernoProject/

[3] *ThinMatrix*, *Youtube*, *2018-04-03*
https://www.youtube.com/user/ThinMatrix

[4] *docs.gl*, *Khronos*, *2018-04-03*
http://docs.gl/

[5] *cplusplus.com*, *cplusplus.com*, *2018-04-03*
http://www.cplusplus.com/

[6] *The C++ Programming Language - Fourth Edition*
Bjarne Stroustrup, *Utgiven 2013*

10 Bilagor

1 Källkod

Källkoden är versionskontrollerad och finns på GitHub.com:
https://github.com/Playturbo/PressureEngine

2 Kompilerad binärfil

Programmet är kompilerat som .exe, men är beroende av ett antal andra filer. Därför tilländahålls den i form av ett .zip arkiv där alla filer är inkluderade.
https://github.com/Playturbo/PressureEngine/releases
Direktlänk till .zip arkivet: **https://goo.gl/Gkk2yW**

3 Visualisering av förändringshistoriken

Med hjälp av programmet Gource skapade jag en visualisering av projektets historik, den finns som olistad film på Youtube.com. (1m 40s)
https://www.youtube.com/watch?v=M2wo7icSnmw