Hur rendering i realtid fungerar

Nuet och framtiden

Axel Björkqvist 1901241

Kandidatavhandling i datateknik

Handledare: Jan Westerholm

Fakulteten för naturvetenskaper och teknik

Åbo Akademi

Våren 2023

Innehållsförteckning

[1. Inledning 3](#_Toc130910423)

[2. Grafikprocessor 4](#_Toc130910424)

[3. Grafikrenderingspipeline 5](#_Toc130910425)

[3.1. Applikationsnivå 5](#_Toc130910426)

[3.2. Geometribearbetning 5](#_Toc130910427)

[3.2.1. Vertexskuggare 5](#_Toc130910428)

[3.2.2. Tessalation 6](#_Toc130910429)

[3.2.3. Geometriskuggare 7](#_Toc130910430)

[3.2.4. Klippning 8](#_Toc130910431)

[3.2.5. Skärmkartläggning 8](#_Toc130910432)

[3.3. Rasterisering 8](#_Toc130910433)

[3.3.1. Triangeluppställning 8](#_Toc130910434)

[3.3.2. Triangelgenomgång 8](#_Toc130910435)

[3.4. Pixelbearbetning 8](#_Toc130910436)

[3.4.1. Pixelskuggare 9](#_Toc130910437)

[3.4.2. Hopslagning 9](#_Toc130910438)

[3.5. Beräkningsskuggare 10](#_Toc130910439)

[4. Viktiga koncept 11](#_Toc130910440)

[4.1. Vertexblandning 11](#_Toc130910441)

[4.2. ? 11](#_Toc130910442)

[5. Exempelprogram 11](#_Toc130910443)

[5.1. Transformer 12](#_Toc130910444)

[5.1.1. Translation 13](#_Toc130910445)

[5.1.2. Rotation 13](#_Toc130910446)

[5.1.3. Förstoring och förminskning 14](#_Toc130910447)

[5.1.4. Skärning 15](#_Toc130910448)

[5.1.5. Normal 15](#_Toc130910449)

[5.1.6. Invers 15](#_Toc130910450)

[5.1.7. Euler 16](#_Toc130910451)

[5.1.8. Kamera 16](#_Toc130910452)

[5.1.9. Projektion 17](#_Toc130910453)

[5.1.10. Konstruktion 17](#_Toc130910454)

[5.2. Kvaternion 17](#_Toc130910455)

[5.3. Kamera 18](#_Toc130910456)

[5.4. Projektion 19](#_Toc130910457)

[5.5. Indata 20](#_Toc130910458)

[5.6. Resultat 20](#_Toc130910459)

[6. Sammandrag 20](#_Toc130910460)

Ordlista

Vertex Punkt i rymden

Primitiv Innebär punkter, linjer och trianglar

Fragment Del av en triangel som delvis eller helt beskriver en pixel vid rasterisation

Mesh En samling primitiver som bildar en yta.

Pass Rendering av en bild.

GPU Grafikprocessor (Graphical processing unit)

CPU Centralprocessor (Central processing unit)

Grafikprocesseringspipeline

Standardiserade steg för att utföra rendering (Graphicsprocessingpipeline)

# Inledning

Avhandlingen utreder vad som krävs för att simulera ett interaktivt tredimensionellt utrymme på en skärm. Detta är rendering i realtid. Teknikens grunder samt framtida utveckling presenteras och sammanställs med tanke på datorspel. Datorspel är i fokus eftersom spelbranschen är huvudkonsumenten av realtidsuppritningsteknologin. Tekniken konkretiseras med hjälp av ett enkelt renderingsprogram som byggs upp steg för steg i denna avhandling. Ämnet är intressant eftersom det har skett stora framsteg i teknikerna och hårdvaran. Rendering kommer att fortsätta vara relevant i framtiden på grund av dess centrala roll inom digitaliseringen.

För att en simulering ska vara interaktivt, måste den reagera på användarens respons. Om simuleringen accepterar kontinuerliga indata från användaren, är det opraktiskt att rendera simuleringen i förtid. Detta beror på att det kan finnas oändligt många möjliga kombinationer av indata som borde alstra en unik sekvens av renderingar. Då borde alla dessa sekvenser renderas och sparas i förtid, vilket är opraktiskt. Om simulationen accepterar indata reglerat, kan en färdigt renderad sekvens dock helt enkelt spelas upp vid önskad tidpunkt. Ett exempel på detta är mellansekvenser i spel, där sekvensen är förhandsbestämd och därför renderad i förväg. Denna avhandling behandlar den tidigare nämnda situationen, där det lönar sig att rendera simuleringen i realtid, det vill säga i takt med användarens indata. Då kan simuleringen hantera vilka indata som helst och rendera den korrekta bilden för varje tidpunkt.

För att simuleringen ska vara övertygande, bör bilder visas tillräckligt snabbt efter varandra på skärmen. Då upplevs simulationen som kontinuerlig, i stället för som en hackig bildserie. En känsla av kontinuitet kan uppnås redan genom att byta bilden sex gånger i sekunden. För att uppnå förväntad kvalitet på simuleringar i dag ska bytet dock ske tio gånger snabbare, dvs. 60 gånger i sekunden. Notationen bild byten i sekunden uttrycks ofta i FPS, ”frames per second” eller i Hz, hertz. För att uppnå denna frekvens av bild byten måste varje bild genereras inom en tidsram av ca 16 millisekunder (1 / 60). På grund av detta krav på hastighet har den moderna komponenten, grafikprocessorn, GPU:n, utvecklats. Komponenten har blivit allt mera vanlig under de senaste årtiondena, eftersom efterfrågan har ökat bland annat i spelbranschen.

Rendering har ändrats drastiskt inom de senaste 20 åren. Tekniken har standardiserats och metoderna som utförs för varje bild har organiserats till en piplinje av olika skeden. Avhandlingen presenterar grafikrenderingspipans skeden i detalj.

# Grafikprocessor

En grafikprocessor (GPU) är ett slags centralprocessor (CPU). En CPU är datorns viktigaste beräkningskomponent. En CPU har ett fåtal kärnor som klarar av att bearbeta självständigt från varandra. En GPU är en CPU som har flera men mindre kraftfulla kärnor. En ensamstående GPU kärna är svag, men den totala mängden beräkningar utförd av alla dess kärnor tillsammans är drastiskt högre än beräkningsförmågan av en CPU.

Om en beräkningstung operation kan spjälkas i flera deloperationer, kan dessa beräknas snabbare av en GPU än en CPU, eftersom den senare endast använder sig av en enda tråd. Därmed beräknar en GPU snabbt parallelliserade beräkningar tack vare den stora mängden kärnor. En GPU är därmed ett verktyg för tunga beräkningar som kan parallelliseras. För att utföra grafikberäkningar snabbt, har grafikkortet eget videominne, v-ram (video ram). Tack vare minnet har GPU:n åtkomst till ofta använda data med minimal fördröjning. En GPU har hårdvara som är dedikerad för interpolering och testning av z-djuphet. ([1, p. 29]).

GPU:n utvecklades för att försnabba grafikberäkningar. Grafikberäkningar består till största delen av tusentals eller miljontals små beräkningar. Dessa kan beräknas parallellt på en GPU, vilket möjliggör rendering i realtid för mera komplicerade scener.

Exempelprogrammet i denna avhandling använder sig av OpenGL biblioteket för att delegera grafikberäkningar till GPU:n.

Varje kärna har ett eget litet minne [1, p. 30]. Detta hjälper minimera väntetid vid till exempel texturering, då textur förfrågan inte behöver väntas på utan kärnan kan påbörja ett annat arbete och återkomma till textureringsskedet då texturinformationen befinner sig i minnet.

Varje enskilt anrop till nyanseringsprogrammet kallas en tråd. Flera trådar grupperas ihop till vågfrontar. Antalet trådar i en vågfront varierar från 8–64 [1, p. 34]. Grupperingen minimerar väntetid genom att synkronisera bytet av beräkningar då minnet hämtas.

Faktumet att GPU teknologi har utvecklats bort från hårdkodad hårdvara mot mera flexibel hårdvara som accepterar olika skuggningsprogram är det viktigaste i utvecklingen.

Alla moderna skuggningsprogram består av samma arkitektur, ISA (Instruction set architecture) [1, p. 35]. Tack vare standardiseringen kan GPU:n mera effektivt delegera beräkningarna till de olika komponenterna. Alla programmen är skrivna i antingen HLSL eller GLSL. Exempelprogrammet i denna avhandling använder OpenGL:s GLSL.

# Grafikrenderingspipeline

Grafikrenderingspipan är en standardiserad ordning av de steg som krävs för att rendera en bild. Huvudindelningen av pipan består av applikationsnivån, geometribearbetning, rasterisering och pixelbearbetning [1, p. 12]. Dessa kategorier har sina egna syften och mål för i vilket skick data ska vara i före nästa steg kan initialiseras. Det finns mycket valbarhet och renderingsstilen är oftast väldigt flexibel i dagens GPU hårdvara. Till exempel kan renderingen utföras stiliserat så att omgivningen liknar det ur en tecknad serie.

## Applikationsnivå

Applikationsnivån skiljer sig från de resterande kategorierna genom att dess operationer exekveras på CPU:n, medan de andra exekveras huvudsakligen på GPU:n. Kategorin är därmed pliktad med att utföra de operationer som GPU:n inte klarar av. Nivån kan bland annat sköta om kollisionsdetektering, globala accelerationsalgoritmer (dragningskraft), animation, fysiksimulationer. Beroende på vilken applikation som är i fråga körs de relevanta beräkningarna. ( [1, p. 13]).

## Geometribearbetning

Geometribearbetningsskedet består av flera mellanskeden: Vertexskuggning, tessalation, geometriskuggning, klippning och skärmkartläggning.





### Vertexskuggare

Modeller transformeras från modell-utrymmet till världs-utrymmet och sedan till kamera-utrymmet för att underlätta beräkningar. ( [1, p. 15]).

Vertex skuggningens första steg är indata assembleraren. Indata assembleraren beräknar primitiver, trianglarna, utifrån en räcka av positioner och en räcka färger. Dessa primitiver skickas vidare i pipelinen. Varje vertex som uppgör primitiverna kan innehålla färgdata eller information om textur koordinater. Vertexerna innehåller också normaler. Dessa normaler definierar åt vilket håll skuggaren ska runda trianglarna som vertexerna bildar (Om tessalation används). Rundande innebär att skuggaren ritar trianglarna med mjuka kanter. Vertex skuggningsberäkningarna är självständiga beräkningar, vilket innebär att de kan beräknas parallellt med GPU:n. Vanligaste stegen efter vertex skuggning är generering av trianglarna, rasterisering och sedan pixel behandling. ( [1, p. 42]).

### Tessalation

Tessalation är ett valbart skede. Tessalation innebär generering av runda ytor av varierande kvalitet beroende på kamerans avstånd till ytan [1, p. 18].

Processen innebär bildandet av nya trianglar ur ett fåtal trianglar. En större mängd trianglar kan bilda en yta som uppfattas rund. Dessa trianglar genereras i GPU:n för att minska på informationsflödet mellan GPU:n och CPU:n. Tusentals trianglar kräver mycket minne och det skulle skapa en flaskhals i dataöverföring om de skulle skickas från CPU:n till GPU:n. Generering av den mängd trianglar kräver mycket beräkningsresurser. Därför är det viktigt att kunna kontrollera mängden beroende på avståndet. Om en rundad yta ligger längre bort från kameran krävs det inte lika mycket trianglar för att uppehålla rundheten. Genom att kontrollera mängden trianglar och därmed mängden beräkningar kan renderingen uppehållas interaktiv.

Tessalation består av tre steg: skrov beskuggning, fördefinierad tessalation samt domän beskuggning.

Skrov beskuggningen tar in meshen som ska tesseleras och dess kontrollpunkter. Kontrollpunkterna definierar hur meshen ska delas inför generering av nya trianglar. Steget utför förberedande beräkningar på dessa kontrollpunkter. Steget beräknar hur många nya trianglar som ska genereras och i vilken ordning. Steget kan också modifiera indatat fritt före behandlingen. Steget skickar data vidare till både fördefinierade tessalationen och domän beskuggningen.

Fördefinierade tessalationen används endast om en tessalationsskuggare är i användning. Detta steg har som uppgift att generera nya vertexer som ska bilda nya trianglar i meshen. Skrov beskuggningen kan kräva att steget generar vertexer som uppgör antingen trianglar, fyrhörningar eller konturlinjer. Konturlinjer kan till exempel användas för att rendera hår. Skrov beskuggningen berättar också hur mycket trianglar som ska genereras innanför gamla trianglar samt hur mycket kanterna på meshen ska delas upp. Efter att de nya vertexerna har genererats skickas de till domänbeskuggaren. Domänbeskuggaren är lik vertex beskuggaren eftersom båda behandlar ett vertex åt gången. De nya vertexerna ges normaler samt texturkoordinater i enlighet med de gamla trianglarna. Domänbeskuggaren bildar sedan de nya trianglarna av vertexen och skickar dem vidare till nästa steg i grafikrenderingpipan. ( [1, p. 44]).

### Geometriskuggare

Geometriskuggning innebär generering av nya vertex. Skedet används oftast för att åstadkomma partikel effekter. Skedet följs av utdata strömning (outdatastream), vilket använder GPU:n som ett geometri motor för att generera nya data [1, p. 18].

Ström utdata kan skickas utanför GPU:ns piplinje. Utdata från Vertex beskuggaren, tessalation eller geometribeskuggaren kan strömmas tillbaka till programmeraren. Piplinjen kan därmed användas för icke-grafiska beräkningar och till och med iterativt behandlande av vertexer för att uppnå önskat resultat. Tekniken kan till exempel användas vid rendering av vatten, där vertexerna behandlas flera gånger genom piplinjen före de renderas. ( [1, p. 48]).

Geometriskuggaren har möjlighet att ändra primitiver till andra primitiver. Detta kunde inte tessalationsskedet göra. Geometribeskuggaren behandlar ett objekt åt gången. Ett Objekt består av en primitiv och möjliga extra vertex. Om primitivet är en linje kan de närstående vertexerna inkluderas. Om primitivet är en triangel kan de närstående trianglarnas tredje vertex inkluderas. Den extra informationen per objekt möjliggör mera funktionalitet. ([1, p. 47]).

Beskuggarens utdata kan vara noll. När geometribeskuggaren och tessalationsteget tas i beaktan kan programmeraren addera nya primitiver, ändra dem och radera dem fritt. Geometribeskuggaren är menad att modifiera indata eller alternativt generera kopior. Kopiorna kan användas för att rendera objektet från olika synvinklar samtidigt eller till exempel skapa päls genom att klistra kopiorna spikigt längs med konturerna [1, p. 47].

Algoritmer som genererar skuggor använder geometribeskuggaren för att generera skuggningskartor eller för att hitta kanterna i en scen [1, p. 47]. Geometribeskuggaren kan behandla samma objekt flera gånger vid behov.

Geometribeskuggaren är inte den bästa komponenten för att generera nya primitiver [1, p. 47]. Detta beror på att beskuggaren garanterar att primitiverna skickas ut i samma ordning som de tas emot. Eftersom beskuggaren använder sig av parallelliserade kärnor måste de hålla ordning på primitiverna, vilket drastiskt försämrar beräkningshastigheten.

Det finns endast tre steg som kan skapa nya beräkningar för GPU:n när rendering körs: Rasterisation, tessalation och geometribeskuggaren [1, p. 47]. Av dessa tre är geometribeskuggaren den mest oförutsägbara användaren av resurser och minne. Detta beror på att dess programmeringsgrad är högst, dvs. programmeraren har mycket kontroll över beräkningarna.

### Klippning

Klippande innebär reducering av de primitiva som befinner sig partiellt innanför kamerans vy. Detta görs för att undvika bearbeta onödig geometri som ändå inte syns i renderingen. Primitiver ”skärs” i gränsen av vyn genom att omforma primitivernas vertexer [1, p. 19].

### Skärmkartläggning

Skärmkartläggning beräknar koordinaterna från enhetskuben till skärmkoordinater. ( [1, p. 20]).

## Rasterisering

Rasterisering innebär att räkna ut vilken primitiv varje pixel tillbehör. Processen är indelad i två mellanprocesser. Triangel förberedande skedet bygger upp trianglarna mellan vertexen. Triangel genomgång beräknar vilka pixlar som befinner sig inuti de genererade trianglarna. Förberedande och genomgången beräknas av oftast av dedikerade transistorer. ( [1, p. 21]).



### Triangeluppställning

### Triangelgenomgång

## Pixelbearbetning

Pixel behandlande består av skuggning och hopslagning. GPUn utför detta enligt önskvärt program. ( [1, p. 22]).



### Pixelskuggare

Skuggarens mest kända uppgift är texturering, dvs. applicerande av en bild på en yta [1, p. 23].

Efter att vertex beskuggning, tessalation och geometribeskuggning har utförts klipps primitiven och förbereds för rasterisation. Vid rasterisation vandras varje triangel för att bestämma vilka pixlar som beskriver den. Triangelns delar som uppgör en pixel eller en del av den kallas fragment.

Pixelskuggaren hanterar dessa fragment. Fragmenten bildas för varje pixel då trianglarna vandras över och dess vertex värden interpoleras. Beroende på hur interpolationen utförs bestäms vilket perspektiv som används i renderingen. För att uppnå perspektivet som vi är vana vid som människor krävs en specifik interpolation medan andra mindre använda perspektiv använder andra interpolationer.

För programmeraren lyder det oftast att vertexskuggarens utdata är pixelskuggarens indata. Pixelskuggaren har åtkomst till fragmentets position på skärmen samt andra variabler som programmeraren definierat på förhand. Pixelskuggaren kan skapa genomskinlighet genom att modifiera fragmentets opacitet och z-djup. Pixel beskuggaren kan också fullständigt förkasta ett fragment och inte generera utdata för det. ( [1, p. 49]).

Då pixelskuggarprogrammet körs på ett fragment kan programmet inte använda sig av grannfragmenters resultat och programmets egna utdata kan inte användas i andra fragmenters beräkningar [1, p. 49]. Om programmeraren vill iterera med pixelskuggaren måste programmet köras i flera pass efter varandra.

Det finns ett minne som alla skuggare använder sig av. Minnet är ett skuggar buffert minne objekt (Shader storage buffer object). Minnet är skyddat av kapplöpning, dvs. att två trådar försöker modifiera minnet samtidigt vilket leder till fel. Minnet garanteras också att trådarna får tillgång till minnet i kronologisk ordning. Tack vare ordningen minskar mängden arbete sevärt i hopslagningsskedet. Problemet med att tvinga ordningen är att trådar kan tvingas vänta på sin tur och fördröja beräkningarna. Fördelen anses vara värt risken. ( [1, p. 49]).

### Hopslagning

Hopslagning tar itu med färg bufferten, som innehåller flera färgvärden per pixel, för att tilldela entydiga värden för varje pixel. Detta betyder att välja den färgen som är längst fram, närmast kameran. Djupheten är sparad som ett z-värde. Färgen som har det minsta z-värdet används för pixeln. ( [1, p. 24]).

I Hopslagningsskedet kombineras färgdata samt z-djupet av fragmenterna beräknade i pixelskuggarskedet. Resultatet skrivs in i rambufferten. Rambufferten är minnet för nästa bild som ska uppvisas på skärmen. Minnet används för att bygga upp den nya bilden för att sedan kunna byta ut den gamla bilden med den nya på en och samma gång.

I detta skede utförs z-buffert och stencil-buffert beräkningarna. Z-buffert beräkningen går ut på att jämföra fragmenters z värde, dvs. beräkna vilket fragment som är närmare skärmen. Det fragment som befinner sig närmast skärmen är den som ska visas på skärmen. Om det närmaste fragmentet är opakt, förkastas den. Om det närmaste fragmentet är genomskinligt i någon grad blandas dess färgvärde med följande fragment.

Många moderna GPU:er utför hopslagning i viss mån före pixelbeskuggning. Detta görs för att inte beräkna pixelbeskuggning i onödan. Om hopslagningen räknar ut att ett fragment kommer att skymmas av ett annat fragment beräknas inte pixelskuggningen för det skymda fragmentet över huvud taget. Detta kallas tidig-Z. Om pixelbeskuggarens program ändrar z-djup eller förkastar fragment kan tidig-Z inte användas, vilket försämrar på prestandan. Det finns möjligheter att tvinga tidig-Z med begränsningar. ([1, p. 53]).

Hopslagning är inte programmerbart men kan konfigureras med en hög grad av frihet. Speciellt blandande av färger har flera kombinationer av operationer som kan väljas. Färg och alpha värden kan adderas, subtraheras, multipliceras och funktioner som minimum och maximum kan appliceras. Bitvisa operationer kan också användas. ([1, p. 53]).

## Beräkningsskuggare

En GPU kan också utföra beräkningar som inte är relaterade med grafiker. Plattformar som CUDA och OpenCL kan användas för att hantera GPU:n som en massiv parallell processor. Beräkningsskuggare (Compute shader) använder sig av samma buffertar och kärnor som de grafiska skuggarna använder. Beräkningsskuggaren skiljer sig från de grafiska skuggarna genom att göra trådarna mera synliga. De indexeras vilket betyder att de kan följas med och kontrolleras effektivare. Trådarna ges ett litet gemensamt minne, oftast i storleksordningen 32kB. ([1, p. 54]).

Beräkningsskuggaren kan använda sig av data genererad på GPU:n [1, p. 54]. Faktumet gör beräkningsskuggaren snabbare än grafiska skuggarna, eftersom de tvingas kommunicera med CPU:n, vilket tar tid.

Beräkningsskuggare används ofta i grafiska beräkningar i form av efterbehandling. Somliga effekter kan exekveras två gånger snabbare med beräkningsskuggaren än pixelbeskuggaren [1, p. 54].

Beräkningsskuggare är användbara i flera fall. Några exempel är partikel system, mesh behandling, bild filtrering och förbättrande av djuphet i bilder samt skuggor [1, p. 54]. Beskuggaren är användbar i de flesta uppgifter där GPU:ns processorer utnyttjas till fullo.

# Viktiga koncept

## Vertexblandning

Vertex blandning går ut på att skapa ett ”skelett” till en modell. Modellens ”hud” reagerar på skelettets rörelser, vilket skapar bättre funktionalitet än att röra på modellen med stel-objekt transformer. Modellens vertex delas in i grupper för varje ben i skelettet. Vertexerna transformeras som vanligt i takt med benen. Vid gränsen av benen sker vertex blandning. Trianglarna som befinner sig i gränsen har vertexer som hör till båda benen. Därmed ”töjs” triangeln, huden, i takt med benens rörelser. Förutom töjningen kan flera trianglar skapas i efterhand för att jämna ut gränsen mellan benen. Varje vertex kan transformeras av flera ben, men med olika vikter. Vikterna kan bero på till exempel avståndet från benen till vertexen. En nackdel med vertex blandning är att det kan förekomma oönskade töjningar och vridningar. ([1, p. 84]).

## ?

Geometri minne återspelning (Geometry cache playback). Mellansekvenser kan skapas i högre kvalitet p.g.a. att simulationen är bestämd i förhand. Ett sätt att animera scenen är att spara alla vertexpositioner för varje bild. Detta är naivt eftersom kravet på minnet blir ohanterbart redan med små mängder av vertexer, till exempel 30 000. Genom att använda tekniken Gneiting kan minneskravet dras ner till 10% av originalet. ([1, p. 92]).

# Exempelprogram

Exempelprogrammets skelett är uppbyggt med hjälp av videoserien ”3D Game Engine Development Tutorial” på videogemenskapet YouTube. Programmet är skriven med Java och använder sig av två bibliotek för att underlätta det triviala. Ena biblioteket är LWJGL (Light Weight Java Game Developement Library) som innehåller OpenGL för att kommunicera med GPUn. Andra biblioteket är slick-util vilket underlättar hämtandet av texturer för texturering. Biblioteken används därmed inte för att exekvera själva renderingen, vilket implementeras i avhandlingen.

## Vektor

En vektor representeras i programmet av tre flyttalsvärden: x, y och z. Vanliga vektor operationer kan implementeras relativt fritt. Rotations operationen kan emellertid sevärt försnabbas genom att använda sig av kvaternioner.



Figur 1 - Metod för vektor rotering mha. kvaternioner

## Kvaternion

Kvaternioner används för att enkelt och säkert representera rotationer och riktningar. Kvaterioner är bättre än eulers transform och matriser. Kvaternioner är stabilare och lättare att bryta ner till sina grundkomponenter.

Kvaternioner är komplexa tal bestående av fyra värden. Av dessa är tre imaginära och ett är reellt. De tre imaginära representerar de tre grundkomponenterna z, y och z medan den reella komponenten kallas w, FÖRKLARING. De imaginära delarna kan hanteras med vanliga vektoroperationer som till exempel addition, förstoring och förminskning, punktprodukt och kryssprodukt. ( [1, p. 76].)

Kvaternion multiplikation används flitigt och implementeras som följande:



Inom programmering representeras kvaternioner med fyra reella tal [1, p. 76].

Enhetskvaternioner används för att enkelt representera vilken som helst rotation i tre dimensioner.

Kvaterioner kan transformeras till en matris och vice versa. Ändringen från en kvaternion till en matris är effektivare än det motsatta. ( [1, p. 79].)

## Transformer

Transformer är beräkningar som ändrar på ett objekts vertex positioner. De kan flyttas, roteras, förstoras och förminskas på ett plan. Transformer kan flytta objekt mellan olika koordinat system och projektera dem på olika sätt. Transformerna utgör grunderna för rendering. Utan transformer skulle rendering inte vara möjligt. Grundläggande transformer är translation, rotation, förstoring och förminskning.

Ett simpelt exempel på en transform är f(x) = 5x. x är en vektor vilkets alla komponenter multipliceras med 5. Detta är en lineär transform. Lineära transformer används för rotation och förstoring eller förminskning av objekt. Transformer kan lagras i en 3x3 matris. Storleken 3x3 grundar sig på de tre koordinataxlarna x, y och z som vi är vana vid i tredimensionella världar. Varje rad beskriver hur mycket inkommande vektorns värden ska ändra på antingen x, y eller z komponenten.

Addition av vektorer utför translation, dvs. förflyttning av en vektor till summan av vektorernas komponenter.

Vid rendering krävs oftast flera transformer under ett pass. För att effektivera beräkningarna bör enskilda transformerna sammanslås till en enda transform. Linjära transformen och en translation transform kan kombineras till en affin transform. En affin transform behåller parallellismen mellan den originella och den transformerade objektets linjer. Det som ändras i transformen är längden och vinklarna. En affin transform kan representeras med en 4x4 matris. Den fjärde raden betecknar den reella komponenten, w. x, y och z är imaginära värden Flera affina transformer kan kombineras till en entydig affin transform, vilket effektiverar beräkningarna. ([1, p. 57]).

Det är skillnad på ordningen som transformerna sammanslås. Olika ordningar av matriser ger olika resultat. Vid hopslagning av translation, rotation och förstoring eller förminskning är det viktigt att bibehålla rätt ordning. En vektor bör först förstoras eller förminskas, sedan roteras och till sist transleras till sin rätta position. Inom programmering ser ordningen ut som följande: V = T(R(S(v))). Förstoring eller förminskning beräknas först, sedan rotation och sedan translation. ( [1, p. 65]).

Stel-objekt transformer är transformer som endast innehåller translationer och rotationer. Detta innebär att transformen inte förvränger objektets former utan endast flyttar på objektet. ( [1, p. 66]).

### Translation

Translation är addition av vektorer. Vektorernas summa utgör den nya vektorn position [1, p. 59]

. Således har vektorn translerats. Vektor beräkningen ser ut på följande sätt:



Notera att riktningsvektorer inte kan transleras.

### Rotation

Rotation utförs längs med en av de tre axlarna i ett tredimensionellt utrymme. Liksom translation, bibehåller transformen avståndet mellan vertexerna och objektet speglas inte. Vid rotationstransformer används ofta en orientationsmatris. En orientationsmatris är en matris som används för att rotera ett orienterat objekt, dvs. ett objekt som har definierade värden för ”upp” och ”framåt”. ([1, p. 60]).

För att härleda matrisberäkningarna för rotation i tre dimensioner är det naturligt att först beskåda rotation i två dimensioner. En vektor v = (x, y) = (r \* cos(0), r \* sin(0))

BILD AV 2D

I tre dimensioner roteras objektet skilt längs med varje axel. Vid närmare analys av matrisberäkningarna framstår att den relevanta axelns värden inte ändras. Om en kub roteras längs med x axeln ändras inte kubens x-värden. Endast y- och z-värden ändras.



Dessa rotationsmatriser kan multipliceras med varandra för att skapa en entydig affin rotationsberäkning för att utföra den önskvärda rotationen på ett objekt. Tillsammans med translation kan objekt även roteras runt valfria punkter. Dessa beräkningar börjar med att translera objektet och punkten så att punkten befinner sig i origo. Detta utförs med translationen T(-p). Sedan utförs rotationen R(phi). Till slut transleras objektet tillbaka med T(p). Tillsammans ser transformen ut som följande: X = T(-p) \* R(phi) \* T(p). Notera att ordningen av matrismultiplikation har skillnad. ([1, p. 60]).

### Förstoring och förminskning

Förstoring och förminskning av vektorer utförs med multiplicering. En vektors komponenter kan multipliceras med samma konstant eller med olika konstanter. Det är vanligast att förstora och förminska alla komponenter lika mycket.



Förstoring och förminskning med negativa värden kan leda till att objektet blir en spegelbild. Detta kan avgöras genom att beräkna determinanten av transformen. PÅMINNELSE OM DETERMINANT. Om determinanten är negativ, blir slutresultatet en spegelbild. Ett objekt som är en spegelbild kan leda till renderingsproblem. Till exempel belysning kan beräknas fel och ytor kan visas svarta. ( [1, p. 62]).

### Skärning

Skärning åstadkommer förvrängda former. En kvadrat kan till exempel skäras till en romb, där arean består lika. Skärning utförs på en axel längs med en annan axel. Den första axeln beskriver vilken komponents värden som ändras och den andra beskriver riktningen för ändringen. ( [1, p. 63]).



### Normal

Normal transformen ser till att ytnormaler hanteras rätt. En ytnormal ska hålla sig rätvinklig mot dess yta. När man förstorar eller förminskar en yta längs med en axel förvrängs ytnormalen. Normal transformen används endast om systemet över huvudlaget transformerar normaler. Till exempel kan ett system i stället beräkna ytnormalen efter transformen med hjälp av kryssprodukten. ( [1, p. 63]).

### Invers

Inverterade transformer används ofta. Till exempel då ett objekt ska roteras runt en punkt translateras punkten och objekten först till origo för att utföra roteringen och sedan translateras de tillbaka, användandes av inversen. En transform kan inverteras enligt de tre följande situationerna med olika krav:

Ett; transformen består av hopslagna simpla transformer med givna parametrar. Då kan inverterade transformen beräknas genom att mata in inverterade parametrarna i de enskilda transformerna och sedan slå ihop dem i motsatt ordning.

Två; transformen är ortogonal, dvs. en rotation. Alla rotationers inverterade form är deras transponant, dvs. matrisen reflekterad längs med huvuddiagonalen.

Tre; Om inte ovanstående kraven uppfylls, används till exempel Cramers regel. Cramers regel är ett bra val p.g.a. att den innehåller en minimal mängd if-satser. ( [1, p. 69].)

### Euler

Euler transformen används för att orientera kameran eller ett objekt i enlighet med koordinataxlarna. Transformen byggs upp av tre rotationer, en för varje koordinataxel.

E(h, p, r) = Rz(r)Rx(p)Ry(h)

Eftersom transformen endast består av rotationer är den ortogonal och kan enkelt inverteras. h står för head, p för pitch och r för roll. I vissa simuleringar kallas head i stället för yaw, till exempel i flygsimuleringar. För att förstå sig på benämningarna kan en tänka sig att kameran är ens egen syn: ändring av head kan jämföras med att skaka på huvudet för att signalera ”nej”. Ändring av pitch är kan exemplifieras genom att nicka. Ändring på roll lutar på huvudet till någondera sida. Detta exempel antar att ”upp” i världen beskrivs av y-axeln.

Euler transformen har sina nackdelar. Då flera Euler transformer hanteras efter varandra kan fel uppstå. Därför används oftast kvaternioner i stället för Euler transformen för att representera riktning. ( [1, p. 70].)

### Kamera



### Projektion



### Konstruktion



## Kamera





## Projektion

Projektering innebär translation av objekt så att scenen sedan kan klippas och renderas. ( [1, p. 92].)

Ortografisk projektion är en simpel projektion som endast plattar till alla objekten i scenen så att de bildar ett tvådimensionellt plan. I ett koordinatsystem där z-koordinaten är ”djupet” ändras z komponenten till 0 medan x och y komponenterna förblir oförändrade. Projektionen är onaturlig för oss eftersom objektens storlekar inte ändras beroende på hur nära de är kameran. Detta åstadkoms i perspektiv projektion. ( [1, p. 93].)

Perspektiv projektion innebär att storleken på objekt beror på hur nära de är kameran. Linjer som är parallella före projektion är oftast inte det efter projektion. I perspektivprojektion medförs ett avstympat parti. Det avstympade partiet är formen av en pyramid där toppen huggits av. Formens volym beskriver vad kameran ser. När rendering sker transformeras avstympade partiet till enhetskuben. Detta görs p.g.a. att följande stegen är sevärt lättare att beräkna i en enhetskub än det avstympade partiet; nämligen skärning och homogenisation. Projektionen beräknas ofta så att kameran befinner sig i origo och tittar nerför den negativa z-axeln. Då beskrivs den avstympade partiets ”topp” av z = n (near) och ”bottnen” av z = f (far). Genom att ändra på det avstympade partiets dimensioner ändras synfältets storlek. Ett bredare botten på det avstympade partiet leder till ett bredare synfält. ( [1, p. 96].)



## Resultat

# Sammandrag

Litteraturförteckning

|  |  |
| --- | --- |
| [1]  | E. H. N. H. Tomas Akenine-Mo¨ller, ”Real-Time Rendering,” i *Fourth Edition*, Boca Raton, Taylor & Francis Group, 2018, p. 1198. |
| [2]  | thebennybox, Regissör, *3D Game Engine Development Tutorial.* [Film]. Youtube, 2014.  |