Borttagning av skymda ytor – Painter's Algorithm och Z-buffering

MICHAEL HJORTHOLT
och ANDREAS PAULSSON

KTH Datavetenskap
och kommunikation

Examensarbete
Stockholm, Sverige 2010

Borttagning av skymda ytor – Painter's Algorithm och Z-buffering

M I C H A E L H J O R T H O L T
o c h A N D R E A S P A U L S S O N

Examensarbete i datalogi om 15 högskolepoäng
vid Programmet för datateknik
Kungliga Tekniska Högskolan år 2010
Handledare på CSC var Lars Kjelldahl
Examinator var Lars Kjelldahl

URL: www.csc.kth.se/utbildning/kandidatexjobb/datateknik/2010/hjortholt_michael_OCH_paulsson_andreas_K10030.pdf

Kungliga tekniska högskolan
Skolan för datavetenskap och kommunikation

KTH CSC
100 44 Stockholm

URL: www.kth.se/csc

Borttagning av skymda ytor – Painter's Algorithm och Z-buffering

Sammanfattning

Rapporten behandlar två algoritmer som båda löser datorgrafikproblemet med skymda ytor. Först ges en kort introduktion till problemet samt till datorgrafik i allmänhet. Därefter presenteras de två algoritmerna, Painter's Algorithm och Z-buffering, och deras för- och nackdelar. Efter det jämförs algoritmerna och hur väl de fungerar i olika scenarion. Baserat på slutsatserna från jämförelsen, diskuteras hur en kombination av algoritmerna skulle kunna skapas, samt vilka för- och nackdelar den kombinerade algoritmen skulle ha. Slutsatser dras att en kombinerad algoritm inte behöver rita om pixlar flera gånger, däremot kommer många andra av de redan existerande nackdelarna kvarstå.

Removal of hidden surfaces – Painter's Algorithm and Z-buffering

Abstract

The report discusses two algorithms which both solves the computer graphics-problem; removal of hidden surfaces. You are first given a short introduction to the problem, as well as a short introduction to computer graphics in general. After that the two algorithms, Painter's Algorithm and Z-buffering is presented together with its advantages and disadvantages. Then a comparison is made between the two. Based on the conclusions of that comparison a possible combination of the two algorithms is discussed, along with the advantages and disadvantages that such a combination would have. The problem with the overdrawing of pixels desists, but the majority of the disadvantages still persist.

Förord

Arbetsfördelning har varit så att avsnitten har delats upp mellan Michael Hjortholt och Andreas Pålsson. Därefter har båda läst igenom rapport och kommenterat och gjort ändringar.

Michael Hjortholt har skrivit följande avsnitt: Sammanfattning, Abstract, Introduktion till datorgrafik och Painter’s Algorithm

Andreas Pålsson har skrivit följande avsnitt: Object Space and Image Space, Z-buffering, Jämförelse av algoritmerna och Algoritmerna kombinerade.

Inledning är skriven av båda.

Innehållsförteckning

Inledning

Bakgrund

Dagens datorer blir allt snabbare och en av de grenar inom datalogin som pressar utvecklingen framåt är datorgrafiken och då speciellt 3D-grafiken. Datorgrafik har sitt ursprung i början av 1960-talet då William Fetter var den första att mynta ordet datorgrafik för att beskriva sitt arbete vid Boeing [4]. De som gjorde de största framstegen inom datorgrafik under 1960-talet var University of Utah som rekryterade många av pionjärerna inom datorgrafik. Det var även University of Utah som gjorde det första stora framsteget inom 3D-grafik genom att utveckla algoritmer för borttagning av skymda ytor [5].

Borttagning av skymda ytor är den mest elementära och viktigaste delen i 3D-grafik som man måste kunna behärska för att kunna skapa en korrekt bild. Skillnaden mellan 2D-grafik och 3D-grafik är djupet. Djupet medför att vissa objekt kan stå framför andra och därmed skymma delar i en bild. Om det inte skulle gå att avgöra vilka objekt som skall synas i en bild, skulle ordningen som objekten ritas ut i, helt bestämma bildens utseende. Det betyder att om man väntar med bakgrunden till sist och först ritar ut allt annat så skulle bara bakgrunden synas eftersom den kommer att rita över allt annat som redan har ritats. Det som algoritmer för borttagning av skymda ytor gör är att räkna ut vad det är som skall synas i den slutgiltiga bilden och vad som kommer att vara skymt av framförvarande objekt. Utan dessa algoritmer för att bestämma skymda ytor skulle den 3D-grafik som vi är vana vid i film och datorer idag inte finnas.

Syfte

Syftet med denna uppsats är att jämföra två grundläggande algoritmer för borttagning av skymda ytor och visa deras för och nackdelar, för att sedan avgöra om det går att kombinera de två algoritmerna till en ny algoritm. En algoritm som då förhoppningsvis kan dra fler fördelar än nackdelar från de båda andra algoritmerna

Problemformulering

Utforska två kända algoritmer för borttagning av skymda ytor, Painter’s Algorithm och Z-buffering, för att sedan se vilka fördelar respektive nackdelar en kombination av dessa algoritmer för med sig. Hur fungerar algoritmerna var för sig? Hur används de? Är det möjligt att kombinera algoritmerna och i vilket område kan den kombinerade algoritmen vara till nytta?

Introduktion till datorgrafik

Datorgrafik är ett extremt vagt uttryck eftersom det egentligen täcker in allt grafiskt som visas med hjälp av en dator. I den här rapporten som handlar om skymda ytor kommer 3D-grafik att tas upp. 3D-grafik är hur man med hjälp av datorn skapar och ritar upp tredimensionella miljöer. Dessa används inom en rad områden, allt från spel och film till modelleringar inom industrin.

Bilder i en dator kan lagras och ritas upp med hjälp av ett rutnät med små, små rutor med olika färger. Varje ruta i bilden kallas för en pixel. Ett enkelt sätt att lagra en bild är att spara en siffra för varje pixel, där siffran representerar den färg som pixeln har. Vanligtvis sparas färgerna enligt RGB-skalan, där tre siffror, mellan 0 och 255, representerar intensiteten för vardera röd, grön och blå. I en svartvit bild räcker det att spara en siffra för varje pixel, den siffran svarar då för intensiteten i en gråskala, där 0 är helt svart och 255 är helt vitt.

En 3D-miljö skapas i en dator som matematiska beräkningar vilka beskriver de olika objekt som finns i miljön, samt deras egenskaper. För att representera objekt i 3D är den vanligaste metoden att skapa polygoner. Polygonernas hörn lagras då som deras koordinater i rummet. Enklast är att använda trianglar. Dessa trianglar sätter man ihop till avancerade figurer och former. Enklare former som t.ex. kuber och klot kan man lagra och rita ut efter deras matematiska formler. Detta betyder att för en sfär behövs endast koordinaterna för mittpunkten samt radien på sfären lagras för att den ska kunna ritas ut.

För att kunna visa dessa matematiska modeller på t.ex. en bildskärm, där ytan är tvådimensionell måste de matematiska modellerna projiceras. Detta kallas för rendering. Man kan beskriva det som att man drar linjer från objektets hörnpunkter i rummet, till kameran (den punkt objektet betraktas från). Dessa linjer passerar genom ett plan, detta plan är bildskärmen. I de punkter där linjerna skär planet ritas hörnpunkter ut och då får man en projicering. Fyller man alla punkter mellan de utritade punkterna får objektet rätt form. Detta kallas för att rastrera.

Om man ritar ut flera objekt kan de rastrerade ytorna överlappa varandra. Då måste det bestämmas vilken yta som skall synas och vilken som skall vara dold. Självklart skall den yta som i rummet är närmast kameran ritas ut. Detta avgörs på ett effektivt sätt med hjälp av olika algoritmer för att ta bort skymda ytor.

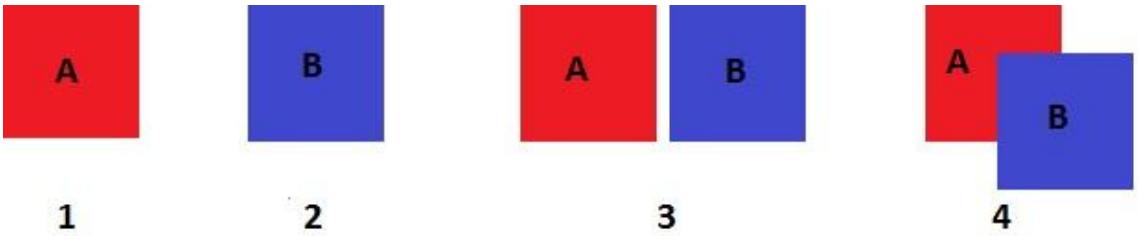
Algoritmer

Object Space and Image Space

Inom borttagning av skymda ytor finns två kategorier av algoritmer. Dessa är Object Space och Image Space [1]. Skillnaden mellan de två är hur man ser på bilden. I en Object Space-baserad algoritm, ser man till varje polygon (objekt) i en scen. En generisk Object Space-algoritm skulle till exempel kunna vara att jämföra alla polygoner parvis.

Ex. Det finns en scen med *k* st. polygoner där två plockas ur, polygon A och polygon B. Sedan jämförs hur dessa två polygoner förhåller sig till varandra sett från kameran. Det finns då fyra möjligheter.

1. A skymmer hela B sett från kameran, bara A ska visas.
2. B skymmer hela A, bara B ska visas.
3. Både A och B är helt synliga, båda ska visas.
4. A och B täcker varandra till viss del, då måste man räkna ut de synliga delarna för varje polygon.



Figur 1: De fyra möjliga polygon-förhållandena

För enkelhetens skull räknar man med att det tar en enda operation att bestämma om en del av polygonen är synlig. Alla polygoner kommer att gås igenom iterativt. Man börjar med den första polygonen och jämför den parvis med de resterande k-1 polygonerna i scenen. Detta talar om vilka delar av polygonen som är synlig. Sedan görs samma operationer för de resterande k-1 polygonerna i scenen. Detta kommer att ge en tidskomplexitet på $O(k^2)$. Genom detta kan man se att Object Space-baserade algoritmer troligen fungerar bäst där antalet polygoner i en scen är lågt.

Image Space-algoritmer i sin tur använder strålar för att avgöra vad som syns. Det går ut på att en stråle skickas för varje pixel. Strålen startar i kameran och går igenom hela scenen. Strålen kommer då att korsa varje plan som våra *k* polygoner skapar, bestämma i vilket plan strålen passerar en polygon, samt för de strålar som passerar en polygon bestämma vilken polygon som är närmast kameran. Sedan är det bara att färga den pixeln med den färg polygonen har där strålen korsar den. Den karakteristiska operationen är att hitta korsningen mellan polygonen och strålen. Om upplösningen är *n*m* pixlar kommer operationen att utföras *n*m*k* gånger, vilket ger tidskomplexiteten O(*k*). Men eftersom algoritmen använder en stråle per pixel är noggrannheten begränsad av antalet pixlar.

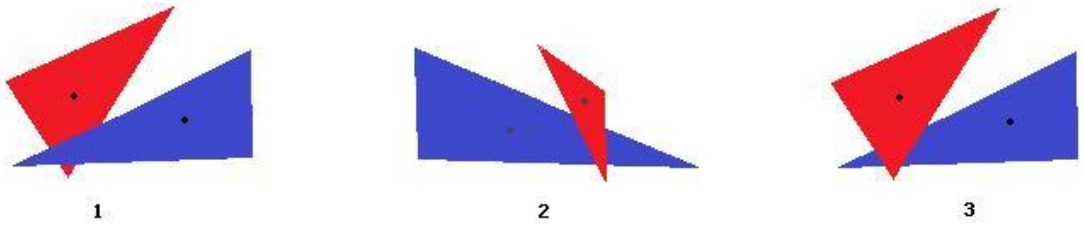
För att sammanfatta kan man säga att Object Space-algoritmer ser till hur alla polygoner förhåller sig till alla andra polygoner i scenen och är till största del beroende av antalet polygoner i scenen. Medan Image Space-algoritmer skickar strålar för varje pixel på skärmen och ser var strålen träffar en polygon i scenen. Den är därför istället beroende av antalet pixlar.

Painter's Algorithm

Painter's Algorithm är en av de enklaste och mest grundläggande algoritmerna för projicering av objekt i en 3D-miljö på en 2D-yta. Som namnet antyder fungerar algoritmen lite som en målare skulle måla en tavla. En oljemålare börjar med att måla det som är längst bort från ögat för att sedan måla saker som skall synas närmare, ovanpå det gamla. Precis likadant fungerar Painter's Algorithm. De objekten som är längst bort från kameran ritas ut först och sedan ritas de objekten ut som är närmare och närmare kameran. De närmsta objekten kommer till slut täcka de delar av de bortre objekten som inte skall synas. Eftersom algoritmen jämför de olika objekten, det vill säga polygonernas placering i scenen räknas den in som en Object Space-algoritm.

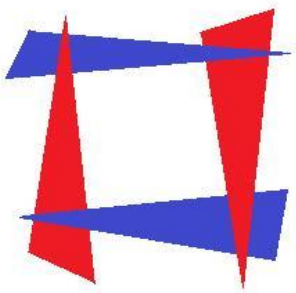
Svårigheten med Painter's Algorithm är att bestämma i vilken ordning polygonerna skall ritas ut. Det finns både enkla och mer avancerade lösningar för att räkna ut detta. En enkel lösning är att välja en bestämd punkt på alla polygoner, t.ex. polygonens mittpunkt och därefter jämföra avstånden mellan mittpunkten och kameran. Därefter ritas polygonerna ut i ordning, startandes med den polygon som har längst avstånd mellan sin mittpunkt och kameran.

Detta fungerar dock tyvärr inte alltid. I fallet i figur 2, skall den blå polygonen täcka den röda som i illustrationen till vänster (1). Men om uppställningen betraktas från sidan, ser man att den blå polygonens mittpunkt sitter längre bak än den röda polygonens (2). Vid utritning kommer då den röda polygonen ritas ut sist och därför felaktigt täcka den blå (3).



Figur : 1. Den tänkta bilden. 2. Bilden sett från sidan, här syns att den blåa polygonens mittpunkt sitter längre bak än den rödas. 3. Resultatet av utritning med Painter's Algorithm.

Ett annat exempel på hur polygoner kan vara placerade för att de inte skall kunna ritas ut på ett korrekt sätt ges i figur 3. Här täcker varje polygon en annan polygon i en cykel. För att kunna rita upp sådana polygonuppställningar krävs en annan teknik. Polygonerna behöver delas upp i mindre bitar innan de avståndsbedöms. Det är här som den annars så enkla Painter's Algorithm blir mer komplex.



Figur 3: Exempel på en polygoncykel som inte kan ritas ut av en enkel Painter's Algorithm.

Ett effektivt sätt att dela upp polygoner i mindre delar är med hjälp av ett BSP-träd (Binary Space Partitioning Tree). En BSP delar polygonerna i mindre bitar och organiserar subdelarna i ett träd.

Detta träd kan sedan gås igenom med hjälp av en djupet först sökning för att rita ut polygonerna i rätt ordning.

Pseudokod för Painter´s Algorithm kan skrivas som följer:

```
public void painter() {
    sort objects by z;
    for all objects {
      for all pixels (x, y) {
         paint object.color;
      }
    }
}
```

Effektiviteten för Painter's Algorithm är starkt beroende på vilken split- och sorteringsalgoritm som används. Värsta fall med en enkel sorteringsalgoritm är O(n^2). Men har en snittid på O(n log n). [6]

Om en BSP används tar det O(n log n) i snitt att göra uppdelningen och sorteringen och sen O(n) för att vandra igenom trädet som skapats. I värsta fall tar uppdelningen och sorteringen O(n^2) och O(n^2) i värsta fall. [6]

Användningsområde

Painter's Algorithm har använts till 3D-spel tidigare , men är idag inte lika vanlig förekommande. Främst beror det på att dagens spel är mycket mer detaljerade och består av fler objekt som behövs ritas ut. Eftersom algoritmen ritar ut samtliga objekt i bilden blir det många onödiga utritningar.

I andra områden, med simplare grafik, som t.ex. Postscript och CSS används Painter's Algorithm för att rita ut objekt i rätt ordning. Dessa bilder brukar dock inte vara tredimensionella i samma mening som moderna datorspel är.

Fördelar

Den stora fördelen med algoritmen är att den är väldigt enkel att förstå. Det är bara att rita över gamla pixlar med nya. Det enkla konceptet gör även att den är tämligen enkel att implementera.

En annan fördel med konceptet att rita alla pixlar i djupled och sen rita över dem, är att transparanta ytor och pixlar inte behöver särskilda beräkningar. Om en genomskinlig yta ligger framför en solid yta är pixeln redan rätt färgad.

Nackdelar

När man tar bort skymda ytor inom 3D grafiken gör man det av två anledningar. Den ena anledningen är att bilden som genereras ska bli vacker och se ut som det är tänkt när objekten i scenen skapades. Den andra anledningen är att slippa beräkna och rita ut det som ändå inte kommer att synas, och då spara på beräkningskraft.

Painter's Algorithm är egentligen inte jättebra för något av fallen. Det är tämligen komplicerat att avgöra vilka objekt och polygoner som ska ligga framför de andra för att få en bra slutbild. Särskilt

svårt är det att skapa en snabb algoritm som delar upp polygoner som täcker varandra och som fungerar allmänt och inte bara för ett specifikt fall.

BSP är bra på detta men det blir aldrig perfekt, dessutom tar det ganska lång tid att skapa trädet (O(n log n)) vilket gör det svårt att använda Painter’s Algorithm i 3D spel och liknande som kräver många omritningar av scenen i realtid.

En stor nackdel med algoritmen är även den överritning som görs hela tiden. Samma pixel kan ritas om flertalet gånger innan bilden är färdig. Den ständiga överritningen innebär att det kan ta mycket lång tid att rita ut en bild om det finns många lager med polygoner i en scen.

Z-buffering

Z-buffering går ut på att när en polygon kommer till algoritmen så räknar man ut avståndet mellan varje pixel i polygonen och bildplanet. Om detta avstånd är mindre än tidigare polygons pixel med samma x- och y-värde betyder det att denna pixeln i polygonen är framför den förra och därmed ersätter man den gamla pixelns färg med den nya som är närmare.

För att använda Z-buffering måste man ha två matriser en framebuffert som sparar färgvärdet för varje pixel som ska ritas ut på skärmen och en z-buffert som sparar djupet för pixlarna. Framebufferten initieras till bakgrundfärgen och z-bufferten till maxdjupet. Både z-bufferten och framebufferten måste vara lika stora som antalet pixlar som man ska rita ut. Så om man ska rita upp en scen som ska ha 1920x1080 pixlars upplösning måste både z-bufferten och framebufferten som man använder vara 1920x1080 för att kunna spara ett djupvärde och färgvärde för varje pixel

När man skapar z-bufferten bestäms samtidigt precisionen eftersom antalet bitar som används för att spara djupet sätts. Detta betyder att om en 16-bitars z-buffert används, finns det 65536 värden som djupet kan anta. Används istället en 24-bitars z-buffert kommer bättre precision erhållas, eftersom det då finns 16777216 olika värden. Bufferten kommer dock att ta mer plats i minnet.

Eftersom z-bufferten har en begränsad precision sett till antal unika djupvärden som den kan använda är det inte bra att köra den på alltför stora ytor. Detta eftersom värdena blir mer utspridda och precisionen blir sämre. För att förhindra detta sätts två klipplan upp, mellan vilka man kör Z-bufferingen. En för det främre klipplanet och en för det bakre klipplanet. Alla polygoner som ligger utanför dessa två klipplan kommer z-buffertalgoritmen inte att ta hänsyn till. Genom att lägga dess gränser närmare eller längre ifrån varandra kan precisionen i z-bufferten varieras. Förhållandet mellan dessa två gränser påverkar även hur z-buffertens olika värden fördelas. Om förhållandet mellan de två gränserna är 100 kommer 90 procent av de olika värdena som z-bufferten kan innehålla, användas av de 10 första procenten av djupet. Om förhållandet istället är 1000, kommer 98 procent av värdena användas av de första 2 procenten av djupet [3].

Pseudokoden för z-buffering kan skrivas så här:

```
For each polygon p{
   For each pixel in p{
      If p.pixeldepth(x,y) < zbuffer(x,y){
        zbuffer(x,y) = p.pixeldepth(x,y)
        framebuffer(x,y) = p.pixelcolor(x,y)
      }
   }
}
```

För varje polygon som kommer in kontrolleras alla pixlar i den polygonen. Om en pixel har ett mindre djup än det djup som är lagrat i z-bufferten, det vill säga den här pixeln är framför den gamla pixeln, ersätts det gamla djupet i z-bufferten med det djupvärde som den nya pixeln har. Samtidigt ersätts det gamla färgvärdet i framebufferten med färgvärdet som den nya pixeln ska ha.

Användningsområden

Z-buffering har inbyggt stöd i nästan alla grafikkort i dagsläget [3] och används både inom Direct3D och OpenGL som är de två största grafikbiblioteken. Därmed används det i nästan alla spel med 3D-grafik som kommer ut.

Fördelar

Online

En av de stora fördelarna med Z-buffering är att den är online. Det betyder att den inte behöver ha all data på en gång innan algoritmen körs. Istället kan algoritmen ta in data i portioner och köra ändå. I det här fallet betyder det att inte alla polygoner som ska visas på skärmen behövs från början när Z-buffering ska köras, utan den kan matas polygoner allt eftersom de kommer. Detta gör att Z-buffering blir effektivare eftersom den slipper stå och vänta på att få alla polygoner. Istället kan den köra igång direkt på de polygoner som den har fått och sedan fortsätta tills den är klar med alla polygoner i scenen.

Lätt att göra i hårdvara

Eftersom Z-buffering är en relativt enkel algoritm som bara kräver extra minnesutrymme för att implementeras är det lätt att bygga in Z-buffering i hårdvara och i dagsläget stödjer nästan alla grafikkort Z-buffering [3]. Det som däremot skiljer sig mellan olika hårdvaruimplementationer är hur många bitars precision de har i sin z-buffert vissa 16-bitar och vissa 24-bitar. Det finns även de som har 32-bitar.

Nackdelar

Minne

En av nackdelarna med Z-buffering är att den kräver mer minne. För att kunna rita ut saker på skärmen måste man först och främst ha en bildmatris eller framebuffer. Den behöver vara lika stor som upplösningen på skärmen. Det vill säga om skärmupplösningen är 1920x1080 så kommer matrisen vara 1920x1080 stor och varje element kommer att innehålla från 32-bitar ner till 16-bitar beroende på inställningar. Detta betyder att framebufferten kommer att ta ca 8MB minne om den innehåller 32-bitar. För att kunna använda Z-buffering måste även en z-buffert finnas, för att lagra djupvärden för varje pixel. Den kommer också vara lika stor som skärmupplösningen. Eftersom det är olika stora tal som väljs att z-buffern kan innehålla, varierar det hur mycket minne som krävs. Om en 16-bitars z-buffer väljs, som då kan innehålla 65536 distinkta värden, kommer den ta ca 4MB minne om upplösningen är 1920x1080. Om istället en 24-bitars z-buffer används, kommer den kunna innehålla 16777216 distinkta värden och får markant bättre precision, men den kommer att ta ca 6MB minne med samma upplösning.

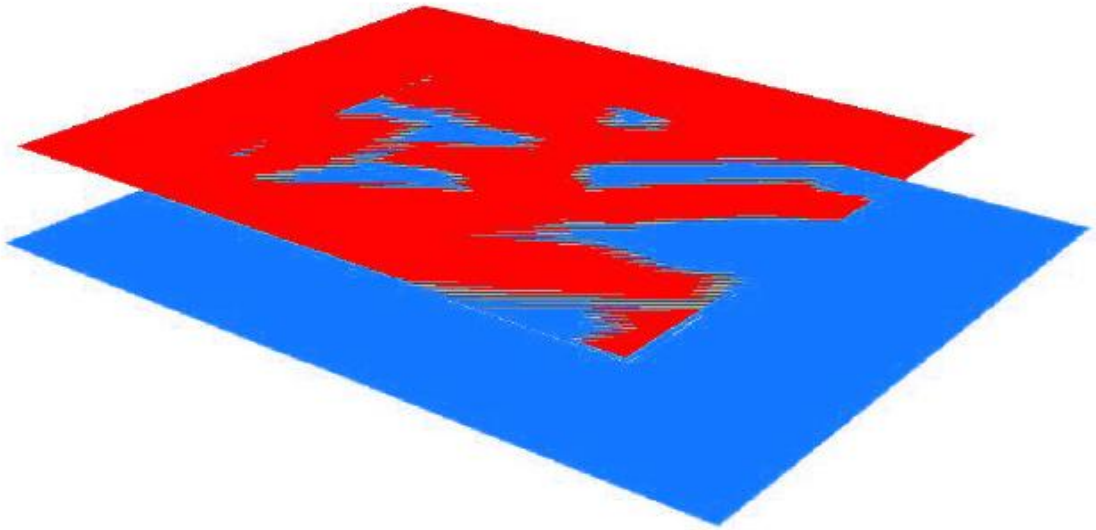
Överritning

Att behöva rita om pixlar flera gånger kan vara ett problem som även uppstår med Z-buffering. Detta eftersom man inte vet i vilken ordning som polygonerna kommer till algoritmen och de skulle kunna i värsta fall komma så att man måste rita om samma pixel varje gång. Men i realiteten är detta ett betydlig större problem för Painter's Algorithm än för Z-buffering.

Z-fighting

Z-fighting uppstår när två polygoner ligger så nära varandra att algoritmen inte kan avgöra vilken polygon som är närmast betraktaren och därmed ska ritas ut. Eftersom algoritmen inte kan avgöra vilken färg som ska målas ut kommer det vara slumpmässigt vilken färg pixeln får vilket ger upphov till att det kommer att ritas ut fragment från båda polygonerna där de korsar varandra. Detta kan uppstå av flera anledningar bland annat på grund av avrundningsfel i uträkningar eller för dålig precision i z-bufferten vilka gör att polygonerna får samma djup även om de inte ligger på samma djup.

Ett sätt att undvik Z-fighting, eller i alla fall göra det mindre vanligt när man har stora ytor som man vill köra Z-buffering på, är att dela upp renderingen och köra Z-buffering på mindre del ytor. På så sätt kan man få bättre precision på varje del yta men till kostnaden att det kommer ta längre tid.



Figur 4: Exempel på Z-fighting

Jämförelse av algoritmerna

De två algoritmerna har mycket olika tillvägagångssätt för att lösa samma problem. Painter's Algorithm bygger på hur man tidigare har gjort för att skymma ytor detta i kontrast till Z-buffering som man skulle kunna säga är ett mer datalogiskt tillvägagångssätt. På grund av deras olika ursprung är de mer eller mindre lämpade för olika typer av scener. Det här är de fyra scener som kommer att jämföras för att visa ytterligheterna i algoritmerna.

1. Enkel scen med låg upplösning. Scenen innehåller 24 polygoner (fyra kuber) och har en upplösning på 160x120.

2. Enkel scen med hög upplösning. Samma som i scen 1 men upplösningen har höjts till 1920x1080.

3. Komplex scen med låg upplösning. Scenen innehåller 5000 polygoner på olika djup och som korsar och går in i varandra. Den har en upplösning på 320x240.

4. Komplex scen med hög upplösning. Samma scen som i exempel 3 men med upplösningen 1920x1080.

Enkel scen med låg upplösning

Om man har en enkel scen det vill säga att det är få polygoner i scenen och låg upplösning kommer båda algoritmerna att gå snabbt. Painter's Algorithm är i viss mån beroende av hur många överritningar[1] den kommer att behöva göra men eftersom det bara är ett fåtal polygoner i scenen så kommer troligen antalet överritningar vara få. Man kan nästan bortse från sortering som krävs för Painter's Algorithm när det är så här få polygoner.

Enkel scen med hög upplösning

I exempel 2 används samma scen som i exempel 1 men med ökad upplösning. Detta kommer att innebära att tiden det tar för Painter's Algorthim kommer att öka eftersom den kommer att behöva färglägga fler pixlar i scenen. Däremot kommer övriga delar av algoritmen inte att påverkas. Z-bufferingen kommer däremot påverkas på flera sätt av den ökade upplösningen. Dels kommer storleken på z-bufferten att öka eftersom den måste vara lika stor som den upplösning som man ska rita ut. Det kommer även ta längre tid att köra genom Z-bufferingen eftersom algoritmen inner loopen som körs för varje polygon kommer ta längre tid på grund av det ökade antalet pixlar.

Komplex scen med låg upplösning

Kollar man på den komplexa scenen i exempel tre, kommer det ta betydligt längre tid att gå igenom för båda algoritmerna eftersom det är betydligt fler polygoner och chansen för överritningar växer med antalet polygoner i scenen. Eftersom upplösningen är låg, kommer antalet pixlar som ritas om varje gång däremot vara lågt men överritning kommer fortfarande spela en stor roll. Detta kommer göra att Painter's Algorithm blir långsammare, eftersom den grundar sig på överritning och komplexa scener medför flera polygoner som kan skymma andra polygoner vilket innebär det fler överritningar. Z-buffering kommer även den lida av överritningsproblem men inte alls i samma omfattning som Painter's Algorithm. För Z-buffering så är överritningsproblemet helt beroende på hur polygonerna

[1] Eftersom den här rapporten handlar om borttagning av skymda ytor och inte tid för att färglägga en pixel kommer det att antas att färgläggning av en pixel tar konstant tid.

kommer till algoritmen. Om de kommer sorterade med polygonen längst bak i scenen först, kommer Z-buffering ha lika många överritningar som Painter's Algorithm. Men om man antar att de inte kommer i någon speciell ordning bör den göra ungefär 50 %[2] färre överritningar än vad Painter's Algorithm gör.

När antalet polygoner blir så här stort, förlorar Painter's Algorithm även mycket på att behöva vänta på att alla polygoner ska blir klara med rendering och rasterisering som görs tidigare. Om man för enkelhetens skulle antar att rendering och rasterisering tar konstant tid för varje polygon betyder det att tiden som Painter's Algorithm måste vänta innan den kan börjar sortera polygonerna växer linjärt med antalet polygoner i scen. Något som Z-buffering inte behöver ta hänsyn till eftersom den kan köra så fort den får en polygon.

Nu är det även så många polygoner i scenen att sorteringen som måste göras för Painter's Algorithm gör sig påmind och gör så att algoritmen går långsammare. Om BSP används för sortering kommer denna sortering ta O(n log n) i snitt.

Komplex scen med hög upplösning

Detta är det exempel som bäst speglar dagens användning av 3D-grafik, en komplex scen med många polygoner och hög upplösning. Precis som i tidigare exempel så påverkas Painter's Algorithm av upplösningshöjningen genom att det tar längre tid att göra färgläggningen av polygonen och därigenom blir det mer kostsamt att göra överritningar. Som nämndes i det tidigare exemplet gör detta att Painter's Algorithm tar allt mer tid och mycket av tiden läggs på att göra överritningar och därmed ta bort saker som har gjorts i onödan. Z-buffering behöver som vanligt mer minne när upplösningen ökar och den inre loopen som går igenom alla pixlar i polygonen kommer även att köras flera gånger. Precis som i det förra exemplet kommer överritning även vara ett problem för Z-buffering men inte i samma utsträckning som för Painter's Algorithm.

Slutsats

Det som man ser i denna jämförelse är att när det är enkla scener med få polygoner, spelar det mindre roll vilken algoritm man använder och hur mycket arbete algoritmerna gör. Men när antalet polygoner ökar syns även hur algoritmerna skiljer sig åt. Vid komplexa scener så märks många av nackdelarna med båda algoritmerna. Painter's Algorithm har flera flaskhalsar med att behöva vänta in alla polygoner och sortera dem innan den kan köra. Den lider även av att den kommer att göra många färgläggningar i onödan på grund av överritningen. Däremot påverkas den inte lika mycket när upplösningen ökar. Till skillnad mot Z-buffering där det just är upplösningen som har den största påverkan förutom antalet polygoner i scenen. Z-bufferingens minnesanvändning ökar också i takt med att upplösningen växer. Men Z-buffering har inte samma problem med överritning som Painter's Algorithm har och ritar garanterat ut en korrekt bild, något som inte är säkert för Painter's Algorithm. Z-buffering är även effektivare eftersom den kan köras så fort den får polygoner och slipper vänta in att alla ska bli klara i tidigare steg.

Som man kan se så fungerar både algoritmerna olika bra under olika förehållanden och båda har sina styrkor och svagheter. Nu återstår det att se om en kombination av de två algoritmerna är bättre.

[2] Den exakta siffran är helt beroende på scenens utseende och i vilken ordning som polygonerna kommer till algoritmen.

Algoritmerna kombinerade

Ett av det största problemet med Painter's Algorithm men som även förekommer med Z-buffering är överritning. För att kunna undvika det kan Painter's Algorithm och Z-buffering kombineras till en algoritm. Problemet med överritning hos Painter's Algorithm uppstår för att den börjar med att måla upp scenen bakifrån och sedan målar över det som den har målat tidigare med polygoner som är längre fram i scenen. För att undvika problemet med att hela tiden måla över de polygoner som man redan har målat ut skulle man kunna göra om Painter's Algorithm så att den istället börjar framifrån. Om den skulle börja framifrån så skulle detta göra att den aldrig kommer att göra en övermålning för när man väl har färglagt en pixel, vet man att den är längre fram än alla andra. Problemet som uppstår är att man inte vet om en pixel har färglagts tidigare eller inte och det är där som z-bufferten kommer in. Med hjälp av z-bufferten kan man hålla reda på vilka pixlar som redan är färglagda och därmed inte ska ändras. Så först så sorterar man alla polygoner i scenen efter djup, efter det så skickar man alla polygonerna i ordning med den som är närmast skärmen först till en Z-bufferingalgoritm som då kommer att köra igenom alla polygoner utan att göra någon överritning.

Fördelar

Fördelen med att kombinera de två algoritmerna på detta sätt är att man helt kan undvik överritning och kan på så sätt tjäna tid på att slippa färgsätta pixlar som inte kommer att synas.

Nackdelar

Nackdelen med denna är att alla nackdelar som dessa algoritmer redan har fortfarande är kvar förutom överritning, plus att Z-buffering inte längre är online eftersom den nu måste vänta in alla polygoner för att göra en sortering innan den kan börja köra.

Litteraturlista

1. ANGEL E. 2009. *Interactive Computer graphics, a top down approach using OpenGL.* 5 uppl. Pearson Education. ISBN 978-0-321–54943-3
2. I. E. Sutherland, R. F. Sproull, R. A. Schumacker, A Characterization of Ten Hidden-Surface Algorithms, ACM Computing Surveys, March 1974
3. Microsoft: "Depth Buffers (Direct3D 9), Microsoft Developers Network 7 April 2010 http://msdn.microsoft.com/en-us/library/bb219616%28VS.85%29.aspx
4. Wayne Carlson: "CGI Historical Timeline" 2 September 2008 http://design.osu.edu/carlson/history/timeline.html#1960
5. Daniel Sevo: "History of Computer Graphics" 12 Augusti 2008 http://www.danielsevo.com/hocg/hocg_1960.htm
6. L. Szirmay-Kalos, T.Foris, Radiosity Algorithms Running in Sub-quadraticTime, Department of Process Control, Technical University of Budapest