Jämförelse av två algoritmer – Painters och Z-buffering

CHARLIE LINDVIKEN

KTH Datavetenskap
och kommunikation

Examensarbete
Stockholm, Sverige 2011

Jämförelse av två algoritmer – Painters och Z-buffering

C H A R L I E   L I N D V I K E N

Examensarbete i medieteknik om 15 högskolepoäng
vid Programmet för medieteknik
Kungliga Tekniska Högskolan år 2011
Handledare på CSC var Lars Kjelldahl
Examinator var Mads Dam

URL: www.csc.kth.se/utbildning/kandidatexjobb/datateknik/2011/
lindviken_charlie_K11058.pdf

Kungliga tekniska högskolan
Skolan för datavetenskap och kommunikation

KTH CSC
100 44 Stockholm

URL: www.kth.se/csc

Abstract

This report contains an analysis of two different algorithms that are used to solve the visibility problem. The visibility problem occurs when 3D-images are rendered onto the image plane. The algorithms analyzed are Painter’s algorithm and Z-buffering. These two help during the process of rendering with deciding the depth of a polygon or object in the picture. Both algorithms have their pros and cons which are discussed. Finally the report consists of an attempt of combining these two algorithms to see if there can be any improvement and if it is possible to implement this combination. There is a small improvement in the combination but many of the flaws are still there. An implementation is possible but it is not encouraged as the Z-buffering algorithm is already built-in into the hardware.

Abstrakt

Den här rapporten analyserar två algoritmer som löser problemet med skymda ytor inom 3D-grafik. Problemet med skymda ytor uppstår när 3D-bilder renderas till bildplanet. De algoritmer som denna rapport behandlar är Painter's och Z-buffering. Dessa algoritmer löser problemet på två olika sätt och de båda har sina för- och nackdelar, vilket presenteras och diskuteras i rapporten. Slutligen består rapporten av ett försök att kombinera de två algoritmerna för att undersöka ifall någon förbättring kan ske. Det visar sig möjligt att kombinera algoritmerna och det finns några förbättringar, men den nya algoritmen lider fortfarande av de gamla nackdelarna. Att implementera den nya algoritmen är möjligt men det är inget att rekommendera, eftersom Z-buffering finns redan dagens hårdvara.

Förord

Denna rapports alla delar har endast skrivits av Charlie Lindviken, då detta arbete har utförts själv utan en arbetspartner.

Innehållsförteckning

Abstract 2
Abstrakt 2
Förord 3
Introduktion 6
Bakgrund 6
Syfte 7
Problemformulering 7
Datorgrafik 8
Teoretisk algoritmanalys 10
Painter's algorithm 10
Förklaring 10
Fördelar 11
Nackdelar 11
Felaktiga värden vid jämförelse 12
Överlappande ytor 13
Användningsområden 13
Z-buffering 14
Förklaring 14
Fördelar 15
Nackdelar 16
Optimering 17
Användningsområden 17
Praktisk jämförelse av algoritmer 18
Painter's algorithm 18
Z-buffering 19
Kombination av algoritmer 20
Förklaring 20
Pseudokod 21
Implementation 22

Slutsats 25
Bilaga 26
C3Vector 26
Polygon 26
Initiering och Painter’s 27
Z-buffering 28
Litteratur 29

Introduktion

Bakgrund

Dagens datorgrafik är en vidareutveckling på vad ett fåtal pionjärer påbörjade i början av 1960-talet. Ivan Edward Sutherland anses vara en av de första som påbörjade arbete inom vår tids grafik. 1961 konstruerade Sutherland ett datorprogram som kunde rita enkel grafik med hjälp av en ljuspenna och en skärm som registrerade dess position.[5] Vid Massachusetts Institute of Technology påbörjade Sutherland sitt arbete och vid slutet av 60-talet hade hans arbete flyttat till University of Utah, därifrån gjordes många framsteg inom den tidiga datorgrafiken.[4]

Det största genombrottet som man gjorde var på University of Utah. Där utvecklade man algoritmerna för borttagning av skymda ytor.[4] Det är den grundläggande tekniken man måste använda sig av när man hanterar 3D-grafik. Problemet med skymda ytor uppstår när man projicerar 3D till 2D. Skärmar kräver att bilder visas i 2D medan världen man konstruerar grafiken är i 3D. Eftersom det finns ett djup i bilderna och objekt kan ligga bakom varandra vilket medför att vissa delar av bilden är skymda. Att man måste bestämma ordningen för objekten man ska rita ut är en självklarhet. Om föremål i förgrunden projiceras först och sedan de objekt som finns i bakgrunden resulterar det i att man bara såg objekten som finns i bakgrunden.

Det är precis det här problemet man behövde lösa innan 3D-grafiken kunde användas. I slutet av 60-talet uppkom ett par algoritmer som löser det här problemet. Algoritmerna gör så att man inte ritar över objekt som ligger närmare ”kameran” med objekt som ligger längre bort och i bästa fall inte alls ritar något som ligger bakom andra objekt. Detta för att spara processorkraft och minne.

Efter många år av utveckling ledde det till slut att nya algoritmer uppstod som löser samma problem. Turner Whitted utvecklade en metod som heter ray tracing.[4] Metoden löser problemet med överritning samtidigt som den färgsätter ytan utifrån dess egenskaper och ljussättning.

Utvecklingen har därifrån eskalerat och nu är realism något som strävas efter, då datorgrafik används i filmer, spel och många olika användningsområden.

Syfte

Syftet med denna uppsats är att jämföra två algoritmer för borttagning av skymda ytor. De ska förklaras noggrant och deras styrkor och svagheter ska presenteras. Ifall de har några användningsområden ska även de presenteras. När analysen är klar ska det undersökas ifall en kombination av de två algoritmerna tillsammans kan ge en mer effektiv algoritm.

Problemformulering

Det är viktigt att få en bra bild om hur algoritmerna fungerar och till en början diskutera hur de fungerar var för sig. Om det är möjligt att kombinera dessa två så att det blir en effektivare algoritm, är det vettigt att försöka implementera den nya algoritmen i ett högnivåspråk? Vad är det för fördelar man får och går det att optimera ännu mera?

Datorgrafik

I rapporten kommer endast 3D-grafik behandlas, även om problemen med skymda ytor återfinns i 2D-grafik. Med hjälp av 3D-grafik kan skapa nöjen så som spel eller film men även bidra med nyttig information. Ett användningsområde är till exempel 3D-renderingar av röntgenbilder. [2]

Med hjälp av en dator och ett grafikbibliotek kan man skapa 3D-objekt i en 3D-rymd. Man kan använda sig av flera olika typer av grafikbibliotek, ett sådant bibliotek är OpenGL. OpenGL är ett plattformsoberoende API som har fördefinierade funktioner för manipulation och skapande av allt från enkla till komplexa 3D-objekt. Med hjälp av ett sådant bibliotek slipper man själv göra de matematiska beräkningarna för att skapa vissa typer av objekt i 3D-miljön.

Att skapa objekt i 3D-rymden kan man göra på flera olika sätt. Enklare geometriska former kan man med enklare matematik beskriva och rita upp utan svårigheter. Kuber och cylindrar är exempel på enklare geometriska former. Typiskt för dem är att man inte behöver mycket information för att skapa dem. Utifrån det kan man sedan placera, rotera och skala dem med hjälp av transformationer.

När det kommer till mer avancerade objekt används polygoner för att bygga dess yta. All realtids 3D-grafik består av objekt som är uppbyggda av polygoner. En polygon är en plan yta som är sluten av kanter. En polygon har inga begränsningar på hur många kanter som den kan innehålla, så länge den sista kanten når den första och bildar en sluten krets. Eftersom kanterna omsluter hela ytan kan man sedan applicera färger på polygonen med hjälp av olika metoder.

När objekten är skapade skall de visas på en skärm. Man måste då projicera 3D-världen ned på ett 2D-plan. Ett sätt att göra det är genom perspektiv projicering, vilket innebär att man drar linjer från kanterna från samtliga synliga objekt mot den virtuella kamerans mitt.[8] Där linjerna skär en yta projiceras sedan bilden på det planet. Man kan tänka sig att ytan som linjerna skär är bildskärmen och kameran är betraktarens öga. Hela det här förfarandet kallas för rendering.

När man projicerat bilden sparas data i varje pixel som finns i skärmen. Skärmarna använder sig av RGB-färgkodning. Varje pixel innehåller information om vilken färg den ska visa sedan. RGB-kodning visar tre grundfärger: Röd, Grön och Blå. Sedan har varje färg ett värde mellan 0 och 255 för att det skall vara möjligt att blanda färger.[3] Att fylla resten av bilden med data, det vill säga färglägga bilden, kallas att rastera. [1]

Det är i renderingsteget som problem kan uppstå. När man har flera objekt i världen som ligger på olika djup gäller det att endast det objekt som är närmast kameran ritas upp, så att inget objekt bakom målas över. Det finns många algoritmer som löser det här problemet, men i den här rapporten behandlas endast Painter’s och Z-buffering.

Teoretisk algoritmanalys

Painter's algorithm

Förklaring

Painter's algorithm är en av de enklare algoritmerna som löser problemet med skymda ytor. Idén med den här algoritmen är att den använder sig av sortering av objekt i z-led.

Algoritmen är en djupsorterings algoritm och kan beskrivas med följande pseudokod:

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

(1)

Körtiden varierar helt på vilket sätt man väljer att sortera djupet på. Med en enklare sorterings algoritm kan körtiden bli $O(n^2)$, men om man väljer en effektivare algoritm kan körtiden för sorteringen gå ner till $O(n * \log(n))$ men uppritningen kommer stanna på $O(n)$[3].

Painter's algorithms namn kommer från tekniken som utnyttjas av målare, där man målar bakgrunden först och sedan målar man över delar med de främre objekten. När man målar över de gamla objekten löser man då problemet med skymda ytor. Visserligen behöver man måla redundanta delar av bilden eftersom det är möjligt att främre objekt skymmer.
Eftersom algoritmen jämför objekten i 3D-rymden så klassificeras den som en object space algoritm[1]. [3]

[1] Object space algoritmen jämför polygoner med varandra i ett visst utrymme av världen[3]

I det här följande fallet fungerar algoritmen utmärkt eftersom hela A ligger bakom B. Ifall $z_a \forall A_{x,y} < z_b \forall B_{x,y}$ är uppfyllt kommer A alltid målas bakom B. Däremot kommer de koordinater som båda delas av A och B att övermålas, vilket är en av nackdelarna med algoritmen.

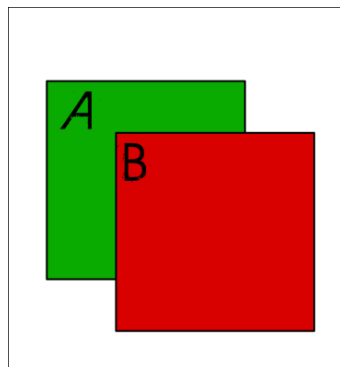

Fig. 1
Objekt B skymmer delvis objekt A. Eftersom objekt A ligger längre bort än vad objekt B gör så målas den över och sedan målas B.

Fördelar

Algoritmen är en enkel lösning som tyvärr medför ett par problem vid mer komplexa 3D-världar. En av de främsta fördelarna är att den är enkel att implementera och att förstå. Något som kan vara svårt inom datorgrafik är transparanta objekt. Det klarar den här algoritmen utan större problem då den ritar över de gamla objekten med det nya genomskinliga och de tidigare färgerna syns genom.

Nackdelar

Dess enkelhet är en bidragande faktor till varför den här algoritmen har märkbara nackdelar. Att sortera kan ta tid och det hindrar algoritmen att rita upp bilden innan alla objekt är sorterade, tillkommer det ett objekt måste sorteringen ske igen. Algoritmen tar en punkt i varje objekt som den ska sortera på. Väljer man inte rätt punkt kan bilden bli felrepresenterad.

I följande fall kommer Painter's ge felaktiga resultat.

Felaktiga värden vid jämförelse

I det här fallet finns det två stycken ytor A och B.

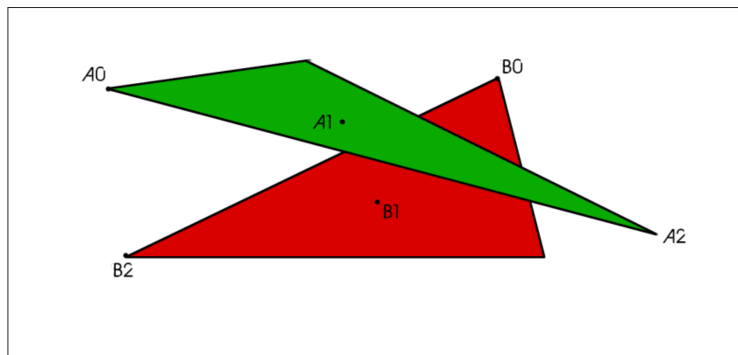

Fig. 2
Objekt A ligger över objekt B, likt fig. 4.

Dessa punkter A_0, A_1, A_2, B_0, B_1 och B_2 har följande koordinater i 3D-rymden:

$$A \begin{cases} A_0 = (1,5,-4) \\ A_1 = (3,4,-2) \\ A_2 = (5,1,0) \end{cases} \quad B \begin{cases} B_0 = (4,5,-1.5) \\ B_1 = (3.5,2.5,-1) \\ B_2 = (1,0.5,-0.5) \end{cases}$$

Enligt koordinaterna är stora delar av A längre bort än vad B är och således ska det objektet ritas upp bakom. Men någonstans mellan A_1och A_2 stiger z-värdena så pass mycket att i punkten A_2 är objektet närmast kameran. Ifall någon annan punkt innan vändningen i z-värdena sker kommer A ritas upp före B.

Bilden skulle inte bli återskapad som i fig. 2, detta skulle resultera i den bild likt fig. 3.

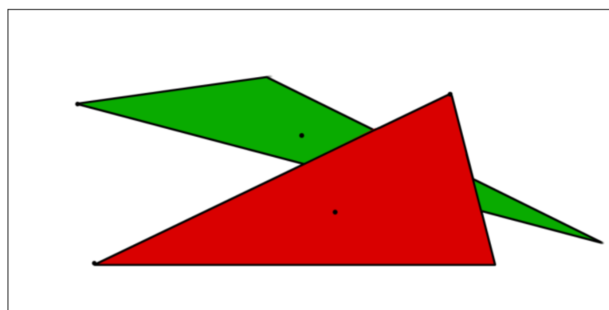

Fig. 3
Resultatet av ett felaktig rendering med hjälp av Painter's algorithm.

Överlappande ytor

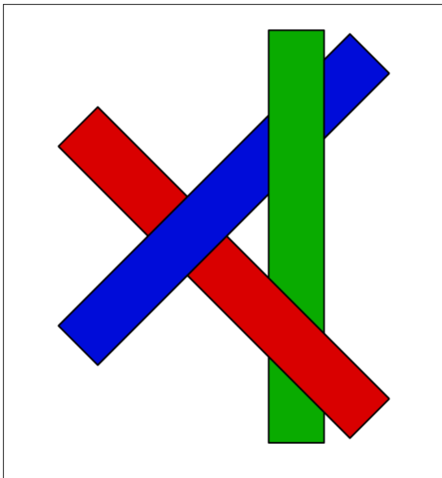

Fig. 4

Tre stycken polygoner som överlappar varandra.

För att lösa dessa problem behövs det att varje polygon bryts upp i mindre polygoner för att sedan kunna jämföras. Ett sätt att dela upp polygoner är med hjälp av Newells algoritm[3]. Utöver att algoritmen inte kan hantera komplexa bilder, utan att man är tvungen att ta hjälp av andra algoritmer, är Painter's algoritm inte beräkningseffektiv.

Inom datorgrafiken strävar man efter att göra bra grafik med så lite beräkningskraft som möjligt. Det här uppfyller inte algoritmen, eftersom man kan behöva måla över samma pixel n gånger, då n är antalet objekt i scenen.

Användningsområden

I ett tidigt skede användes det inom datorspel men när kraven på realtidsgrafik ökade kunde inte algoritmen hålla måttet. Att rendera flera bilder per sekund blir alldeles för jobbigt för algoritmen. Om man vill lösa tidigare nämnda problem med överlappande polygoner behöver man använda andra algoritmer. För att skapa ett BSP-träd tar det $O(nlog(n))$ lång tid. Detta kan behövas på många polygoner och vilket försämrar prestandan. [8]

Den används fortfarande inom ett par områden där grafiken är avsevärt enklare, mestadels i 2D. Postscript använder sig av Painter's för rendering av bilder.[7]

Z-buffering

Förklaring

Z-buffering är en vanlig teknik inom dagens grafikrendering.[8] Den utvecklades från Painter's algoritm för att vara mer effektiv. Det Z-buffering gör att den tar positionen för en polygon vid en pixel och jämför sedan dess djup med tidigare sparat värde för den positionen. Ifall avståndet är kortare skall det nya värdet sparas ned annars fortsätter man med nästa pixel i polygonen. För att bilden ska kunna ritas upp behövs det därför två stycken matriser som sparar olika data.

Den första matrisen sparar ned djupet, den kallas z-buffert eller depth-buffert. Färgen som skall sparas hamnar i framebufferten. Initialt sätts framebufferten till bakgrundsfärgen för bilden och z-bufferten får max djup.

```
//Initialize
for each position{
zBuffer[x,y]=(16-bit||24-bit||32-bit).MAX;
frameBuffer[x,y]=#BACKGROUND;
}
//Compute depth
for each polygon{
    for each pixel{
        if (pixel[x,y].depth < zBuffer[x,y]){
            zBuffer[x,y]=pixel[x,y];
            frameBuffer[x,y]=pixel[x,y].color;
        }
    }
}
```

(2)

Z-bufferten sparar ned värden i flyttalsformat. När man skapar z-bufferten bestämmer man även dess precision. Det vill säga hur många olika värden på djupet man kan urskilja i bilden. Med en 16-bitars z-buffert får man endast urskilja $2^{16} = 65{,}536$ olika värden. Om man vill ha en högupplöstbild med mycket djup finns det risk att bilden blir grynig då 16-bitar inte räcker till. 24-bitars och 32-bitars Z-buffertar är mer precisa än 16-bit med värden från 0 till 16,777,216 respektive 4,294,967,295.

Både Z-bufferten och framebufferten måste vara lika stora. Det vill säga att de måste ha samma (x,y)-värden. Om upplösningen för en bild är 1366x768 måste båda bufferterna ha 1366 x-värden samt 768 y-värden. Detta är för att de ska täcka hela bilden. [8]

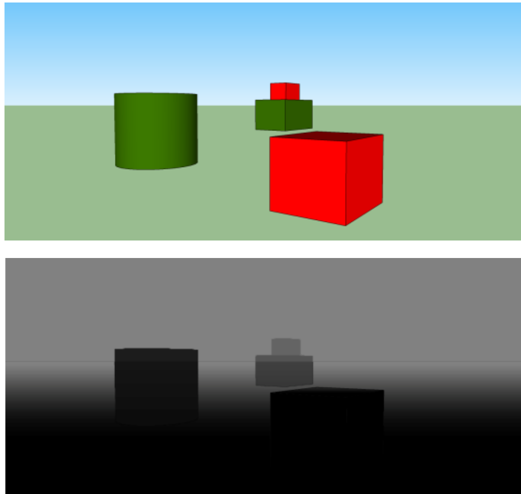

Fig. 5
Övre bild: Ett bild renderad med Z-buffering
Undre bild: Z-buffertens gråskale representation av samma bild

Fördelar

Algoritmen är enkel att implementera med hjälp av OpenGL eller DirectX[9][10]. Z-buffertar stöds av de flesta grafikkorten idag, men de som kan skilja är olika precision på djupet.
En annan fördel med Z-buffering är att det inte behöver vara en polygon man renderar. Egentligen kan det vara vad som helst, så länge algoritmen kan bestämma värden till frame-bufferten och z-bufferten.[2]

Nackdelar

Att spara ner alla de distinkta värdena som finns i 16-bits till 32-bits Z-bufferts kräver plats i minnet. Om man använder sig av upplösningen 1366x768 med 32-bits framebuffert behöver man närmare 4MBytes i minnet. Men som tidigare nämnts krävs det en matris till för att ta hand om djupet i bilden med. Om Z-bufferten skulle vara 24-bits med upplösningen 1366x768 pixlar skulle det behövas ytterligare 3MBytes.

Ett vanligt problem som uppstår vid användning av Z-buffering är att den lämnar artefakter när precisionen inte räcker till. Det här fenomenet kallas Z-fighting och de är vanligare när man har mindre precision på z-bufferten. När Z-fighting uppstår kan inte algoritmen avgöra vilken yta den ska rita utan det blir en slumpad färg. [3]

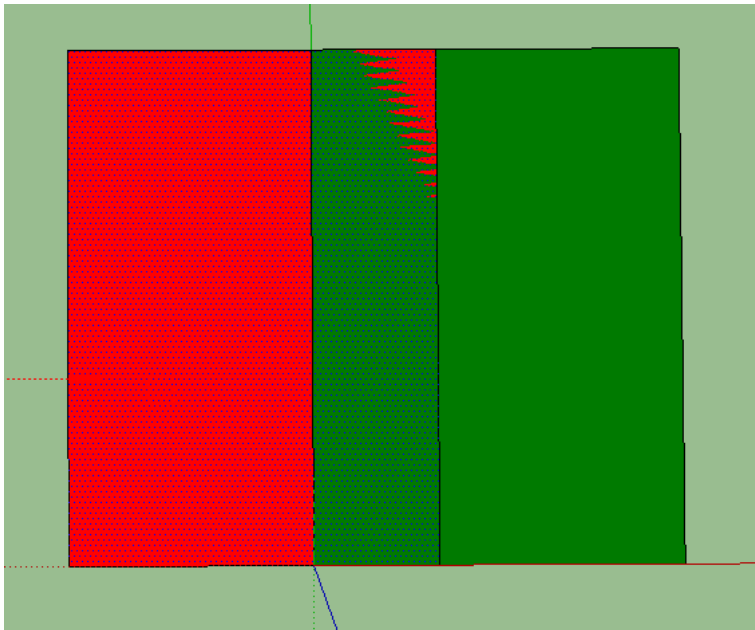

Fig. 6

Exempel på Z-fighting

Förutom att det kan uppstå fel vid renderingen av närliggande ytor är Z-buffering drabbat av samma problem som Painter's algoritm. Båda algoritmerna har problem med att de rita över samma pixel flera gånger.

Optimering

Det går att enkelt optimera Z-buffering med hjälp av andra algoritmer. Det innebär att man kan applicera andra algoritmer som drar nytta av Z-bufferten och kan hjälpa till att snabba upp körtiden. En av dessa algoritmer är Z-culling, där objekt som är helt täckta av redan existerande polygoner ignoreras. Detta betyder att många objekt inte behöver ritas upp. Den sorten av algoritmer heter occlusion culling algorithms.[3]

Användningsområden

Som tidigare nämnts stöds Z-buffering av både Direct3D och OpenGL, vilka är två stora grafikbibliotek. Vilket betyder att de flesta spel som kommer ut idag använder sig av Z-buffering.

Algoritmen behöver inte endast användas för att avgöra djupet i en bild utan kan även användas för att skapa skuggor i bilden. Med hjälp av Z-buffering kan man snabbt få fram skuggor för solida objekt. Däremot är transparanta objekt svårare att hantera med Z-buffering. Att bygga upp komplexa geometriska former med hjälp av enklare former är en teknik som används ofta. Även där används Z-buffering för att ta hjälp att konstruera formen. [12]

Praktisk jämförelse av algoritmer

Följande scenarion har satts upp för att jämföra hur de båda algoritmerna fungerar praktiskt:

1. Lågupplöst bild med fåtal polygoner
2. Högupplöst bild med fåtal polygoner
3. Lågupplöst bild med många polygoner
4. Högupplöst bild med många polygoner

De lågupplösta bilderna skall representera en monitor som har upplösningen 640x480pixlar vilket motsvarar VGA kvalitet. High-Defintion[2] kvalitet används för de högupplösta bilderna vilket motsvarar 1920 x1080pixlar.

Fåtal polygoner menas att det är 50 polygoner i bilden och många polygoner menas med 10,000 polygoner.

Dessa fyra fall valdes eftersom det är just de som kan representera specifika fall, som är möjliga i verkligheten. De båda algoritmerna har olika områden vilket ger sämre prestanda. Där ena blir långsammare då antalet objekt är högt medan den andra blir sämre då upplösningen är högre. De här fallen återspeglar de kombinationer som är möjliga.

Painter's algorithm

1. I det här fallet körs algoritmen snabbt. Detta är för att den endast behöver sortera en liten mängd polygoner. Överritning är inget större problem även om det skulle ske.

2. Sorteringen kommer att ske på samma tid eftersom antalet polygoner inte har förändrats. Uppritningen av bilden kommer ta längre tid, eftersom antalet pixlar har ökat med 675 %.

[2]High-Defenition –En benämning på en viss upplösning.

3. Algoritmen kommer att gå relativt långsamt fastän upplösningen är låg, eftersom antalet polygoner kommer att vara stort. Problem kan uppstå vid sortering beroende på vilken algoritm man använder. Efter sorteringen kommer algoritmen bli mindre beräkningseffektiv eftersom överritningar kommer att ske oftare eftersom den inte håller koll på om något är uppritat sedan tidigare.

4. Likt skillnaden mellan de två första exemplen lider algoritmen av problem med antalet uppritningar då antalet pixlar är avsevärt mycket större. Painter's algoritm blir lidande då antalet polygoner blir större då antalet överritningar och sorteringar tar tid. Det här fallet är ett ypperligt exempel på hur en modern film kan vara renderad.

Z-buffering

1. Det här fallet är i princip identiskt med Painter's algoritm. Det som kan skiljas är mängden minne som krävs för att köra. Med en 24-bits framebuffert och z-buffert krävs det totalt 1.6MBytes[3] minne.

2. Det som ändras med högre upplösning är minnet eftersom matriserna måste bli större för att kunna representera bilderna.

3. Likt tidigare algoritm i det här fallet lider Z-buffering av överritning. Men det beror helt på vilken ordning man väljer polygonerna i bilden. I värsta fall kan den bli lika långsam som Painter's ifall man ritar ut alla bakomliggande polygoner. Detta går att undvika ifall man använder sig av Z-culling tillsammans med Z-buffering.

4. Det här fallet påminner mycket om Painter's fall. Överritning är en självklarhet eftersom det är många polygoner men även uppritningen kräver längre tid då pixlarna är större. Förutom de tidigare problemen kräver Z-buffering avsevärt mer minne jämför med en lågupplöst bild. Med samma precision som i fall 1 har minneskravet stigit till närmare 12Mbytes. Likadant som i fall 3 kan detta optimeras med hjälp av borttagning av polygoner.

[3] Detta är beräknat med formeln: (höjden * bredden * antalet bitar)

Kombination av algoritmer

Förklaring

Något de båda algoritmerna lider av är överritning av tidigare polygoner. Detta går att lösa genom en kombination av dessa två algoritmer.

För att kombinera dessa behövs det att en ändring sker i Painter's eftersom överritning sker då man sorterar på djupet och ritar upp de objekt längst bak först. Om man vänder på algoritmen och ritar de främsta objekten först med vissa villkor skulle det lösa problemen. Detta fungerar endast ifall villkoret är att en pixel aldrig får målas över efter den målats en gång.

Att kombinera detta med Z-buffering innebär att Painter's algoritm har koll på vilka pixlar som är redan ritade eftersom djupvärdet hamnar i Z-bufferten. Det betyder att de objekten som är skymda inte ritas ut.

Det som drar ned körtiden är sorteringen. Eftersom Z-buffering inte kan köras förens efter sorteringen av Painter's är klar. Detta medför att den nya algoritmen inte är lika snabb som Z-buffering självt.

Pseudokod

Detta är ett exempel på hur den nya algoritmen kan konstrueras.

```
//Initialize
double precision = 16-bit||24-bit||32-bit;
for each position{
    zBuffer[x][y]= precision;
    frameBuffer[x][y]= Color background;
}
world_is_sorted = false;
List world<objects>;

void paintBuffer(){
    if(world_is_sorted){
        zbuffer(world);
    }else{
        objects.sort(z-values);
        world_is_sorted = true;
        paintBuffer();
    }
}

void zbuffer(Graphics g){
    for each object in world{
        for each polygon in object{
            interpolate through y-values{
               for each line interpolate x-values{
                if(polygon.getDepth(x,y) < zBuffer[x][y]){
                    zBuffer[x][y] = polygon.getDepth(x,y);
                    frameBuffer[x][y] = polygon.color;
                }
            }
        }
    }
}
```

Implementation

Det som skulle skilja en egen implementation från den riktiga är att z-bufferten skulle hamna i mjukvaran istället för hårdvaran. Dagens Z-buffertar är inbäddade i hårdvaran[12] och optimerade till den grad att de fungerar väldigt effektivt. En mjukvarubuffert kommer troligtvis i alla avseenden bli långsammare än de som ligger på grafikkorten. Det språket som valdes att använda var java. Java valdes för att se ifall det var möjligt att få en implementation av algoritmerna, samt att kunskapen fanns inom språket redan. För uppritningen användes java Applet då detta hanterar enklare grafik enkelt.

Beroende vilket språk man använder påverkar den slutgiltiga körtiden, då java kan vara långsammare än andra programmeringsspråk som är närmare hårdvaran.

Algoritmerna i sig är lätta att förstå, men det uppstår ett visst antal problem som måste kringgås eftersom renderingen skall flyttas till mjukvaran. De problemen tog tid att lösa så att algoritmen kunde köras på ett effektivt sätt.

Först och främst behövdes vad som skulle renderas bestämmas och vilken datastruktur det hela skulle ha. Eftersom inget färdigt grafikbibliotek användes i denna implementation krävdes det två stycken nya datastrukturer. Eftersom de nya strukturerarna ska innehålla data för 3D-rymden behöver det atomära elementet inne hålla x,y och z koordinater. Efter att en punkt skapats måste en samling av punkter skapas för att bilda en polygon.

En polygon kan, som tidigare nämnts, ha oändligt många kanter men i detta fall består alla endast av 3st kanter för enkelhetens skull. Denna struktur behöver implementera javas interface *Comparable*, vilket tillåter ändring av den naturliga sorteringsordningen av objekt. Eftersom varje polygon har tre vektorer med tre olika värden på djupet (z-koordinaten) behövs detta interface implementeras. När man kallar på sorteringsalgoritmen kommer den senare jämföra med alla z-värden. Polygonstrukturen har informationen om vilken färg den har. Detta medför att alla punkter i polygonen delar samma färg. Det går att lägga färgkodningen i koordinaterna vilket skulle medföra att en polygon kan skifta färg vid renderingen.

När polygonerna är konstruerade lades dessa i en av javas egna datastrukurer, denna struktur är ArrayList. Denna valdes för att det går att enkelt sortera med javas inbyggda sorteringsalgoritm. Att använda den inbyggda algoritmen fungerar endast ifall man, som tidigare nämnts, implementerat *Comparable* på polygonerna.

Alltså var följande tre strukturer ett krav att konstruera innan renderingsalgoritmerna kunde påbörjas:

- *C3Vector* – en vektorstruktur som sparar ner koordinater för en punkt.
    - Innehåller tre stycken flyttalsvärden som representerar koordinater i 3D-rymden.
- *Polygon* – en samling av vektorer som bildar en yta vid interpolation
    - Strukturen behöver ärva metoder från *Comparable*[4] för att ha möjlighet att sortera i djup.
- *Objektcontainer* – En lista med samling av polygoner som kan illustreras som objekt.
    - I och med att Polygoner kan sorteras kan även dessa objekt sorteras efter den närmaste polygonens z-värde.

När strukturen är klar måste z-bufferten och framebufferten skapas, vilka motsvarar vektorer av vanliga integers. Under initieringen av programmet, då objekten skapats och läggs till i listan startas den första algoritmen vilket är Painter's.

Den algoritm som valdes för djupsortering var javas inbyggda sorterings algoritm vilket har den garanterade körtiden $O(n * \log(n))$[11].

Den algoritm som tog mest tid att implementera var Z-buffering algoritmen. Eftersom renderingen utförs i den här delen av programmet. Eftersom inget grafikbibliotek användes behövdes vissa metoder kodas från grunden igen. Eftersom 3D-rymden måste ner på ett 2D-plan måste alltså en metod för projicering först skapas. För enkelhetens skulle användas ortonogal projicering. Att ta hänsyn till vinklar skulle troligtvis påverka körtiden negativt, då fler beräkningar skulle behöva utföras.

Z-buffering metoden behöver rita upp polygonerna och för detta behövs en metod som interpolerar mellan de punkterna som bygger upp polygonen. Detta kan göras med linjärinterpolation i 2 dimensioner, eftersom projiceringen sker tidigare, med följande algoritm:

$$y = y_0 + (x - x_0)\frac{(y_1 - y_0)}{(x_1 - x_0)}$$

När linjerna mellan punkterna har interpolerats används samma algoritm längs y-koordinaterna mellan de två linjerna som ligger på samma nivå. Vid varje steg under interpoleringen kontrolleras z-koordinaterna med de tidigare värdena i z-bufferten. När detta är klart är polygonen uppritad.

Så följande tre metoder ska finnas med i Z-buffering metoden:

- Projicering
    - Få ner koordinaterna ned på 2D-planet.
- Interpolering
    - Interpolera fram alla koordinater mellan vektorerna sparade i polygonen.
- Jämförelse z-koordinater
    - Att vid varje pixel inom polygonen jämföra dess z-värde med Z-buffertens z-koordinat vid samma koordinat ifall information skulle sparas ned.

Se bilaga kod för varje metod som beskrivits ovan.

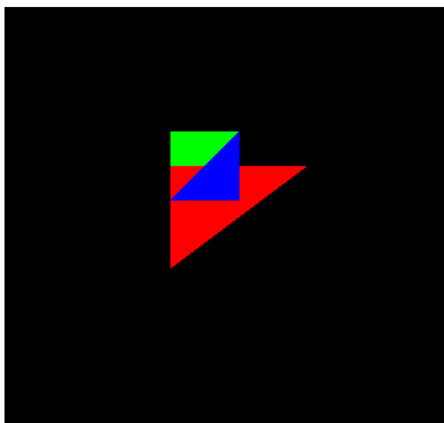

Fig. 7
Bild renderad av kombinationen av Painter's och Z-buffering bestående av 3 polygoner.

Slutsats

De båda algoritmerna lider av sina nackdelar. De båda verkar vara likvärdiga då antalet polygoner är få och upplösningen är låg. När någon av dessa ökar blir den ena eller den andra algoritmen lidande. Ifall antalet pixlar ökas kommer Z-buffering bli sämre, eftersom ju fler pixlar det är desto mer utrymme kräver algoritmen i minnet. Visserligen blir båda algoritmerna långsammare på grund av det nya antalet pixlar som måste färgas, men det är något som är gemensamt för de båda. Om antalet polygoner däremot skulle höjas skulle Painter's bli mer lidande. När det blir fler polygoner att rita blir det fler beräkningar för algoritmen att göra. Även om sorteringsalgoritmen är effektiv kommer det dröja innan sorteringen är klar.

Att sammanfoga algoritmerna för att få en litet effektivare lösning på problemet var en god tanke. Vid närmare granskning är den kombinerade algoritmen en sämre version av Z-culling. Eftersom kombinationen med Painter's måste vänta in tills att alla polygoner är sorterade innan den kan börja beräkna djupet.

Att sedan implementera kombinationen är däremot mycket svårare än förväntat. Problem som är redan lösta i tidigare grafikbibliotek måste återskapas och på så sätt blir det långsammare än vad som krävs för real-tids grafik. Tiden det tog att implementera allt var uppskattningsvis lite mer än en arbetsvecka, det vill säga lite över 40 effektiva arbetstimmar. Detta bestod bara för kodningen och inte förarbetet med datastrukturerna. Det svåraste att implementera var Z-buffering algoritmen då förståelsen över hur interpoleringen skulle ske tog längst tid att lösa. Även om det tog tid att lösa var det lärorikt att få möjlighet att förstå sig på hur en mjukvarurenderare fungerar i praktiken.

Efter implementationen och analysen av den kombinerade algoritmen visade det sig vara möjligt att utan hjälp av Painter's att optimera Z-buffering så att den blir effektivare. Detta måste göras utan Painter's eftersom sorteringen hindrar real-tids rendering. Optimeringen kan göras med hjälp av occlusion culling algoritmer så som back-face culling, view culling och Z-culling.

Bilaga

C3Vector

```
private class C3Vector {
		double x,y,z;

		private C3Vector(){
			x = 0.0;
			y = 0.0;
			z = 0.0;
		}
		private C3Vector(double x, double y, double z){
			this.x = x;
			this.y = y;
			this.z = z;
		}
	}
```

Polygon

```
	private class Polygon  implements Comparable{
		public C3Vector c1,c2,c3;
		Color c;
		public Polygon(C3Vector c1, C3Vector c2, C3Vector c3, Color
c) {
			this.c1 = c1;
			this.c2 = c2;
			this.c3 = c3;
			this.c = c;
		}

		@Override
		public int compareTo(Object o) {
			Polygon tmp = (Polygon)o;
			C3Vector temp;
			if(c1.z > c2.z){
				if(c1.z>c3.z){
					temp = c1;
				}else{
					temp = c3;
				}
			}else{
				temp = c2;
			}
			if(getZ(tmp).z > temp.z){
				return 1;
			}else if (getZ(tmp).z == temp.z) {
				return 0;
			}else{
				return -1;
			}
		}
	}
```

```
public C3Vector getZ(Polygon p){
                if(p.c1.z > p.c2.z){
                        if (p.c1.z > p.c3.z) {
                                return p.c1;
                        }
                        return p.c2;
                }else{
                        return p.c3;
                }
        }
```

Initiering och Painter’s

```
        public void init() {
                framebuffer = new double[W*H];
                zbuffer = new double[W*H];
                clearZbuffer();
                List p = new ArrayList();
                for (int i = 0; i < polygons.length; i++) {
                        p.add(polygons[i]);
                }
                Collections.sort(p, ORDER);
                Collections.reverse(p);
        }
```

Z-buffering

```
public void zbuff(Polygon funk) {
            int vertices = 3;
            int x,y;
            int ymin = H;
            int ymax = 0;
            double pLeft[] = new double[6];
            double pRight[] = new double[6];

            //Check all vertices in the polygon to find where yMax and yMin is
            projectVertex(funk.c1,p[0]);
            ymin = Math.min(ymin, (int)p[0][1]);
            ymax = Math.max(ymax, (int)p[0][1]);
            projectVertex(funk.c2,p[1]);
            ymin = Math.min(ymin, (int)p[1][1]);
            ymax = Math.max(ymax, (int)p[1][1]);
            projectVertex(funk.c3,p[2]);
            ymin = Math.min(ymin, (int)p[2][1]);
            ymax = Math.max(ymax, (int)p[2][1]);


            //When found, loop through only the height of the polygon.
            for (y = ymin ; y <= ymax ; y++) {

                        pLeft[0] = W;
                        pRight[0] = 0;

                        //Linear interpolation y= y0 + (x-x0)((y1-y0)/(x1-x0))...
//wikipedia
                        for (int i = 0 ; i < vertices ; i++) {
                                    int j = (i+1) % vertices;
                                    if (p[i][1] < y != p[j][1] < y) {
                                                x = (int)(p[i][0] + (p[j][0]-
p[i][0])*((y-p[i][1])/(p[j][1]-p[i][1])));
                                                if (x < pLeft[0])
                                                            for (int k = 0 ; k < 6
; k++)
                                                                        pLeft[k] =
(int)(p[i][k] + (p[j][k]-p[i][k])*((y-p[i][1])/(p[j][1]-p[i][1])));
                                                if (x >= pRight[0])
                                                            for (int k = 0 ; k < 6
; k++)
                                                                        pRight[k] =
(int)(p[i][k] + (p[j][k]-p[i][k])*((y-p[i][1])/(p[j][1]-p[i][1])));
                                    }
                                    }

                        // Loop all X-values in current y-value

                        for (x = (int)pLeft[0] ; x < (int)pRight[0] ; x++) {
                                    int currentPixel = y * W + x;
                                    //Get z-value at [x,y]
                                    int z = (int)(pLeft[2] + (pRight[2]-
pLeft[2])*((x-pLeft[0])/(pRight[0]-pLeft[0])));
                                    //If z>zbuffer[x,y]
                                    if (z > zbuffer[currentPixel]) {
                                                zbuffer[currentPixel] = z;
                                                //set color
                                                framebuffer[currentPixel] =
(double)funk.c.getRGB();
                                    }
                        }
            }
}
```

Litteratur

1. Peter Shirley et al. (2005). *Fundamentals of computer graphics*

2. ParaView – Open Source Scientific Visualization
   http://www.paraview.org/

3. James D. Foley, Andries Van Dam, Steven K. Feiner and John F. Hughes (1995), *Computer graphics: principles and practice (2nd ed.)*

4. E. Sutherland, R. F. Sproull, R. A. Schumacker*, A Characterization of Ten Hidden-Surface Algorithms, ACM Computing Surveys*, March 1974

5. http://www.cs.um.edu.mt/~sspi3/HistoryOfGraphics.pdf
   (2011-02-18)

6. Florida Tech: A Short History of Computer Graphics
   http://cs.fit.edu/~wds/classes/graphics/History/history/history.html
   (2011-02-18)

7. Painter's & Z-Buffers and Polygon Rendering
   http://www.cs.cmu.edu/afs/cs/project/anim/ph/463.96/pub/www/notes/zbuf.2.pdf
   (2011-03-30)

8. Mark de Berg, Otfried Cheong, Marc van Kreveld, 2008
   Computational Geometry: Algorithms and Applications, Second Edition

9. OpenGL.org – The Depth Buffer
   http://www.opengl.org/resources/faq/technical/depthbuffer.htm
   (2011-04-04)

10. Microsoft Direct3D 9- Depth Buffers
    http://msdn.microsoft.com/en-us/library/bb219616%28v=vs.85%29.aspx
    (2011-04-04)

11. Java 2 Platform SE v1.4.2 API
    http://download.oracle.com/javase/1.4.2/docs/api/java/util/Collections.html#sort%28java.util.List,%20java.util.Comparator%29
    (2011-04-04)

12. Theoharis Theoharis, Georgios Papaioannou, Evaggelia-Aggeliki Karabassi,
    The magic of the Z-buffer: A survey
    http://graphics.cs.aueb.gr/graphics/docs/papers/z.pdf
    (2011-04-04)