Voxelbaserad rendering med ”Marching Cubes”-algoritmen

Patrik Andersson – Kandidatarbete – 2009

Avdelning för Interaktion och Systemdesign
Blekinge Tekniska Högskola
Kandidatarbete i Datavetenskap, DV1303

Kontaktinformation

Författare: Patrik Andersson
Adress: Trestegsvägen 33
691 54 Karlskoga, Sverige
Email: paan06@student.bth.se

Handledare:
Olle Lindeberg

Abstrakt

Det finns flera olika metoder och tekniker för tredimensionell rendering, alla med olika för- och nackdelar som lämpar sig för olika applikationer. Voxelbaserad rendering har använts flitigt inom vetenskapliga områden, främst inom det medicinska för visualisering av volymetrisk data. Tekniken används nu inom flera olika områden för tredimensionell rendering, t.ex. i datorspel, i matematiska applikationer och vid geologisk rekonstruktion.

I den här rapporten kommer voxelbaserad rendering med Marching Cubes-algoritmen undersökas för att se hur den lämpar sig för realtidsapplikationer. Området behandlas dels teoretiskt, men även praktiskt då en implementering av Marching Cubes gjordes för att genomföra några tester för att se hur prestandan påverkades.

Av testerna framkom det tydligt att algoritmen lämpar sig väl för realtidsapplikationer och dagens grafikkort. Viss optimering krävs dock för att kunna utnyttjas på bästa sätt.

Nyckelord: Rendering, Voxel, Marching Cubes, Volymetrisk Data

Abstract

There are lots of different methods and techniques for three-dimensional rendering, everyone with different advantages and disadvantages that suits different applications. Voxel-based rendering has been used frequently within the scientific area, mainly within medicin for visualization of volumetric data. The technique is now used within many different areas for three-dimensional rendering, e.g. in computer games, in mathematical applications and in geological reconstruction.

In this report voxel-based rendering with Marching Cubes algorithm will be researched to see how it suits for real-time applications. The area will partly be dealt with theoretically, but also practically as an implementation of Marching Cubes was done to run some tests to see how the performance was affected.

From the tests it appeared cleary that the algorithm is well suited for real-time applications and today's graphics card. Though some optimization is needed to fully take advantage of it.

Keyword: Rendering, Voxel, Marching Cubes, Volymetric Data

Innehållsförteckning

1 Introduktion

1.1 Inledning:

Voxelbaserad rendering är ingen ny teknik vad gäller tredimensionell rendering. Det har funnits sedan tidigt åttiotal, då främst inom det vetenskapliga området. En voxel, som är en ihopsättning av volymelement, är en tredimensionell motsvarighet till en pixel. Voxlar beskrivs alltså av en längd, djup och bredd, d.v.s. en volym, till skillnad från en pixel som beskrivs av en längd och bredd, d.v.s. en area. Med hjälp av flera ihopkopplade voxlar och speciella algoritmer kan man skapa väldigt komplexa formationer av geometri i realtidsapplikationer. Detta kan ha stora fördelar i t.ex datorspel och vid visualisering av medicinsk data (magnetröntgen etc.).

I slutet av nittiotalet var det ett flertal kommersiella datorspelsmotorer som använde voxelbaserad rendering istället för ”konventionellt” tillvägagångssätt. Det verkade dock aldrig som att denna teknik slog igenom. Nu på senare tid ser det däremot ut att till viss del ha kommit tillbaka i datorspel. Spelföretaget Crytek t.ex. använder sig av voxlar i sin ”level editor” (Sandbox 2) för att kunna skapa tunnlar, överhäng etc. i terrängen. Miner Wars är ett exempel på ett annat spel som använder voxelbaserad terräng för att på så sätt få den fullt förstörbar. Det största problemet, eller snarare svårigheten med voxlar, är att det kan bli väldigt minnes- och processorkrävande. Olika tekniker finns tillgängliga som optimerar eller förhindrar detta.

Den algoritmen som kommer tas upp och undersökas i rapporten heter Marching Cubes, och är kortfattat en tredimensionell ytkonstruktionsalgoritm. Av volymetrisk data eller matematiska formler konstruerar algoritmen en tredimensionell kontur (en ”polygon mesh”).

1.2 Hypotes och huvudfråga:

Rapporten kommer fokusera på att undersöka hur voxelbaserad rendering med Marching cubes fungerar i realtidsspel och andra applikationer, samt hur framtiden ser ut för denna teknik.

Hur fungerar voxlelbaserad rendering med Marching cubes i realtidsapplikationer?

1.3 Frågeställning:

Vad menas med voxelbaserad rendering?
Hur fungerar Marching Cubes?
Vilken typ av applikationer kan dra nytta av denna renderingsmetod?
Vilka för- och nackdelar finns det i nuläget för denna algoritm?
Hur ser framtiden ut för voxelbaserad rendering med Marching Cubes?

1.4 Syfte:

Jag tror användningen av voxelbaserad rendering kommer bli allt vanligare i framtiden. Därför vill jag fördjupa mig inom just detta område. Jag hoppas även kunna hjälpa andra att få en bättre förståelse för voxelbaserad rendering med Marching Cubes.

1.5 Avgränsning:

Området kommer dels att beskrivas teoretiskt, men även praktiskt. För att kunna genomföra några tester, implementerades Marching Cubes-algoritmen. Rapporten kommer enbart att behandla hur man kan visualisera tredimensionella objekt genom att rendera dem som polygoner med hjälp av Marching Cubes och med ”DirectX 10.0” som grafiskt API (Application Programming Interface).

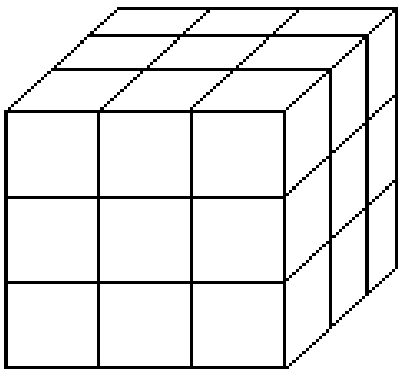
1.6 Målgrupp

Rapporten kommer rikta sig till personer med förkunskaper inom tredimensionell programmering med DirectX API och HLSL[1], där kännedom om spelprogrammering är en fördel. I rapporten kommer alltså inte detaljer gås igenom för att förklara ”grundläggande” saker vad gäller tredimensionell programmering.

2 Teoretisk bakgrund

2.1 Vad menas med voxelbaserad rendering?

Voxelbaserad rendering, eller volymrendering som man också kan kalla det, är en teknik för att visualisera tredimensionell data (volymetrisk data) som en tvådimensionell bild. Ett annat sätt att visualisera volymetrisk data, är genom att bygga upp den med hjälp av enklare primitiver (polygoner) [10]. Till en början, och även idag, användes volymrendering framför allt för vetenskaplig visualisering, t.ex. inom medicin [9]. Tekniken bakom volymrendering bygger på att en bestämd volym delas in i en serie mindre, jämnstora volymer (voxlar), så ett tredimensionellt rutnät bildas (figur 1) [6, 10]. Vad som menas med begreppet voxel, är helt enkelt ett volymelement. Precis som en pixel definierar en area, definierar en voxel en volym. Olika algoritmer går sedan igenom alla voxlarna för att antingen ersätta dem med lämplig polygon, eller färgsätta en pixel baserat på datan i voxlarna.



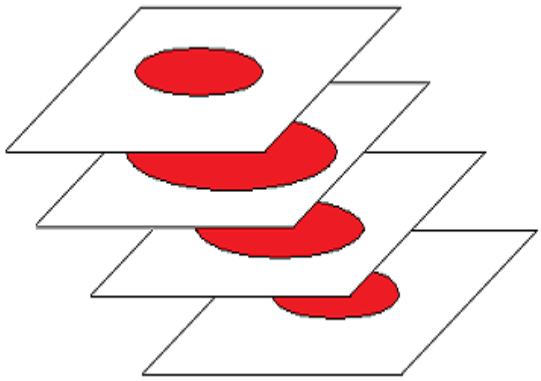
Figur 1: En definierad volym indelad i voxlar.

Volymrendering är indelad i två paradigmer för att representera volymetrisk data. Den ena är ”Direct Volume Rendering” (DVR) och den andra är ”Indirect Volume Rendering” (IVR) [12]. En vanlig algoritm i DVR är ”Volume ray casting”. På senare tid har det blivit möjligt för privatpersoner att använda sig av Volume ray casting för realtidsapplikationer, då det går att

1 High Level Shader Language

implementera algoritmen på moderna grafikkort, med bl.a. fördelen av parallell bearbetning. Tidigare var algoritmen allt för krävande för att kunna användas i realtidsapplikationer, utan att dra nytta av speciellutvecklad hårdvara. Volume ray casting är en bildbaserad algoritm [13] som ger väldigt bra bildkvalité eftersom ingen geometri (polygoner) är inblandad, något som förhindrar t.ex. ”aliasing”. Kortfattat fungerar algoritmen så att en stråle skjuts ut från en synvinkel (d.v.s. en kamera i tre dimensioner) genom ett vyplan, och vidare genom en volym innehållande volymdata. Stickprovs tas med jämna mellanrum då strålen färdas genom volymen. När strålen lämnar volymen vägs alla stickprov ihop för att bilda rätt färg för pixeln. Detta görs för varje pixel i vyplanet, vilket kan bli väldigt dyrt för realtidsapplikationer med hög upplösning [12, 13]. Marching Cubes är ett exempel på IVR [12], den algoritm som kommer att behandlas i rapporten. Förenklat går Marching Cubes igenom alla voxlarna och ersätter dem, beroende på volymdatan, med lämplig yta (polygon). När alla voxlarna har blivit besökta, representeras till slut en tredimensionell geometri av volymdatan.

Volymdatan kan t.ex. beskrivas genom flera lager av tvådimensionella bilder som representerar en tredimensionell geometri. Upplösningen på bilderna måste vara lika hög och avståndet mellan dem lika stort. De måste dessutom vara sekventiellt sparade för att kunna användas (figur 2) [11]. Datan kan även beskrivas med hjälp av matematiska formler för att skapa enklare geometrier så som sfärer, cylindrar och kuber [2].



Figur2: Ett antal 2D-bilder som tillsammans kan beskriva ett tredimensionellt objekt.

Volymrendering används främst i applikationer där hög detaljrikedom är viktigt, eller där komplex geometri ska renderas, t.ex. vid visualisering av medicinsk data och geometrier i datorspel. Tekniken användes ganska flitigt inom datorspel i slutet av nittiotalet, framför allt för att rendera terränger då det blev mer ”realistiskt” än konventionell rendering. Senare upphörde den dock mer eller mindre helt att användas. En stor anledning till det var att tekniken var så pass processor- och minneskrävande, att dåtidens hårdvara inte riktigt hade den prestanda som krävdes. På senare tid har den dock kommit tillbaka mer och mer inom kommersiella spel, eftersom hårdvaran inte längre är någon begränsande faktor. Tekniken har dessutom optimerats och utvecklats mer för realtidsapplikationer.

2.2 Hur uppkom Marching Cubes och vad är det?

År 1984 jobbade elektroingenjören Carl Crawford för ”GE's Medical Systems Business Group”, ett företag som ingår i GE[2]-koncernen. Under sin tid där utvecklade han bl.a. algoritmer för datoriserad bildutrustning, som t.ex. datortomografi och magnetröntgen. Han jobbade även med att utveckla en

2 General Electric

ny ”GE-produkt”, som kallades ”Graphicon”. Det skulle bli en ny renderingsmotor för volymetrisk datavisualisering inom medicin. I ett seminarium som Crawford höll, beskrev han sin vision om en polygonbaserad renderingsmotor GE skulle ta fram (d.v.s. Graphicon), samt dess möjligheter. Den dåvarande renderingsmotorn som GE hade, använde sig av ”Cuberilles”, en teknik som hade en stor nackdel [15]. Ytorna som algoritmen genererade blev nämligen ojämna, vilket påverkade detaljrikedomen i bilderna [4]. Crawford ”utmanade” alla som deltog i seminariet att komma på ett sätt att ersätta den dåvarande Cuberille-teknologin, mot en polygonbaserad teknologi. William E. Lorensen och Harvey E. Cline deltog båda i seminariet, och blev väldigt intresserade av utmaningen då de ville jobba med medicinska applikationer. Genom att anta utmaningen hoppades de få in en fot i ”GE Medical Systems”. Vid den tiden jobbade Lorensen och Cline för ”GE's Corporate Research and Development Center”. De båda hade erfarenhet av att rekonstruera tredimensionella ytor, och det var precis vad den kommande Graphicon skulle göra [15]. Den nya renderingsmotorn skulle dock inte använda sig av ”interferograms”, vilket Lorensen och Cline hade använt sig av, utan av volymetrisk data för att generera en yta.

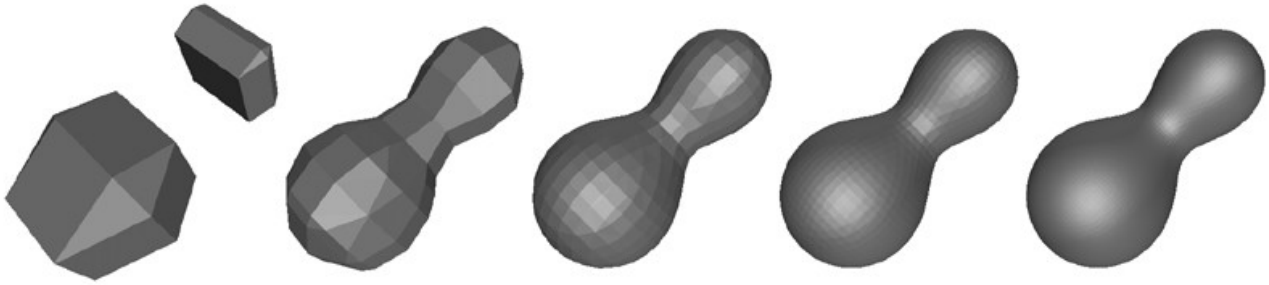
År 1985 ansökte Lorensen och Cline om mjukvarupatent på sin algoritm [8], trots att det de kom fram till var en relativt uppenbar lösning till ytgenereringsproblemet som föregående algoritmer haft problem med. En liknande algoritm, som kallades ”Marching Tetrahedrons”, utvecklades av andra i samband med det för att på så sätt kringgå patentskyddet [11]. Numera har patentet upphört eftersom det gått mer än 20 år sedan patentansökan godkändes, vilket innebär att vem som helst kan använda sig av deras algoritm. Marching Cubes, som blev namnet på algoritmen Lorensen och Cline utvecklade, publicerades 1987 [6] på en konferens som kallas SIGGRAPH (Special Interest Group on GRAPHics and Interactive Techniques).

Marching Cubes-algoritmen är en teknik som används för att representera en isoyta (”isosurface”) som en polygon mesh. Isoytan beskriver en tredimensionell kontur. Principen bakom algoritmen är, som nämnts tidigare, att dela in en definierad volym i en serie mindre volymer (voxlar eller celler). Genom att ”marschera” igenom alla voxlar, därav namnet, och testa deras hörnpunktsvärden mot ett så kallat isovärde (”isovalue”), ersätts voxeln med lämplig(a) polygon(er). Hur polygonen/polygonerna ska se ut bestäms av två olika tabeller: en kanttabell och en triangeltabell. Kanttabellen beskriver hur polygonen/polygonernarna ska ligga, medan triangeltabellen beskriver hur många trianglar som ska bildas. Som mest kan fyra trianglar skapas om Lorensens och Clines ursprungliga triangeltabell används. Med en modifierad triangeltabell kan som mest fem trianglar skapas [5, 12].

Upplösningen på geometrin blir mer detaljrik desto mindre varje voxel är (figur 3). Prestandan påverkas dock beroende på antalet voxlar som finns i den definierade volymen; ju mindre varje voxel är, desto mer voxlar blir det i den definierade volymen. Detta leder till sämre prestanda då fler voxlar måste testas mot volymdatan. Med DirectX 10.0 och senare versioner, samt med ”Shader Model 4.0” (som kom i samband med DirectX 10.0), är det dock möjligt att implementera Marching Cubes på GPU[3]n. Detta tack vare ett nytt stadium som tillkom i den programmerbara ”pipelinen”, som kallas ”Geometry Shader”. Detta stadie exekveras efter ”Vertex Shader”n och före ”Pixel Shader”n. Geoemtry Shadern gör det möjligt att generera enkla primitiver (som t.e.x punkter och trianglar) av de primitiver som Vertex Shadern först tog emot av applikationen. Det är möjligt att generara fler primitiver än vad Vertex Shadern skickar in till Geometry Shadern, vilket t.ex. kan användas för att öka detaljrikedomen hos en geometri. Mindre primitiver än vad som tas emot kan även genereras, något som kan användas för att skapa detaljeringsgrad (LoD[4]). Genom att implementera Marching Cubes på GPUn, är det möjligt att erhålla en prestandavinst jämfört med

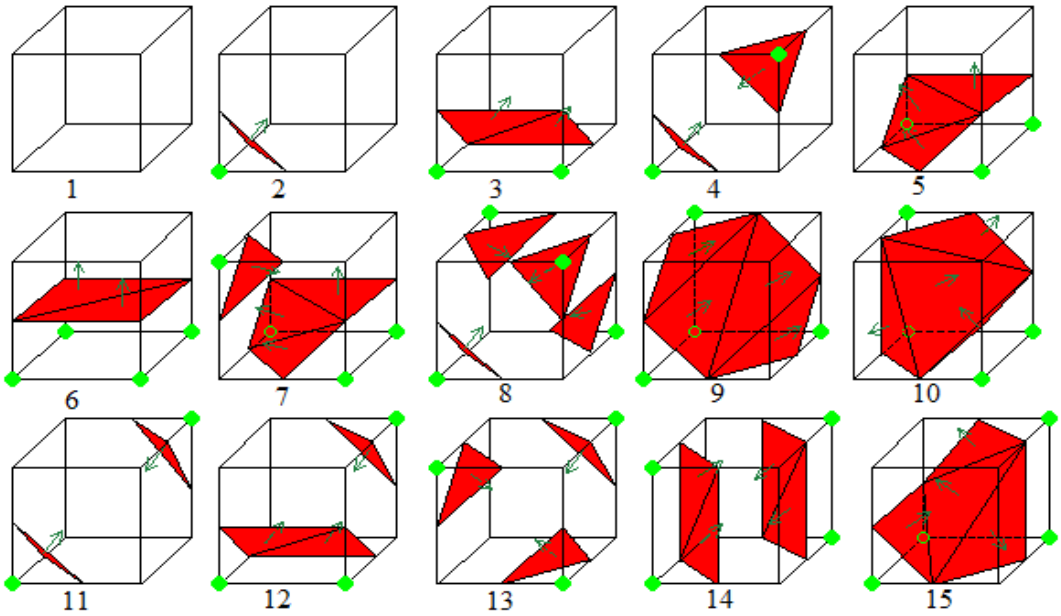
3 Graphics Processing Unit
4 Level of Detail

om beräkningarna skulle ha gjorts på CPU[5]n, eftersom den sekvensbearbetar data vilket inte är lämpligt för väldigt komplex geometri. Där passar istället parallell bearbetning bättre [1, 5].



Figur 3: En modell med olika upplösningar som ger olika detaljrikedom. Ju längre åt höger, desto mindre är voxlarna.

Det finns 2^8, d.v.s. 256 olika kombinationer för hur geometrin i en voxel kan se ut, om man enbart tillåter att de 8 hörnen antingen kan vara högre eller lägre än det förutbestämda isovärdet [10]. Genom att ta hänsyn till reflektion, symetrisk rotation och invertering, kan de 256 olika kombinationerna beskrivas med hjälp av 15 unika fall (figur 4) [6, 7, 10, 11, 15]. Ibland räknas inte fall 1 (voxeln längst upp till vänster i figur 4) med, då den inte bidrar med något (den är tom). Då är det alltså 14 unika fall som räknas.



Figur 4: De 15 unika fallen. De gröna punkterna beskriver vilket/vilka hörn som är lägre än isovärdet, och de gröna pilarna beskriver hur normalen ser ut[6].

3 Marching Cubes

3.1 Hur fungerar Marching Cubes algoritmen?

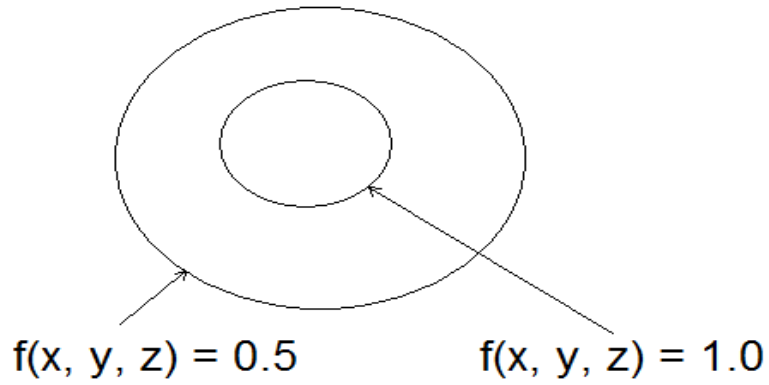
Marching Cubes är en algoritm för polygonisering av en isoyta, d.v.s. den producerar en yta (polygon) av ett skalärt datafält [10]. En isoyta är en yta som representerar punkter av konstant

5 Central Processing Unit

6 Bilden är inte helt korrekt, då kub 10 och kub 15 är varandras spegling.

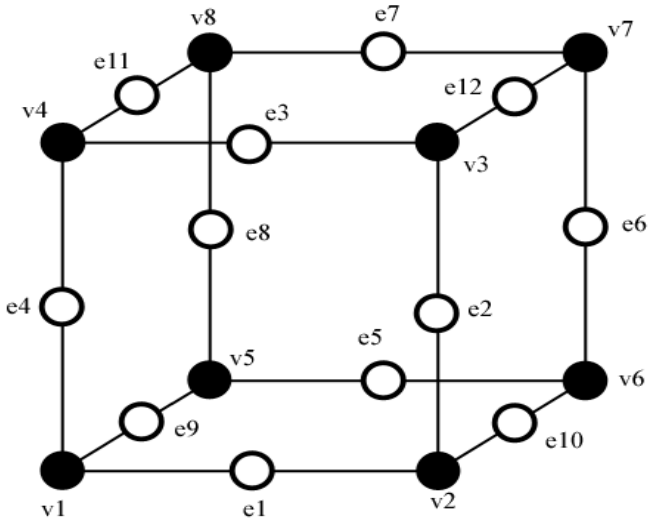
värde i tre dimensioner, och beskriver en kontur. Med hjälp av endast en funktion, kan alla punkter i tre dimensioner (x, y, z) beskrivas med ett decimaltal $f : R^3 => R$. Decimaltalen beskriver vanligtvis densitetsinformation eller avstånd till en yta. Om en funktion *f(x, y, z)* definierar ett skalärt datafält, kan en isoyta S satisfiera ekvationen *f(x, y, z) = konstant* [2].

Som nämnts tidigare, är en definierad volym indelad i K*M*N voxlar, där varje voxel motsvarar en kub. En voxel beskrivs med hjälp av åtta hörn, där varje voxel delar fyra hörn med dess närliggande voxlar. På så sätt är de sammankopplade med varandra, och oavsett i vilken ordning de besöks, så bildas samma yta. Algoritmen går till som så, att varje voxels åtta hörn blir tilldelade ett värde (decimaltal). Dessa värden jämförs med ett fördefinierat värde, ett så kallat isovärde, för att se vilka hörn som är innanför eller utanför ytan [10, 11, 12]. Antingen är ett hörns värde högre eller lägre än isovärdet. På så sätt reduceras antalet olika kombinationer på hur geometrin i en voxel kan se ut till 256. Det bestämmer även hur geometrin ska se ut beroende på vilka hörnvärden som är högre eller lägre än isovärdet. Om ett eller flera hörn har värden mindre än isovärdet, och ett eller flera hörn har högre värden, betyder det att voxeln ska bidra med någon slags polygon för att representera isoytan. Annars, d.v.s. om alla hörnens värden antingen är lägre eller högre än isovärdet, bidrar inte voxeln med något. Isovärdet påverkar geometrins storlek (figur 5). Ju närmre noll isovärdet ligger (det får dock inte vara exakt noll), desto mer syns det av isoytan. Om isoytan t.ex. representerar en människohand, kan man med hjälp av isovärdet specificera vad som ska visas. När isovärdet är nära noll syns hela handen, men om värdet är t.ex. 0,5 kanske bara inre delar syns (som t.ex. blodkärl och nerver) [11].

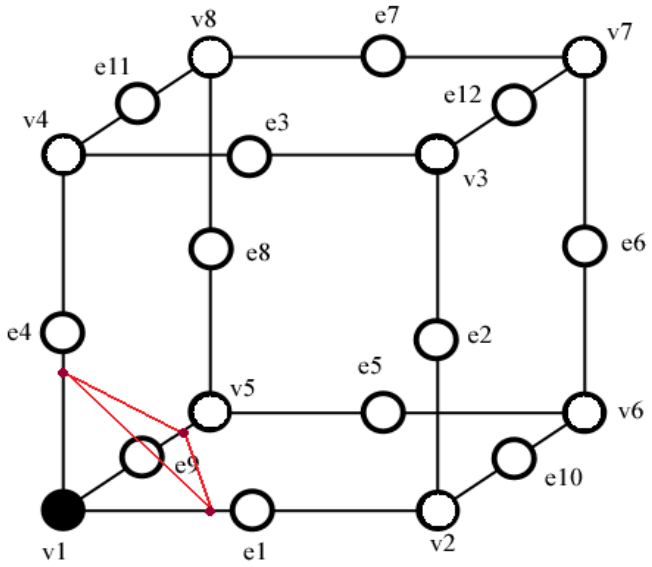


Figur 5: Två sfärer med olika isovärden ger olika radier.

För att underlätta bestämning av alla olika geometriska kombinationer, har ett teckensystem anammats för voxlarnas hörn och kanter (figur 6) [7]. Varje hörn i en voxel motsvaras av 1-bit i ett 8-bitars register, som brukar kallas kubregister. Hörn v1 till v8 i figur 6 motsvaras alltså av ett värde mellan 1 och 128 där v1 = 1, v2 = 2, v3 = 4, v4 = 8, v5 = 16, v6 = 32, v7 = 64 och slutligen v8 = 128. Hörn v1 t.ex., har ett register som ser ut så här 0000 0001 (binärt) om hörnvärdet är lägre än isovärdet. Om hörnvärdet är högre, sätts biten till 0, d.v.s. registret blir 0000 0000. Som ett exempel undersöks fall 2 i figur 4 av de 15 unika. I just det fallet är alla hörnens (v2-v8) värden över isovärdet, medan v1 ligger under. En polygon ska då bildas som skär längs kanterna e1, e4 och e9 (figur 7). Vilka kanter som skärs, bestäms av en kanttabell där alla de 256 olika skärningskombinationerna redan är uträknade. Kanttabellen använder kubregistret och returnerar ett 12-bitars nummer, där varje bit motsvarar en kant. I exemplet ovan, där hörn v1 låg under isovärdet, ger alltså kanttabellen det binära numret 0001 0000 1001, vilket motsvarar kant e1, e4 och e9.



Figur 6: Exempel på ett teckensystemet för voxlarna, där v står för ”vertex” (hörn) och e står för ”edge” (kant).



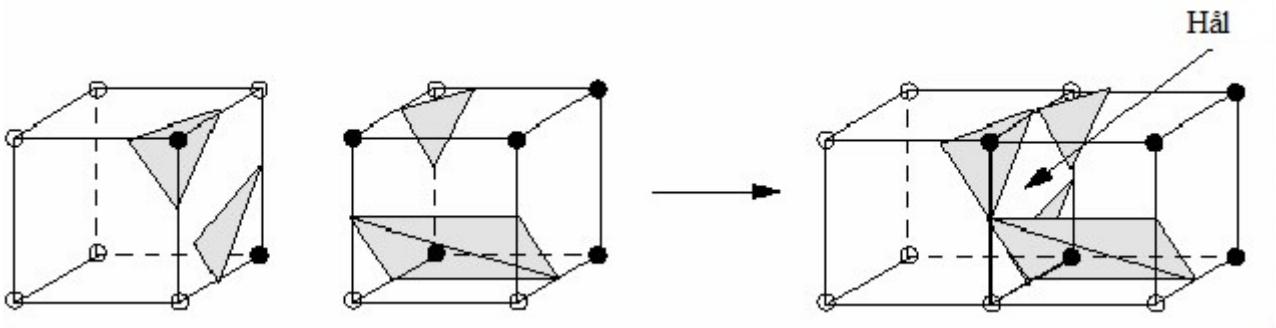
Figur 7: Fall 2 från de 15 möjliga kombinationerna. Polygonen (röd) skär e1, e4 och e9. Hörn v1 (svart punkt) ligger under isovärdet.

Exakt var skärningen av kanterna sker, d.v.s. var polygonens hörn ska vara, beror på relationen mellan isoytvärdet och värdet vid hörnen v1-v2, v1-v5 och v1-v4. Detta görs med hjälp av linjär interpolering [10]. Om t.ex. värdet vid v1 är 0,1 och värdet vid v2 är -0,3, ska polygonens hörn placeras 25% av vägen längs kanten från v1 till v2, då värdet blir noll [1]. Detsamma gäller relationen mellan de andra hörnen. Vid uträkning för detta kan följande ekvation användas: P = P1 + (isovärde – V1)(P2 – P1) / (V2 – V1), där P1 och P2 är hörnen vid en skärningskant och där V1 och V2 är värdena vid hörnen. En polygon är uppbyggd av ett visst antal trianglar, där varje triangel kräver tre punkter för att kunna skapas. För att hålla reda på vilka tre punkter som ska bilda rätt triangel, samt hur många trianglar som ska bildas för att representera ytan, används en tabell som brukar kallas triangeltabell. Tabellen använder, precis som hörntabellen, ett kubregister för att ta hänsyn till de 256 olika fallen. I exemplet ovan där v1 låg under isovärdet, skulle triangeltabellen

returnera följande värden { 3, 11, 2, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1 }, vilket medför att en triangel ska bildas mellan kant 3, 11 och 2 [10]. Om en kant inte bidrar med något, sätts den till -1. När algoritmen väl loopat igenom alla voxlar, en och en [5, 6], representeras tillslut en isoyta som en polygon mesh.

3.2 Brister med Marching Cubes

Ett problem med Marching Cubes vad gäller närliggande voxlar är att de kan skapa tvetydigheter eller ”hål” då de kopplas samman (figur 8) [3, 6, 9]. Detta sker då de båda innehåller olika fall. Det är en nackdel vad gäller Marching Cubes-algoritmen. Detta går dock att lösa genom att t.ex. tillföra fler fall som komplement till de redan 256 olika fallen. Det finns även andra algoritmer som löser detta problem. T.ex. kan en snarlik algoritm, som utvecklades i samband med Marching Cubes, lösa dess tvetydigheter. Denna algoritm heter Marching Tetrahedrons. Det finns både för- och nackdelar med de olika algoritmerna. Eftersom varje voxel i Marching Tetrahedrons är indelad i 6 stycken tetraeder, med fyra hörn var istället för åtta i en voxel, minskar kanttabellen med de olika skärningskombinationerna betydligt. Detta eftersom algoritmen går igenom en tatreader åt gången. En annan fördel med Marching Tetrahedrons är att varje voxel får 19 kanter (6 kanter per tetraeder, gånger 6 tetraeder plus en gemensam kant) istället för 12 som Marching Cubes använder. I och med fler kanter, leder det till att detaljrikedomen blir något bättre med Marching Tetrahedrons, men representationen av isoytan kan dock bli missvisande. Algoritmen blir även mer komplicerad och minnesbehovet ökar, då antalet polygoner (trianglar) ökar[5, 9, 10].



Figur 8: Två närliggande voxlar bildar ett hål.

Hur exakt och detaljerad representationen av en isoyta blir med Marching Cubes, har bl.a. att göra med antalet voxlar, som nämnts tidigare. Ju mer voxlar det är, ju högre blir också prestandakraven då fler voxlar måste testas. Eftersom det är volymer det handlar om, ökar antalet test som måste genomföras markant för varje voxel som läggs till. En volym som t.ex. har 2*2*2 = 8 voxlar mot en som har 3*3*3 = 27 voxlar, visar att det blir betydligt fler voxlar även fast det bara är 1 voxel mer i längd, djup och bredd. Olika optimeringstekniker börja dyka upp mer och mer för att tillåta fler voxlar, vilket leder till ökad prestanda och detaljrikedom.

3.3 Vilka användningsområden kan dra nytta av Marching Cubes?

Marching Cubes utvecklades för att användas inom det medicinska området för tredimensionell visualisering av volymetrisk data [15]. Det är än idag det största användningsområdet. Exempel på andra användningsområden som kan dra nytta av Marching Cubes är datorspelsprogrammering och vid geologiska rekonstruktioner [9].

Algoritmen vänder sig främst till applikationer med komplex geometri, eller där visualisering av

inre strukturer är viktigt, eller där hög detaljrikedom och realism är ett krav (t.ex. representation av en människokropp).

För realtidsapplikationer som t.ex. datorspel, där antalet bilder per sekund (FPS[7]) bör vara högt för att få en jämn/naturlig rörelse, är det viktigt att optimera algoritmen, eller snarare hanteringen av voxlar. Stora öppna terränger i t.ex. datorspel, skulle bli allt för krävande utan att voxlarna struktureras upp. Den definierade volymen skulle behöva bli lika stor som terrängen, och för att inte tappa detaljrikedom skulle den behöva innehålla ofantligt många voxlar. Olika datastrukturer gör det möjligt att frånkomma dessa problem. Det går t.ex. att använda en ”Octree”-struktur för att på så sätt kunna räkna bort voxlar som inte syns i bild. Ett annat sätt är att variera storleken på voxlarna beroende på hur de förhåller sig till kameran, d.v.s. ju längre bort från kameran, desto större voxlar. Även många andra optimeringstekniker finns tillgängliga för att ge möjlighet till användning av volymrendeirng och Marching Cubes i realtidsapplikationer där t.ex. prestanda och detaljrikedom är ett krav.

3.3.1 Användningsområde inom medicin

Datortomografi, även kallat skiktröntgen, används inom medicinsk diagnostik för att avbilda delar av människokroppen, t.ex. huvudet. Genom att föra en patients kropp igenom en roterande ”trumma” som sänder ut en röntgenstråle igenom kroppen ett visst antal gånger per rotation (figur 9), kan en serie tvådimensionella tvärsnittsbilder skapas. Dessa visar olika skikt av kroppen och beroende på vävnad erhålls olika färgnyanser. Tvådimensionella anatomibilder kan dock vara svåra att föreställa sig i tre dimensioner utan speciell träning och erfarenhet. Marching Cubes gör det möjligt att representera dessa tvådimensionella bilder i tre dimensioner, vilket t.ex. gör att det blir lättare att bilda sig en uppfattning när det gäller fastställande av diagnos. Vissa krav ställs på de tvådimensionella bilderna för att algoritmen ska kunna representera en korrekt tredimensionell yta. Upplösningen måste vara samma på alla bilder, då motsvarande pixel i varje tvärsnittsbild ska överensstämma med varandra. Bilderna får alltså inte vara roterade eller flippade olika, eftersom de motsvarande pixlarna inte skulle stämma överens. Ordningen bilderna togs i spelar också stor roll för hur den tredimensionella representationen blir. De måste vara ordnade sekventiellt för att på rätt sätt kunna representeras som en tredimensionell geometri. Om t.ex. första halvan av bildserien är tagen av huvudet, och av misstag en bild av magen kom in efter det, skulle det innebära att huvudet satt ihop med magen, vilket ju inte stämmer [11].

Andra tillämpningar inom det medicinska området där Marching Cubes kan användas för att representera en tredimensionell geometri av ett antal tvådimensionella bilder är bl.a. magnetröntgen och ultraljud.

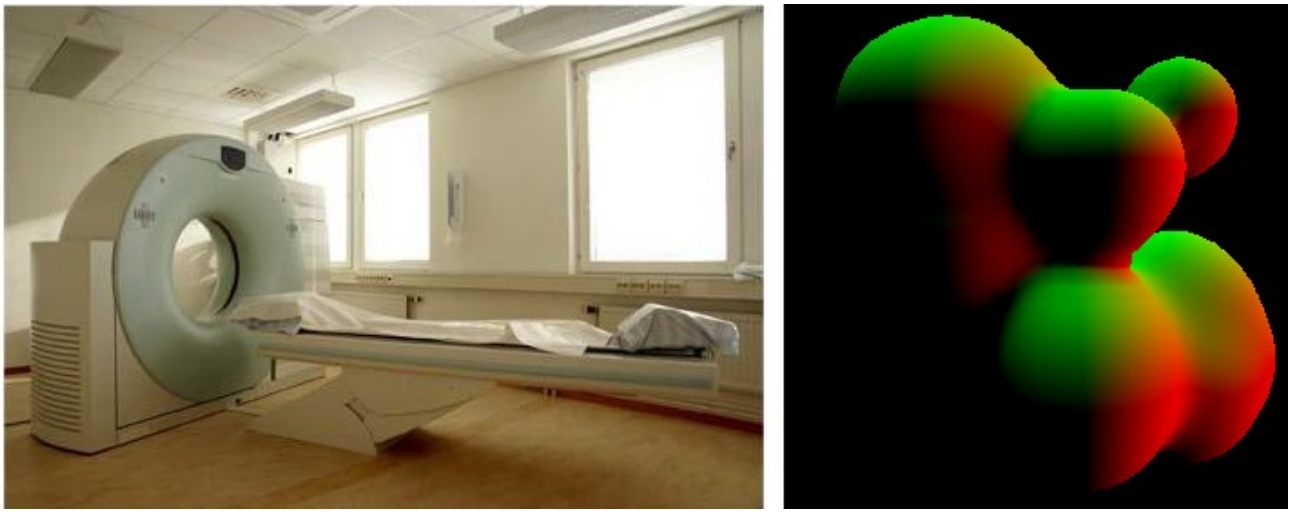
3.3.2 Användningsområde inom datorspel

Volymrendering är ingen ny teknik vad gäller rendering av datorspelsgrafik (främst terränger) då flera kommersiella datorspel använde sig av det i slutet av nittiotalet. Några exempel är ”Outcast”, ”Deltaforce”-serien och ”Commanche”-serien.

De flesta grafikkort arbetar med polygoner för att bygga upp tredimensionella objekt, något som gör Marching Cubes väldigt lämpad att använda för att uppnå olika effekter och resultat. ”Metaballs” t.ex. kan dra nytta av Marching Cubes-algoritmen. En metaball är helt enkelt en sfär som interagerar med andra metaballs (sfärer), då de ”smälter” ihop vid kontakt (figur 9). Denna teknik lämpar sig väl för bl.a. simulering av vätskor och organiska former, något som kan vara användningsbart inom

7 Frames Per Second

datorspel [2]. Ett annat användningsområde inom spelprogrammering är vid rendering av terränger. I dagsläget används mest så kallade ”heightmaps” i datorspel för att representera terränger. En heightmap är en fil som innehåller data motsvararande hur ytan av en terräng ska se ut. Nackdelen med det är att enbart berg och dalar kan visas, då det bara är ett ”topplager” av data som beskriver terrängen. Genom att använda volymrendering och Marching Cubes, är det möjligt att skapa väldigt komplexa terränger med t.ex. grottor och överhäng [1]. Det går alltså inte med heightmaps. En fullt förstörbar terräng skulle även vara möjlig med denna teknik, vilket är något som ökar realismen ytterligare i datorspel.
Om t.ex. en projektil träffar terrängen, går det att sätta voxlarnas hörnpunkter inom en viss radie från nedslaget till ett negativt värde. Detta medför att en krater skulle bildas i terrängen.



Figur 9: Till vänster: Datortomografi. Till höger: Flera metaballs som smälter in i varandra.

4 Implementering av Marching Cubes

4.1 Metaballs

För att se Marching Cubes fungera i praktiken, implementerade jag metaballs, som nämnts tidigare. Metaballs är sfärer som beskrivs med hjälp av en implicit ekvation (ekvation 1) där en gradiens kan beräknas direkt (ekvation 2) [2].

$$\sum_{i=1}^{N} \frac{r_i^2}{\left\| \mathbf{x} - \mathbf{p}_i \right\|^2} = 1$$

Ekvation 1: En implicit ekvation.

$$\mathbf{grad}(f) = -\sum_{i=1}^{N} \frac{2 \cdot r_i^2}{\left\| \mathbf{x} - \mathbf{p}_i \right\|^4} \cdot (\mathbf{x} - \mathbf{p}_i)$$

Ekvation 2: Gradiensen beräknas direkt i en implicit ekvation.

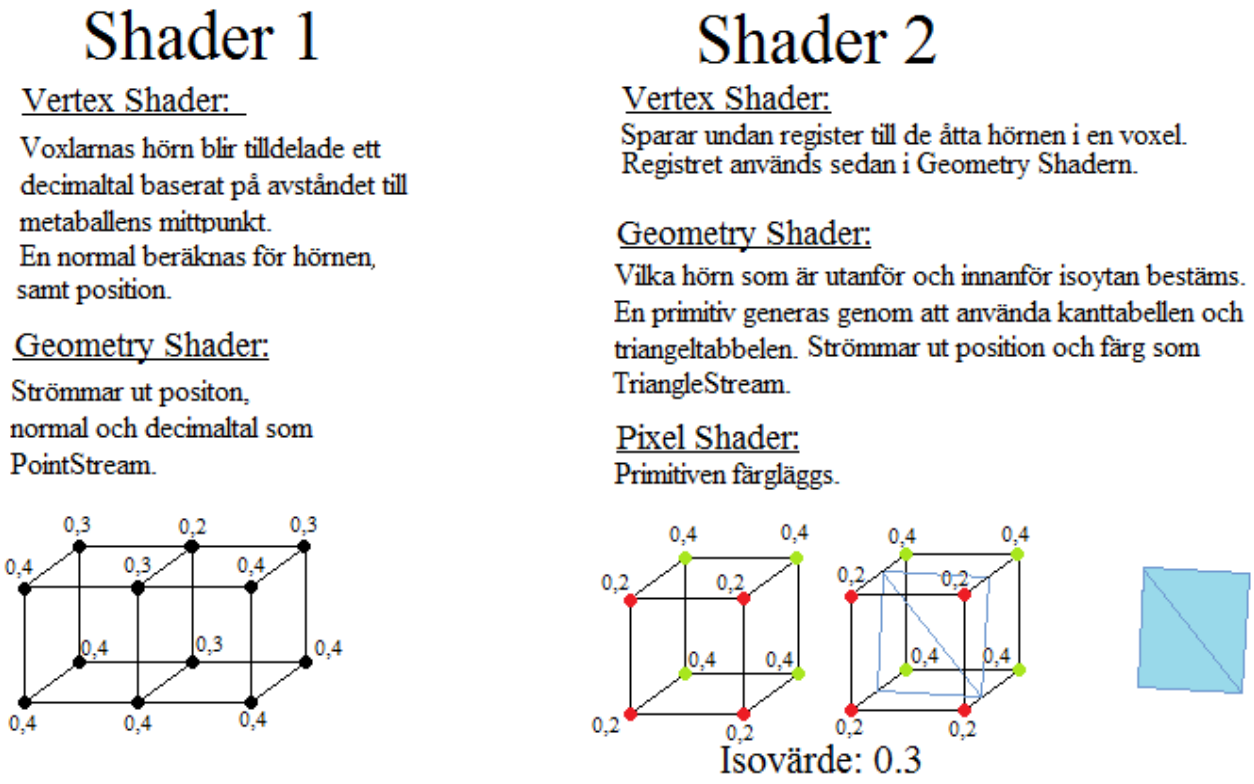
Den implicita ekvationen beräknar ett decimaltal (x) för varje voxels hörn vid en given radie (r) och mittpunkt (p). Detta beräknas för varje metaball, där (i) indikerar vilken metaball som avses, och N motsvarar antalet metaballs. Decimaltalet är baserat på avståndet till mittpunkten, så ju närmre en punkt är mitten, desto större blir värdet. Isovärdet bestämmer hur stor radien på en metaball blir. Ju lägre isovärdet är (dock större än noll), desto större blir radien på sfären, och tvärt om ju högre det är. Om det är mer än en metaball, är det möjligt att addera ihop decimaltalen, för att på så sätt se hur de påverkar varandra. Det medför att de ”smälter” ihop med varandra vid sammanstötning och därmed ökar den totala volymen. Gradiensen beräknar hur normalen för varje punkt (voxlarnas hörn) ser ut. Normalen ska vara vinkelrät mot ytan.

4.2 Implementeringsutförande

För att få en bättre förståelse om hur algoritmen fungerar, valde jag alltså att implementera den. Som grafiskt API använde jag DirectX 10.0, och som programmeringsspråk C++. Microsoft Visual Studio 2008 valde jag som IDE[8], d.v.s. programutvecklingsmiljö. Eftersom det är möjligt att implementera Marching Cubes helt på GPUn tack vare Geometry Shader (GS), och på så sätt vinna prestanda (eventuellt), valde jag att göra det.

Efter att ha läst en hel del om hur det går att implementera Marching Cubes på GPUn [2], samt undersökt Nvidias ”SDK[9] sample” för att se hur de implementerat metaballs med Marching Tetrahedrons [14], satte jag igång med programmeringen. När väl DirectX och Win32 var initialiserat, började jag programmera det som krävdes för att algoritmen skulle fungera. Till att börja med skapade jag ett tredimensionellt rutnät, som motsvarar alla voxlar. Detta rutnät skickas upp till GPUn, då varje punkt i rutnätet motsvarar en voxels hörn. Hur stor volym det ska vara, samt hur många voxlar som ska finnas i volymen, går att ändra. För att kunna se hur stor volymen är samt antalet voxlar, valde jag att ha möjligheten att rita ut det tredimensionella rutnätet som punkter. Det underlättar felsökning en hel del. Eftersom jag bestämde att implementera Marching Cubes på GPUn, använde jag mig av “Shaders” och “HLSL”. Det blev nästa steg att uföra i implementering. Jag använde mig av två shaders (Shader 1 och Shader 2) för att dela in Marching Cubes-algoritmen i olika steg, vilket gör att det blir lite mer överskådligt anser jag. Vertex Shadern i Shader 1 beräknar metaballs genom att få en mittpunkt och en radie från applikationen, samt att den tilldelar voxlarnas hörn (d.v.s. alla punkter i det tredimensionella rutnätet) med ett värde (decimaltal) beroende på hur de förhåller sig till sfären/sfärerna. En normal för ytan räknas även ut. Geometry Shadern strömmar sedan ut position, normal och hörnpunkternas värde (decimaltalet) som punkter (“PointStream”). Ingen Pixel Shader används i Shader 1, då den fungerar som en uppdatering av all data som Marching Cubes behöver för att fungera. Ingenting ritas heller ut på skärmen i detta steg. En buffer används innehållande bl.a. hörnpunktsvärden, för att kunna användas i nästa shader (Shader 2). Den andra shadern, d.v.s. Shader 2, bestämmer hur geometrin i varje voxel ska se ut, samt att den ritas ut som polygoner (trianglar). Både hörn och triangeltabellen ligger i denna shader. Vertex Shadern får in register från applikationen, som håller reda på de åtta olika hörnen i en voxel. Geometry Shadern behandlar en voxel åt gången, d.v.s. åtta hörn, och kollar vilket/vilka av hörnen som ligger under och över isovärdet. När det är gjort beräknas skärningspunkterna ut med hjälp av kanttabellen, för att sedan producera och forma rätt antal trianglar med hjälp av triangeltabellen. Upp till fyra trianglar kan som mest strömmas ut (“TriangleStream”) från Geometry Shadern, då det är det maximal antalet trianglar Marching Cubes-algoritmen kan generera. Trianglarna färgläggs sedan i Pixel Shadern med hjälp av normalerna, som då även gör det möjligt att se om de verkar korrekta. Nedan är en bild (figur 10) som illustrerar vad de båda shaders har för uppgift.

8 Integrated Development Environment
9 Software Development Kit



Figur 10: Uppgifterna Shader 1 och Shader 2 har.

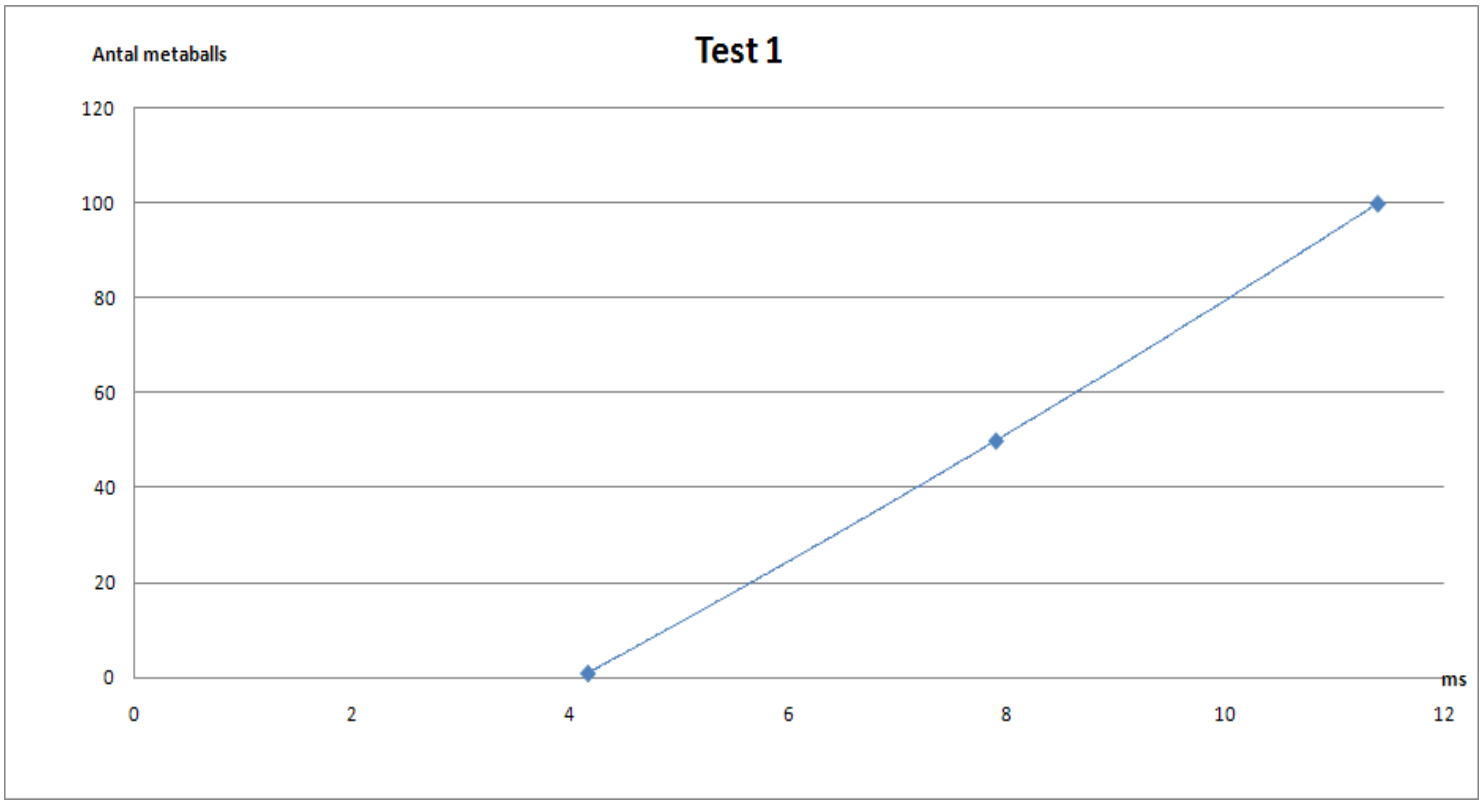
Det var betydligt svårare än vad jag först trodde att implementera Marching Cubes. Mycket av det berodde på att jag inte riktigt visste hur jag skulle överföra teorin till praktiken. Hade jag inte haft möjligheten att kunna studera källkod från en implementering [14], skulle jag nog inte ha klarat av det. När väl allt fungerade, efter mycket om och men, genomförde jag några tester. De gick ut på att se hur bilduppdateringsfrekvensen (i millisekunder) ändrade sig beroende på olika inställningar.

4.3 Testresultat

Jag genomförde två olika tester för att se hur bilduppdateringsfrekvensen betedde sig beroende på givna förutsättningar. Datorns specifikationer som testerna genomfördes på var följande:

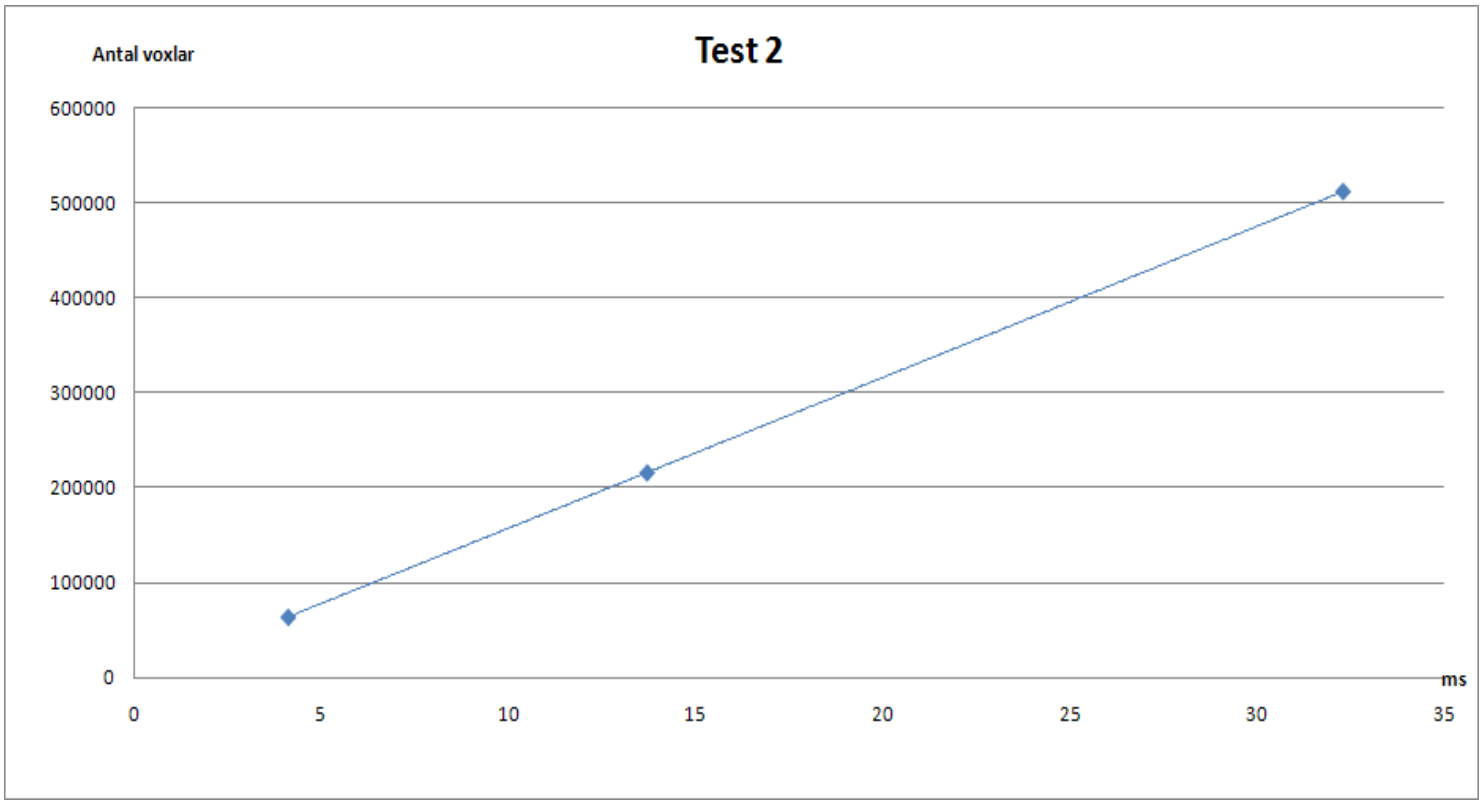
Processor: AMD Athlon 64 X2 Dual Core Processor 6000+ 3,01 Ghz
Minne (RAM): 4.00 GB
Grafikkort: NVIDIA GeForce 8800 GTX 768 MB OC 2
Operativsystem: Windows Vista Ultimate 64-bit, SP 2

Det första testet bestod av att se hur bilduppdateringsfrekvensen påverkades av antalet metaballs, d.v.s. antalet geomtrier. Den definierade volymen och antalet voxlar var samma oavavsett hur många metaballs det var, för att ge så lika förutsättningar som möjligt. Volymen var 20*20*20 enheter, med 64000 voxlar á 0,5*0,5*0,5 enheter och isovärdet var 0,9. Metaballs:en rörde sig slumpmässigt i x, y och z riktning och alla hade samma radie på 2,0 enheter. Antalet metaballs var 1, 50 och 100. Testresultat såg ut på följande vis (figur 11).



Figur 11: Diagram över resultatet för det första testet.

Andra testet bestod av att se hur antalet voxlar i den definierade volymen påverkade bilduppdateringsfrekvensen. Den definierade volymen var 20*20*20 enheter precis som föregående test. Antalet metaballs var noll för att inte ta hänsyn till att någon geometri ritas ut. Antalet voxlar var 40^3, 60^3 och 80^3. Testresultatet såg ut på följande vis (figur 12).



Figur 12: Diagram över resultatet för det andra testet.

5 Resultat och diskussion

De testerna som genomfördes i rapporten är troligen inte helt representativa. Datorn som användes är nästan ett år gammal, och med tanke på att hårdvara utvecklas i så pass rask takt, kan denna till viss del anses föråldrad. Å andra sidan ger testerna ändå en god inblick i hur bilduppdateringsfrekvensen varierar beroende på de olika förutsättningarna.

I det första testet, då prestandan mättes genom att variera antalet metaballs, framkom det tydligt att bilduppdateringsfrekvensen är fullt duglig även med 100 stycken metaballs. Det verkar alltså inte som att Geometry Shader:n påverkar prestandan så värst mycket trots att antalet primitiver (trianglar) som genereras ökar betydligt med antalet metaballs (geometrier). Självklart minskar bilduppdateringsfrekvensen ju mer geometri som måste renderas, men absolut inte i så stor utsträckning som jag hade trott innan. Från det att bara en metaball renderas till att 50 stycken renderas, ökade millisekunderna från cirka 4,2 till ungefär 7,9; från 50 stycken metaballs till 100 stycken ökade de till runt 11,4.

Vad som framgick av det andra testet, var att antalet voxlar, d.v.s. deras storlek i den definierade volymen, definitivt kan vara ett problem för vissa typer av realtidsapplikationer. För att det ska vara möjligt att ha en stor definierad volym, med väldigt små voxlar i, d.v.s. ett stort antal som ger en hög detaljrikedom, krävs någon form av optimering om det ska vara acceptabelt.

Båda testerna som genomfördes visar en linjär ökning av tidskomplexiteten i samband med att antalet metaballs och voxlar ökar. Detta medför bl.a. att det relativt lätt går att räkna ut t.ex. maximalt antal metaballs en realtidsapplikation kan ha, för att uppnå godtycklig bilduppdateringsfrekvens.

6 Framtid

I rapporten undersöktes hur voxelbaserad rendering med Marching Cubes-algoritmen fungerar i reapltidsapplikationer då ett antal tester genomfördes. Med dess breda användningsområde tror jag definitivt att det finns potential i framtiden. Trots att algoritmen är över 20 år gammal, håller den ändå en hög standard som går att tillämpa i dagens grafiska realtidsapplikationer.

Exempel på ett potentiellt användningsområde är inom datorspelsindustrin, där strävan efter realism i datorspel (t.ex. fullt förstörbar terräng) är stor, samt att dagens grafikkort är lämpade för att jobba med polygoner. Det är inga kommersiella datorspel som använt sig av just Marching Cubes, vilket jag tror har att göra med mjukvarupatenten som algoritmen hade. Numera är det dock fritt fram för alla att använda sig av den, eftersom patentet gick ut 2005. Något som jag dock tror kommer hota Marching Cubes-algoritmen i framtiden, är Volume ray casting eller liknande algoritmer. Volume ray casting är bildbaserad, d.v.s. den använder sig inte av någon slags geometri för att bygga upp volymetrisk data. I och med det, behöver det aldrig bli något bekymmer med för mycket eller för komplex geometri. Det medför att det kommer vara möjligt att ha otroligt detaljrik och avancerad grafik, utan att prestandakraven ökar. Utvecklingen av datorspel (andra grafiska realtidsapplikationer med för den delen) kommer samtidigt bli ”enklare” då optimeringar och liknande för att reducera antalet geometrier inte behövs. Det lär dock dröja ett tag innan denna teknik har etablerat sig, då den i dagsläget är allt för krävande.

7 Slutsats

Hur fungerar då Marching Cubes för realtidsapplikationer? Jag skulle vilja hävda att den fungerar väldigt bra när det gäller en mindre definierad volym med relativt stort antal voxlar i. Den fungerar också bra med en större definierad volym, men då med färre antal voxlar i. Vad som framkom av testerna visar helt klart att ”dagens” hårdvara, så som grafikkortet, är tillräckligt bra för att uppnå acceptabla bilder per sekund i realtidsapplikationer utan att optimera algoritmen.

Referenser

Litteratur:

[1] Hubert Nguyen, ”GPU Gems 3”, Pearson Education, Inc., 2007, ISBN 0-321-51526-9.

Artiklar:

[2] Sarah Tariq, ”Next Generation Effects in DirectX10”, Nvidia Siggraph presentation 2006, Boston.
Kan laddas hem från: http://developer.download.nvidia.com/presentations/2006/siggraph/dx10-effects-siggraph-06.pdf
Senast besökt, 2009-08-12

[3] Chin-Feng Lin, Don-Lin Yang and Yeh-Ching Chung, ”A Marching Voxels Method for Surface Rendering of Volume Data”, Proceedings of the International Conference on Computer Graphics 2001.
Kan laddas hem från: http://www.cs.nthu.edu.tw/~ychung/conference/cgi_2001.pdf
Senast besökt, 2009-08-12

[4] Hamish Carr, Thomas Theußl, Torsten Möller, ”Isosurfaces on Optimal Regular Samples”, ACM International Conference Proceeding Series; Vol. 40, Proceedings of the symposium on Data visualisation 2003, Frankrike.
Kan laddas hem från: http://www.cs.sfu.ca/~torsten/Publications/Papers/vissym03.pdf
Senast besökt, 2009-09-01

[5] Christopher Dykan, Gernot Ziegler, ”High-speed Marching Cubes using Histogram Pyramids”, EUROGRAPHICS 2007.
Kan laddas hem från: http://www.mpi-inf.mpg.de/~gziegler/hpmarcher/techreport_histopyramid_isosurface.pdf
Senast besökt, 2009-08-12

Internet:

[6] Michael Glanzing, Muhammad Muddassir Malik, M. Eduard Gröller, ”Locally Adaptive Marching Cubes through Iso-Value Variation”, Österrike, 2009.
Kan laddas hem från: http://www.cg.tuwien.ac.at/research/publications/2009/glanznig-2009-LAMC/glanznig-2009-LAMC-Full%20paper.pdf
Senast besökt, 2009-08-12

[7] http://users.polytech.unice.fr/~lingrand/MarchingCubes/algo.html
Senast besökt, 2009-08-12

[8] United States Patent, Patent Nr. 4,710,876, 1987.
http://www.google.com/patents/download/System_and_method_for_the_display_of_sur.pdf?id=T1Q3AAAAEBAJ&output=pdf&sig=ACfU3U14l_Y7ush3LINAnGAsXSwds5LcyQ&source=gbs_overview_r&cad=0
Senast besökt, 2009-08-12

[9] Thomas Lewiner, Hélio Lopes, Antônio Wílson Vieira and Geovan Tavares, ”Efficient implementation of Marching Cubes cases with topological guarantees”.
Kan laddas hem från: http://www.ks.uiuc.edu/Research/vmd/projects/ece498/surf/lewiner.pdf
Senast besökt, 2009-08-12

[10] Paul Bourke, ”Polygonising A Scalar Field”, 1994.
http://local.wasp.uwa.edu.au/~pbourke/geometry/polygonise/index.html
Senast besökt, 2009-08-12

[11] Burak Erem, Nicolas Dedual, ”Surface Construction Analysis using Marching Cubes”, Northeastern University, Boston.
Kan laddas hem från: http://www.ndedual.com/wp-content/uploads/marchingcubes.pdf
Senast besökt, 2009-08-12

[12] Dirk Bratz, Michael Meißner, ”Voxels versus Polygons: A Comparative Approach for Volume Graphics”, University of Tübingen, Tyskland.
Kan laddas hem från: http://www.gris.uni-tuebingen.de/people/staff//bartz/Publications/paper/vg99.pdf
Senast besökt, 2009-08-12

[13] http://en.wikipedia.org/wiki/Volume_rendering
Senast besökt, 2009-08-12

[14] http://developer.download.nvidia.com/SDK/10.5/Samples/MetaBalls.zip
Senast besökt, 2009-08-12

[15] http://www.marchingcubes.org/index.php/Marching_Cubes
Senast besökt, 2009-08-12

Bilder:

Figur 3: http://local.wasp.uwa.edu.au/~pbourke/geometry/polygonise/polygonise3.gif
Senast besökt, 2009-08-12

Figur 4: http://users.polytech.unice.fr/~lingrand/MarchingCubes/resources/marchingCnormales.gif
Senast besökt, 2009-08-12

Figur 6: http://users.polytech.unice.fr/~lingrand/MarchingCubes/resources/CubeNumbering.gif
Senast besökt, 2009-08-12

Figur 7: http://users.polytech.unice.fr/~lingrand/MarchingCubes/resources/CubeNumbering.gif
Modifierad av författaren.
Senast besökt, 2009-08-12

Figur 9 (vänstra bilden): http://www.direct-rontgen.se/backa/images/Siemens2.jpg
Senast besökt, 2009-08-12