Anders Fjeldstad
andfj645@student.liu.se

INLEDNING

Detta är en kort text som gör ett försök att på ett överskådligt sätt sammanfatta ämnet vetenskaplig visualisering i allmänhet och kursen TNM056 Vetenskaplig visualisering (Ken Museth, Linköpings tekniska högskola) i synnerhet. Innehållet bygger till mesta delen på de föreläsningar som givits under kursen, men även till viss del på kapitlen fyra till sju i boken "The Visualization Toolkit" (tredje utgåvan) av Schroeder, Martin och Lorensen samt på diverse forskningsartiklar – jag är inte så noga med att ange källor, det är bara att acceptera. Syftet med texten är att läsaren ska få en snabb inblick i ämnet tillsammans med några mer noggranna genomgångar av viktiga begrepp.

Innan vi börjar vill jag passa på att påpeka att den här texten inte har genomgått någon noggrann korrekturläsning överhuvudtaget, så det kan mycket väl hända att det förekommer trista stavfel eller andra språkliga manövrar som inte är helt regelrätta – en vild blandning av svenska och engelska facktermer är ett exempel på detta. Jag hoppas att dessa inte ska vara så frekventa att det totala intrycket försämras. Dessutom är det inte helt omöjligt att jag gjort faktamässiga misstag. Jag tror inte att det är så, men jag tänkte ändå säga det för att vara på den säkra sidan.

Hursomhelst, trevlig läsning! Och lycka till på tentan, om du läser inför en sådan...

VAD ÄR VETENSKAPLIG VISUALISERING?

Vetenskaplig visualisering är ett ämne inom datorgrafik som går ut på att hitta på metoder för att visa upp vetenskaplig data som bilder på ett sätt som gör att betraktaren får ut mer än om denne tittat direkt på bakomliggande information. Tanken är att man genom sådana visualiseringar ska hjälpa betraktaren att *förstå* data och att låta denne *utforska*, *undersöka* och *analysera* den.

Ofta vill man visualisera förlopp som normalt ligger bortom vad den mänskliga synen kan hantera. Det kan till exempel vara att titta på sådant som rör sig väldigt snabbt, sådant som är mycket litet, sådant som ligger långt borta eller att se insidan på solida objekt. En viktig och vanlig tillämpning hittar man inom medicin, där man till exempel kan utforska resultatet från en magnetröntgen i 3D och undersöka vävnaden i en människa.

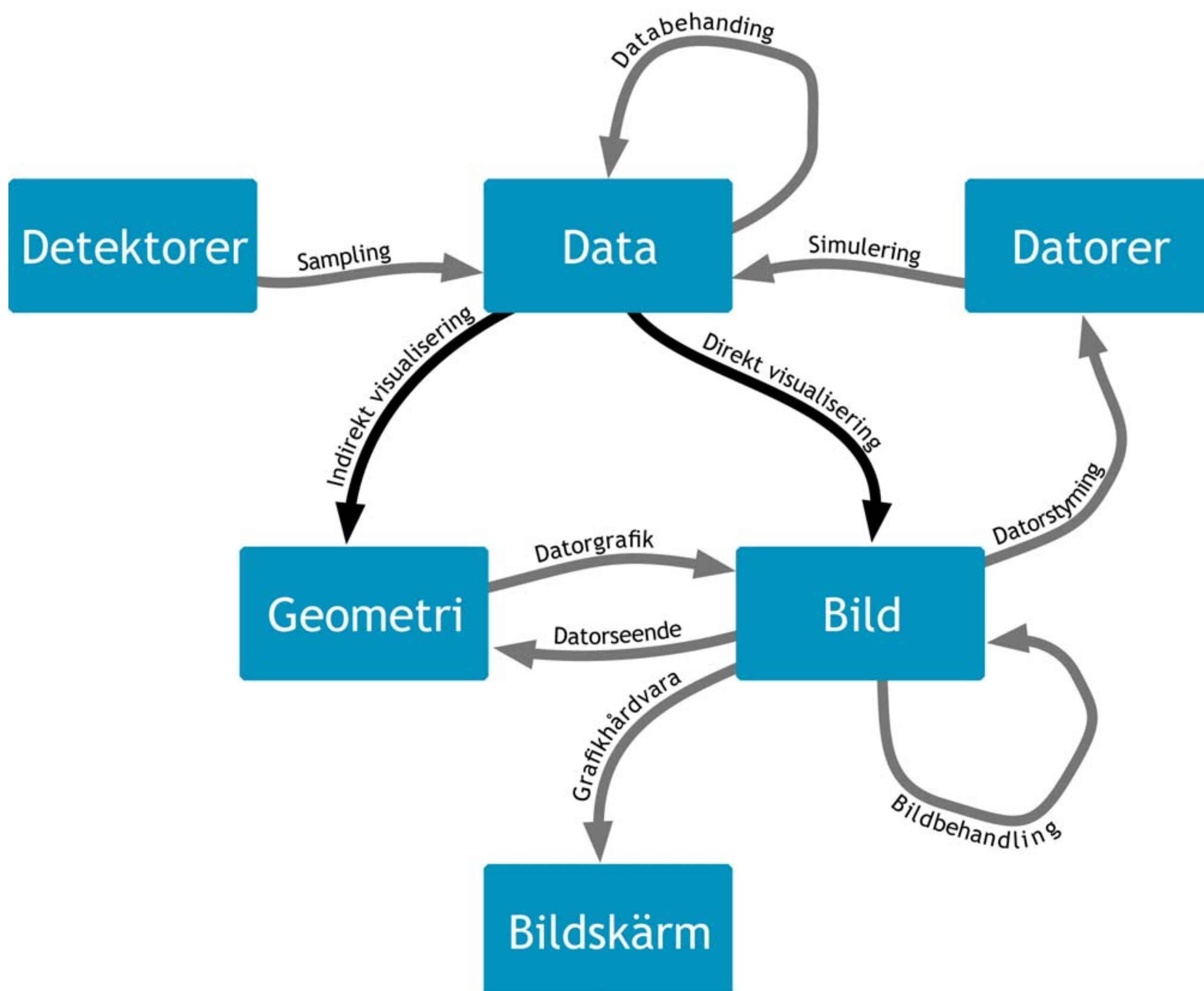
Även om avancerade renderingstekniker och 3D-grafik ofta används så är det viktiga inte att skapa fina bilder, utan att bilderna ska kunna förmedla den information de representerar – man är alltså inte nödvändigtvis ute efter fotorealism.

Visualisering – att visa upp information i form av bilder – är ett kraftfullt sätt att föra fram ett budskap. Det man kan se med sina egna ögon tenderar man att tro på – förutsatt att man inte betvivlar att tekniken fungerar. Därför är vetenskaplig visualisering både viktigt och komplicerat.

Anders Fjeldstad
andfj645@student.liu.se

VISUALISERINGSSYSTEMATIK

Hur passar då visualisering in i datorgrafikområdet och vilka olika processer finns? För att göra det litet klarare kan man titta på figuren nedan, som jag ritat av från Ken Museths föreläsningsmaterial.



Till att börja med så måste man ha någon form av (vetenskaplig) data. Denna kan antingen komma från datorsimuleringar eller från ”verkligheten” via samplingar (mätningar). Oavsett vilken metod man använder så kommer man få data som består av ett antal *diskreta* värden.

Nästa steg är att förbehandla data. Exempelvis kanske man vill konvertera från ett format till ett annat eller räkna ut en sekundär datatyp utifrån det man fått från simuleringen eller experimentet. När detta är klart är det dags att välja hur man ska visualisera resultatet. Här har man två huvudsakliga val: antingen går man direkt till en bild (så kallad *direkt* visualisering) eller så beräknar man *geometri* som ett mellansteg och använder sedan kända metoder från datorgrafiken för att skapa själva bilden. Det senare alternativet kallas *indirekt* visualisering.

Anders Fjeldstad
andfj645@student.liu.se

DATAINSAMLING

Som jag nyss nämnde så kan man samla in den data man ska visualisera antingen genom simuleringar i en dator eller genom att man samplar ett verkligt förlopp. För att ge en liten inblick i hur det praktiskt går till tänkte jag beskriva några datainsamlingstekniker från den kategori där man gör fysikaliska mätningar. Eftersom det hela tiden handlar om sampling – även vid datainsamling genom datorsimuleringar – så är det bra om vi börjar med att titta på en grundläggande sats.

NYQUIST-SHANNONS SAMPLINGSTEOREM

Det finns en matematisk sats som kallas för *Nyquist-Shannons samplingsteorem* som beskriver förhållandet mellan frekvenserna i den kontinuerliga signalen och samplingsfrekvensen.

> *En kontinuerlig funktion som beror av en variabel kan återskapas exakt från en mängd sampel med konstant avstånd till varandra om och endast om samplingsfrekvensen är minst dubbelt så hög som den högsta frekvensen i funktionen.*

Vad menas då med detta? Enkelt beskrivet så kan man säga att små detaljer i den kontinuerliga signalen inte kommer att synas i den diskreta signalen om det är för stort avstånd mellan samplen. Detta är ju ganska självklart. Att samplingsfrekvensen ska vara just dubbelt så hög är inte lika självklart, men eftersom det är bevisat så behöver vi inte bry oss om *varför* det är så, bara *att* det är så.

Rent matematiskt kan satsen uttryckas såhär:

$$2\Delta x < \lambda_{input}$$

Här är den litet annorlunda uttryckt – avståndet mellan närliggande sampel ska som mest vara hälften så stort som insignalens våglängd.

TEKNIKER FÖR DATAINSAMLING

Detta avsnitt ska berätta litet mer om olika tekniker för det datainsamling. De exempel som tas upp är hämtade från den medicinska världen, eftersom det är ett stort tillämpningsområde där vetenskaplig visualisering kan göra mycket nytta. Det finns naturligtvis många andra vetenskapliga mättekniker som kan ge upphov till data som går att visualisera (seismologiska mätningar, laserscanning, radar, ultraljud och så vidare) men de kommer inte att behandlas här.

Jag använder de engelska termerna medvetet eftersom jag tror att de är lättare att känna igen från litteraturen än de svenska, särskilt i kursen TNM056 (som ges på engelska).

Magnetic Resonance Imaging (MRI)

Magnetic Resonance Imaging (MRI) är en teknik där människan som ska undersökas läggs inuti en maskin som skapar ett mycket starkt magnetiskt fält runt denne. Sedan utnyttjar man att väteatomernas rotation i vatten inuti människan kommer att förändra magnetfältet märkbart för att skapa en bild av hur densitetsfördelningen av väteatomer i vävnaden ser ut. Data som kommer ut är en volym, alltså tredimensionell information.

Anders Fjeldstad
andfj645@student.liu.se

MRI är bra på att visa olika typer av mjuk vävnad, vilket är användbart om man till exempel försöker hitta en tumör. Ben och andra hårda strukturer visas däremot inte särskilt bra. Värt att nämna är att utsignalen har relativt låga brusnivåer, men att den kan variera väldigt mycket från tillfälle till tillfälle för en och samma försöksperson.

Vidare får man säga att MRI är förhållandevis säkert att använda. Så länge som det inte finns några lösa metallföremål i närheten (och så länge man inte har pacemaker eller liknande) så är det inte farligt för en människa att genomgå en MRI-scanning. Detta skiljer MRI från till exempel Computed Tomography som jag beskriver nedan.

Functional MRI (fMRI)

Functional MRI (fMRI) fungerar som MRI med tillägget att man gör mätningar över en viss tid, så utgående data varierar med tiden. Detta används för att mäta aktivitet i hjärnan.

Diffusion MRI (DW-MRI eller DT-MRI)

Diffusion MRI (DW-MRI eller DT-MRI beroende på utdatatyp) mäter vattnets spridning i hjärnan. Denna teknik utnyttjar att vatten sprids olika i olika typer av vävnad. Exempelvis kan vatten endsast spridas *längs med* nervtrådar och tumörer hindrar vatten från att spridas överhuvudtaget. Utdata är antingen rena skalärer (DW-MRI) eller tensorer (matriser i detta fall) (DT-MRI).

Computed Tomography (CT)

Computed Tomography (CT) baseras på röntgenteknik. En människa placeras inuti en maskin där roterande strålningskällor och motstående detektorer framförallt avbildar ben och liknande hårda strukturer från ett stort antal vinklar runt rotationsaxeln. Resultatet är (efter viss efterbehandling) ett antal tvärsnittsbilder av den scannade människan. Traditionellt har specialister använt bilderna direkt och i sina huvuden skapat sig en bild av hur de interna strukturerna i den undersökta personen ser ut. Med vetenskaplig visualisering kan man bygga ihop tvärsnitten till tredimensionella volymer som går att undersöka på ett mycket mer intutitvt sätt.

CT är en teknik som producerar en utsignal med höga brusnivåer. Dessutom är det faktiskt farligt att genomgå en CT-scanning eftersom man utsätts för strålning. Dosen är proportionell mot upplösningen på utsignalen, så vill man ha riktigt detaljerade bilder så kommer man att utsätta försökspersonen för mycket strålning, vilket naturligtvis inte är att rekommendera. Tekniken används dock allt mer i brottsundersökningar där försökspersonen inte lever och därmed inte tar någon skada av undersökningen.

Single Photon Emission CT (SPECT)

Single Photon Emission CT (SPECT) är en variant av CT där man injicerar ett radioaktivt spårämne i försökspersonen innan scanningen. Spårämnet avger gammastrålar som fångas upp av en gammakamera, och resultatet kan ge en bild av bland annat blodflödet i hjärnan.

Anders Fjeldstad
andfj645@student.liu.se

Positron Emission Tomography (PET)

Positron Emission Tomography (PET) bygger som SPECT på att man injicerar ett radioaktivt spårämne i personen som ska undersökas. Spårämnet avger positroner som efter en kort sträcka kolliderar med elektroner och avger två gammastrålar var. Dessa fångas i sin tur upp av gammakameror och ger en bild av blodflödet i hjärnan. PET fungerar alltså ungefär som SPECT, men ger bilder med högre kontrast.

Sammanfattningsvis kan man säga att MRI och CT upptäcker strukturer medan fMRI, DW/DT-MRI, SPECT och PET avbildar aktivitet.

Cryosection Imaging

En alternativ metod för avbildning av vävnad och struktur är Cryosection Imaging. Denna teknik bygger på att man manuellt och handgripligt skivar upp försöksobjektet (som måste vara nedfryst), färglägger intressanta delar av tvärsnitten och tar bilder av varje skiva. En uppenbar nackdel är förstås att försöksobjektet förstörs, att det tar lång tid och att det kan vara svårt att se precis hur de olika skivorna ska radas upp relativt varandra. Fördelar är att man får högupplösta bilder med bra kontrast.

Anders Fjeldstad
andfj645@student.liu.se

PIPELINES FÖR VISUALISERING

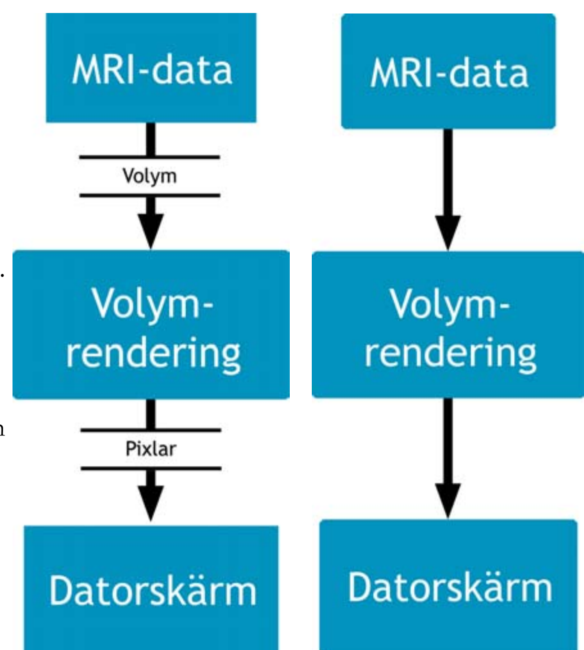
Att gå från vetenskaplig data till vettiga bilder är komplicerat och innefattar ofta många steg. För att processen ska bli mer överskådlig har man tagit fram olika ”pipelines” som beskriver hur visualiseringen går till. Det finns två huvudtyper av pipelines och jag ska berätta om båda två, men först tänkte jag gå igenom några begrepp som de har gemensamt.

En pipeline innehåller källor (sources), filter (filters) och sänkor (sinks). En källa producerar någon form av data, i vårt fall vetenskapliga sådana, och skickar ut dessa som en utsignal. Filter förändrar sin insignal och skickar resultatet som utsignal. Sänkor tar emot en insignal och konsumerar denna – ingen utsignal ges. Källor, filter och sänkor kan kopplas ihop (seriellt eller parallellt) för att åstadkomma det eftersökta resultatet.

Det finns två grundläggande regler som man måste ta hänsyn till när man ritar upp sin pipeline, oavsett vilken modell man väljer att använda. Dels har vi *multiplicitet*, som talar om hur många in- respektive utsignaler som en enhet (block) i piplinen får ha. Exempelvis kan man tänka sig att ett ”union-block” ska ta in två eller flera insignaler och skicka ut en utsignal, medan ett medelvärdesbildande filter tar in en insignal och skickar ut en utsignal. Sedan har vi *datatyp*, som avgör *vad* det är som får skickas in i ett block och vad det är som kommer ut. Detta är viktigt att hålla reda på – det går till exempel inte att skicka in intensitetsvärden i ett block som förväntar sig koordinater och normaler.

DEN FUNKTIONELLA PIPELINEMODELLEN

Den första typen av pipeline kallas för *funktionell*. I denna beskrivs källor och sänkor grafiskt som rektangulära block och filter som rektangulära block med rundade hörn. Signalerna mellan blocken ritas ut som pilar. Dessutom anges signalernas datatyp mellan blocken *explicit* genom att man skriver ut typen mellan parallella, horisontella linjer. Se den vänstra figuren här intill för ett exempel.



DEN NÄTVERKSBASERADE PIPELINEN

Den andra pipelinetypen kallas för *nätverksmodell*. Den är mindre strikt än den funktionella i och med att alla block (källor, filter och sänkor) betecknas likadant, som rektangulära block med rundade hörn. Signalernas datatyper ritar man inte ut överhuvudtaget – de anges *implicit*, vilket innebär att den som ska använda sig av pipelinen måste lista ut vilka datatyper som är aktuella baserat på sammanhanget. Den högra figuren här intill visar ett exempel på pipeline enligt nätverksmodellen.

EXEKVERING AV PIPELINEN

Det finns även två *exekveringstyper* för sådana här pipelines. Den första kallas *implicit pipelineexekvering*. Den är baserad på efterfrågan, så att när ett block behöver uppdateras (till exempel bildskärmen) så skickas det en förfrågan ”bakåt” i pipelinen tills antingen den första källan nås eller tills man stöter på ett block som är aktuellt och inte behöver få sin insignal uppdaterad. Sedan utförs exekveringen bakifrån och framåt, så att det block där propageringen av uppdateringsförfrågan stannade är det som uppdateras först, och det block som ursprungligen begärde uppdateringen exekveras sist. Denna modell används exempelvis i programpaketet VTK.

Anders Fjeldstad
andfj645@student.liu.se

Den andra typen av exekvering heter *explicit pipelineexekvering*. Där finns det en central enhet som kallas *executive* som har kopplingar till alla block och som håller reda på vilka som måste exekveras. Detta innebär förstås att alla block hela tiden måste tala om för executive-enheten när de modifierats. Denna metod används till exempel i det kommersiella visualiseringsprogrammet AVS.

TERMINOLOGI FÖR DATAMÄNGDER

Inom alla typer av vetenskaplig visualisering hanterar man datamängder av olika slag – det är ju det som är hela vitsen med alltihop. För att man ska ha ett gemensamt språk för att beskriva vad man håller på med så finns det en vedertagen terminologi som är nödvändig att känna till. Det är vad detta avsnitt handlar om.

Som jag nämnt tidigare så är underliggande data en *diskret sampling* av en kontinuerlig signal av något slag. Sampelpunkterna kallas för *hörnpunkter* (*vertices)* eller helt enkelt *punkter* (*points*). Själva sampelvärdena kallas för *attribut*. För att man i visualiseringen ska kunna hantera punkter som ligger mellan hörnpunkterna så måste man interpolera på något sätt, och för att detta i sin tur ska bli lättare så delar man in punkterna i *celler* (det finns olika celltyper, och jag kommer prata mer om dem litet längre ned). Den information som talar om *hur hörnpunkterna är sammankopplade* i en cell kallas för cellens *topologi*. Denna förändras inte av transformationer som rotation, translation och icke-uniform skalning. Det gör däremot cellens *geometri*. Med geometri menar man hörnpunkternas koordinater. Alla hörnpunkter och celler bildar tillsammans ett *rutsystem* (*grid*). Högst upp i denna hierarki har vi *datamängden* som består av rutsystemet och alla attribut.

En viktig sak att observera här är att man är noga med att skilja på geometri och topologi för en cell. Detta gör man för att förenkla arbetet med cellerna och för att man inte ska behöva lagra lika mycket information för att beskriva datamängden. En punkt kan ju ingå i flera celler, och då är det bättre att ha en separat lista över punkter med dess koordinater och en annan lista med information om hur punkterna sitter ihop – annars skulle man bli tvungen att upprepa samma koordinater flera gånger.

OLIKA TYPER AV DIMENSIONALITET

I en datamängd finns det både *beroende-* och *oberoende* variabler. Hörnpunkter och celler är oberoende, medan attribut är beroende (eftersom de beror av hörnpunkterna). Man har sedan olika benämningar för dimensionalitet beroende på vilken sorts variabel man talar om. När man talar om något som är *flerdimensionellt* (*multidimensional*) syftar man på oberoende variabler. När man å andra sidan talar om något som är *multivariat* (*multivariate*) syftar man på beroende variabler.

KONVEXITET HOS CELLER

En cell sägs vara *konvex* om den innehåller alla räta linjer som man kan rita mellan dess hörnpunkter – annars är den *icke-konvex*. Oftast vill man bara ha konvexa celler eftersom rastrering på hårdvara och mjukvara går smidigare, och många numeriska metoder kräver att cellerna är konvexa.

DEGENERERADE CELLER

En cell sägs vara *degenererad* (*degenerate*) om den nästan är identisk med någon cell av lägre dimensionalitet. Som ett exempel kan man säga att en triangel (tvådimensionell) där alla hörnpunkter i princip ligger på en och samma räta linje är degenererad, eftersom den liknar en linje (endimensionell). Många renderingsalgoritmer och numeriska metoder producerar dåliga resultat om cellerna är degenererade.

Anders Fjeldstad
andfj645@student.liu.se

REGELBUNDNA DATAMÄNGDER

Om en datamängds alla hörnpunkter ligger på ett regelbundet rutnät säger man att datamängden är just *regelbunden* (*regular*). Fördelen med sådan data är att den har en så klar struktur att man inte behöver lagra information om hur hörnpunkterna är kopplade till varandra. Dessutom behöver man inte lagra alla hörnpunkters koordianter explicit – det räcker om vi vet vilken dimension datamängden har, var dess origo ligger och hur långt det är mellan två närliggande hörnpunkter. Det är även lätt att interpolera mellan hörnpunkter för att få fram mellanliggande attribut.

OREGELBUNDNA DATAMÄNGDER

Om datamängdens hörnpunkter ligger spridda på ett ostrukturerat sätt kallas datamängden *oregelbunden* (*irregular*). Detta är mer svårhanterligt eftersom man måste specificera cellernas geometri och topologi explicit, och interpolation blir klurigare. Däremot kan det vara väldigt bra om man har en underliggande data med stora områden där en intressant kvantitet varierar långsamt och mindre isolerade områden som har stora variationer som man vill återge. I det fallet kan man placera många hörnpunkter där man behöver detaljrikedom och färre där det inte finns något intressant.

CELLTYPER

En cell kan faktiskt bestå av en enda hörnpunkt, och i det fallet så kallas cellens typ för just ***hörnpunkt*** (vertex). Vidare kan den utgöras av flera hörnpunkter utan någon synlig koppling, och då har den istället typen ***polyhörnpunkt*** (polyvertex). En annan celltyp är ***linjen*** (line) som består av två hörnpunkter sammanbundna med en rät linje. En utökning av denna är ***polylinjen*** (polyline) som består av flera hörnpunkter på en rad, sammanbundna med räta linjer.

Nu kommer den första tvådimensionella celltypen, ***triangeln*** (*triangle*) som består av tre punkter sammanbundna med räta linjer. Flera sådana kan sättas ihop till en ***triangelremsa*** (*triangle strip*) som är ytterligare en celltyp. Nästa steg är fyra hörnpunkter sammanbundna med varandra med räta linjer så att en ***fyrhörning*** (*quadrilateral*) bildas. En ***pixel*** är ett specialfall av denna, där man kräver att kantlinjerna ska vara parallella med koordinataxlarna. En ***flerhörning*** (*polygon*) är nästa steg, där man bildar en sluten cell genom att förbinda ett godtyckligt antal hörnpunkter med varandra.

Sedan kommer de tredimensionella celltyperna, och den enklaste av dem är en ***tetraeder*** (*tetrahedron*) som består av fyra sammanbundna trianglar, som en pyramid med triangulär bas. Utökar vi denna med ytterligare fyra hörnpunkter får vi en ***hexaeder*** (*hexahedron*) som är ett åttahörnigt block. ***Voxeln*** (*voxel*) är en sorts hexaeder med kravet att alla dess sidor ska vara parallella med antingen x-, y- eller z-planet.

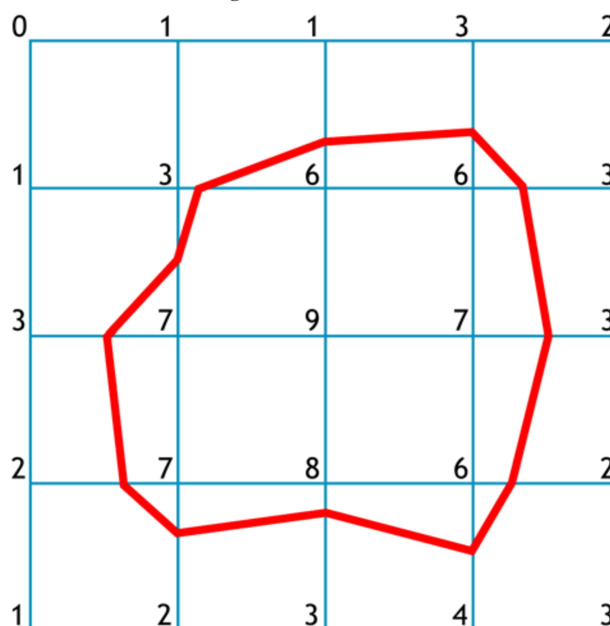
Anders Fjeldstad
andfj645@student.liu.se

KONTUREXTRAHERING (CONTOURING)

Det är ofta intressant att extrahera konturer vid vetenskaplig visualisering. Det kan exempelvis röra sig om att rendera ytan eller insidan på ett scannat människohuvud. Grundproblemet ligger i att mappa datamängdens celler och attribut till geometri som grafikhårdvaran kan visa upp på en datorskärm. Rent strikt så innebär konturextrahering (som heter *contouring* på engelska) att man definierar *gränssnitt* mellan olika områden i datamängden.

MANUELL KONTUREXTRAHERING

För att visa principen med konturextrahering tänkte jag ta ett exempel där man ritar en kontur i 2D helt manuellt. Antag att vi har ett regelbundet rutsystem med heltalsattribut i hörnpunkterna. Uppgiften är att rita en kontur (kurva) i rutnätet som motsvarar ett fixt värde, låt oss säga 5. Det vi gör då är att vi söker upp alla kantlinjer där hörnpunkterna i respektive ände har värden som ligger över respektive under 5 (eller hörnpunkter som exakt har värdet 5). Vi markerar var på kantlinjen som vi anser att 5 ligger. När vi gjort detta för alla berörda kantlinjer kan vi dra streck mellan våra markeringar. Resultatet blir en kontur. I figuren här intill visas exemplet, som är hämtat från Ken Museths föreläsningsmaterial.



I exemplet gjorde vi en förenkling som kommer visa sig spela roll längre fram. Vi antog att en rät linje kunde approximera den kurva inom varje cell som representerar ett konstant värde (i detta fall 5). Detta är inte generellt sant.

I tekniska termer så använde vi linjär interpolation mellan hörnpunkterna för att hitta skärningspunkten mellan en kantlinje och nivåkurvan, men sedan utökade vi *inte* detta till bilinjär interpolation för att hitta resten av kurvans punkter i cellen, utan vi drog helt enkelt ett rakt streck mellan skärningspunkterna.

AUTOMATISK KONTUREXTRAHERING I 2D: MARCHING SQUARES

Man vill förstås inte sitta och rita konturer för hand – det tar för lång tid och är allmänt tråkigt. Istället används automatiska metoder. En mycket välkänd sådan för 2D-datamängder är *Marching squares* (även känd som *Marching pixels*). Denna kan beskrivas som en algoritm i tre steg:

1. Gå igenom alla pixlar och markera vilka kantlinjer som skärs av den sökta nivåkurvan.

2. Beräkna skärningspunkterna genom linjär interpolation.

3. Koppla samman skärningspunkterna genom att titta i en tabell där alla möjliga kombinationer av kurvsegment finns listade. Tabellslagningen görs genom att man låter varje ”status” för varje kantlinje (alltså huruvida kanten skärs av kurvan eller ej) utgöra bitarna i ett fyrabitarstal, som i sin tur fungerar som index i tabellen. Sålunda finns det 16 möjliga fall om vi räknar med ”helt innesluten av kurvan” och ”helt utanför kurvan”.

Anders Fjeldstad
andfj645@student.liu.se

UTÖKNING TILL 3D: CONTOUR STITCHING

Okej, nu har vi alltså en metod för att hitta konturer i 2D som fungerar automatiskt. Denna skulle vi kunna använda på alla tvärsnitt från en CT-scanning, men sedan vill vi ju gå över till 3D för att kunna rita upp fina nivåytor och liknande...hur gör vi det? En metod som försöker genomföra denna övergång är Contour Stitching. Den går ut på att man helt enkelt förbinder konturerna i närliggande tvärsnitt med varandra, vilket resulterar i en nivåyta. Det finns dock ett grundläggande problem som gör att denna metod inte är praktiskt användbar – vad händer om det i det ena tvärsnittet finns en konturkurva, och i det andra finns två? Hur ska man sätta ihop dessa? Tyvärr är det bara att acceptera faktum – man har inte tillräckligt med information för att lösa det problemet. Alltså lämnar vi Contour Stitching därhän.

AUTOMATISK KONTUREXTRAHERING I 3D: MARCHING CUBES

Det finns en mycket välanvänd automatisk metod för konturextrahering i 3D som fungerar (med vissa tillägg – mer om det nedan). Den heter *Marching cubes* och är, precis som det låter, en utökning av Marching squares. Algoritmen ser ut såhär:

1. Gå igenom hörnpunkterna för alla voxlar i datamängden och klassificera dem som antingen innanför eller utanför den sökta nivåytan.

2. Skapa ett åttabitars index för voxeln baserat på hörnpunkternas klassificering (på samma sätt som i Marching squares).

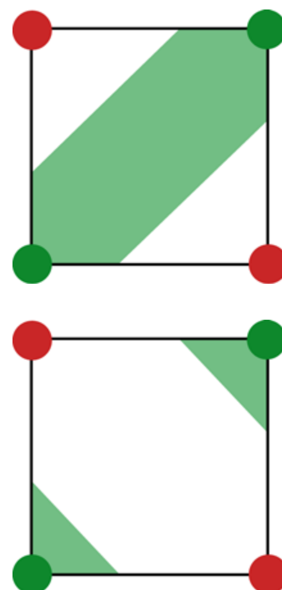
3. Hämta en kantlista från en tabell genom att slå upp det index som beräknats för voxeln.

4. Beräkna hörnpunkter för varje kantlinje i listan genom att linjärinterpolera mellan voxelns hörnpunkter för den aktuella voxelkanten.

5. Beräkna normalvektorer för de uträknade hörnpunkterna. En normal för en voxelhörnpunkt fås genom att ta den normaliserade gradienten i punkten. Normalen för nivåytehörnpunkten fås sedan genom linjär interpolation mellan närliggande voxelhörnpunkters normaler.

Problem med tvetydiga fall

Det finns flera problem med originalversionen av Marching cubes. Ett allvarligt sådant är att det finns fall där olika kantlistor passar, och om man inte ser upp så kan det därför skapas hål i den resulterande ytan. Ett exempel i 2D visas i figuren här till höger.



Hur ska man komma tillrätta med det här? Det finns flera lösningsförslag. Ett är att man ***modifierar tabellen*** och lägger till nya kantlistor som fungerar i de tvetydiga fallen innan man sätter igång. På det sättet slipper man hål, fast man kan inte vara säker på att den yta man får verkligen ser ut som den ska göra överallt.

Anders Fjeldstad
andfj645@student.liu.se

Ett annat sätt är att dela in kuberna i tetraedra, och då kallas algoritmen för ***Marching tetrahedra*** eller ***Marching simplexes***. Man slipper problemen med hål, men å andra sidan finns det flera sätt att dela in en kub i tetraedra, vilket kräver att man håller tungan rätt i mun och gör likadant överallt. Dessutom får den resulterande ytan många fler trianglar, vilket gör att den blir jobbigare att rendera. Det bästa sättet att lösa problemet med de tvetydiga fallen är dock att använda en teknik som heter *asymptotic decider,* som jag beskriver mer ingående nedan.

Asymptotic decider

Asymptotic decider är en matematisk metod som i varje tveksamt fall talar om vilken av kantlistorna som ska användas. Den bygger på att man tänker sig att värdet över voxelsidan varierar bilinjärt. Vi gjorde ju tidigare förenklingen att använda räta linjer för att förbinda de punkter där nivåytan skär voxelkanterna, men egentligen ska de vara *hyperblar*. Man har kommit fram till att dessa går att utnyttja för att avgöra vilken kantlista man ska använda i ett tveksamt fall.

Principen är att man räknar ut var hyperblarnas asymptoter skär varandra, och tar fram det interpolerade värdet i den punkten. Om värdet är större än nivåytans värde väljer man den ena kantlistan, annars väljer man den andra.

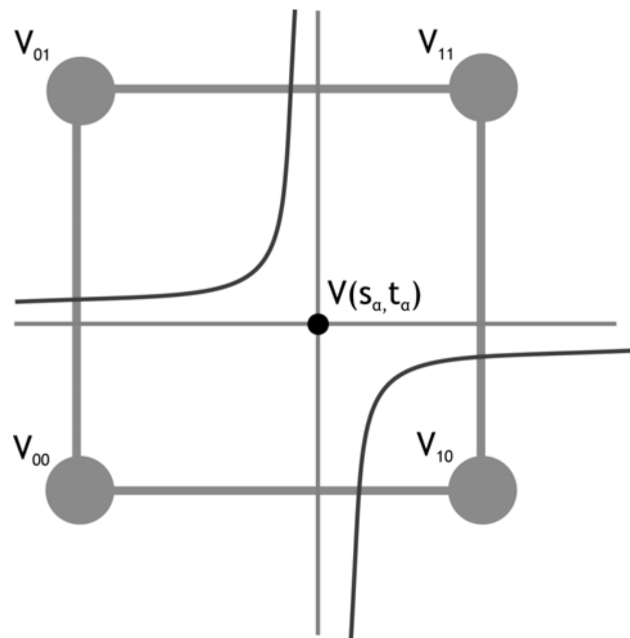
Positionen för asymptoternas skärningspunkt (s_α, t_α) kan man komma fram till genom att hitta gradientens nollställen för voxelsidans ytvärde, vilken man i sin tur får från formeln för bilinjär interpolation:

$$V(s,t)=\begin{bmatrix}1-s, s\end{bmatrix}\begin{bmatrix}V_{00} & V_{01}\\ V_{10} & V_{11}\end{bmatrix}\begin{bmatrix}1-t\\ t\end{bmatrix}$$

$$\frac{\partial V}{\partial s}=0 \Rightarrow t_\alpha=\frac{V_{00}-V_{10}}{V_{00}+V_{11}-V_{01}-V_{10}}$$

$$\frac{\partial V}{\partial t}=0 \Rightarrow s_\alpha=\frac{V_{00}-V_{01}}{V_{00}+V_{11}-V_{01}-V_{10}}$$

$$V(s_\alpha,t_\alpha)=\frac{V_{00}V_{11}-V_{01}V_{10}}{V_{00}+V_{11}-V_{01}-V_{10}}$$



Acceleration för ökad effektivitet

Ofta är det bara en relativt liten andel av volymens voxlar som skärs av en viss nivåyta. Marching cubes går ju igenom samtliga voxlar, vilket innebär att den kommer att titta på en massa voxlar som är helt ”tomma” och som inte bidrar till det visuella resultatet på något sätt – men de tar ändå upp en massa tid under beräkningarna. Man vill alltså hitta på någon smart teknik för att slippa undersöka en massa voxlar som är ointressanta.

En sådan är ***span space search*** där man låter varje voxel representeras av sitt största och minsta värde. När man sedan vill extrahera en nivåyta med ett visst värde räcker det att titta på de voxlar som har ett intervall som spänner över det sökta värdet. Sökningen accelereras ytterligare genom att man delar upp voxlarna i en trädstruktur av KD-typ, där man kan förkasta en hel gren om inte föräldernoden har ett intervall som passar in på det sökta värdet.

Anders Fjeldstad
andfj645@student.liu.se

En annan metod är att använda ***octrees***, där man bygger upp en trädstruktur med voxlar som löv och allt större grupper av voxlar (fyra och fyra, 16 och 16 och så vidare) som noder högre upp i trädet. I varje nod lagrar man minimum och maximum för alla underliggande grenar, så att man snabbt kan förkasta ointressanta voxlar. En nackdel med denna typ av acceleration är förstås att det innebär ett visst arbete att bygga upp själva trädstrukturen, men det kan vara nödvändigt för att man ska kunna rendera tillräckligt snabbt för att uppnå interaktivitet – särskilt i de fall där man vill kunna låta betraktaren dynamiskt styra nivåytans värde.

Ytterligare ett sätt att slippa gå igenom samtliga voxlar i datamängden är att använda *konturpropagering* (*contour propagation*). Man startar i en voxel som man vet skärs av den sökta nivåytan och fortsätter åt de håll som man ser att ytan gör. Det finns dock ett stort problem: hur ska man veta vilken punkt man ska starta i? Man vill slippa söka igenom datamängden voxel för voxel – det är ju det som hela idén går ut på. En lösning är att använda sig av så kallade *Extrema graphs*.

Extrema graphs

Extrema graphs baseras på ett enkelt samband mellan nivåytor och extrempunkter i datamängden. Om man förbinder alla extrempunkter med linjer så vet man att *godtycklig sluten nivåyta* (eller nivåkurva i 2D) kommer att skära någon av linjerna. Detta gör att man kan hitta en startpunkt för konturpropagering genom att följa linjerna mellan extrempunkterna.

Ett problem är dock att *öppna* nivåytor – alltså sådana som delvis ligger utanför datamängden – inte garanterat kommer att hittas. Det kan man dock komma runt genom att utnyttja faktumet att dessa garanterat skär datamängdens kanter. Den fullständiga algoritmen för Extrema graphs ser då ut såhär:

1. Hitta de linjer mellan extrempunkterna som har ett intervall som innehåller det givna värdet för nivåytan.

2. Stega längs dessa utvalda linjer och hitta startpunkter för propageringen.

3. Konturpropagera från de valda startpunkterna och hitta alla slutna nivåytor.

4. Sök igenom alla kantpunkter och hitta eventuella punkter med samma värde som nivåytan.

5. Konturpropagera från dessa punkter och hitta öppna nivåytor.

VISUALISERING AV VEKTOR- OCH TENSORFÄLT

Det är inte alltid lätt att visualisera vektor- och tensorfält, men det finns en rad olika metoder som var och en har sina för- och nackdelar. I detta kapitel tänkte jag försöka sammanfatta några sådana och samtidigt förklara en del begrepp som är bra att känna till.

Först och främst är det viktigt att vi är på det klara med vad ett *fält* är för något. Enligt litteraturen så definieras det som ”en funktion som mappar punkter i rymden till något”. Detta något kan vara ett numeriskt värde, en färg, en riktning eller vektor, en kombination av alla dessa eller något annat. Ett särskilt sorts fält är *flödesfältet* som jag tänkte berätta litet mer om härnäst.

Anders Fjeldstad
andfj645@student.liu.se

FLÖDESFÄLT

Ett flödesfält är ett vektorfält som beskriver transport av någon fysikalisk kvanitet. Det kan vara luft- eller vattenströmmar, någon form av energifält, blodcirkulationen i en människokropp eller liknande. Det finns mängder av tillämpningsområden för visualisering av flödesfält och det finns en massa olika tekniker för att göra det, och det är vad det här avsnittet ska handla om.

Flöden kan klassificeras baserat på sina egenskaper. Jag kommer att använda de engelska termerna här, mest för att jag inte känner till vad alla heter på svenska.

Ett flöde kan vara *unsteady,* och med det menar man att det förändras med tiden. Om det å andra sidan är *steady* så är det oberoende av tiden. Om flödet är *incompressible* så betyder det att det inte kan påverka densiteten hos material som införs i flödet. Motsatsen är att flödet är *compressible*. Vidare kan ett flöde vara *rotational*, vilket innebär att det har rotationer (som virvelvindar, till exempel). Om det inte finns någon benägenhet till rotation kallas flödet *irrotational.*

Dessa egenskaper kan representeras med matematik. Om *divergensen* (*divergence*) för flödesfältet är noll så är det incompressible, eftersom divergensen är ett mått på hur mycket densiteten förändras i en given punkt i fältet. Divergensen beräknas som summan av de partiella derivatorna av fältet, alltså:

$$\nabla \cdot \boldsymbol{V} = \frac{\partial V_x}{\partial x} + \frac{\partial V_y}{\partial y} + \frac{\partial V_z}{\partial z}$$

Den matematiska egenskapen *rotation* beskriver naturligtvis huruvida flödesfältet innehåller rotationer – om den är noll så finns det inga. I 2D är resultatet ett skalärfält som beskriver den maximala rotationen i varje punkt. I 3D får man istället ut ett vektorfält där varje vektors längd motsvarar den maximala rotationen och dess riktning är ortogonal mot det plan i vilket rotationen sker. Beräkningen ser ut såhär:

$$\nabla \times \boldsymbol{F} = \frac{\partial V_y}{\partial x} - \frac{\partial V_x}{\partial y} \qquad (2\boldsymbol{D})$$

$$\nabla \times \boldsymbol{F} = \begin{bmatrix} \frac{\partial V_z}{\partial y} - \frac{\partial V_y}{\partial z} \\ \frac{\partial V_x}{\partial z} - \frac{\partial V_z}{\partial x} \\ \frac{\partial V_y}{\partial x} - \frac{\partial V_x}{\partial y} \end{bmatrix} \qquad (3\boldsymbol{D})$$

GEOMETRISKA IKONER

Ett enkelt och ibland kraftfullt sätt att visualisera ett vektor- eller tensorfält är genom *geometriska ikoner*. En ikon är någon slags primitiv som placeras ut i fältet och genom sin storlek, orientering och utseende förmedlar någon form av information om underliggande data. Den enklaste formen av ikon är en så kallad *hedgehog*.

Anders Fjeldstad
andfj645@student.liu.se

Hedgehogs

En hedgehog är en endimensionell ikon, oftast en linje eller en pil, som placeras ut i det fält man vill visualisera. Den är mest användbar i 2D, eftersom endimensionella objekt är svåra att riktigt få grepp om i 3D. Risken är dessutom att visualiseringen blir för rörig när många hedgehogs hamnar för nära varandra eller överlappar. Man måste vara mycket noga med fördelningen och skalningen av hedgehogs för att det ska vara en bra metod. Eftersom man hanterar endimensinella ikoner kan man inte berätta så mycket om den data som finns i bakgrunden. Fler möjligheter blir tillgängliga om man går över till *glyphs*.

Glyphs

En *glyph* är en geometrisk ikon med två eller tre dimensioner. Glyphs används på samma sätt som hedgehogs, och har ungefär samma begränsningar, även om den ökade dimensionaliteten tillåter en att visa upp litet mer komplicerade egenskaper hos datamängden. Samtidigt krävs det att man ökar avståndet mellan ikonerna jämfört med hedgehogs för att betraktaren ska kunna se detaljerna.

Ett gemensamt problem för alla metoder som bygger på geometriska ikoner är att de på grund av sin diskreta natur inte alltid är så bra på att visualisera flöden, eftersom flödets kontinuerliga karaktär bara visas i ett ändligt antal ikoner.

PARTIKELLINJER

Ikoner är som sagt inte bra på att visa kontinuerliga flöden annat än i ganska enkla visualiseringar. Ett något bättre alternativ är att använda *partikellinjer*. Det finns en rad olika varianter på sådana med ganska små skillnader mellan sig, men jag ska försöka reda ut begreppen här.

Stream lines

Om man drar en linje efter en partikel som rör sig i ett flödesfält som inte tillåts variera med tiden (antingen genom att tiden är ”frusen” eller på grund av att fältet är steady) får man vad som kallas en *stream line*. Även om dessa kan visa nyttig information i sin princip så är de svåra att verifiera på experimentell väg, eftersom det troligen är svårt att få det verkliga flödesfältet att hållas konstant över tiden om det är unsteady.

Streak lines

Streak lines kan man skapa genom att man i varje tidssteg injicerar en partikel på en fast position och sedan låter partiklarna (som man sammanbundit med linjer) röra sig med flödet. Här får alltså fältet variera med tiden.

Path lines

Om man placerar en ensam partikel i flödesfältet och drar en linje efter den när den rör sig med tiden så får man en *path line* (även kallat *particle trace*). Här kan också fältet variera med tiden.

Time lines

En *time line* får man om man på samma gång placerar in ett antal sammankopplade partiklar i fältet och låter flödet flytta dem, så att de liknar en vågfront. Fältet kan variera med tiden.

Anders Fjeldstad
andfj645@student.liu.se

Begränsningar och alternativ

Eftersom partikellinjer är endimensionella har man precis som med de geometriska ikonerna problem med att visualiseringen kan bli rörig. Dessutom är det svårt att se till exempel små virvlar och liknande eftersom man inte kan se hur en endimensinell linje vrider sig runt sin egen axel. En lösning är att låta en polygon – till exempel en triangel – följa varje partikellinje och vrida sig med vektorfältet. Denna teknik kallas för *stream polygon*. Motsvarande går att göra med triangelremsor (*stream ribbons*) och cirklar (*stream tubes*).

TEXTURBASERAD FLÖDESFÄLTSVISUALISERING

Även om partikellinjer är bättre än ikoner på att visa upp karaktären hos flödesfält så bygger teknikerna fortfarande på att man ritar ett diskret antal linjer, vilket naturligtvis medför att man kan missa viktiga variationer. Det finns *texturbaserade* tekniker som kan vara mycket kraftfulla när det gäller att visa upp hela flödesfält, förutsatt att fältet är i två dimensioner. Den mest kända kallas för ***Line integral convolution*** (LIC).

I LIC beräknar man *för varje punkt* i utbilden en faltning mellan en stream line i punkten och de värden i en texturbild som stream line:en överlappar. Värdet av faltningen (vilket förstås är en skalär) blir alltså intensitetsvärdet i utbilden för den aktuella pixeln. Texturbilden kan vara vilken bild som helst, men ofta använder man helt enkelt vitt brus.

VOLYMRENDERING

I många tillämpningar är huvudproblemet att visualisera någon form av volymetrisk data. Detta är fallet vid de flesta typer av medicinsk visualisering, till exempel när man har CT-scannat en patient och vill titta på benfrakturer eller liknande.

Visualiseringen kan i grunden gå till på två olika sätt. Antingen visualiserar man volymen direkt genom speciella volymrenderingstekniker, eller så beräknar man geometri som ett mellansteg och använder konventionella datorgrafikrenderingsmetoder för att visa upp resultatet på datorskärmen. Jag har redan talat om tekniker som hittar nivåytor och skapar geometri, så detta avsnitt ska handla om direkt volymrendering.

Volymrendering kan i sig delas upp i olika kategorier baserat på vilken approach som används. Ett par grundläggande sådana är rendering med *image-order* respektive *object-order*.

IMAGE-ORDER-RENDERING

Rendering med image-order innebär att en eller flera *rays* (*strålar* på svenska) sänds från projektionscentrum genom varje pixel i bilden man skapar. Strålarna går igenom volymen och stöter eventuellt på intressanta voxlar som sedan kan bidra till den akutella bildpunktens färg. Eftersom strålarna inte nödvändigtvis träffar rakt på voxlarnas hörnpunkter krävs någon form av interpoleringsteknik för att kunna få ut voxelvärden för godtyckliga positioner.

OBJECT-ORDER-RENDERING

När man renderar med object-order börjar man antingen längst bak eller längst fram i volymen. Man går sedan igenom alla voxlar och projicerar ner dem på bildplanet. Detta går delvis att göra med hårdvaruacceleration i vanliga grafikkort genom att man skapar ett antal plan som är parallella med bildplanet och fördelar dessa i volymen. Planen textureras sedan med volymen som 3D-textur (där man tar hänsyn till både färg och opacitet) och grafikkortets blendingteknik används för att skapa den slutgiltiga bilden.

Anders Fjeldstad
andfj645@student.liu.se

PROJEKTIONSTYPER

Det finns flera projektionstyper för rendering med image-order eller object-order. Tre vanliga sådana är m*aximalintensitetsprojektion*, *nivåyteprojektion* och *transparensprojektion*.

Maximalintensitetsprojektion (MIP)

Maximalintensitetsprojektion är en mycket enkel projektionstyp där den aktuella bildpunktens intensitet sätts till den maximala intensiteten längs strålen. Detta fungerar ganska bra för att till exempel visa blodådror och liknande, men har en stor begränsning i att ingen djupkänsla bevaras i bilden.

Nivåyteprojektion

Nivåyteprojektion bygger på en enkel men användbar princip: visa upp de voxlar som motsvarar en nivåyta av ett visst värde. Strålen som skjuts in i volymen stannar när den når en voxel med ett värde som motsvarar den yta man söker. Man kan sedan skugga ytan genom *depth-shading* (som innebär att punkter på ytan blir mörkare desto djupare in i volymen de är) eller *surface-shading* (vilket medför att man uppskattar ytnormaler i varje punkt som sedan används i en konventionell ljusberäkning, till exempel Phong).

Transparensprojektion

Den mest generella typen av projektion vid volymrendering heter transparensprojektion (även kallad *compositing*) och bygger på en modell där man tänker sig att varje punkt i volymen absorberar respektive emitterar en viss mängd ljus. Om vi låter ρ_j och ω_j vara emittansen respektive opaciteten för punkt j samt låter I_{j-1} vara den utgående intensiteten från punkt j-1 så fås den utgående intensiteten för punkt j genom:

$$I_j = \rho_j + I_{j-1}(1 - \omega_j)$$

Detta går att generalisera för att få ett uttryck som ger den totala intensiteten efter n punkter:

$$I_n = \sum_{k=0}^{n-1} \rho_k \prod_{j=k+1}^{n} (1-\omega_j) + \rho_n$$

Transparensprojektionen går att göra antingen framifrån och bakåt eller bakifrån och framåt i volymen. Det finns fördelar med båda – om man går framifrån och bakåt kan man sluta beräkningarna när opaciteten når ett visst tröskelvärde, medan om man går bakifrån och framåt så behöver man inte ackumulera $(1-\omega_k)$ eftersom man i varje steg kan räkna ut ett delresultat.

För att kunna använda transparensprojektion måste man tilldela absorbtion och emittans till varje voxel. Absorbtion brukar man beräkna som en funktion av voxelns värde och gradienten i voxeln. Emittansen får man genom en ljusberäkning, till exempel Phong, där man tar hänsyn till betraktarens position, ljuskällornas position samt volymens gradient (som ger normalen) i voxeln.

Anders Fjeldstad
andfj645@student.liu.se

METODER FÖR ACCELERATION

Det är inte sällan som de volymer man vill rendera är mycket stora – upplösningen ökar i takt med att datainsamlingstekniken blir mer avancerad. Samtidigt vill man kunna jobba med visualiseringen på ett interaktivt sätt, och helst utan att behöva använda en superdator. Om man använder de renderingstekniker som beskrivits ovan direkt kommer man att märka att de inte är tillräckligt snabba för att uppfylla dessa förutsättningar. Därför behövs accelerationsmetoder. Här nedan tar jag upp några sådana.

Adaptiv ray-casting

Ett sätt att minska ned antalet beräkningar är att inte skjuta en stråle genom varje pixel i bilden, utan istället till exempel varannan eller liknande. För att få värden i mellanliggande pixlar använder man interpolation. När man de strålar man skickar ut stöter på områden i volymen där variansen verkar vara stor, kan man skjuta ut fler strålar just i närheten för att bättre kunna få med detaljer i datamängden. Detta kallas för *adaptiv ray-casting*.

Early ray termination

Early ray termination innebär precis som det låter att man ”avslutar” varje stråle tidigare än vad man gjort annars. Detta kan användas vid transparensprojektion när man går framfrån och bakåt i volymen. Man slutar helt enkelt att bygga på intensitetsvärdet när den ackumulerade opaciteten närmar sig ett visst tröskelvärde, till exempel 1.

Space leaping

Space leaping är namnet på en familj av accelerationstekniker som alla bygger på att man inte samplar sina strålar i de delar av volymen som är tomma. Detta kräver förstås att man vet var de intressanta objekten befinner sig, vilket i sin tur bara kan göras med genom någon form av förberäkning. Ett sätt är att man innesluter objekten i konvexa polyedra och använder två djupbuffertar för att hålla reda på ungefär var objekten börjar och slutar i förhållande till betraktaren. Ett annat är att man i varje tom voxel skriver in avståndet till närmaste objektvoxel, så att en stråle som stöter på den tomma voxeln direkt kan hoppa till objektets yta. Ytterligare en variant på space leaping är att man samplar strålen med jämna (men relativt stora) avstånd, och gör finare sampling bara mellan de sampel vars värden skiljer sig mycket.

Interaktiv rotation med låg upplösning

En vanlig teknik är att man renderar en bild med låg upplösning medan betraktaren roterar objektet, och gör fullständiga beräkningar först när rotationen upphört. På det sättet får användaren en känsla för hur objektet ser ut även under interaktionen, medan detaljer kan studeras noga när objektet är stationärt.

Anders Fjeldstad
andfj645@student.liu.se

SEGMENTERING

Jag beskrev tidigare konturextrahering och kom då in på Marching cubes och liknande. Nu ska vi titta på *segmentering*, som handlar om att definiera gränssnitt mellan olika delar av datamängden. Konturextrahering är en sorts segmentering, men ofta segmenterar man datamängden baserat på mer information än bara intensitetsvärden, till exempel kanter, kurvatur och liknande. Faktum är att man skiljer på segmentering där man tar hänsyn till kanter och segmentering där man inte gör det. I det första fallet delar man upp volymen baserat på just var det finns kanter, och i det andra fallet tar man fram områden där intensiteterna varierar mjukt.

MANUELL SEGMENTERING

En enkel segmenteringsteknik är att låta en expert manuellt markera vilka delar av volymen som är intressanta. Resultatet blir ofta mycket bra, men det kan ta lång tid och dessutom är det inte alltid som man har tillgång till en expert som kan genomföra jobbet. Alltså vill vi hellre ha automatiska metoder för segmentering.

TRÖSKELKLASSIFICERING

I *tröskelklassificering* (*threshold classification*) specificerar man helt enkelt ett tröskelvärde med avseende på cellintensitet eller derivata, och sedan renderas alla celler som stämmer in på värdet. Eftersom metoden är binär i den mening att en cell antingen tillhör den utvalda mängden eller inte så kommer det inte finnas några mjuka övergångar, och man får därför *aliasingeffekter* i den renderade bilden, vilket man inte vill ha.

REGION GROWING

Region growing fungerar som tröskelklassificering förutom att man även specificerar en startpunkt, och från den punkten så växer det valda området hela tiden åt de håll där cellerna motsvarar det uppsatta kriteriet. På det sättet kan man noggrannare välja ut enstaka objekt, men man har fortfarande samma problem med aliasing som tidigare.

LIVE WIRE

Live wire är en halvautomatisk teknik där användaren specificerar ett antal fasta punkter på den kontur eller yta som man vill få fram. Sedan räknar programmet automatiskt ut hur dessa punkter kan bindas samman så att en energifunktion minimeras. Typiskt betyder detta att konturen kommer att följa skarpa kanter.

SNAKES ELLER AKTIVA KONTURER

En helautomatisk metod är *snakes* eller *aktiva konturer* (*active contours*). Här utgår man från en parametrisk kurva som sedan deformeras för att minska energin enligt en funktion. Denna funktion är utformad så att den deformerade kurvan kommer att följa kanter samtidigt som den bervarar sin mjukhet. Snakes fungerar normalt bara i 2D.

LEVEL SETS

En vidareutveckling av snakes är *level sets*. Jag ska försöka förklara vad level sets går ut på, trots att jag själv inte är helt säker på att jag fattat det helt och hållet. Här måste jag alltså reservera mig för eventuella felaktigheter.

Ett level set är ett sätt att beskriva en deformerande yta som en nivåyta till någon underliggande funktion. I praktiken så brukar man definiera ett *zero level set* som en nivåyta som sammanfaller med ytan på det objekt i volymen (datamängden) som man vill rendera. Man definierar sedan en tredimensinell *avståndsfunktion* som i varje punkt anger avståndet till den närmaste punkten på zero level set. Denna avståndsfunktion kallar man ϕ. Nästa steg är att definiera en *hastighetsfunktion* som i varje punkt definierar hur snabbt ens level set rör sig i *normalriktningen*, där

Anders Fjeldstad
andfj645@student.liu.se

normalen i varje punkt pekar utåt i förhållande till level set:et. Hastighetsfunktionen kan definieras så att level set:et hela tiden har mjuk kurvatur och följer kanter i avståndsfunktionen. Man brukar säga att det finns *externa krafter* i hastighetsfunktionen som tar hänsyn till egenskaper hos datamängden (som till exempel kanter) och *interna krafter* som ser till att den resulterande ytan har en bra kurvatur lokalt sett.

När man använder level sets för att visualisera ett objekt i en volym skapar man ett ”start-level set” som man utgår ifrån. Detta kan till exempel ligga på volymens kanter. Man låter sedan detta level set deformeras med tiden enligt hastighetsfunktionen tills man når det intressanta objektets yta.

Level set-metoden har flera fördelar jämfört med att direkt försöka få ut geometri från volymen. Dels så vet man att ytan kommer att vara ”fysiskt realiserbar” i och med att den inte kan skära sig själv (detta följer av level set:ets matematiska natur) och dels så är det inga problem att avbilda ytor som till exempel delar upp sig i flera objekt och liknande, vilket gör att det är möjligt att genomföra avancerade modelleringar med hjälp av level sets. Ett level set har heller aldrig några hål.

ÖVERFÖRINGSFUNKTIONER

Ett mycket kraftfullt sätt att visa upp olika delar av en datamängd är att använda *färgkodning*. Principen är att man låter cellernas värden mappas till färger och opaciteter, baserat på någon (eller några) överföringsfunktioner. I till exempel medicinsk visualisering kan det vara nyttigt att se tydlig skillnad i färger för till exempel en tumör och frisk vävnad eller blodådror och benstruktur. Hur dessa överföringsfunktioner tas fram är ett komplicerat problem. I dagsläget får man alltid de bästa överföringsfunktionerna genom att låta en expert designa dem manuellt baserat på erfarenhet, men naturligtvis skulle man vilja ha automatiska metoder för att åstadkomma samma sak.

RENDERING AV PUNKTMOLN

Punktmoln är en typ av datamängd som uppstår om man till exempel laserscannar ett objekt eller liknande. Man får ett mycket stort antal punkter som man sedan vill visualisera. Ett sätt är att helt enkelt rita ut en punkt på datorskärmen för varje punkt i datamängden. Detta blir snabbt ganska rörigt. Man kan förbättra resultatet genom att beräkna en *normal* för varje punkt och sedan använda denna för skuggning och för att ta bort punkter som befinner sig på baksidan av objektet (back-face culling). En punkt tillsammans med sin normal kallas för *surfel* vilket är en kortform av *surface element*.

Punkternas massiva antal är dock fortfarande ett betydande problem om man vill låta betraktaren interagera med visualiseringen i realtid. Hur kan man göra renderingen snabbare utan att förstöra den visuella kvaliteten för mycket?

SPLATTING

Ett sätt att accelerera punktmolnsrenderingen är att använda *splatting*. Tekniken bygger på en trädstruktur av så kallade *bounding spheres* som hjälper till att hålla en dynamisk detaljnivå som kan styras med avseende på önskad renderingshastighet och bildkvalitet.

Man bygger upp trädstrukturen genom att först skapa en sfär runt varje punkt, där varje sfär ska ha tillräckligt stor radie för att överlappa sina grannar något. Sedan upprepar man proceduren rekursivt, fast varje gång låter man fyra sfärer ingå i den nya (större) sfären. Till slut har man en enda stor sfär som innehåller alla andra. En sfär kallas för en *nod* i trädet, och man utnyttjar relationerna mellan nodens position och radie och dess barns motsvarande egenskaper för att skapa en minneseffektiv representation av hela trädet.

Anders Fjeldstad
andfj645@student.liu.se

För att ytterligare accelerera renderingen kan man låta varje nod innehålla information om inom vilka vinkelintervall som dess barns normaler ligger, för att snabbt kunna avgöra om noden har några barn som ska synas från betraktarens position.

Renderingsalgoritmen startar med rotnoden, alltså den sfär som inkapslar alla andra sfärer, och ser ut såhär:

1. Om noden inte är synlig från den aktuella betraktningsvinkeln, skippa noden (och därmed alla dess barn).
2. Annars, om noden är ett *löv* (alltså en punkt i punktmolnet), rita det på skärmen.
3. Annars, kontrollera om nodens projicerade storlek på skärmen är mindre än ett visst tröskelvärde, rita noden på skärmen.
4. Annars, gå vidare till nodens barn.