Anders Fjeldstad
andfj645@student.liu.se

SAMMANFATTNING

AV

TNM079 MODELLERING OCH ANIMERING

INNEHÅLL

Några inledande ord....3
Geometrirepresentationer....3
Datastrukturer för polygonmeshes....4
Independent face lists....4
Indexed face lists....4
Full adjacency lists....5
Partial adjacency lists....5
Winged edge....5
Half edge....5
Meshförenkling....6
Grundläggande operationer....6
Generell algoritm....6
Felmått....6
Subdivision surfaces....7
Principen med subdivision surfaces....7
Den bakomliggande matematiken....8
Ett par subdivision-metoder....10
Loop-subdivision....10
Catmull-Clark-subdivision....10
Implicit modellering....11
Booleanska operationer på implicita funktioner....11
R-funktioner....12
Rendering av implicita ytor....12
Level sets....12
Vad är ett level set?....13
Några egenskaper hos level sets....14
Beräkning av ytnormalen....14
Closest point transform (CPT)....14
Modellering med level sets....15
Animering av vätskor och rök....16
Lösningsstrategi....17
Steg 1: Addera externa krafter....17
Steg 2: Self-advection....18
Steg 3: Diffusion....18
Steg 4: Projektion....19
Förbättringar till Stams lösningsmetod....19
Vorticity confinement....19
Monotonisk kubisk interpolation....19
Allmänt om skillnader mellan simulering av rök och vätska....20
En liten avslutning....20

NÅGRA INLEDANDE ORD

Hej! Vad roligt att du är här. Texten du ska till att läsa är mitt försök till sammanfattning av teorin i kursen TNM079 Modellering och animering av årgång 2006 som ges av Ken Museth, ITN, Linköpings tekniska högskola. Sammanfattningen är på intet sätt ett heltäckande dokument utan är tänkt att ge dig en snabb inblick i ämnet utan att vara vare sig för djup eller för grund.

En egenskap hos den här texten som du säkert både kommer att lägga märke till och irritera dig på är en viss blandning av engelska och svenska termer, egenhändigt tillyxade översättningar och liknande. Tror du att du ser en ond särskrivning – titta igen; det är mest troligt ett engelskt uttryck helt brutalt inklämt i ett annars svenskt sammanhang.

Med detta sagt hoppas jag ändå att du har en liten stunds trevlig läsning framför dig, kanske med en och annan aha-upplevelse under färden!

GEOMETRIREPRESENTATIONER

Geometriska ytor kan representeras på flera olika sätt när vi talar om datorgrafik. *Polygonbaserade* modeller är i många sammanhang den vanligaste varianten, och bygger på att man kopplar samman ett antal hörnpunkter så att en *mesh* (ett polygonnät) bildas. Den vanligaste polygontypen är triangeln eftersom en triangels tre hörnpunkter garanterat ligger i samma plan, vilket underlättar vid rendering av modellen. Anledningen till att polygonbaserade representationer är så populära är att dagens grafikkort är utvecklade speciellt för att kunna rastrera trianglar mycket snabbt. Det är alltså en enkel och ofta effektiv representation, men den har även nackdelar. Om man till exempel vill ändra en polygonmodells upplösning eller morfa ihop två objekt så får man problem. En annan dumhet är att en polygonmodells yta kan skära sig själv, vilket gör att det inte är säkert att modellen är fysiskt realiserbar – den representerar alltså kanske inte längre något verkligt objekt.

Matematiska funktioner kan användas för att beskriva geometriska objekt. Ett exempel som smidigt kan representeras av funktioner är höjdkartor, där höjden över havet är beroende av var på kartan man är, alltså en funktion av (minst) två variabler. Denna form är bra eftersom det är lätt att till exempel beräkna ytnormaler – man använder helt enkelt gradienten till funktionen.

Vidare finns det *parametriska ytor*, där var och en av de spatiella koordinaterna (x, y, z) bestäms av parametriseringar (funktioner) som i sin tur beror av två variabler. Genom att variera variablernas värden får man olika punkter på ytan som resultat.

Iso-ytor till matematiska funktioner kan också ses som representationer för ytan av ett objekt. Ett speciellt exempel är *level sets*, där funktionen i fråga är en *avståndsfunktion* som i varje punkt av definitionsmängden talar om hur stor avståndet till ytan är. Mer om level sets senare.

En särskild sorts geometrirepresentation är *surfels*, vilket betyder ”punkter med normaler”, ungefär. Eftersom en uppsättning surfels inte är ett sammanhängande objekt är det kanske tveksamt om man kan kalla dem för en ytrepresentation, och jag kommer inte ge dessa någon större plats eftersom de inte tas upp så mycket i kursen.

Dessa olika ytrepresentationer kan delas in i två kategorier: *explicita* och *implicita*. En explicit representation talar i klartext om precis var ytan finns, vilket alltså är fallet med polygonmodeller, höjdkartor (funktioner) och parametriska ytor. Iso-ytor är implicita eftersom de definierar ytan på ett omvänt sätt; ytan går genom alla punkter (x, y, z) som gör att funktionen i fråga får ett bestämt (iso-)värde. Surfelsrepresentationen räknas varken som explicit eller implicit eftersom den motsvarar en sampling av ytan utan någon topologisk information inkluderad.

Man kan dessutom dela in ytrepresentationerna i *Lagrange-representationer* och *Eulerska representationer*. Lagrange-representationerna är centrerade kring själva ytan, så att ytan i förhållande till sig själv är exakt representerad medan den kan ligga var som helst i rymden. I de Eulerska representationerna är det rymden som är fix och ytan definieras utifrån detta koordinatsystem. Detta medför att iso-ytor och höjdkartor räknas som Eulerska representationer medan övriga är Lagrange-representationer.

DATASTRUKTURER FÖR POLYGONMESHES

En polygonmesh består per definition av polygoner, där varje polygon har kanter och hörnpunkter. Man vill kunna använda meshen till en mängd olika saker, allt från rendering till förenkling. För att detta ska gå snabbt krävs en datastruktur som erbjuder effektiv sökning samtidigt som den inte tar onödigt mycket plats i minnet. Det finns flera vedertagna datastrukturer och som man kan tänka sig så blir de allt mer avancerade och komplexa ju bättre de uppfyller dessa grundläggande kriterier. Här nedan beskrivs några olika varianter. För att inte förvirra använder jag dess engelska namn.

Independent face lists

Independent face lists är precis vad det låter som – listor av polygoner utan någon information om eventuella beroenden sinsemellan. Detta är den absolut enklaste datastrukturen för representation av polygonmeshes. Rent konkret består den av en lista som innehåller koordinater för hörnpunkter. Om polygonerna är trianglar så antar man helt enkelt att de tre första hörnpunkterna i listan definierar den första triangeln, de tre följande definierar den andra och så vidare. Enkelheten i datastrukturen gör det mycket svårt att ta reda på egenskaper hos ytan som till exempel vilka trianglar som ligger bredvid varandra. Vidare används onödigt mycket minne eftersom koordinaterna till hörnpunkter som delas mellan flera polygoner lagras flera gånger.

Indexed face lists

Indexed face lists är en utökning av independent face lists där man lagrar hörnpunkternas koordinater i en egen lista och sedan låter polygonlistan referera till hörnpunkterna genom deras index i listan. På det sättet sparar man en massa minne eftersom man inte upprepar koordinaterna för en och samma punkt flera gånger. Fortfarande finns det dock inget stöd för att ta reda på närliggande trianglar, kanter och liknande. Tack vare sin enkelhet är denna datastruktur väldigt populär.

Full adjacency lists

En till synes komplett lösning (när den används i kombination med indexed face lists) är *full adjacency lists*. Denna struktur innehåller tre omfattande tabeller: en för kanter, en för polygoner och en för hörnpunkter. I varje tabell finns information om närliggande kanter, polygoner och hörnpunkter till den aktuella primitiven (som är en kant om man tittar i kanttabellen och så vidare). Genom att använda denna metod kan man alltså få tillgång till precis all information om närliggande primitiver. Nackdelen är förstås att den tar upp en massa onödig plats i minnet eftersom informationen överlappar. Detta utnyttjas i nästa datastruktur som jag tar upp.

Partial adjacency lists

Partial adjacency lists innehåller precis som full adjacency lists all information man kan tänkas vilja ha när det gäller närliggande primitiver till kanter, polygoner och hörnpunkter. Skillnaden är att minnet utnyttjas effektivare genom att man bara lagrar viss närliggande-information för varje primitiv, och sedan kombinerar man information från flera primitiver för att lista ut hur helheten ser ut.

Winged edge

En annorlunda idé är att utgå från polygonernas *kanter* och associera all information till dessa. Detta är grunden för *winged edge*-representationen. För varje kant sparas index till två hörnpunkter, två polygoner och fyra andra kanter (höger respektive vänster kant bakåt och framåt relativt aktuell kant). Detta innebär att man kan traversera kanter av polygoner och själva polygonerna på ett smidigt sätt.

Half edge

Half edge är en variant av winged edge där man ser varje kant som två ”halvkanter” med riktning (en halvkant åt vardera håll för en given kant). För varje halvkant sparas referenser till nästa respektive förra halvkant, motstående halvkant (den som har motsatt riktning), starthörnpunkt och den polygon som halvkanten hjälper till att definiera genom moturs listning av hörnpunkter. Genom att använda half edge kan man på ett generellt sätt representera ytor med ytterkanter (vars halvkanter längs ytterkanten inte har några motstående halvkanter).

MESHFÖRENKLING

En yta som representeras av en polygonmesh har en detaljrikedom som bestäms av meshens upplösning. Detta innebär att om en betraktare tittar på ytan på nära håll så önskar man en högupplöst mesh medan en avlägsen betraktare lika gärna kan visas en lågupplöst variant, eftersom en mesh av en given upplösning alltid tar lika lång tid att rendera hur liten den resulterande bilden än är. Alltså: man vill kunna välja upplösning på modellen beroende på avståndet till betraktaren (eller objektets storlek på datorskärmen). Samtidigt vill man naturligtvis att den lågupplösta versionen av objektet ska ha samma karaktäristiska utseende som fullstorleksversionen – det är alltså inte bara att plocka bort hörnpunkter och polygoner hur som helst. Här behövs metoder för meshförenkling (mesh decimation på engelska).

Grundläggande operationer

Översiktligt kan man säga att det finns tre grundoperationer som man kan använda för att minska antalet polygoner vid meshförenkling: *vertex clustering*, *vertex removal* och *edge collapsing*.

Vertex clustering innebär att man ersätter flera närliggande hörnpunkter med en enda. Detta kan dock medföra att objektets topologi förändras genom att tidigare fristående delar sätts ihop.

Vertex removal innebär att man tar bort utvalda hörnpunkter och alla närliggande trianglar och sedan triangulerar det resulterande hålet med ett mindre antal nya trianglar. Även om metoden ofta ger bättre resultat än vertex clustering i och med att topologin inte kan ändras så är själva trianguleringen icke-trivial och kan därför vara svår att implementera för det generella fallet.

Edge collapsing är ett specialfall av vertex clustering och går ut på att ta bort kanter genom att slå ihop en kants två hörnpunkter till en enda. Även denna metod kan ändra objektets topologi.

Generell algoritm

Meshförenkling innefattar generellt följande steg:

1. Beräkna ett ”felmått” associerat med varje möjlig grundoperation.
2. Ordna operationerna efter stigande felmått.
3. Genomför operationerna en efter en tills en given feltoleransgräns är nådd eller tills modellen består av det eftersträvade antalet trianglar. Efter varje utförd operation måste resterande operationers felmått beräknas på nytt.

Felmått

Den generella meshförenklingsalgoritmen bygger på att man för varje operation kan beräkna det associerade ”felet” och till detta behövs alltså något slags geometriskt mått – till exempel avståndet mellan den nya punkten och den föregående, avståndet mellan den nya punkten och den föregåendes trianglars plan och så vidare.

Metoden *quadric error metric* bygger på kvadraten av avståndet mellan den nya hörnpunkten (vid edge collapsing eller vertex clustering) och vart och ett av de plan som originalhörnpunktens trianglar låg i. Genom att initialt bygga upp en ”quadric” (4x4-matris) för varje hörnpunkt utifrån dess trianglars plan och sedan låta nya hörnpunkters quadrics vara summan av dess hopslagna punkters quadrics får man ett effektivt sätt att mäta det introducerade felet. För detaljerad information, läs ”Surface Simplification Using Quadric Error Metrics” av Garland och Heckbert (1997).

SUBDIVISION SURFACES

Precis som vi kan vilja få fram lågupplösta versioner av en polygonmesh så kan det vara användbart att räkna fram en modell med *högre* upplösning. Vid animering är det lättare för artisten att hantera en lågupplöst modell under arbetet, men man vill ändå att den slutgiltiga renderingen ska vara högupplöst. Här kommer *subdivision surfaces* in i bilden.

Principen med subdivision surfaces

Grundidén med subdivision surfaces är att en *mjuk* yta kan ses som en ”gränsyta” som man till slut får om man delar upp en styckvis linjär yta (polygonyta) i tillräckligt fina delar. Uppdelningen, eller förfiningen, är en iterativ process där fler trianglar läggs till i varje steg. Den nya ytan har som syfte att vara mjukare än den gamla, och man vill därför ofta att den ska ha kontinuerliga första- och andraderivator. Hur de nya hörnpunkterna placeras bestäms av kringliggande gamla hörnpunkters positioner viktat med några speciella funktioner, exempelvis så kallade *B-splines*.

Man kan säga att man ser polygonernas hörnpunkter som *kontrollpunkter* i en parametrisk yta. Genom en förfiningsregel beräknar man nya kontrollpunkter, vilket innebär att fler punkter introduceras och gamla kanske flyttas. Ju fler gånger man förfinar kontrollpunktnätet, desto mer börjar det likna den parametriska yta som det representerar. I och med denna insikt kan man rendera *kontrollpunktnätet* precis som vilken polygonmesh som helst (istället för att rendera den parametriska ytan) och genomföra ytterligare förfiningssteg om man behöver en mjukare yta.

Det finns två typer av subdivision: *interpolerande* och *approximerande*. Vid interpolerande subdivision går den parametriska ytan garanterat genom originaluppsättningen av kontrollpunkter, medan så inte behöver vara fallet i approximerande subdivision.

Den bakomliggande matematiken

För att riktigt förstå hur subdivision surfaces fungerar måste man ta en titt på teorin bakom parametriska ytor, och det blir förstås lättare om man till och med nöjer sig med kurvor, så det tänkte jag göra här. Som exempel kommer kubiska B-splines att användas, alltså B-splines av grad 3.

Ett segment i en parametrisk kubisk B-spline-kurva definieras som:

$$P(u)=\sum_{i=0}^{3} P_i B^i(u)$$

B-funktionen kan skrivas om som en translation av sig själv, såhär:

$$B^i(u)=B(u-i)$$

Genom att sätta upp ett antal kontinuitetsförutsättningar som kopplar samman *B*-funktionerna kan man lösa ut dem och komma fram till:

$$B^0(u)=\frac{1}{6}u^3$$
$$B^1(u)=-\frac{1}{2}u^3+\frac{1}{2}u^2+\frac{1}{2}u+\frac{1}{6}$$
$$B^2(u)=\frac{1}{2}u^3-u^2+\frac{2}{3}$$
$$B^3(u)=-\frac{1}{6}u^3+\frac{1}{2}u^2-\frac{1}{2}u+\frac{1}{6}$$

Det finns dock en alternativ definition av B-splines (av grad *l*) som bygger på upprepad *faltning*:

$$B_l(t)=\int B_{l-1}(s)\,B_0(t-s)\,ds \qquad B_0(t)=\begin{Bmatrix}1\,om\,0\leqslant t<1\\0\,annars\end{Bmatrix}$$

Okej, detta ser kanske inte självklart ut, men det fina är att man kan gå vidare till:

$$B_l(t)=\frac{1}{2^l}\sum_{k=0}^{l+1}\binom{l+1}{k}B_l(2t-k)$$

Uttrycket ser ännu mer avskräckande ut, men det kan hjälpa oss att förstå en mycket viktig egenskap: en *B*-funktion av en given grad (*l*) kan skrivas som en linjärkombination av *translaterade* (*k*) och *avsmalnade* (2*t*) kopior av sig själv. Och om en och samma basfunktion kan delas upp i flera så innebär det att man också kommer att behöva använda fler kontrollpunkter. Fler kontrollpunkter innebär högre upplösning i kontrollpunktnätet, vilket är precis vad subdivision går ut på.

Om vi nu tänker oss att vi har en uppsättning kontrollpunkter i vektorn ***p***:

$$\boldsymbol{p}=\begin{bmatrix}\vdots\\ p_{-2}\\ p_{-1}\\ p_0\\ p_1\\ p_2\\ \vdots\end{bmatrix}$$

Vidare har vi basfunktionerna i vektorn ***B****(t)*, och här har jag struntat i graden (eftersom den alltid är 3 vid kubiska B-splines) och indexet (eftersom vi inte vill hantera punkter i slutet av kurvan, vi koncentrerar oss på det generella fallet):

$$\boldsymbol{B}(t)=[\ldots \quad B(t+2) \quad B(t+1) \quad B(t) \quad B(t-1) \quad B(t-2) \quad \ldots]$$

Kurvan kan nu alltså skrivas som $\boldsymbol{B}(t)\,\boldsymbol{p}$. Genom att använda faktumet att man kan skriva om elementen i $\boldsymbol{B}$ som summor av avsmalnade kopior får vi:

$$\boldsymbol{B}(2t)=[\ldots \quad B(2t+2) \quad B(2t+1) \quad B(2t) \quad B(2t-1) \quad B(2t-2) \quad \ldots]$$

Relationen mellan ***B****(2t)* och ***B****(t)* kan skrivas med hjälp av en *subdivision-matris* ***S***:

$$\boldsymbol{B}(t)=\boldsymbol{B}(2t)\boldsymbol{S}$$

Elementen i $\boldsymbol{S}$ fås förstås från ”förfiningsekvationen” ovan:

$$\boldsymbol{S}_{2\mathrm{i}+k,i}=\frac{1}{2^l}\binom{l+1}{k}$$

Kurvan ges nu alltså av $\boldsymbol{B}(t)\,\boldsymbol{p}=\boldsymbol{B}(2\mathrm{t})\boldsymbol{S}\,\boldsymbol{p}$ vilket innebär att vi har fått nya kontrollpunkter som beräknas av $\boldsymbol{S}\,\boldsymbol{p}$. Naturligtvis kan man göra om denna förfiningsprocess hur många gånger man vill, till exempel *j* gånger, varpå kurvan fås av $\boldsymbol{B}(t)\,\boldsymbol{p}=\boldsymbol{B}(2^j t)\boldsymbol{S}^j\,\boldsymbol{p}$. Kontrollpunkterna efter *j* subdivision-steg fås av uttrycket $\boldsymbol{S}^j\,\boldsymbol{p}$.

Om man använder kubiska B-splines för sin subdivision så kommer kontrollpunkterna garanterat konvergera mot den parametriska ”gränsytan” som de definierar. Dessutom vet man att punkterna efter varje subdivision-steg kommer att beskriva en ”mjukare” yta än i steget innan. Vidare klarar ytan att utsättas för affina transformationer utan problem.

Alla dessa fina egenskaper är dock inte självklara för andra val av basfunktioner. Man kan undersöka huruvida kriterierna uppfylls för en given subdivision-matris genom att undersöka dess egenvärden, λ_i, och egenvektorer $\boldsymbol{x}_i$. Dessa är villkoren:

- **Konvergens:** $|\lambda_i| \leqslant 1$
- **Invarians under affina transformationer:** $\boldsymbol{x_0} = \mathbf{1}, \lambda_0 = 1, |\lambda_i| < 1 \qquad i > 0$
- **Mjukhet ("smoothness"):** $\lambda_0 = 1 > |\lambda_1| > |\lambda_i| \qquad i > 1$

Ett par subdivision-metoder

Jag kommer att gå igenom två olika metoder för subdivision av polygonytor, men först måste jag ta upp ett par definitioner som används mycket i sammanhanget:

- **Valens** är ett uttryck inom grafteori som betyder ”antalet sammanbundna punkter till en given punkt”. Detta används även i subdivision-sammanhang när man talar om hörnpunkter i polygonmeshes. Om en hörnpunkt har valens 6 så betyder det att den har sex stycken ”hörnpunktsgrannar”, alltså att hörnpunkten är kopplad till sex andra hörnpunkter genom polygonkanter.
- **Extraordinära hörnpunkter** är hörnpunkter med valens skild från 4 i quad-meshes och valens skild från 6 i triangel-meshes. Denna typ av punkter kan medföra problem vid subdivision eftersom man kan behöva vidta speciella åtgärder för att den önskade graden av kontinuitet ska bibehållas.

Loop-subdivision

Loop-subdivision är en enkel approximerande subdivision-metod som fungerar på triangel-meshes och använder ”three-directional box spline”. Den garanterar att både första- och andraderivatorna är kontinuerliga överallt utom i extraordinära hörnpunkter, där endast förstaderivatan garanteras vara kontinuerlig.

Förfiningsprocessen är relativt enkel; varje triangel delas upp i fyra nya genom att varje kant delas i två delar (tre nya hörnpunkter per triangel introduceras alltså).

Catmull-Clark-subdivision

En annan approximerande subdivision-metod är *Catmull-Clark-subdivision*, där kontrollmeshen är en quad-mesh. Denna är mer komplex än Loop och baseras på ”tensor product bicubic spline”. Metoden producerar ytor som garanterat har kontinuerliga första- och andraderivator överallt.

Förfiningen innebär att man beräknar en ny kontrollpunkt som placeras inuti en existerande quad (och därigenom delar upp den i fyra nya quads), sedan en ny punkt för varje existerande kant och sedan flyttar man de gamla kontrollpunkterna.

IMPLICIT MODELLERING

Polygonmodeller är visserligen populära i och med att de går så snabbt att rendera med dagens grafikkort, men de passar inte för alla ändamål. Om man till exempel jobbar med att ta fram CAD-modeller av maskindelar så är det viktigt att ritningarna man tar fram går att skicka till en maskin som sedan tillverkar den verkliga delen – och fysisk realiserbarhet kan inte garanteras så länge man modellerar med polygonmodeller. Här passar implicita geometrirepresentationer bättre eftersom de inte kan ha "hål", inte kan låta ytorna skära sig själva, klarar av komplexa topologiförändringar och går att kombinera genom enkla booleanska operationer.

Det finns två typer av implicita funktioner: *analytiska* och *numeriska*. En analytisk funktion är kontinuerlig och har därmed ingen upplösning (eller oändlig upplösning, hur man nu vill se det). Numeriska funktioner är diskreta samplingar av sina kontinuerliga motsvarigheter. I resten av detta avsnitt antar vi att vi har med analytiska funktioner att göra, men principerna som presenteras fungerar för det diskreta fallet också. Det antas också att funktionen i fråga har värden som är negativa inuti objektet, är noll på ytan och är positiva utanför objektet.

Booleanska operationer på implicita funktioner

Det finns en modelleringsgren inom datorgrafik som heter "Constructive Solid Geometry" (CSG) och som går ut på att man bygger upp en komplex modell utifrån kombinationer av enkla former. Själva kombineringen gör s med de booleanska operationerna *union*, *snitt* och *differens* som är lånade direkt från mängdläran – tolkningen är att man ser objekten som mängder i R^3.

Det visar sig vara mycket enkelt att implementera de booleanska operationerna för implicita funktioner. Om F_A och F_B är implicita funktioner som representerar objekten A respektive B så ges resultatet av de booleanska operationerna av:

$$F_{A \cup B} = min(F_A, F_B)$$
$$F_{A \cap B} = max(F_A, F_B)$$
$$F_{A - B} = max(F_A, -F_B)$$

En möjlig oönskad följd av booleanska operationer mellan två implicita funktioner är att *tolkningen* av den *resulterande* funktionen inte behöver vara densamma som tolkningarna av de *ingående*. Antag att de ingående funktionerna beskriver avståndet till ytan, med negativa tal inuti objektet. Om man genomför en union-operation kommer funktionen inte längre överallt att motsvara avstånd eftersom överlappande delar fortfarande har värden från originalfunktionerna. För att lösa detta måste man "reinitialisera" funktionen efter sammanslagningen, alltså formulera om den så att den återigen blir en avståndsfunktion (om det var en sådan man utgick ifrån).

Ett annat problem är *mjukheten* hos den resulterande funktionen. Vid booleanska operationer kommer skarpa kanter introduceras där originalobjekten skurit varandra på grund av operatorernas binära natur, vilket gör att resultatets förstaderivata inte kan garanteras vara kontinuerlig överallt. En lösning på detta är att approximera de booleanska operationerna med nya funktioner som har mjuka övergångar.

R-funktioner

R-funktionerna är approximationer till de booleanska operatorerna som erbjuder en mjukare övergång från positivt till negativt för de ingående funktionerna. Om man skickar in två implicita funktioner i en R-funktion får man ut en funktion vars tecken överallt bestäms av de ingående funktionernas tecken. På det sättet liknar R-funktionerna logiska switchar, vilket är precis vad de booleanska operatorerna är.

Ett problem som uppstår när man använder sådana här approximationer för att efterlikna booleanska operationer är att mjukheten som man får är *global* på det sättet att man inte kan kontrollera vilka övergångar som ska vara mjuka och vilka man vill ha skarpa. Detta kan leda till förlust av detaljer samt att objekt smälter ihop utan att man vill det.

Rendering av implicita ytor

Det finns två sätt att rendera implicita ytor:

- **Direkt metod:** man använder *ray-casting* för att beräkna skuggning av ytorna. Rays skjuts ut från projektionscentrum genom bildplanet och in i scenen. Genom att undersöka hur de studsar mot objektens ytor och genom att man känner till ytegenskaper och ljuskällor så kan man bygga upp en bild på bildplanet, ray för ray. Fördelen är att resultatet blir snyggt, med reflektioner mellan objekt och liknande om man så önskar. Nackdelen är att det går långsamt.

- **Indirekt metod:** man går över till en polygonrepresentation av funktionen, troligen genom en algoritm som Marching Cubes (vilken i sin tur kräver att man samplar den implicta ytan i ett rutnät av voxlar). När man har en polygonmesh istället för en implicit funktion kan man genomföra ljusberäkningar och annat på valfritt sätt. Nackdelen är förstås att det tar tid att skapa polygonmodellen (vilket måste göras om ifall den implicita ytan förändras). Fördelen är att renderingen går snabbt (kan utnyttja hårdvaruacceleration) när man väl har den.

En bra egenskap hos implicita ytor är att det är mycket lätt att ta fram normalen, något som behövs i alla ljusberäkningar. Den ges helt enkelt av funktionens gradient, vilken är enkel att beräkna oavsett om funktionen är analytisk eller numerisk.

LEVEL SETS

Som det förra avsnittet förklarade så finns det stora fördelar med att använda implicita ytor jämfört med explicita inom vissa typer av modellering. Där nämndes CAD-tillämpningar – ett annat exempel skulle kunna vara simulering av *fluids*, alltså flytande och flyktiga element som vätskor och rök. Så fort man har en yta vars topologi förändras på komplexa sätt blir representationer som polygonmeshes ohanterbara och mycket klumpiga. Implicita ytor fungerar bättre, och *level sets* är en särskild typ av sådan som har visat sig vara mycket användbar.

Vad är ett level set?

Ett level set är en implicit funktion som har tolkningen "en teckenförsedd avståndsfunktion" (*signed distance function*). I varje punkt i rymden ger funktionen det kortaste avståndet till *gränssnittet*, eller ytan. Dessutom innehåller definitionen av ett level set en tidsparameter, och genom att variera denna kan man förändra ytans form och topologi över tiden. Ett level set är alltså en iso-yta som kan variera med tiden. Rent matematiskt definieras ett level set såhär:

$$S(t)=\{\boldsymbol{x}(t)\in\mathbb{R}^3 : \phi(\boldsymbol{x}(t),t)=k\}$$

Här är *S(t)* iso-ytan, ***x****(t)* är punkter på ytan, *t* är tidsparametern och *k* är iso-värdet (vilket typsikt alltid är noll eftersom det är själva ytan vi är intresserade av). ϕ är den så kallade *level set-funktionen*. För att ϕ ska vara en teckenförsed avståndsfunktion krävs att följande ekvation (som kallas *Eikonal equation*) är uppfylld:

$$|\nabla\phi|=1$$

Okej, så vad händer då när tidsparametern förändras? Förändring är ju derivata, så vi deriverar level set-funktionen med avseende på tiden. Eftersom den beror av flera variabler är det partiell derivering som gäller. Genom att applicera kedjeregeln kommer man fram till:

$$\frac{d\phi}{dt}=0=\frac{\partial\phi}{\partial t}+\frac{d\boldsymbol{x}}{dt}\cdot\nabla\phi$$

Genom att flytta runt lite samt dividera och multiplicera högerledet med normen av level set-funktionens gradient kan man få fram:

$$\frac{\partial\phi}{\partial t}=-\frac{d\boldsymbol{x}}{dt}\cdot\frac{\nabla\phi}{|\nabla\phi|}|\nabla\phi|$$

Tittar vi noga på detta så inser vi att $\frac{d\boldsymbol{x}}{dt}$ motsvarar hastigheten och $\frac{\nabla\phi}{|\nabla\phi|}$ är ytnormalen.

Dessa två kan bakas ihop till den så kallade *speed-funktionen* F vilket gör att vi kan skriva uttrycket för den partiella derivatan av level set-funktionen med avseende på tiden som:

$$\frac{\partial\phi}{\partial t}=-F|\nabla\phi|$$

Speed-funktionen talar alltså om hur level set:et rör sig i ytnormalens riktning. Detta är en *skalär* formulering – det finns en vektorverision också:

$$\frac{\partial\phi}{\partial t}=-\boldsymbol{V}\cdot\nabla\phi$$

Denna kan tolkas som att level set:et "flyter med" ett vektorfält ***V***. Om jag fattat saker och ting rätt så kan denna variant användas om man till exempel vill simulera en vattenyta.

Några egenskaper hos level sets

Genom sin natur har level sets flera goda egenskaper som gör dem smidiga att arbeta med. Här presenteras ett par viktiga sådana.

Beräkning av ytnormalen

Ytnormalen hos ett level set är simpel att få fram. Som jag tidigare nämnt så ges den av uttrycket:

$$\boldsymbol{n} = \pm\frac{\nabla\phi}{|\nabla\phi|}$$

Men eftersom vi vet att ett level set är en teckenförsedd avståndsfunktion och för att garanterat vara en sådan så måste den uppfylla $|\nabla\phi| = 1$ så blir normalen helt enkelt:

$$\boldsymbol{n} = \pm\nabla\phi$$

Closest point transform (CPT)

En annan viktig operation som man ständigt behöver ha tillgänglig är den så kallade *closest point transform* (CPT) som för varje punkt i rymden talar om vilken punkt på ytan som ligger närmast – det är ju avståndet däremellan som funktionen ska ha som värde överallt. Om vi antar att vi har en (okänd) punkt $\mathbf{x}_0$ på ytan och en annan (känd) punkt $\mathbf{x}_1$ en bit utanför så inser vi att vi kan få reda på $\mathbf{x}_1$ om vi känner till $\mathbf{x}_0$ och har ytnormalen i $\mathbf{x}_0$:

$$\boldsymbol{x}_1 = \boldsymbol{x}_0 + h\,\boldsymbol{n}$$

Här är h avståndet mellan punkterna och $\boldsymbol{n}$ är ytnormalen i $\mathbf{x}_0$. Tänker vi efter lite så inser vi att detta är ekvivalent med:

$$\boldsymbol{x}_1 = \boldsymbol{x}_0 + \phi(\boldsymbol{x}_1)\nabla\phi(\boldsymbol{x}_0)$$

Detta skulle alltså kräva vi känner till ytnormalen i punkten $\mathbf{x}_0$, men det gör vi inte eftersom vi ju inte känner till själva $\mathbf{x}_0$. Dock så kan vi konstatera att eftersom gradienten (normalen) i alla punkter pekar åt det håll som funktionen växer snabbast så kommer den vara konstant längs den räta linje som förbinder $\mathbf{x}_0$ och $\mathbf{x}_1$. Detta betyder i sin tur att vi lika gärna kan beräkna gradienten i $\mathbf{x}_1$. Problemet är löst! Vi har hittat closest point transform:

$$CPT(x_1) = \boldsymbol{x}_0 = \boldsymbol{x}_1 - h\,\boldsymbol{n} = \boldsymbol{x}_1 - \phi(\boldsymbol{x}_1)\nabla\phi(\boldsymbol{x}_1)$$

Modellering med level sets

Precis som i det generella fallet av implicita ytor så går det bra att använda booleanska operationer för att modellera med level sets, och det har dessutom utvecklats tekniker där man kan kontrollera var *lokal* utjämning ska göras, så att man kan undvika att detaljer försvinner och att objekt smälter ihop (läs "Algorithms for Interactive Editing of Level Set Models" av Museth och gänget för detaljerad information).

Det är även möjligt att implementera olika *morfologiska operationer* som utökar verktygslådan ytterligare. Två mycket användbara sådana är *öppning* och *stängning*. Dessa känns igen från bildbehandling och definieras generellt såhär:

- **Öppning** är en *krympning* (erosion) följd av en *expansion* (dilation). Denna operation är bra för att ta bort små oönskade objekt och för att dela upp objekt som sitter ihop endast med ett smalt parti.
- **Stängning** är det omvända: en expansion följt av en krympning. Stängning eliminerar små hål och sprickor samt jämnar ut ytor.

Eftersom ett level set är en iso-yta kan öppning med faktor *d* åstadkommas såhär:

1. Välj level set nummer *d* inåt i objektet jämfört med original-level set:et. Detta motsvarar krympningen.
2. Utgå från det valda level set:et och reinitialisera avståndsfunktionen relativt denna.
3. Välj level set nummer *d* utåt från den nya ytan. Detta motsvarar expansionen.

Stängning är samma grej fast tvärt om. Dessa morfologiska operationer kan vara bra för att snygga till en modell som ursprungligen skapats genom exempelvis laserscanning, där ljusreflektioner i objektets yta kan introducera brus i form av smala "taggar" på objektytan.

ANIMERING AV VÄTSKOR OCH RÖK

Vätskor och rök, under samlingsnamnet *fluids*, är svåra att simulera och animera. Det finns ett par ekvationer som anses vara en bra modell för fluid-flöden, nämligen *Navier-Stokes-ekvationerna:*

$$\frac{\partial \boldsymbol{u}}{\partial t} = -(\boldsymbol{u}\cdot\nabla)\boldsymbol{u} + \nu\nabla^2\boldsymbol{u} - \frac{1}{\rho}\nabla p + \boldsymbol{f}$$
$$\nabla\cdot\boldsymbol{u} = 0$$

Här är $\boldsymbol{u}$ ett vektorfält som beskriver flödet i ett visst ögonblick, ν är viskositeten hos vätskan, ρ är densiteten, p är trycket och $\boldsymbol{f}$ är en extern kraft (till exempel gravitation eller lyftkraft). Hela den första ekvationen uttrycker bevarandet av rörelsemängd medan den andra slår fast att fältet ska bevara massa.

Låt oss titta lite närmare på den första ekvationen. Den första termen, $(\boldsymbol{u}\cdot\nabla)\boldsymbol{u}$, beskriver *self-advection* vilket betyder att flödesfältet "flyter med sig själv". Låter lite abstrakt kanske, men så är det i alla fall. Den andra termen, $\nu\nabla^2\boldsymbol{u}$, handlar om *diffusion*. Detta innebär att de olika delarna av flödesfältet utbyter krafter med varandra på något sätt. Den tredje termen, $\frac{1}{\rho}\nabla p$, motsvarar hur tryckförändringar påverkar flödesfältet. Den sista termen, $\boldsymbol{f}$, visar hur externa krafter ändrar fältet.

Dessa ekvationer kan alltså på ett realistiskt sätt modellera ett flödesfält. Nackdelen är att de numeriska metoder som kan lösa dem exakt kräver mycket små tidssteg för att vara stabila och tar därmed lång tid att beräkna. Vi befinner oss dock i datorgrafikens värld, där det är utseendet som räknas – ser det bra ut så är det bra. I och med att vi kan godta en simulering som inte är strikt fysikaliskt korrekt så går det faktiskt att lösa ekvationerna i realtid på en vanlig dator, om man är listig i implementationen av lösaren.

Och listig var just vad en man vid namn Jos Stam var när han år 1999 i sin artikel "Stable Fluids" föreslog en lösningsmetod som är stabil även för stora tidssteg. Här utnyttjas att det finns ett samband mellan hastighetsfältet och tryckfältet, och han använder en matematisk sats som heter *Helmholtz-Hodge Decomposition* som säger att ett givet vektorfält alltid kan skrivas som en summa av ett divergensfritt och ett rotationsfritt fält. Om vi definierar ett vektorfält $\boldsymbol{w}$ som:

$$\boldsymbol{w} = -(\boldsymbol{u}\cdot\nabla)\boldsymbol{u} + \nu\nabla^2\boldsymbol{u} + \boldsymbol{f}$$

Så får vi vid insättning i första ekvationen ovan:

$$\frac{\partial \boldsymbol{u}}{\partial t} = w - \frac{1}{\rho}\nabla p$$

Om vi definierar en *projektionsoperator* $\boldsymbol{P}$ som projicerar ett vektorfält på sin divergensfria del (vilket innebär att den subtraherar det rotationsfria fältet) som:

$$\boldsymbol{P}(\boldsymbol{w}) = \boldsymbol{w} - \nabla q$$

Här är q ett okänt skalärfält. Vi applicerar $\boldsymbol{P}$ på båda leden i den första Navier-Stokes-ekvationen och får:

$$\boldsymbol{P}\left(\frac{\partial \boldsymbol{u}}{\partial t}\right) = \boldsymbol{P}\left(\boldsymbol{w} - \frac{1}{\rho}\nabla p\right)$$

Vi vet att $\boldsymbol{P}(\boldsymbol{u}) = \boldsymbol{u}$ och $\boldsymbol{P}(\nabla p) = \boldsymbol{0}$ vilket innebär att den ovanstående ekvationen kan förenklas till:

$$\frac{\partial \boldsymbol{u}}{\partial t} = \boldsymbol{P}(\boldsymbol{w}) = \boldsymbol{w} - \nabla q$$

Skalärfältet q kan ses som ett icke-fysiskt tryckfält som garanterar att massa bevaras. Man kan man få reda på hur det ser ut genom att lösa *Poissonekvationen*:

$$\nabla \cdot \boldsymbol{w} = \nabla^2 q$$

Det vi kommit fram till nu är en enda ekvation som är en kombination av de ursprungliga Navier-Stokes-ekvationerna:

$$\frac{\partial \boldsymbol{u}}{\partial t} = \boldsymbol{P}\left(-(\boldsymbol{u}\cdot\nabla)\boldsymbol{u} + v\nabla^2\boldsymbol{u} + \boldsymbol{f}\right) = \boldsymbol{P}(\boldsymbol{w}) = \boldsymbol{w} - \nabla q$$

Lösningsstrategi

Stams idé var att lösa denna differentialekvation i flera steg enligt nedan:

$$\boldsymbol{u}_{old} \rightarrow \textit{externa krafter} \rightarrow \textit{advection} \rightarrow \textit{diffusion} \rightarrow \textit{projektion} \rightarrow \boldsymbol{u}_{new}$$

Här tänkte jag gå igenom varje steg för sig.

Steg 1: Addera externa krafter

Första steget är att lägga till externa krafter till det gamla vektorfältet. Med "gamla" menar man fältet från förra tidssteget. Såhär går det till:

$$\boldsymbol{w}_0(\boldsymbol{x}, t) = \boldsymbol{u}_{old}(\boldsymbol{x}, t) + \Delta t\, \boldsymbol{f}(\boldsymbol{x}, t)$$

Om man till exempel har en vind som blåser eller något objekt som rör sig i vektorfältet så är det här som dess krafter ska in, i $\boldsymbol{f}$ alltså.

Steg 2: Self-advection

Self-advection vet jag inget bra svenskt ord för, men det handlar om att vektorfältet flyttar på sig själv, det "flyter" längs sina egna krafter. För att det ska bli lättare att lösa detta steg antar man att $\boldsymbol{w}$ inte förändras under själva tidssteget, vilket naturligtvis är en förenkling.

Det hela handlar om att flytta vektorfältet längs sig själv, vilket innebär att man vill hitta nya vektorer för varje position i fältet baserat på vektorerna i det gamla. Man skulle kunna tänka sig att man tar vektorn på position *i* och tidsintegrerar framåt för att få reda på var vektorn hamnar efter ett tidssteg. I praktiken gör man dock det hela "baklänges" – man utgår från det nya vektorfältet $\boldsymbol{w_1}$ och för varje position tittar man var den positionen var för ett tidssteg sedan, man integrerar alltså bakåt. På det sättet får man garanterat en vektor per position (voxel), även om man generellt måste interpolera fram den baserat på gamla vektorer. Denna metod (att integrera baklänges) kallas för *Semi-Lagrangian-integration*. I matematiska termer ser det ut såhär:

$$\boldsymbol{w}_1(\boldsymbol{x}, t) = \boldsymbol{w}_0(\boldsymbol{p}(\boldsymbol{x}, -\Delta t))$$

Här är $\boldsymbol{p}$ en *stream-line* i vektorfältet (alltså en tidsberoende position) där $\boldsymbol{p}(\boldsymbol{x}, 0)$ motsvarar den aktuella positionen i det nya vektorfältet $\boldsymbol{w_1}$.

Steg 3: Diffusion

Det tredje steget i lösningsgången tar hand om diffusionen i vektorfältet, alltså att hastigheten på ett ställe i fältet påverkar hastigheterna runt omkring. Den ekvation man måste lösa är:

$$\frac{\partial \boldsymbol{w}_1}{\partial t} = \nu \nabla^2 \boldsymbol{w}_1$$

Man skulle kunna använda explicit Eulerintegration, men den blir instabil för stora tidssteg – och vi vill kunna använda stora tidssteg för att snabba upp beräkningarna eftersom vi hanterar realtidsgrafik. Vi kan istället använda *implicit integration* som alltid är stabil:

$$(\boldsymbol{I} - \nu \Delta t \nabla^2)\boldsymbol{w}_2 = \boldsymbol{w}_1$$

$\boldsymbol{I}$ är identitetsmatrisen. Detta ekvationssystem är "sparse" och kan lösas numeriskt på ett effektivt sätt.

Steg 4: Projektion

Till sist vill vi att det resulterande flödesfältet ska vara divergensfritt – det ska alltså bevara massa som man stoppar in i det. Som vi tidigare kommit fram till kan man åstadkomma detta genom att applicera vår projektionsoperator på det, så att den rotationsfria delen av fältet subtraheras och resultatet blir divergensfritt. För att kunna göra detta måste vi lösa Poissonekvationen för att få fram det okända skalärfältet q och sedan dra bort gradienten av q från vårt flödesfält:

$$\nabla \cdot \boldsymbol{w}_2 = \nabla^2 q$$
$$\boldsymbol{w}_3 = \boldsymbol{w}_2 - \nabla q$$

Det visar sig att även Poissonekvationen i diskret form blir ett "sparse" ekvationssystem och kan lösas effektivt. Vips så har vi fått ett divergensfritt flödesfält för det nya tidssteget och vi är klara!

Förbättringar till Stams lösningsmetod

Även om Stams metod ovan producerar riktigt fina resultat i förhållande till beräkningstiden så finns det ett par förbättringar som inte är särskilt svåra att implementera och som föreslogs av Stam själv (tillsammans med ett par andra killar) två år senare i artikeln "Visual Simulation of Smoke".

Vorticity confinement

På grund av de numeriska metoder som används i Stams fluid-lösare dör virvlar och annat i flödesfältet ut tidigare än de gör i verkligheten. En smart lösning är att "stoppa tillbaka" lite rotation här och där i fältet där det är mest troligt att det behövs. Detta kanske låter lite godtyckligt, men det har trots allt en fysisk motivering och det ger ett mer detaljerat och realistiskt vektorfält som resultat.

I praktiken börjar man med att beräkna flödesfältets rotation (vorticitet). Detta varierar förstås över fältet beroende på var eventuella virvlar befinner sig. Man vill hålla liv i virvlarna genom att lägga in extra rotation i dem, och för att hitta deras positioner beräknar man gradienten på rotationen, eftersom denna pekar mot de punkter där rotationen är som störst. Sedan räknar man ut åt vilket håll "extra-rotationen" ska snurra och lägger till den till flödesfältet. Enkelt!

Monotonisk kubisk interpolation

I Stams ursprungliga lösare använder man vanlig bilinjär (eller trilinjär om man har 3D-fluids) interpolation i self-advection-steget när man utför Semi-Lagrangian-integreringen för att få reda på de nya krafterna i fältet. För att förbättra den visuella kvaliteten är det möjligt att istället använda en kubisk interpolationsmetod. Problemet med detta är att kubisk interpolation generellt kan innebära "överslag" vilket i sin tur kan göra lösaren instabil. På grund av detta tog Stam och gänget fram en "monotonisk" kubisk interpolation som inte har några överslag. Den tar visserligen längre tid att beräkna än vanlig linjär interpolation men resultatet blir mycket snyggare.

Allmänt om skillnader mellan simulering av rök och vätska

Stams fluid-lösare är i sin grundform anpassad för röksimuleringar och i det demo som han publicerat är det just rök som visas upp. Det som varierar med vektorfältet är då rökens densitet.

Om man istället vill simulera vatten så har man en liknande men ändå annorlunda situation. Det är fortfarande Navier-Stokes-ekvationerna som gäller, men nu har man ett fysiskt *gränssnitt* mellan två element som man vill visualisera – vattenytan alltså. Normalt så bortser man från flöden i luften och löser bara ekvationerna för vattenfyllda voxlar. Sedan så kan man med fördel använda ett level set för att representera vattenytan, eftersom den förändras med tiden och byter topologi på ett komplext sätt när det plaskar och går vågor.

EN LITEN AVSLUTNING

Sådär, nu är min sammanfattning av kursen slut! Jag hoppas att du kommit till några nya insikter eller åtminstone blivit säkrare på det du redan visste. Kommentarer och annat får du gärna skicka till mig om du vill. Till sist vill jag bara tacka för uppmärksamheten, för detta är den sista sammanfattningen jag skriver under min tid på medieteknik här i Norrköping! Lycka till framöver!