Utvidgningar av OpenGL 1(4)

OpenGL är standardiserat. Men samtidigt utvecklas grafikkorten i mycket rask takt och blir allt kraftfullare. Därför inför olika tillverkare utökad funktionalitet. Flera sådana äldre utvidgningar har tagit sig in i standarden (ett tidigt exempel är *glBindTexture*, som gör det lätt att arbeta med mer än en texturkarta). Man kan lätt undersöka vad som finns tillgängligt av hårdvara och utvidgningar. Se avsnitt 14 i OpenGL-häftet. Partiella utskrifter från programmet `GL_INVESTIGATE2005.c` på ett par OH fram redovisas nedan. Man skiljer på utvidgningar som ARB (Architecture Review Board, som har hand om OpenGL) accepterat och sådana som bara några leverantörer kommit överens om (EXT) eller bara en infört (t ex NV från NVIDIA eller SGI från Silicon Graphics).

Datorerna i 6220 (PC med Linux, grafikkort NVIDIA FX6600)
Version: 2.0.0 NVIDIA 76.64
Leverantör: NVIDIA Corporation
Grafiksyst: GeForce 6600/PCI/SSE2/3DNOW!
GLSL-VERSION: 1.10 NVIDIA via Cg 1.3 compiler
Utvidgningar: ... **GL_ARB_fragment_program** GL_ARB_fragment_program_shadow **GL_ARB_fragment_shader** GL_ARB_occlusion_query GL_ARB_point_parameters GL_ARB_point_sprite **GL_ARB_shader_objects GL_ARB_shading_language_100 GL_ARB_vertex_shader** GL_EXT_Cg_shader GL_EXT_point_parameters GL_EXT_secondary_color GL_EXT_separate_specular_color GL_EXT_texture3D GL_EXT_texture_compression_s3tc GL_EXT_texture_cube_map GL_HP_occlusion_test GL_NV_fragment_program GL_NV_occlusion_query GL_NV_point_sprite GL_NV_texture_compression_vtc ...
Antal aux-buffertar: 4
Antal bitar per pixel i djupbuffert: 24
Antal röda bitar per pixel: 8
Antal alfabitar per pixel: 0
Antal stencil bitar per pixel: 8
Antal röda ackumuleringsbitar per pixel: 16
Stödjer stereo?0
Stödjer dubbel-buffring?1
GLUT_WINDOW_BUFFER_SIZE= 32

Vissa av nollorna beror på att vi inte i programmet begärde motsvarande resurs.

Utvidgningar av OpenGL 2(4)

Ett grävande program (GL_INVESTIGATE2005.c)

```
#include <GL/glut.h>
int main (int  argc,char *argv[]) {
    GLint a[5];  GLboolean b[5];  const char *v;
    glutInit (&argc, argv);
    glutInitDisplayMode (GLUT_DOUBLE | GLUT_DEPTH |
                GLUT_RGBA| GLUT_STENCIL );
    // Måste vara med
    glutCreateWindow ("Undersökning: Se terminalfönstret");
    v = glGetString(GL_VERSION);printf("Version: %s\n", v);
    v=glGetString(GL_VENDOR);printf("Leverantör: %s\n", v);
    v=glGetString(GL_RENDERER);printf("Grafiksystem: %s\n",v);
    v=glGetString(GL_SHADING_LANGUAGE_VERSION);
         printf("GLSL-VERSION: %s\n",v);
    v=glGetString(GL_EXTENSIONS);printf("Utvidgningar: %s\n", v);
    glGetIntegerv(GL_AUX_BUFFERS,a);
    printf("Antal aux-buffertar: %d\n", a[0]);
    glGetIntegerv(GL_DEPTH_BITS,a);
    printf("Antal bitar per pixel i djupbuffert: %d\n",a[0]);
    glGetIntegerv(GL_RED_BITS,a);
    printf("Antal röda bitar per pixel:  %d\n", a[0]);
    glGetIntegerv(GL_ALPHA_BITS,a);
    printf("Antal alfabitar per pixel:  %d\n", a[0]);
    glGetIntegerv(GL_STENCIL_BITS,a);
    printf("Antal stencil bitar per pixel:  %d\n", a[0]);
    glGetIntegerv(GL_ACCUM_RED_BITS,a);
    printf("Antal röda ackumuleringsbitar per pixel:  %d\n",
           a[0]);
    glGetBooleanv(GL_STEREO,b);
    printf("Stödjer stereo?%d\n", b[0]);
    glGetBooleanv(GL_DOUBLEBUFFER,b);
    printf("Stödjer dubbel-buffring?%d\n", b[0]);
    if (glutExtensionSupported("GL_EXT_polygon_offset"))
         printf("Visst stödjer jag det du frågade om!\n");
    //GLUT kan också ge besked, t ex antal bitar i bildminne
    printf(" GLUT_WINDOW_BUFFER_SIZE= %d\n",
        glutGet(GLUT_WINDOW_BUFFER_SIZE));
}
```

Programmet finns i `$DG/DEMOS`.

Utvidgningar av OpenGL 3(4)

En hel del utvidgningar har tagit sig in i senaste standard (specifikation) OpenGL 2.0 (september 2004; 1.5 kom oktober 2003). Det finns några problem.

- Dokumentationen finns inte alltid fullt ut i specifikationen (hittas via `www.opengl.org`) , utan i lösa dokument (`http://oss.sgi.com/projects/ogl-sample/registry/`).
- Standarden avviker i vissa avseenden från tidigare NV- eller ATI-versioner.

Här följer en lista över **några** utvidgningar som tagits in i OpenGL:s kärna (det gör att ARB i namnen i princip kan uteslutas):

- 2.0 GLSL (OpenGL Shading Language) och shading objekt, texturer utan 2^n-kravet, ...
- 1.5 Occlusion query, fragmentprogram (enbart utvidgning), ...
- 1.4 Vertex-program (enbart utvidgning), shadow, ...
- 1.3 Multitexturering, texturkomprimering, kubtexturer, alla matriser kan fås på transponerad form vilket är naturligare och förenklar, ...
- 1.2 3D-texturer, bildbehandlingsdel, ...
- 1.1 Hörnvektorer, texturobjekt, polygonoffset, ...

Utvidgningar av OpenGL 4(4)

De senaste av ARB accepterade utvidgningarna är
26-27 (Juni 2002/Sept 2002):*GL_ARB_vertex_program*, *GL_ARB_fragment_program* (**Anm**. Microsoft claims to own intellectual property related to this extension), som har sina ursprung i ATI- och NV-utvidgningar.
28. *GL_ARB_vertex_buffer_object*
29. *GL_ARB_occlusion_query*
30-32 (Juni 2003): *GL_ARB_shader_objects*,*GL_ARB_vertex_shader*, *GL_ARB_fragment_shader*
33 (Juni 2003): *GL_ARB_shading_language_100*
34 (Juni 2003): *GL_ARB_texture_non_power_of_two*
35. *GL_ARB_point_sprite*
36. *GL_ARB_fragment_program_shadow*
37 (Juli 2004): *GL_ARB_draw_buffers* (standard i 2.0)
38 (Juni 2004): *GL_ARB_texture_rectangle*
39. *GL_ARB_color_buffer_float*
40. *GL_ARB_half_float_pixel*
41. (Okt 2004) *GL_ARB_texture_floa*t
42. (Dec 2004) GL_ARB_pixel_buffer_object

Flera av dessa är inaktuella i och med att motsvarigheter tagit sig in i OpenGL 2.0.

Utvidgningar i egen kod på PC 1(2)

Vill man på en PC (med någon medlem i familjen Windows) dra nytta av utvidgningar i OpenGL (inkl OpenGL 2.0) måste man i minst ett avseende förfara litet annorlunda (än i andra system) vid egen kodning.

- Självklart måste du ha grafikkort (och tillhörande programvara) som stöder utvidgningen. Det bör observeras att Microsofts ursprungliga dynamiska bibliotek `opengl32.dll` inte förändras, vilket nog hade varit naturligt. I stället tillhandahåller korttillverkaren en ICD (installable client driver), som anropas av `opengl32.dll`.
- Du måste också ha (finns t ex via OpenGLs webbsida `www.opengl.org`) `glext.h` och placera den tillsammans med `gl.h`. Under Linux inkluderas den från `gl.h`, men vanligen inte under Windows. Skriv därför `#include <GL/glext.h>` efter motsvarande glut-rad. Härigenom får man tillgång till de flesta nya namn. Rätt filer följer rimligen med vid dator/kort-köp, men de kan behöva placeras lämpligt.
- I koden bör du kontrollera att den aktuella utvidgningen är tillgänglig, t ex enligt modellen
```
if (glutExtensionSupported("GL_ARB_multitexture")) { ... }
else ...
```
Detta kan göras tidigast efter `glutCreateWindow ("...")`.
- Om du vill använda ett nytt funktionsnamn (som ej hör till kärnan) måste du förfara enligt modellen
```
glMultiTexCoord2fARB = (PFNGLMULTITEXCOORD2FARBPROC)
            wglGetProcAddress("glMultiTexCoord2fARB");
```
Samma placering som i förra punkten. Detta krångliga sätt, som inte alls behövs under Linux eller Solaris, hänger ihop med Microsofts åsikt att `opengl32.dll` inte får ändras.

Utvidgningar i egen kod på PC 2(2)

Sista punkten på föregående sida vållar mycket besvär och gör att programmen inte utan vidare är transportabla. På nätet kan man hitta färdiga initieringsprocedurer, som underlättar. Just nu är nog GLEW - OpenGL Extension Wrangler[1] (se `http://glew.sourceforge.net`) mest inne).

Man skriver då i st f
```
#include <GL/glut.h>
```
raderna
```
#define GLEW_STATIC 1
#include <GL/glew.h>
#include <GL/glu.h>
#include <GL/glut.h
```
och före glutMainLoop();
```
glewInit();
```

Kompilering under Linux:
```
$gcc -o Prog.out Prog.c  -I/users/course/TDA360/LINUX/GLEW/
glew/include  -L/users/course/TDA360/LINUX/GLEW/glew/lib  -
lGLEW -lglut -lGLU -lGL
```
och under Windows (hos oss; tillfälliga placeringar; och ger just nu fel)
```
gcc -o Prog.exe Prog.c -I../PC/GLEWPC/src/glew/include -L../
PC/GLEWPC/src/glew/lib -lglew32 -lglu32 -lopengl32
```
Före körning under Linux
```
setenv LD_LIBRARY_PATH /users/course/TDA360/LINUX/GLEW/glew/
lib:$LD_LIBRARY_PATH
```
Liknande för Windows

1.Boskapsskötare enl ordboken

Mesa och OpenGL 1.5 1(2)

Mesa (`www.mesa3d.org`) är en OpenGL-emulator, som i sin senaste version 6.0 (dec 2003) (6.2 okt 2004) uppges klara allt i OpenGL-specifikationen 1.5 (även GLSL tycks det). Utan att man har program/maskinvara för OpenGL! Mesa bygger bara vidare på datorns bottengrafiksystem, dvs t ex X eller Windows. En ansenlig prestation av främst upphovsmannen Brian Paul. Men snabbt går det inte. Motiven för vidareutveckling verkar något svaga, sedan SGI släppt källkod för OpenGL. Det förefaller dock som om Mesa i viss mån utnyttjar eventuellt installerad OpenGL. Inga specialåtgärder behövs under Windows. Man inkluderar bara som vanligt `GL/glut.h`.
För den som inte har tillgång till bra grafikkort kan det användas för att pröva t ex vertex- och fragmentprogrammering (uppgift 7 på laboration 3). En annan användning kan vara felsökning. Men långsamt.

Hur på kurskontot? (ej kontrollerat 2005)

- Kompilering och länkning med
`MESALINK DittOpenGLProg`
- Körning med `DittOpenGLProg`
eller `MESARUN DittOpenGLProg`
- I `$DG/MESA6.0/Mesa-6.0/progs` finns ett antal demoprogram (källkod och exekverbara program).

På PC?

Binärer för Visual C++ finns att hämta. Se kurssidan. För gcc har jag inte lyckats fullt ut, men något finns via kurssidan. OBS! Mesa har egna DLL-er för OpenGL.

Beräkningsgeometri: Allmänt

Vi skall här se på några geometriska problem som är av datorgrafiskt intresse. **Beräkningsgeometri** (eng computational geometry) anses handla om algoritmer (och datastrukturer) för geometriska problem. Det är alltså frågan om konstruktion och analys av metoder för konkret praktisk lösning av problem och inte den andra sidan av geometrin nämligen härledning av egenskaper. Vi presenterar inte fullständiga algoritmer och gör inte heller ordentliga prestandaanalyser. Och vi tar bara upp några få problem.

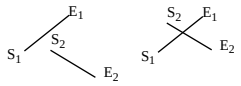
Vi förutsätter i allmänhet att ändpunkter etc har heltalskoordinater, dvs att vi arbetar i skärmkoordinatsystemet. Det gör att vi kan förutsätta att alla beräkningar görs exakt.

Vi har tidigare mött flera problem som skulle kunna sägas tillhöra området. Och som vi sa redan i början av kursen och alla vid det här laget märke till spelar matematik en betydande roll inom datorgrafik.

Referens: O'Rourke, J: Computational Geometry in C, Cambridge University Press, 1994.

Beräkningsgeometri: Linjers skärning i 2D

Vi har två linjesegment och det gäller att avgöra om de skär varandra och i så fall bestämma skärningspunktens koordinater. Ett segment



med startpunkt i S = (Sx,Sy) och slutpunkt i E = (Ex,Ey), kan matematiskt beskrivas dels på parameterform

$P = S+t(E-S)$, $0 \le t \le 1$,

dels på tradionell form

$F(x,y) \equiv Ax + By + C = 0$ med begränsning av t ex x,

där A=(Ey-Sy), B=(Ex-Sx), C=ExSy-SxEy.

Algoritm 1: Beteckna parametern för de båda linjerna med *t* respektive *u*. Vi får då

$S_1 + t(E_1 - S_1) = S_2 + u(E_2-S_2)$

som är ett ekvationssystem med två ekvationer för de två obekanta *t* och *u*. Om linjerna inte är parallella kan vi alltså lösa ut *t* och *u*. Skärning karakteriseras av att $0 \le t,u \le 1$. Annars är det segmentens förlängningar som skär varandra.

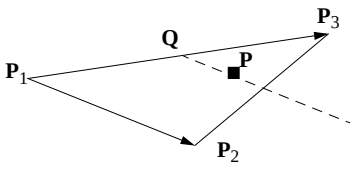
Om de flesta segment som vi vill undersöka skär varandra är detta en fullt acceptabel metod. Om däremot de flesta inte gör det, så lägger man ner onödigt mycket möda och följande metod är effektivare.

Algoritm 2: Vi undersöker först om ändpunkterna hos den andra linjen ligger på ena sidan om den första, dvs om $F_1(S_2)$ och $F_1(E_2)$ har samma tecken. I så fall finns ingen skärning. På samma sätt undersöks om $F_2(S_1)$ och $F_2(E_1)$ har samma tecken. I så fall finns heller ingen skärning. Annars skär segmenten varandra och skärningspunkten kan beräknas med linjär interpolation utifrån de beräknade värdena.

Beräkningsgeometri: Barycentriska koordinater

Ibland behöver man kunna beskriva punkter i en triangel. Vi såg ett sådant exempel i samband med morfing och möter ett annat på nästa OH.

Varje punkt **P** i en triangel



kan skrivas

$$P = P_1 + v(P_3 - P_1) + u(P_2 - P_1)$$

där $0 \le u$, $0 \le v$ och $u+v \le 1$, vilket framgår av figuren. Vi går ju från P_1 i vektorn P_3-P_1:s riktning till Q och därefter längs den streckade linjen, som har samma riktning som vektorn P_2-P_1. Kanten P_2 -P_3 motsvarar att u+v=1.

Alternativt utnyttjar vi att en triangel är konvex och att punkterna i den därför genereras med $P = \sum \alpha_j P_j$, med $\sum \alpha_j = 1$ och $0 \le \alpha_j \le 1$ eller omskrivet $P = (1-\alpha_2-\alpha_3)\cdot P_1 + \alpha_2 P_2 + \alpha_3 P_3$, som överensstämmer med den tidigare med $u = \alpha_2$ och $v = \alpha_3$. Man brukar kalla $(\alpha_1,\alpha_2,\alpha_3)$, där $\alpha_1 = 1-\alpha_2-\alpha_3$, för **barycentriska koordinater** (Möbius 1827 eller tidigare). Vi kanske kan kalla (u,v) för **triangelkoordinater**.

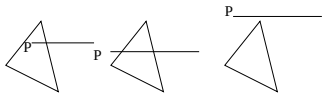
Beräkningsgeometri: Punkt i polygon 1(3)

Bl a vid pekning på objekt kan det vara av intresse att veta om en punkt ligger inuti en polygon eller inte. En del grafikbibliotek har en inbyggd funktion (typiskt namn *PointInPolygon*) för detta i skärmkoordinater. Specialfallet att polygonen är en rektangel med axelparallella sidor är simpelt. I det allmänna fallet kan det vara lämpligt att hålla reda på en omslutande rektangel, mot vilken grovtest sker.

Specialfallet triangel

Algoritm 1: Vi kan uttrycka punkten P i triangelkoordinater (u,v) genom att lösa ett litet ekvationssystem. Vi behöver därefter bara kontrollera om $0 \le u$, $0 \le v$ och $u+v \le 1$.

Algoritm 2: En helt annan algoritm räknar antalet skärningar mellan en högerriktad horisontell linje från den aktuella punkten och triangelsidorna.



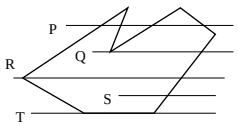
Är antalet skärningar 1, ligger punkten inuti, annars ligger den utanför. Test sker mot triangelsida efter triangelsida, dvs vi har ett problem av samma slag som i avsnittet om linjers skärning, som dock underlättas något av att den ena linjen är horisontell. Beträffande ett par komplikationer se nedan.

Beräkningsgeometri: Punkt i polygon 2(3)

Allmänt

Vi kan göra varianter av båda de tidigare algoritmerna. Låt oss börja med den sista.

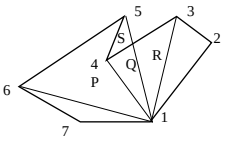
Algoritm 2: På samma sätt som förut räknar vi antalet skärningar mellan en horisonell linje och polygonsidorna. Är antalet udda ligger punkten inuti, annars ligger den utanför. Detta bygger på att när den horisontella linjen passerar en sida går man in i eller ut ur polygonen.



T ex ger punkten P i figuren fyra skärningar och ligger utanför medan S ger en skärning och ligger innanför. Om linjen går genom ett av polygonens hörn stämmer vårt resonemang inte riktigt. T ex ger R skärning med tre sidor, men ligger utanför. Förklaringen är att i polygonhörnet längst till vänster får vi bara en inpassage och bara en av de två skärningarna där skall räknas (ett annat sätt att inse det är att tänka sig R flyttad litet i höjdled vilket inte påverkar utanförskapet). I fallet Q skall däremot varje skärning räknas eftersom hörnet ger en utpassage följd av en inpassage. Detta kan uttryckas som att för spetsar i x-led tar man bara med en av skärningarna, medan för spetsar i y-led båda skall räknas. Horisontella kanter ger ju egentligen oändligt många skärningar, men ignoreras helt. T ex ger T två skärningar och ligger utanför. En likartad situation uppstår när det gällde rastrering av en allmän polygon.

Beräkningsgeometri: Punkt i polygon 3(3)

Algoritm 1: Vi väljer något hörn och tittar efter om punkten ligger i någon av de successiva trianglar som bildas, dvs i figurens fall 123, 134, 145, 156, 167.



Är polygonen konvex överlappar inte trianglarna och bildar tillsammans polygonen. Då räcker det att testa mot triangel efter triangel och att avbryta vid eventuell träff. Är den icke-konvex, som i figuren, måste samtliga trianglar undersökas och är antalet träffar udda så ligger punkten inuti polygonen, annars utanför. T ex ligger P och R i var sin triangel och ligger inuti polygonen. Q ligger i tre trianglar, 134 och 145 och 156, och innanför polygonen. S ligger i två trianglar, 145 och 156, och utanför.

Båda algoritmerna har en tidskomplexitet av O(N), där N är antalet sidor i polygonen. I allmänhet anses Algoritm 2 vara den snabbaste.

Beräkningsgeometri: Area och omkrets

En polygon i xy-planet med hörn $\mathbf{P}_0$, $\mathbf{P}_1$, $\mathbf{P}_2$, ..., $\mathbf{P}_{n-1}$ och $\mathbf{P}_n$=$\mathbf{P}_0$, där $\mathbf{P}_k = (x_k,y_k)$ är given. Det är välbekant att för omkretsen gäller

$$Omkretsen = \sum_{k=1}^{n} \sqrt{(x_k - x_{k-1})^2 + (y_k - y_{k-1})^2}$$

Det är ju bara att summera de enskilda kanternas längder för vilka vi använder avståndsformeln.

Mindre känt är att arean är

$$Arean = \frac{1}{2} \cdot \left| \sum_{k=1}^{n-1} \langle x_k \cdot y_{k+1} - x_{k+1} \cdot y_k \rangle \right| .$$

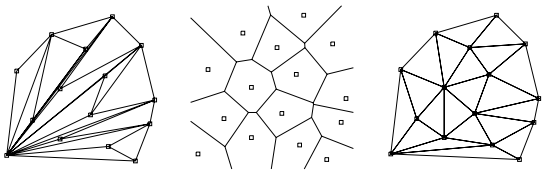
Förvånansvärt nog är denna formel "enklare" än den för omkretsen.

Det är däremot avsevärt svårare att bevisa ytformeln (man har nytta av begrepp som ytintegral och kurvintegral från den flerdimensionella analysen). Beviset är inget för oss. I enklare fall dock direkt ur motsvarande formel för en triangel (se t ex Beta).

Det kan anmärkas att OpenGL använder summan för att bestämma en polygons framsida. Och att summan också är en beståndsdel i det andra sättet att bestämma en normal till en polygonyta. Båda sakerna har vi varit inne på tidigare.

Beräkningsgeometri: Delaunaytriangulering

Problemet är att givet en N st punkter bilda ett triangelmönster med hörn i punkterna. En tillämpning kan vara att man har mätvärden i punkterna och vill interpolera fram värden i andra punkter. Linjär interpolation över trianglar gör man ju lätt. En annan är FEM (Finita Element Metoden). En tredje, s k morfing. I figuren nedan är de givna punkterna markerade med små fyrkanter. Längst till höger visas en lyckad triangulering.



Rakt på

Med N st punkter finns det $N^2/2$ möjliga kanter. En algoritm är att gå igenom dessa och rita kanter som inte skär redan ritade (återigen har vi nytta av linjeskärningsalgoritmer). Detta leder till ett resultat av typen i vänstra figuren ovan, dvs i allmänhet inte så bra.

En girig metod

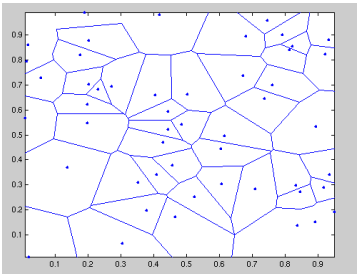
En bättre metod är att sortera de tänkbara $N^2/2$ kanterna i växande ordning och sedan gå igenom följden och rita kanter som inte skär redan ritade. Härigenom reduceras risken att trianglarna får långa sidor. Sorteringen (med t ex kvicksorteringen) är en $O(N^2 \log N)$ process. Resten av metoden kan (med knep) utformas så tidskomplexiteten totalt är sådan. Resultatet kan bli ungefär som i högra figuren ovan.

Delaunaytriangulering

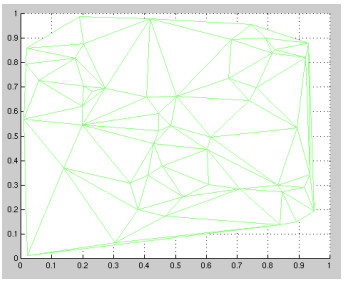
En bättre men mera komplicerad metod kallas Delaunaytriangulering. Det är en sådan som visas till höger ovan. En viktig egenskap är att metoden maximerar den minsta triangelvinkeln, dvs den ger mindre spetsiga trianglar än någon annan metod. Man kan konstruera algoritmen så att den får tidskomplexiteten O(N log N). Vi kan inte här gå in på några detaljer utan anger bara huvudstegen. Först bildas ett s k Voronoi-diagram, se mellersta figuren ovan. Till varje given punkt bildar man en Voronoi-polygon som utgörs av de punkter i planet som ligger närmre den givna punkten än någon annan. Att det rör sig om en polygon är uppenbart eftersom man successivt skär bort halvrymder. Nästa steg är att sammanbinda de ursprungliga punkter som har en gemensam polygonkant. Under rätt allmänna betingelser erhålles en triangulering. Det bör påpekas att spetsiga trianglar och långa triangelkanter inte alltid kan undvikas.

Voronoi/Delaunay med MATLAB

```
>> x = rand(1,50); y = rand(1,50);
>> voronoi(x,y)
```



```
>> tri = delaunay(x,y);
>> trimesh(tri,x,y, zeros(size(x)))
>> view(2)  % standardbetraktning i 2D
```



```
>>
```

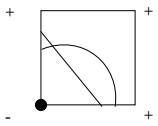
Marscherande kuber (marching cubes) 1(4)

Bakgrund:

- 2D
  1. Hur ritar man allmänna kurvor bestämda av en ekvation f(x,y)=0?
     Ex. $x^2 + y^2 - 1 = 0$ (cirkel)
     Ex. $x^{2/3} + y^{2/3} - 1 = 0$ (asteroid)
  2. Hur ritar man nivåkurvor till 2D-data givna över ett ekvidistant rutnät?
- 3D
  1. Hur ritar man allmänna ytor bestämda av en ekvation f(x,y,z)=0?
     Ex. $x^2 + y^2 + z^2 - 1 = 0$ (sfär)
     Ex. $x^4 + y^4 + z^4 - y^2z^2 - z^2x^2 - x^2y^2 - x^2 - y^2 - z^2 + 1 = 0$ (Kummers yta)
  2. Hur ritar man nivåytor till 3D-data givna över ett ekvidistant kubnät?

För speciella kurvor och ytor kan man hitta en parameterframställning och med dess hjälp lätt plocka fram approximerande trianglar (t ex). Det gäller ju cirkel- och sfärfallet men även för asteroiden ($x = \cos^3(t)$, $y = \sin^3(t)$). Men vi är intresserade av allmänna kurvor och ytor.

En idé är att bestämma en begränsande kvadrat till kurvan resp en kub till ytan och dela in kvadraten i mindre kvadrater resp kuben i mindre kuber och bestämma en approximerande linjär kurva resp yta för resp delobjekt. Låt oss hastigt se på en sådan situation i 2D. Funktionen antages beräknad i hörnen med värden vars tecken anges i figuren



Den verkliga funktionskurvan kan se ut som bågen i figuren. Men för små delkvadrater ligger den i närheten av den approximerande räta linje som ritats. Ändpunkterna till denna beräknas lätt med linjär interpolation.

Marscherande kuber (marching cubes) 2(4)

I figurens fall finns bara en tolkning, men det finns också situationer där mer än en tolkning är möjlig, t ex

Med tillräckligt små delkvadrater uppstår inte detta problem.

Låt oss övergå till det allmänna fallet i 3D. Det finns två huvudalternativ

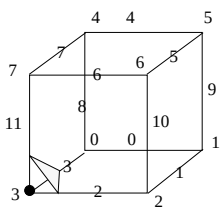
- Strålföljning
- Marscherande kuber

Vi vet att strålföljning ger ett strålande resultat (och man kan i allmänhet göra som för sfärer även om det blir besvärligare att lösa ekvationerna), men är en kostsam metod, så låt oss koncentrera oss på "marscherande kuber". Vi tänker oss att ytan är sluten (inte så viktigt; mest för formuleringarnas skull) och att

- f(x,y,z) = 0 på ytan, f(x,y,z) < 0 innanför och f(x,y,z) > 0 utanför

Förberedande steg: Bestäm ett rätblock inom vilket ytan (eller den intressanta delen av den finns). Dela in denna i t ex 100x100x100 delkuber.

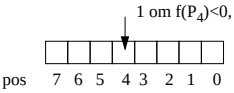
Algoritmens steg per delkub:



$f(P_i) \geq 0$ för alla hörnpunkter utom P_3.

Marscherande kuber (marching cubes) 3(4)

1. Bestäm vilka av hörnen som ligger inuti objektet.
   Leverera svaret i form av ett 8-siffrigt binärt tal



   Ex: Om enbart hörn 3 inuti => 00001000 = 8
   Ex: Om hörnen 0-3 inuti => 00001111 = 15

2. Slå nu i en förpreparerad tabell upp vilka kanter som skärs av ytan. Som ingångsnummer används det framräknade talet och raderna nedan är märkta med det. Därefter följer en uppgift om antalet trianglar och sedan för varje triangel motsvarande kantnummer.

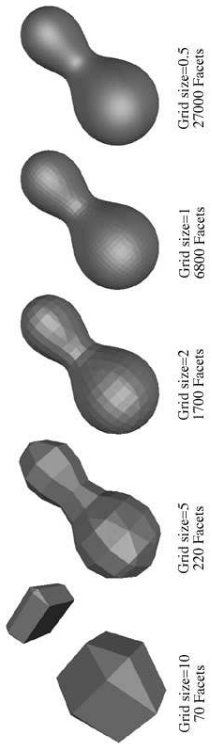
```
  0   0
......
  8   1 3 11 2
......
 15   2 8 11 10 8 10 9
......
255   0
```

3. Bestäm nu med linjär interpolation varje skärningspunkt. Vi får därmed hörnpunkterna för samtliga trianglar som berör kuben. Maximalt behövs 5 trianglar per kub.
4. Rita upp trianglarna. För att resultatet skall bli bra behövs belysning.

Även i det tredimensionella fallet finns tvetydiga fall.

Referens: Paul Bourke (http://astronomy.swin.edu.au/~pbourke/), som gjort massor med bra datorgrafikmaterial. Jag har bearbetat hans program och läst en uppsats. Det finns andra utformningar av algoritmen. Metoden presenterades 1987 av W.E.Lorensen, men har senare bearbetats av många.

MK: Ett exempel från Paul Bourkes sidor 4(4)



Ytan är en s k blob (eller snarare två).

F ö från
Comp.Graphics.Algorithms
Frequently Asked Questions
http://www.exaflop.org/docs/cgafaq

Subject 5.11: What is the status of the patent on the "marching cubes" algorithm?

United States Patent Number: 4,710,876
Date of Patent: Dec. 1, 1987

Inventors: Harvey E. Cline, William E. Lorensen'
Assignee: General Electric Company

Title: "System and Method for the Display of Surface Structures Contained Within the Interior Region of a Solid Body"

Filed: Jun. 5, 1985

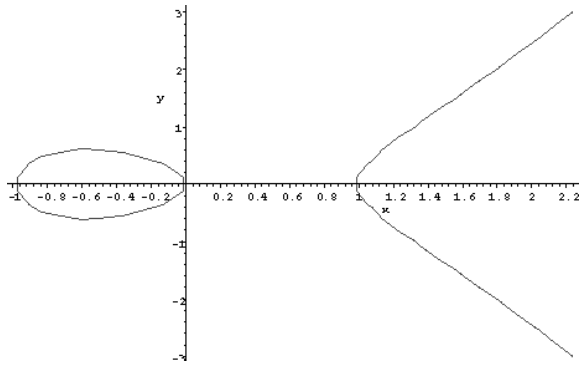
På andra håll:

"I believe the patent expires 17 years after issue."

"Not much we can do about that til around 2003 or so when the MC patent expires,"

Implicita kurvor/ytor i Maple

```
with(plots);
implicitplot(y^2-x^3+x,x=-3..3,y=-3..3);
```



Just detta fall är ju enkelt att analysera. För givet x har vi en andragradsekvation i y, $y^2=x^3-x$, dvs till varje x finns 0 eller 2 (ev sammanfallande) y-värden. För att lösning skall finnas måste $x^3-x>=0$, dvs $x(x^2-1)>=0$, dvs x>=1 eller -1<=x<=0, precis som figuren visar.

Maple använder en samplingsteknik för sin ritning, vilket förklarar den "fula" kurvan. Samplingstätheten kan visserligen justeras. Exakt hur Maple går tillväga kan man ta reda på genom att studera källkoden för `implicitplot` och de rutiner den använder sig av.

Maple har motsvarande kommando för ytor.

CSG-operationer 1(2)

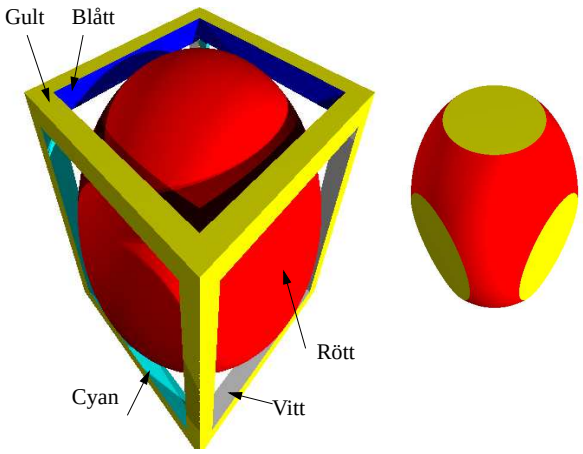
I modelleringsprogram brukar man kunna använda CSG-operationerna (CSG = Constructive Solid Geometry) + (union) och - (skillnad) samt skärning. Kallas även boolska operationer. Matematiken bakom går vi inte in på.

Exempel (POVRay): Som bl a visar hur lätt man kan tillverka "omöjliga" föremål.

```
union {
   // Ett rött klot
   sphere {<0, 0, 0>, 1.75
           pigment { Red }
   }
   // Från en kub subtraheras tre rätblock
   difference {
      object {UnitBox scale 1.5 pigment {Yellow}}
      object {UnitBox scale <1.51, 1.25, 1.25>
              pigment {White}
      }   // "clip" x
      object {UnitBox scale <1.25, 1.51, 1.25>
              pigment {Blue}
      }   // "clip" y
      object {UnitBox scale <1.25, 1.25, 1.51>
                      pigment {Cyan}
      }   // "clip" z
   }
   rotate y*45
}
```

Instoppat i kod med lampor kan resultatet bli det som syns på nästa sida. Väsentligen `granite.pov` men modifierad av trycktekniska skäl.

CSG-operationer 2(2)



Ytfärgen från de subtraherade rätblocken smiter tydligen av sig. Till höger skärningen mellan en sfär och en kub enligt koden

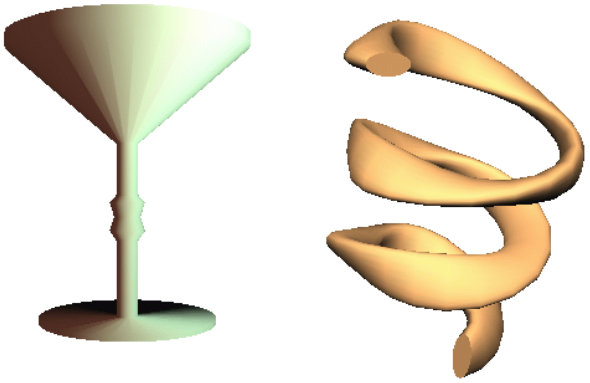
```
intersection {
   sphere {<0, 0, 0>, 1.75
           pigment { Red }
   }
   object {UnitBox scale 1.5 pigment {Yellow}
   }
   rotate y*45
}
```

OpenGL:s GLU har också CSG-operationer men de gäller bara polygoner.

Svep- och extrusionsbibliotek

Detta leder osökt in på två andra operationer vi mött tidigare under kursen.

Linas Vepstas (`http://linas.org/gle/index.html`) skapade 1991 till en föregångare till OpenGL ett sådant bibliotek kallat GLE (tube and extrusion library). I GLUT-distributionen ingår version 3.0.7 (Dec 2001). Via webbplatsen hittar man en senare 3.1.0 (Mars 2003).



Prova: T ex `$DG/EXEMPEL_GLUT/gle/helix4`) och använd musen.

Egen kompilering (PC):

```
gcc helix4.c  mainsimple.c -lgle -lglut32 -lglu32 -lopengl32
                           -luser32 -lgdi32
```