Scientific visualization, Vetenskaplig visualisering
MVE080, MAI510

Thomas Ericsson
Computational mathematics
Chalmers/GU

office: L2075
phone: **772 10 91**
e-mail: **thomas@chalmers.se**
www: **http://www.math.chalmers.se/Math/Grundutb/CTH/mve080/1011/**

Assignments, copies of handouts (lecture notes etc.) can be found at the www-address (look under Diary).

Some of the images from this introduction is not available in the handout, though (they take too much space).
[Image] denotes a missing image (images).

Contents

Here comes the syllabus (kursplan):

Aim
The solution of computational problems with the help of computers often generate large data sets. This course deals with how computer graphics can be used to visualize data in order to give a better understanding of the problem and its solution. In simple cases the solution can perhaps be represented as a curve. More complicated problems have solutions in the form of surfaces or volumes, maybe even time dependent. Many mathematical problems may not generate so large data sets but require an understanding of more three dimensions.

Goal
At the conclusion of the course, the participant should find it natural to think in visualization terms, be able to produce insightful graphics in a number of common cases, be quite familiar with Matlab graphics, and be acquainted with OpenGL and ParaView.

Prerequisites
Basic courses in mathematics, numerical analysis, programming and data structures. Basic Matlab programming. This is an introductory course so no prior knowledge of computer graphics is required.

Content
Introduction to visualization. Different techniques for visualizing surfaces, volumes and other common mathematical objects. Animation. Interaction. An orientation about the construction of user interfaces. OpenGL, ParaView and advanced Matlab graphics.

Matlab, easy and to get started. OpenGL to see how some basic computer graphics is done. ParaView, more capable than Matlab (but harder to use).

Computer graphics concepts, such as transformations and shading models, necessary to use and understand the graphics software. A sufficient amount of C to finish the the computer assignments.

Organization
Lectures and computer assignments. The assignments, which make up a substantial part of the course, consist of several problems where the student will use Matlab, OpenGL and ParaView to solve different visualization problems.
The problems are fetched from numerical analysis and applied mathematics.

The assignments vary in difficulty. Some are routine tasks (would take me minutes) while others require a bit of programming.

Literature
Lecture notes, articles and manuals.

Reference literature: F S Hill, S M Kelley, Computer Graphics using OpenGL, 3d ed., Pearson, 2008 or Edward Angel, Interactive Computer Graphics, A Top-Down Approach with OpenGL, Pearson Education 2008, 5rd ed.

The topic of these titles are not strictly visualization, they are standard computer graphics books.
See `http://www.opengl.org/documentation/books/` for more titles. On the next page I will list more literature.

Examination
Compulsory computer assignments and take-home exam.

We have two lectures and two labs per week. See the schedule on www. Show me your solutions to the assignments at lab-times. You do not have to hand in any reports.

More books from my shelf

Here comes a list of books which I collected with this course in mind. For other books, see the references on the home page. Some E-books are available via the Chalmers library home page. Finally there are man-pages and PDF-manuals.

- S. K. Card, J. D. Mackinlay, B. Shneiderman, Readings in Information Visualization: Using Vision to Think Morgan Kaufmann, 1999.
- C. D. Hansen, C. R. Johnson (eds.), Visualization Handbook av Johnson, Academic Press, 2004.
- R. Spence, Information Visualization, Addison-Wesle, 2001.
- D. Thompson, J. Braun, R. Ford, OpenDX: Paths to Visualization, 2nd ed. 2004, `http://www.vizsolutions.com`
- D. A. Norman, The Design of Everyday Things, Basic Books, 2002.
- C. Ware, Information Visualization Perception for Design, Elsevier, 2004.
- M. K. Agoston, Computer Graphics & Geometric Modeling, Implementation and Algorithms, Springer, 2004. There is one Computer Graphics and Geometric Modeling: Mathematics, which I do not have.
- S. R Buss, 3D Computer Graphics: A Mathematical Introduction with OpenGL, Cambridge UP, 2003.
- H. C. Hege, K. Polthier (eds.), Mathematical Visualization, Algorithms, Applications and Numerics, Springer, 1998.
- J. O'Rourke, Computational Geometry In C, Cambridge UP, 1998.
- D. F. Rogers, An Introduction to Nurbs: With Historical Perspective Morgan Kaufmann, 2000.
- A. Unwin, M. Theus, H. Hofmann, Graphics of Large Datasets, Springer, 2006. To the math-library.

Some typical visualization problems

The primary goal of Scientific Visualization, is to provide insight into scientific data. We often need a deeper understanding of a phenomenon, need to draw conclusions, make predictions. (Computer) graphics can (often) give us the help we need, after all:

"An image says more than a thousand words (or numbers)"

Scientific visualization usually has a natural physical or mathematical representation or background. We may want to visualize the flow of air around aircraft or the roots of an equation. When visualizing the data, we would probably make an outline of the aircraft and draw a coordinate system for our roots. [Image]

A related area is information visualization. It is less common with a physical background and it may not even be important. A classical example is Harry Beck's map of the London underground (1931). [Image]
See `http://en.wikipedia.org/wiki/Harry_Beck` for example.

Previous maps based on the actual layout, the geography, of the underground had not worked well. Beck's map, on the other hand, leaves the physical reality behind and shows the order of stations, where lines cross etc. It captures what is essential for the traveler.

Another example is given by business graphics (pie charts etc.), e.g. visualizing the number of admitted and graduated students for different programmes at Chalmers/GU.

This course will deal with scientific visualization.
You have already dealt with this in previous classes. Plotting the graph of a scalar function of a scalar variable, `plot(x, y)` provides almost a complete understanding of the function.

There are however harder visualization problems, where we only get a partial understanding, e.g. looking at $w = f(x, y, z)$, given a function f. [Image] Understanding $w = f(x, y, z, t)$ completely may be hopeless.

Another cause of problems may be the amount of data.
Computers are fast, and when a program has executed a few hours the output can be enormous, several Gbytes.
To visualize this amount of data may be difficult, but a thousand numbers may be hard enough.

It is not always easy to say what is a meaningful image.
Tastes differ as does the ability to interpret 3D-plots, for example. So this course will show different ways of visualizing data, but there is rarely a unique solution to a visualization problem (or to the assignments).

Use your imagination. If one plot is not that helpful there may be another, better, way to visualize the data. Trial and error may be a successful method.

An example, the cosine function

In Matlab it is possible to compute $\cos z$ where z is a complex number. Suppose we would like to understand how this function behaves. Since we know a lot of mathematics we can easily list several properties.

Let a, b be real numbers, then

$$\cos(a+ib) = \frac{e^{i(a+ib)} + e^{-i(a+ib)}}{2} = \cdots =$$

$$\frac{(e^b+e^{-b})\cos a}{2} - i\,\frac{(e^b-e^{-b})\sin a}{2}$$

If $z \in C$ then the following is true, for example:

$$\cos(z+2\pi) = \cos z, \quad \cos z = \cos(-z), \quad \cos\bar{z} = \overline{\cos z}$$

So, it is sufficient to study $0 \leq Re(z) \leq 2\pi$ and $Im(z) \geq 0$.

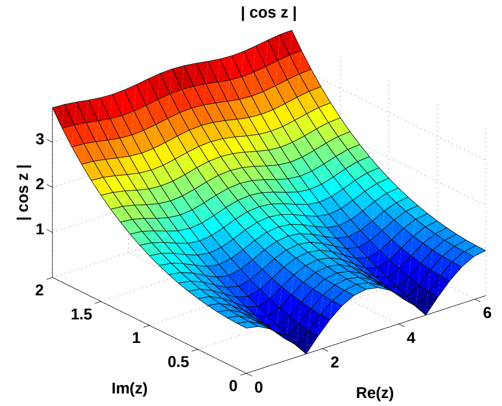
For large b

$$|\cos z| \approx \frac{e^b|\cos a - i\sin a|}{2} = \frac{e^b}{2}$$

In a real application it may not be possible to use mathematics this way. Perhaps the function is too complicated, or perhaps worse, we may not have an expression for the function. We may have to rely on a computer program that returns $f(z)$ for a given z.

The visualization of $\cos z$ is still a bit hard since we are dealing with four real dimensions. Here are a few alternatives (not all good).

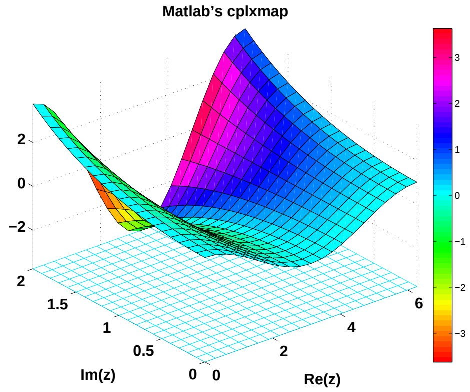
The obvious first try, plot $|\cos z|$ as a function of $(Re(z), Im(z))$.



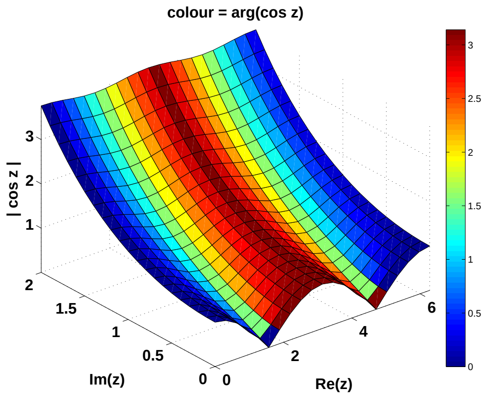
In Matlab this would be in colour, where the colour corresponds to $|\cos z|$.

It captures some of the behaviour: periodicity, what happens for large $Im(z)$. We have lost the sign information, and introduced corners (like $x \longrightarrow |x|$).
On the other hand, this image may be exactly what we need. It is possible to use more fancy graphics, no grid but a smooth surface using light etc. [Image]

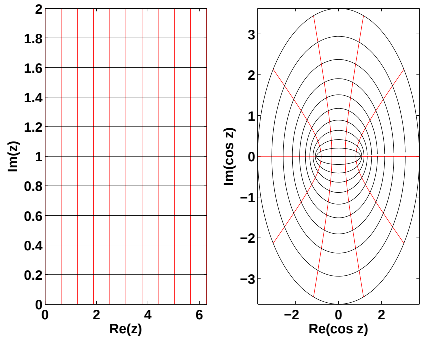
The next image was done with Matlab's cplxmap-command. It plots $Re(\cos(z))$ as a function of $(Re(z), Im(z))$. The colour is used for $Im(\cos(z))$. I have added a color bar. I have a problem with this plot. The shape of the surface dominates over the colour information.



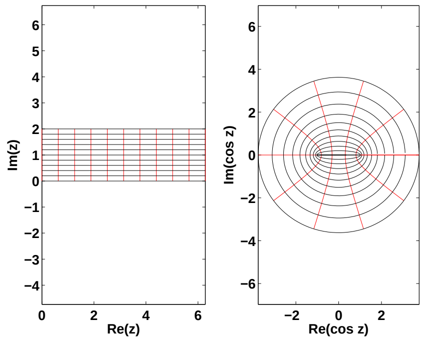
A similar idea is to plot $|\cos z|$ as before but to let the colour show the argument, so if $\cos z = r\ e^{i\varphi}$ we use colour for φ and height for $r = |\cos z|$.



In the next plot we do not loose any space-dimension. A grid in the $(Re(z), Im(z))$-plane is mapped onto $(Re(\cos z), Im(\cos z))$. We see the periodicity in a new way. Lines with constant imaginary-part seem to be mapped onto closed curves.

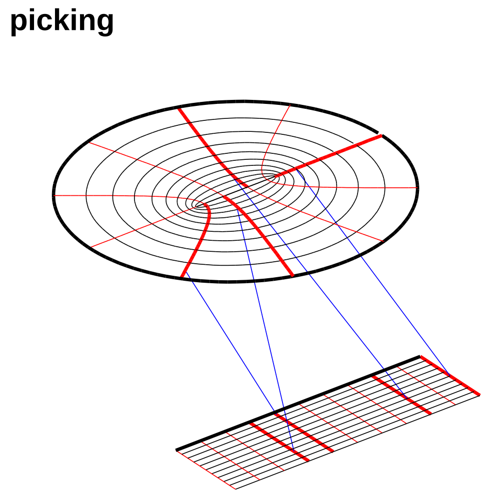


The plot is not quite truthful. Matlab tries to fill out the window, which may cause different scaling between the axes (a circle may look like an ellipse). After correction, `axis equal`, we see some new features.



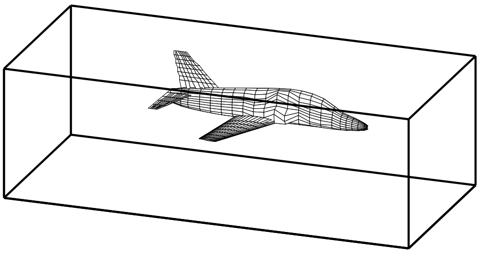
It seems like we have right angles between the curves in the right diagram. So are the angles in the z-grid preserved? Yes. Those who have read complex analysis may recognize this as a special case of a more general theorem. cos is a conformal mapping and hence preserves angles (whenever the derivative is non-zero).

One drawback with this plot is that is hard to know what line corresponds to which $\cos z$-curve. Perhaps we could use some interaction with the mouse, clicking on a line in the left window would make the corresponding curve in the right window blink, change colour or something.



We end with two images where we plot $Re(\cos z)$ and $Im(\cos z)$ in two windows or together in one. [Image].
Several other alternatives remain.

Now to another problem, mesh generation in 3D. The difficulty is not the number of dimensions this time, but the huge amount of data.



Discretize (divide into small volume elements) the air in the box and outside the aircraft. Mesh generation (using **m3d**, an ITM-project, Swedish Institute of Applied Mathematics) on one RS6000 processor:

```
wed may 29 12:54:44 metdst 1996  So this is old stuff
thu may 30 07:05:53 metdst 1996

183463307 may  2 13:46 s2000r.mu

tetrahedrons in structured mesh:   4 520 413
tetrahedrons in unstructured mesh: 4 811 373
```

Does the program work? Does it refine the mesh it the right places? Make nice images for the annual report (and for those supplying the money). [Image] (several).

There are of course many other visulization problems. Here are a few [Image] showing a simulation of an open cavity problem. Others will turn up on the lectures or in the labs.

Starting Matlab

```
> matlab -help

Here is an edited list:


-h|-help           - Display arguments.
-nodisplay         - Do not display any X commands. The
                    MATLAB desktop will not be started.
                    However, unless
                    -nojvm is also provided the Java
                    virtual machine will be started.
-nosplash          - Do not display the splash screen
                    during startup.
-nodesktop         - Do not start the MATLAB desktop.
                    Use the current terminal for
                    commands. The Java virtual
                    machine will be started.
-nojvm             - Shut off all Java support by not
                    starting the Java virtual machine.
                    In particular the MATLAB
                    desktop will not be started.
```

I use `matlab -nodesktop`
To get short help, type `help command` For more help use the GUI (Graphical User Interface) or `doc command` There are thick PDF-manuals available (through the GUI) as well. Start Help and click on MATLAB, choose Printable (PDF) Documentation. The basic graphics manual is 679 pages and the 3D-manual an additional 212 pages.

For this to work you have to tell MATLAB what browser you are using. Netscape is default (and we do not have it). This is one way to fix it (not necessary in Matlab 2009b):

```
cd
mkdir matlab
cd matlab
cp /chalmers/sw/sup/matlab-2008b/toolbox/
    local/docopt.m .  (I have broken the line)

edit docopt.m and change line 77 in the file
      doccmd = 'netscape';
to
     doccmd = 'mozilla';    or
     doccmd = 'firefox';
```

Programming in Matlab

- A full programming language, if, for,...
- The basic datatype, a double precision matrix in several dimensions. In new Matlab-versions there are more types, such as single precision and integers.
- No type declarations. variables are created when needed.
- Interactive. Partially interpreted.
- New programming style; vector based.
- Object oriented (to some extent).
- Easy to use graphics.
- Can add toolboxes and compiled code.

A tutorial is available. Look under MATLABs help. You can also see the Matlab-book by Jönsson (Swedish).
One should learn to work with vectors and matrices instead of using loops and elements. Shorter, faster and easier to read. It is convenient to write the labs as m-files (instead of typing commands and using the history mechanism).

On the following pages comes a short and fast review of Matlab. There will probably be new things for you as well. Some of the commands below can be performed using the GUI instead.

```
>> v = [1 -5 7 8 -3]  % or comma as delimiter
v =
     1    -5     7     8    -3

>> a = v(2) + v(5)
a =
    -8

>> v(2) = 25
v =
     1    25     7     8    -3

>> v(2) + v(3)
ans =   % default "answer", % = comment
    32

>> who

Your variables are:

a         ans       v

>> sin(v(1))
ans =
    0.8415

>> format short e
>> sin(v(1))
ans =
   8.4147e-01
```

```
>> format long e
>> sin(v(1))
ans =
     8.414709848078965e-01

>> format short
>> sin(v(1))
ans =
    0.8415

>> format bank
>> sin(v(1))
ans =
          0.84

>> format hex
>> sin(v(1))
ans =
   3feaed548f090cee

>> format compact  % for less space

>> help format

 FORMAT Set output format.
  etc.

>> doc format   % opens Matlab's browser
               % (or use the GUI)
```

Note that this changes the output format and not the internal binary representation.

```
>> w = 1:6
w =
     1     2     3     4     5     6

>> w = 1:2:8
w =
     1     3     5     7

>> w = 7:-2:-4
w =
     7     5     3     1    -1    -3

>> w = 7:-2:8
w =
   Empty matrix: 1-by-0

>> 1.5:0.856:6.7  % complex numbers do not work
ans =
 1.5000 2.3560 3.2120 4.0680 4.9240 5.7800 6.6360

>> w = [1; 2; 3]
w =
     1
     2
     3

>> w = [1; 2; 3];   % no printing
>> w
w =
     1
     2
     3
```

```
>> w = 1; 2; 3     % ; separates commands
ans =
     3
>> w
w =
     1

>> a = 1:3; b = 5:7;
>> c = a + b
c =
     6     8    10

>> a = 1:3; b = 5:8;
>> c = a + b
??? Error using ==> plus
Matrix dimensions must agree.

>> size(a)
ans =
     1     3    % size(a, 2) is 3 etc.
>> size(b)
ans =
     1     4

>> b = (5:7)'
b =
     5
     6
     7
```

```
>> c = a + b
??? Error using ==> plus
Matrix dimensions must agree.

>> size(b)
ans =
     3     1

>> a = a’
a =
     1
     2
     3
>> c = a + b
c =
     6
     8
    10

>> sqrt(c’)
ans =
    2.4495    2.8284    3.1623

>> a = 1:3, b = 10 * (3:-1:1)
a =
     1     2     3
b =
    30    20    10

>> a * b
??? Error using ==> mtimes
Inner matrix dimensions must agree.

>> a .* b
ans =
    30    40    30
```

```
>> a ./ b
ans =
    0.0333    0.1000    0.3000

>> a / b   % something different
ans =
    0.0714

>> a .\ b
ans =
   30.0000   10.0000    3.3333

>> a \ b
ans =
         0         0         0
         0         0         0
   10.0000    6.6667    3.3333


>> a \. b
??? a \.
       |
Error: Unexpected MATLAB operator.

>> a .ˆ b
ans =
           1     1048576       59049

>> a.ˆ2 .* b.ˆ3
ans =
       27000       32000        9000

>> 1 + a
ans =
     2     3     4
```

```
>> 1 ./ a
ans =
   1.0000e+00   5.0000e-01   3.3333e-01


>> i
ans =
        0 + 1.0000i
>> j
ans =
        0 + 1.0000i
>> sqrt(-1)
ans =
        0 + 1.0000i

>> q = [1+i 2-3*i 6+6*i]   % 2-3i works as well
q =
 1.0000 + 1.0000i  2.0000 - 3.0000i  6.0000 + 6.0000i

>> q’
ans =
   1.0000 - 1.0000i
   2.0000 + 3.0000i
   6.0000 - 6.0000i

>> q.’
ans =
   1.0000 + 1.0000i
   2.0000 - 3.0000i
   6.0000 + 6.0000i

>> real(q)   % is applied on the whole vector
ans =
     1     2     6
```

```
>> imag(q)
ans =
     1    -3     6

>> abs(q)
ans =
    1.4142    3.6056    8.4853

>> exp(i * pi) % pi is predefined
ans =
  -1.0000 + 0.0000i

>> format short e
>> exp(i * pi)
ans =
 -1.0000e+00 + 1.2246e-16i

>> sqrt(2)ˆ2 - 2
ans =
   4.4409e-16

>> sin(pi)
ans =
   1.2246e-16

>> v = 1:10

v =
  1  2  3  4  5  6  7  8  9  10

>> s = 0;
>> for k = 1:10
     s = s + v(k);
   end
>> s
s =
    55
```

```
>> s = sum(v)  % there is prod as well
s =
    55
```

Matrices

```
>> A = [1 2 3; 4 5 6]
A =
     1     2     3
     4     5     6

>> A’
ans =
     1     4
     2     5
     3     6

>> A(2, 3) = 66
A =
     1     2     3
     4     5    66
```

```
>> A(3, 3) = 9   % A is increased dynamically
A =
     1     2     3
     4     5    66
     0     0     9

>> 1 + A(3, 4)
???  Index exceeds matrix dimensions.

>> A = [1 2; 3 4]
A =
     1     2
     3     4

>> B = [3 4; 1 2]
B =
     3     4
     1     2

>> A * B
ans =
     5     8
    13    20

>> A + B
ans =
     4     6
     4     6

>> A .* B
ans =
     3     8
     3     8
```

```
>> A ./ B
ans =
   3.3333e-01   5.0000e-01
   3.0000e+00   2.0000e+00

>> A .\ B
ans =
   3.0000e+00   2.0000e+00
   3.3333e-01   5.0000e-01

>> A / B   % roughly A * inv(B)
ans =
     0     1
     1     0

>> A \ B   % roughly inv(A) * B
ans =
  -5.0000e+00  -6.0000e+00
   4.0000e+00   5.0000e+00

>> Aˆ2
ans =
     7    10
    15    22

>> A.ˆ2
ans =
     1     4
     9    16

>> A.ˆA
ans =
     1     4
    27   256
```

```
>> sqrt(A)
ans =
   1.0000e+00   1.4142e+00
   1.7321e+00   2.0000e+00

>> sqrt(-A)
ans =
           0 + 1.0000e+00i           0 + 1.4142e+00i
           0 + 1.7321e+00i           0 + 2.0000e+00i

>> R = rand(3)
R =
   9.5013e-01   4.8598e-01   4.5647e-01
   2.3114e-01   8.9130e-01   1.8504e-02
   6.0684e-01   7.6210e-01   8.2141e-01

>> R = rand(3, 2)     % rand
R =
   4.4470e-01   9.2181e-01
   6.1543e-01   7.3821e-01
   7.9194e-01   1.7627e-01

>> R = randn(3, 2)    % NOTE randN
R =
  -1.9790e-02   2.5730e-01
  -1.5672e-01  -1.0565e+00
  -1.6041e+00   1.4151e+00

>> D = diag(1:2:5)  % diag(matrix) returns the
D =                 % diagonal in a vector
     1     0     0
     0     3     0
     0     0     5
```

```
>> D = diag(1:2:5, -1) + diag(1:2:5, 1)
D =
     0     1     0     0
     1     0     3     0
     0     3     0     5
     0     0     5     0

>> I = eye(3)
I =
     1     0     0
     0     1     0
     0     0     1

>> B = magic(3)
B =
     8     1     6
     3     5     7
     4     9     2

>> IB = inv(B)
IB =
   1.4722e-01  -1.4444e-01   6.3889e-02
  -6.1111e-02   2.2222e-02   1.0556e-01
  -1.9444e-02   1.8889e-01  -1.0278e-01

>> B * IB
ans =
   1.0000e+00            0  -1.1102e-16
  -2.7756e-17   1.0000e+00            0
   6.9389e-17            0   1.0000e+00

>> IB * B
ans =
   1.0000e+00            0  -2.7756e-17
            0   1.0000e+00            0
            0   1.1102e-16   1.0000e+00
```

```
>> ones(2, 3)
ans =
     1     1     1
     1     1     1

>> zeros(2)
ans =
     0     0
     0     0

>> S = reshape(1:6, 2, 3)
S =
     1     3     5
     2     4     6

>> sum(S)
ans =
     3     7    11

>> sum(S’)
ans =
     9    12

>> sum(S, 2)
ans =
     9
    12

>> sum(sum(S))
ans =
    21

>> cumsum(1:7)
ans =
     1     3     6    10    15    21    28
```

```
>> M = magic(3)
M =
     8     1     6
     3     5     7
     4     9     2

>> sort(M)
ans =
     3     1     2
     4     5     6
     8     9     7

>> M(:)’
ans =
     8     3     4     1     5     9     6     7     2

>> s = sort(ans)
s =
     1     2     3     4     5     6     7     8     9
```

There are matrices of higher order:

```
>> A1 = [1 2;3 4]
A1 =
     1     2
     3     4

>> A2 = [5 6; 7 8]
A2 =
     5     6
     7     8

>> A(:,:,1) = A1;
>> A(:,:,2) = A2;

>> A
A(:,:,1) =
     1     2
     3     4
A(:,:,2) =
     5     6
     7     8
```

```
               1 ----- 2
              /|      /
             / |     /
            3 ------ 4
               |
               5 ----- 6   ---> 2nd index
              /|      /
             / |     /
            7 ------ 8
           /   |  third index
first index    V
```

```
    (1, 1, 1)  1 ----- 2  (1, 2, 1)
              /|      /
             / |     /
 (2, 1, 1)  3 ------ 4  (2, 2, 1)
               |
    (1, 1, 2)  5 ----- 6  (1, 2, 2)
              /       /
             /       /
 (2, 1, 2)  7 ------ 8  (2, 2, 2)
```

Index vectors

```
>> v = 0.1 + (1:7)
v =
    1.1    2.1    3.1    4.1    5.1    6.1    7.1

>> v(1:3:7)   % 1:3:7 = [1 4 7]
ans =
    1.1    4.1    7.1

>> M = magic(5)
M =
    17    24     1     8    15
    23     5     7    14    16
     4     6    13    20    22
    10    12    19    21     3
    11    18    25     2     9

>> M(:, 2)
ans =
    24
     5
     6
    12
    18

>> M([2 5], :)
ans =
    23     5     7    14    16
    11    18    25     2     9

>> M([2 5], [2 4])
ans =
     5    14
    18     2
```

`end` is practical in constructions like these:

```
>> M(:, end)
ans =
    15
    16
    22
     3
     9

>> M(end, :)
ans =
    11    18    25     2     9

>> M(end, end)
ans =
     9

>> M([1 3], [end-3:end])
ans =
    24     1     8    15
     6    13    20    22
```

An alternative is of course:

```
>> [m, n] = size(M)
m =
     5
n =
     5

>> M(m, :)
ans =
    11    18    25     2     9
```

A bit more original is:

```
>> M(:, [1 1 2])
ans =
    17    17    24
    23    23     5
     4     4     6
    10    10    12
    11    11    18
```

Is used by `meshgrid`.

```
>> x = 1:3
x =
     1     2     3

>> y = -2:0
y =
    -2    -1     0

>> [X, Y] = meshgrid(x, y)
X =
     1     2     3
     1     2     3
     1     2     3
Y =
    -2    -2    -2
    -1    -1    -1
     0     0     0
```

Can be computed this way:

```
>> x = x(:)’; X = x(ones(length(y), 1), :)
X =
     1     2     3
     1     2     3
     1     2     3

>> y = y(:); Y = y(:, ones(1, length(x)));
```

I used this quite often:

```
>> [X, L] = eig(M)
X =
    0.4472   -0.6780   -0.6330    0.0976    0.2619
    0.4472   -0.3223    0.5895    0.3525    0.1732
    0.4472    0.5501   -0.3915    0.5501   -0.3915
    0.4472    0.3525    0.1732   -0.3223    0.5895
    0.4472    0.0976    0.2619   -0.6780   -0.6330

L =
   65.0000         0         0         0         0
         0   21.2768         0         0         0
         0         0  -13.1263         0         0
         0         0         0  -21.2768         0
         0         0         0         0   13.1263

>> [l, pntr] = sort(diag(L))
l =
  -21.2768
  -13.1263
   13.1263
   21.2768
   65.0000

pntr =
     4
     3
     5
     2
     1

>> X = X(:, pntr);
```

`min` can return a pointer vector as well. Suppose we would like to find the row- and column indices for the largest element in a matrix (we assume it is unique).

```
>> M
M =
    17    24     1     8    15
    23     5     7    14    16
     4     6    13    20    22
    10    12    19    21     3
    11    18    25     2     9

>> [col_max, row_p] = max(M)
col_max =
    23    24    25    21    22
row_p =
     2     1     5     4     3

>> [max_M, col_p] = max(col_max)
max_M =
    25
col_p =
     3

>> M(row_p(col_p), col_p)
ans =
    25
```

```
>> M(1:2, 3:4) = M([2 5], [2 4])
M =
    17    24     5    14    15
    23     5    18     2    16
     4     6    13    20    22
    10    12    19    21     3
    11    18    25     2     9

>> A = ones(3, 1) * (1:3)
A =
     1     2     3
     1     2     3
     1     2     3

>> B = A(:, 3:-1:1)
B =
     3     2     1
     3     2     1
     3     2     1

>> A = A’
A =
     1     1     1
     2     2     2
     3     3     3

>> C = A(3:-1:1, :)
C =
     3     3     3
     2     2     2
     1     1     1
```

Logical vectors

```
>> v = 0.1 + (1:7)
v =
    1.1    2.1    3.1    4.1    5.1    6.1    7.1
```

```
>> v > 4
ans =
     0     0     0     1     1     1     1
>> v(v > 4)
ans =
    4.1    5.1    6.1    7.1

>> v([0 0 0 1 1 1 1])
??? Subscript indices must either be real positive
    integers or logicals.

>> v(logical([0 0 0 1 1 1 1]))
ans =
    4.1    5.1    6.1    7.1
```

Logical operators:

```
>> v(2 < v & v < 5)
ans =
    2.1    3.1    4.1

>> v(v <= 2.1 | 6 <= v)
ans =
    1.1    2.1    6.1    7.1
```

Count occurrences

```
>> sum(v ˜= 3.1)    % == equality, ˜= not equal
ans =               % unsafe for floating point
     6

>> any(v ˜= 3.1)
ans =
     1

>> all(v ˜= 3.1)
ans =
     0
```

```
>> all(v ˜= 3.5)
ans =
     1

>> find(v > 4)
ans =
     4     5     6     7

>> M = magic(4)
M =
    16     2     3    13
     5    11    10     8
     9     7     6    12
     4    14    15     1

>> M > 11
ans =
     1     0     0     1
     0     0     0     0
     0     0     0     1
     0     1     1     0

>> M(M > 11)
ans =
    16
    14
    15
    13
    12

>> i = find(M > 11)
i =
     1
     8
    12
    13
    15
```

```
>> m = M(:);

>> m(i)
ans =
    16
    14
    15
    13
    12

>> [j, k] = find(M > 11)
j =
     1
     4
     4
     1
     3
k =
     1
     2
     3
     4
     4
```

Creating matrices from parts

```
>> A = magic(2)
A =
     1     3
     4     2

>> b = [1; 3]
b =
     1
     3
>> C = [A, b; b’, 7]
C =
     1     3     1
     4     2     3
     1     3     7

>> b = (1:3)’;

>> F = [b b(3:-1:1) [b([3 1]); 10]]
F =
     1     3     3
     2     2     1
     3     1    10

>> [F(:, end:-1:1), F’; F(end:-1:1, :), F]
ans =
     3     3     1     1     2     3
     1     2     2     3     2     1
    10     1     3     3     1    10
     3     1    10     1     3     3
     2     2     1     2     2     1
     1     3     3     3     1    10
```

Three dimensional matrices

```
>> A1 = [1 2; 3 4] + 0.1;
>> A2 = [5 6; 7 8] + 0.1;
>> A(:,:,1) = A1;
>> A(:,:,2) = A2;

>> A
A(:,:,1) =
    1.1000    2.1000
    3.1000    4.1000
A(:,:,2) =
    5.1000    6.1000
    7.1000    8.1000

>> A(A > 3)
ans =
    3.1000
    4.1000
    5.1000
    7.1000
    6.1000
    8.1000

>> i = find(A > 3)
i =
     2
     4
     5
     6
     7
     8

>> A(:)’
ans =
1.100  3.100  2.100  4.100  5.100  7.100  6.100  8.100
```

`[i, j, k] = find...` does not do anything useful in this case. Here is an alternative using loops:

```
i = []; j = []; k = [];

for r = 1:size(A, 3)
  [row, col] = find(A(:, :, r) > 3);
  i = [i; row];
  j = [j; col];
  k = [k; r * ones(size(row))];
end

ind = [i, j, k]

ind =
     2     1     1
     2     2     1
     1     1     2
     2     1     2
     1     2     2
     2     2     2

v = [];
for i = 1:6
  v(i) = A(ind(i, 1), ind(i, 2), ind(i, 3));
end

v
>> v
v =
  3.1000  4.1000  5.1000  7.1000  6.1000  8.1000
```

Linear systems

```
>> A = [1 -1 1; 1 2 3; 4 5 6]
A =
     1    -1     1
     1     2     3
     4     5     6
>> b = [0 1 0]'
b =
     0
     1
     0
>> x = A \ b
x =
   -0.9167
   -0.1667
    0.7500
>> r = b - A * x
r =
   1.0e-15 *
    0.1110
         0
         0
```

Cell arrays

An array where the elements can be of different types:

```
>> c{1, 1} = sqrt(2);
>> c{1, 2} = [1 2; 3 4];
>> c{2, 1} = 'Hejsan';
>> c{2, 2} = 1:5;

>> c
c =
    [1.4142e+00]    [2x2 double]
    'Hejsan'        [1x5 double]

>> c{1, 2}(2, 2)
ans =
     4

>> celldisp(c)
c{1,1} =
   1.4142e+00
c{2,1} =
Hejsan
c{1,2} =
     1     2
     3     4
c{2,2} =
     1     2     3     4     5

>> cc={1:3, 'hej'; c, eye(2)}
cc =
    [1x3 double]    'hej'
    {2x2 cell  }    [2x2 double]

>> cc{2,1}{1,2}(1, :)
ans =
     1     2
```

```
>> C = cell(2)
C =
     []     []
     []     []
```

Another data structure where we can store elements of different types is the struct (record, post in Sw). We name the element with a string and not an index.

```
>> s = struct('type', 'circle',                              ..
              'geom',  struct('c', [1 3], 'r', 1.2), ..
              'color', [1 0 0])
s =
     type: 'circle'
     geom: [1x1 struct]
    color: [1 0 0]

>> s.type
ans =
circle

>> s.geom
ans =
    c: [1 3]
    r: 1.2000e+00

>> s.geom.c
ans =
     1     3

>> s.geom.c(2)
ans =
     3
```

```
>> s.color(2) = 1;

>> s.color
ans =
     1     1     0
```

One can have arrays of structs (of the same kind)

```
>> v(1).fn = ’Thomas’;
>> v(1).ln = ’Ericsson’;
>> v(2).fn = ’Anders’;
>> v(2).ln = ’Andersson’;

>> v(3).new = ’oops’    % a new member
v =
1x3 struct array with fields:
    fn
    ln
    new
>> v(1).new         % all the structs in the array
                   % get this new member
ans =
     []
```

There are if-statements etc.

```
>> a = 2.25;
>> if a > 1
     disp(’a > 1’)
   else
     disp(’a <= 1’)
   end

a > 1

>> a = 0.2;
>> if a > 1
     disp(’a > 1’)
   else
     disp(’a <= 1’)
   end

a <= 1

>> help if   % for elseif etc.
```

`&&`, `||` for lazy scalar and, or.

```
>> a = 2.25;
>> if a, disp(’****’), end
****

>> a = 0;
>> if a, disp(’****’), end
```

Handling characters

```
>> s = ’AabcDd’
s =
AabcDd
```

```
>> s + 0   % double(s) works as well
ans =
    65    97    98    99    68   100

>> whos
  Name      Size         Bytes  Class

  s         1x6             12  char array

Grand total is 6 elements using 12 bytes

>> S = [s; s(6:-1:1)]
S =
AabcDd
dDcbaA

>> s = ’sirapiparis’;
>> palin = all(s == s(end:-1:1))
palin =
     1

>> s(1)=’a’;
>> palin = all(s == s(end:-1:1))
palin =
     0

>> s1 = ’ABC’;
>> s2 = ’ 12’;

>> s1 + s2
ans =
    97   115   117

>> char(ans)
ans =
asu
```

Tuning Matlab programs

The timings below are for Matlab R2009b. Matlab 6.5, and later versions, has a JIT-accelerator (Just In Time) which speeds up for-loops etc.

- Use the builtin compiled routines.
  The Matlab-language is interpreted.
- Work on the matrix/vector-level, not on element-level.
  “New” programming style.
- Take care when using the dynamic memory allocation.
  Preallocate.

Some examples:

```
  % Matrix sum. n = 3000 in all examples

  for j = 1:n
    for k = 1:n
      A(j, k) = A(j, k) + B(j, k);
    end
  end
```

Takes 0.11 s.

`A = A + B;` requires 0.018 s.

```
clear A
for k = 1:n
  A(:, k) = x;  % could have different arrays
end
```

Takes 96 seconds.

```
A = zeros(n);   % preallocate
for k = 1:n
  A(:, k) = x;
end
```

Takes 0.05 s.

`W` is a 10000 × 15-matrix and `x` is a column vector having 10000 elements.

```
y = W * W' * x;            y = W * (W' * x);

Takes 2.1 s                0.0006 s
```

Note that it may be impossible just to form `W * W'` even though `y = W * (W' * x);` gives no problem.

M-files and functions

- For short tests we may type commands by hand and use the history mechanism, arrow keys etc. to modify statements. Possible to use emacs-commands on the command line. `Ctrl-a` beginning of line, `Ctrl-e` end of line, `Ctrl-d` remove character, `Ctrl-k` kill (remove) the rest of the line etc. Can match the beginning of a string: `im`↑ press up-arrow, matches line starting with `im`.
  For those using the GUI there is a Command History window, as well.
- For longer tests (assignments) we create an m-file script (or a function) with an editor (e.g. Matlab's own). If the filename is `name.m` we execute the file by typing `name` in Matlab.

Scripts do not take any parameters. Matlab just reads from the file instead of reading commands from the command window. Sometimes functions are more useful or necessary. Here is a simple example. We disregard the fact that Matlab has a function for computing the median. We store the function on the file `median.m` If the name of the function and file are different you have to use the filename to invoke the function.

```
function med = median(v)
% med = median(v) computes the median of
%       the elements in the vector v

  n = length(v);        % number of elements in v
  if n == 0
    med = 0;
  else
    s = sort(v);        % s is local to the function
    if rem(n, 2) == 0
      n2  = n / 2;
      med = 0.5 * (s(n2) + s(n2 + 1));   % even
    else
      med = s((n + 1) / 2);              % odd
    end
  end
```

We can think of the parameters as being passed by “call by value”, but “call by reference” is used for variables that are not changed. We could have written

```
v = sort(v);   % replace v
...
med = 0.5 * (v(n2) + v(n2 + 1));   % even
```

This does not change the array in the calling program. The variables `n`, `n2`, `s` and `med` are local to the function. We give the function a value by giving the return-variable, `med`, a value.

```
>> help median

  med = median(v) computes the median of
        the elements in the vector v

>> v = randn(1, 4)
v =
   -1.8092   -0.6337   -0.4533    0.2840

>> median(v)
ans =
   -0.5435

>> median([v, 5])
ans =
   -0.4533
```

There are several types of functions:

- Anonymous functions (short function not stored in a file)
- Subfunctions (several functions in one file)
- Nested functions (functions inside other functions)
- Overloaded functions (polymorphic functions)
- Private functions (functions in `dir_name/private` are only visible to functions in `dir_name`)

Let us look at the first three types. An anonymous function is created by

```
fhandle = @(argumentlist) expr
```

`expr` is a simple expression and `@` a so-called function handle.

```
>> f = @(x) x .* exp(-x)
f =
    @(x) x .* exp(-x)

>> f([-1 0 1])
ans =
   -2.7183         0    0.3679

>> quadl(f, 0, 1)  % integrate
ans =
    0.2642

>> sin(f(2))
ans =
    0.2674

% A cell array of functions.
% Be careful with blanks. See the manual

>> funcs = {@(x)x.*exp(-x), @(x)x.*sin(-x), ...
            @(x)x.*cos(-x)};

>> for k=1:3, quadl(funcs{k}, 0, 1), end
ans =
    0.2642
ans =
   -0.3012
ans =
    0.3818
```

```
>> comm = @(A, B) A * B - B * A;
>> C = [1 2; 3 4];
>> comm(C, C')
ans =
    -5    -3
    -3     5

% Using "external" variables
>> a = 10;
>> mul_10 = @(z) a * z
mul_10 =
    @(z) a * z

>> mul_10(2)
ans =
    20

>> a = 20;  % does not change the function
>> mul_10(2)
ans =
    20
```

One disadvantage with ordinary m-file functions is that they tend to produce many files. It is possible to put several functions in one file. The first function in the file, the primary function, is visible from outside, but the functions coming after, the subfunctions, are only visible to the primary function or to other subfunctions in the same file. So something like this:

```
function w = f(x, y, z)
 w = x + g(z);
 ...

function s = g(t)
  ...
  s = ...
```

Another alternative is to use nested functions,

```
function w = f(x, y, z)
 w = x + g(z);
 ...

  function s = g(t)
    s = ...
    ...
  end  % necessary

end  % necessary
```

Read more in the manual about scope for variables and functions.
A function can take zero or more input arguments and return zero or more output arguments.

```
function [b_plus_c, sum_A] = func(A, b, c)

  b_plus_c = b + c;
  sum_A    = sum(A(:));


>> F = [1 2;3 4]
F =
     1     2
     3     4
>> h = [1 3]; g = [2 5];

>> [uu, vv] = func(F, h, g)
uu =
     3     8
vv =
    10

>> z = func(F, h, g)
z = 3     8
```

One can choose to ignore output arguments (new in Matlab R2009b):

```
>> [˜, vv] = func(F, h, g)
vv =
    10

>> [uu, ˜] = func(F, h, g)
uu =
     3     8
```

It is also possible to ignore input arguments (will come later).

It is possible to, inside the function, see the number of arguments.

```
  function [out1, out2, out3] = func(in1, in2, in3, in4
  n_in_arg = nargin;
  n_out_arg = nargout;

  if n_in_arg == 4
    ...
  elseif n_in_arg == 3
   ...
etc.
```

It is possible to have optional input (output) parameters, so the number of parameters of a function may change between calls. See the documentation for `varargin` and `varargout` for details.

Global variables

Variables in functions are local to the function. We use the parameters to communicate with other routines. Another way is to use global variables.

```
>> global a b   % In Matlab, or the calling routine
>> type func

function func

global a b      % A matching global declaration

a = a + 1;
b = b * 10;

>> a = 1; b = 2;
>> func
>> a

a =
     2

>> b

b =
    20
```

Persistent variables

A variable which is local to a function does not keep its value between calls. To make it keep the value, we use a persistent declaration. A persistent variable is initialised to the empty matrix.

```
>> type pers
function num_calls = pers
  persistent k  % persistent num_calls does not work

  if isempty(k)
    k = 0;
  end

  k = k + 1;

  num_calls = k;

>> pers
ans =
     1
>> pers
ans =
     2
>> pers
ans =
     3


>> clear pers
>> pers
ans =
     1
```

A few tips

Debugging: there is a Matlab-debugger, but it is usually sufficient to remove semi-colons (to print variables).
The `keyboard`-command is convenient when we want to stop in functions. Resume execution by typing the letters `return`.

```
>> y = cos(0)
y =
     1
>> cos = 8
cos =
     8
>> y = cos(0)
??? Subscript indices must either be real
    positive integers or logicals.

>> which cos
cos is a variable.  % checks variable first
                    % then function
>> clear cos        % remove definition

>> which cos
built-in (/chalmers/sw/ ... /cos)  % double method

% Even more amusing
>> cos = 1:4
cos =
     1     2     3     4

>> cos(1)
ans =
     1
```

The `clear`-command takes several parameters. Here are a few. For a full description, see the documentation.

`clear` removes all variables from the workspace.
`clear variables` does the same thing.
`clear global` removes all global variables.
`clear functions` removes all compiled M- and MEX-functions.
`clear all` removes all variables, globals, functions and MEX links.
`clear var1 var2 ...` clears the variables specified.
`clear fun` clears the function specified.
Clear does not affect the amount of memory allocated to the Matlab process under unix.

Some commands have been written in C while others are m-files,

```
>> type cos
cos is a built-in function.

>> which ls
/chalmers/sw/sup/matlab-7.1/toolbox/matlab/general/ls.m

>> type ls   % lists the m-file (not included)
>> dir       % (DOS-command) faster
```

More unix-like stuff. `cd`, `path` etc. `matlab` and `VIS` are directories.

```
                   /users/math/thomas
                   /        |        \
              visual.m    matlab     VIS
                                      |
                                   visual.m

>> cd ~             % ~ home dir
>> cd               % print current directory
/users/math/thomas

>> pwd              % an alternative
ans =
/users/math/thomas

>> which visual              % one visual.m here
/users/math/thomas/visual.m

>> cd matlab
>> pwd
ans =
/users/math/thomas/matlab
```

```
>> which visual              % but no one here
visual not found.

>> cd ../VIS

>> pwd
ans =
/users/math/thomas/VIS

>> which visual
/users/math/thomas/VIS/visual.m  % and one here

>> cd ../matlab
>> which visual
visual not found.

>> path(path, '../VIS')
>> which visual
../VIS/visual.

>> path    % lists the path

        MATLABPATH

/users/math/thomas/matlab
/chalmers/sw/sup/matlab-7.1/toolbox/matlab/general
/chalmers/sw/sup/matlab-7.1/toolbox/matlab/ops
/chalmers/sw/sup/matlab-7.1/toolbox/matlab/lang
/chalmers/sw/sup/matlab-7.1/toolbox/matlab/elmat

etc.
```

Handling files

`save filename` saves all workspace variables to a binary file **`filename.mat`**. The data may be retrieved with **`load`**.

If **`filename`** has no extension, **`.mat`** is assumed. **`save`**, by itself, creates **`matlab.mat`**.

`save filename var` saves only **`var`**.
`save filename var1 var2 var3` saves only **`var1`**, **`var2`** and **`var3`**.

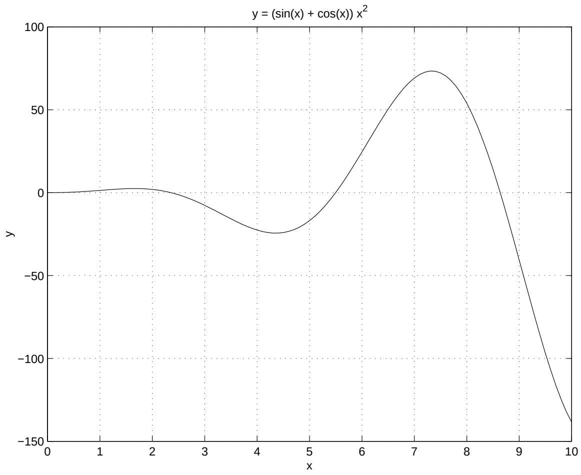
`save filename var -ascii` or **`save -ascii filename var`** saves in human readable form, 8-digit ASCII.
`save -ascii -double filename var` saves in 16-digit ASCII.

If one needs more control over the format, **`fprintf`** can be used. This routine accepts the same type of formatting codes as C. **`fscanf`** is a more general routine for reading data. **`help iofun`** gives a long list of I/O-related routines.

Graphics in Matlab

I assume you have seen some basic plotting, but here comes a few simple examples. First 2D-plots:

```
>> x = 0:0.1:10; % or linspace
>> plot(x, (sin(x) + cos(x)) * x.^2)
>> grid on
>> xlabel('x')
>> ylabel('y')
>> title('y = (sin(x) + cos(x)) x^2')
```

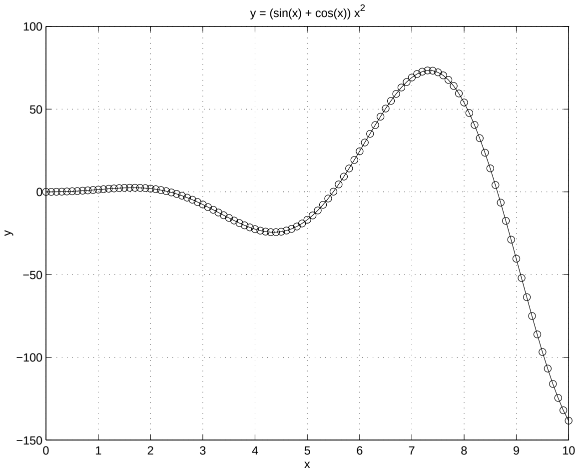


I usually make the lines thicker, increase the size of numbers and letters when showing transparencies, but we have not learnt that yet, and I do not want to give the wrong impression. So that is why it is hard to read the text in the plots.

```
>> hold on
>> plot(x, (sin(x) + cos(x)) * x.^2, 'o')
>> help plot

 PLOT   Linear plot.
    PLOT(X,Y) plots vector X versus vector Y.
  ....
```

An alternative to hold on/off:

```
>> y = (sin(x) + cos(x)) * x.^2;
>> plot(x, y, '-', x, y, 'o')
```



A good sequence is:

```
figure(1)  % create otherwise put on top
hold off
plot...
hold on
plot ...

figure(2) % makes a new figure window
```

`plot(1, 1:3, ’o’)`is equivalent to `plot([1 1 1], 1:3, ’o’)`
`plot(1:3, 1, ’o’)`is equivalent to `plot(1:3, [1 1 1], ’o’)`

`plot(x, y)` works as expected even if `x` is a row vector and `y` is a column vector (or vice-versa).

If `x` is a vector and `Y` is a matrix, then `plot(x, Y)` is equivalent to plotting `plot(x, Y(:, 1))`, ..., `plot(x, Y(:, end))` or `plot(x, Y(1, :))`, ..., `plot(x, Y(end, :))` whichever lines up. If the matrix is quadratic, the columns are used.

`x` cannot be a scalar.

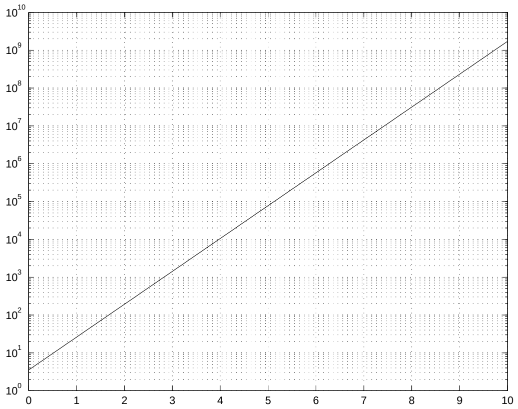
It is analogous for `plot(X, y)`.

`plot(X, Y)` where also `X` is a matrix plots `Y(:, k)` as a function of `X(:, k)`.

Do not forget the x-values. `plot(y)` is equivalent to `plot(1:length(y), y)`

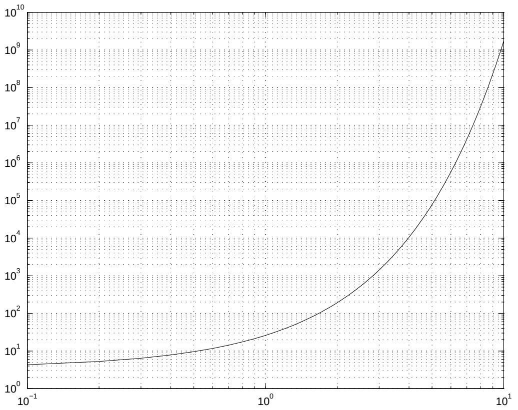
`plot(Y)` is equivalent to `plot(1:m, Y(:, 1))`, ..., `plot(1:m, Y(:, end))`where `m = size(Y, 1)`.

```
>> x = 0:0.1:10;
>> y = 3.52441 * exp(2 * x);
>> semilogy(x, y)
>> grid on
>> print -deps semilogy.eps

% head semilogy.eps   In unix
%!PS-Adobe-2.0 EPSF-1.2
%%Creator: MATLAB, The Mathworks, Inc.
%%Title: semilogy.eps
%%CreationDate: 09/01/ 0  22:34:29
%%DocumentNeededFonts: Helvetica
%%DocumentProcessColors: Cyan Magenta Yellow Black
%%Pages: 1
%%BoundingBox:    66   210   548   592    <--- NOTE
%%EndComments
```



Do not use screen dumps (raster images, gif, jpeg or similar) for simple plots (necessary of you add light).

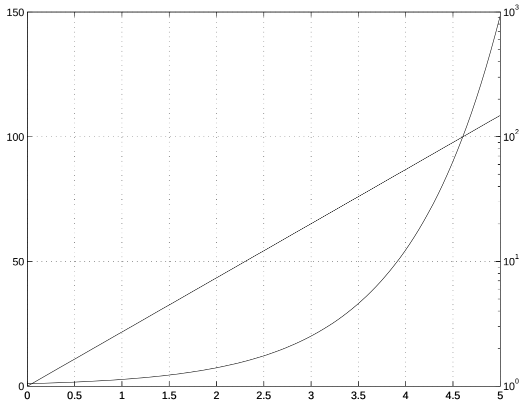
```
>> loglog(x, y)
>> grid on
```



One can plot with different styles, e.g. `plot(x, y, ’r:’)`, plot with a red dotted line. Type `help plot` for details.

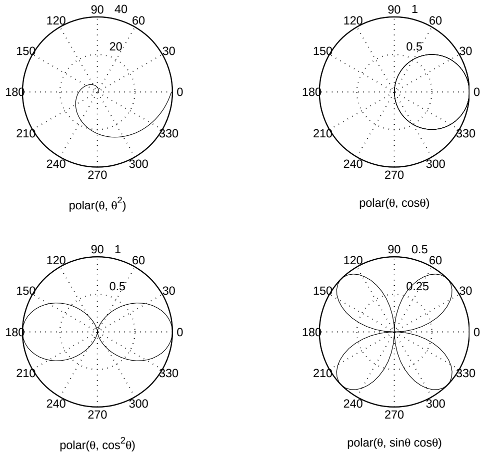
Two y-axes.

```
>> x = linspace(0, 5);
>> y = exp(x);
>> plotyy(x, y, x, y, ’plot’, ’semilogy’)
>> grid on
```



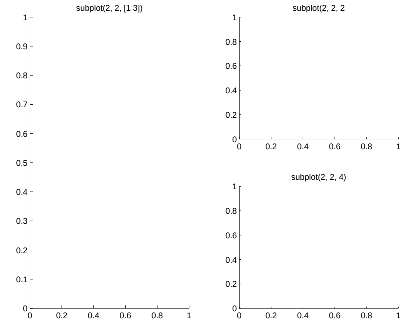
Polar coordinates and subplot

```
theta = linspace(0, 2* pi);
subplot(2, 2, 1)    % 2 x 2-matrix of plots
polar(theta, theta.^2)
xlabel('polar(\theta, \theta^2)')

subplot(2, 2, 2)
polar(theta, cos(theta))
xlabel('polar(\theta, cos\theta)')

subplot(2, 2, 3)
polar(theta, cos(theta).^2)
xlabel('polar(\theta, cos^2\theta)')

subplot(2, 2, 4)
polar(theta, sin(theta)*cos(theta))
xlabel('polar(\theta, sin\theta cos\theta)')
```

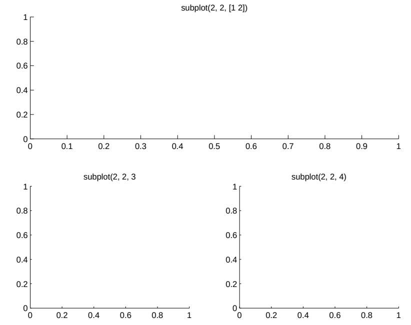


Here are two other ways to use `subplot`.

```
subplot(2, 2, [1 3]), title('subplot(2, 2, [1 3])')
subplot(2, 2, 2),     title('subplot(2, 2, 2')
subplot(2, 2, 4),     title('subplot(2, 2, 4)')
```



```
subplot(2, 2, [1 2]), title('subplot(2, 2, [1 2])')
subplot(2, 2, 3),     title('subplot(2, 2, 3')
subplot(2, 2, 4),     title('subplot(2, 2, 4)')
```



Matlab understands simple TeX/LaTeX-expressions such as:

```
Greek letters: \alpha, \beta, ..., \Gamma, \Xi

Index: \alpha_2^3, \alpha^{m + n}

Integrals: \int_a^b f(x) dx
```

Here is the LaTeX-code:

$$\alpha, \beta, ..., \Gamma, \Xi$$

$$\alpha_2^3, \alpha^{m+n}$$

$$\int_a^b f(x)dx$$

Matlab cannot cope with more complicate expressions, such as:

```
\sum_{k=0}^{n-1}\ ax^k = a\ \frac{x^n-1}{x-1},\ x\ne 1
```

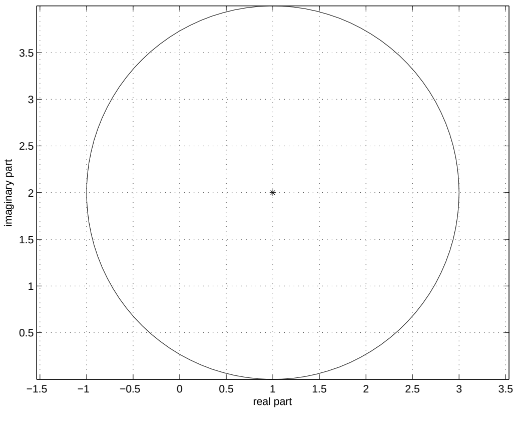
$$\sum_{k=0}^{n-1} ax^k = a\ \frac{x^n-1}{x-1},\ x \ne 1$$

unless ones changes the string's **`Interpreter`**-property to **`latex`** and surrounds the string with **`$ $`**.
It does not always seem to work properly though, I had problems with minus-signs, for example.

`text` and **`gtext`** can be used to place text in a plot (as can the menu in the plot window). **`ginput`** can be used to read the position of the mouse.

One can plot complex numbers

```
>> theta = linspace(0, 2* pi);
>> iu = sqrt(-1)
iu =
       0+ 1.0000i
>> circle = 1 + 2 * iu + 2 * exp(iu * theta);
>> plot(circle)
>> axis equal % or axis('equal'). NOT axis square
>> grid on
>> hold
Current plot held
>> plot(1 + 2 * iu, '*')
>> xlabel('real part')
>> ylabel('imaginary part')
```

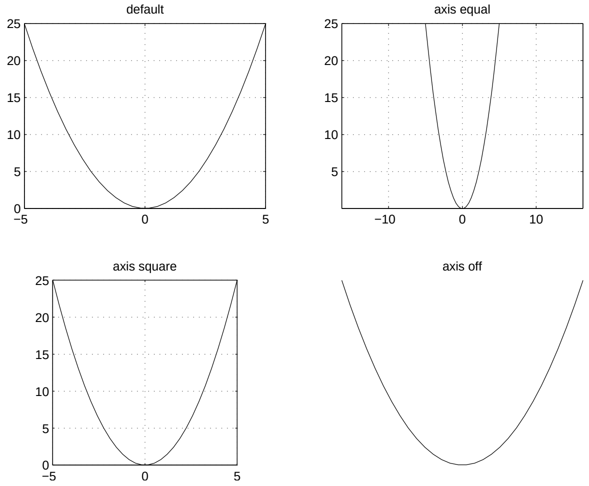


A word on the `axis` command.
`axis equal` gives correct scaling, a circle does not look like an ellipse and a sphere not like an ellipsoid. Not to be confused with
with `axis square` makes the axis box square (regardless of the extent in x and y).
`axis vis3d` freezes the aspect ratio so that plot is not deformed during rotation.
`axis off` turns off axis. `axis on` turns on axis.
It is possible to set the axis limits:

```
axis([xmin xmax ymin ymax])
axis([xmin xmax ymin ymax zmin zmax].)
```

There are eight more options, type `help axis` or `doc axis` for more (all) details.

An example: $y = x^2, |x| \leq 5$.

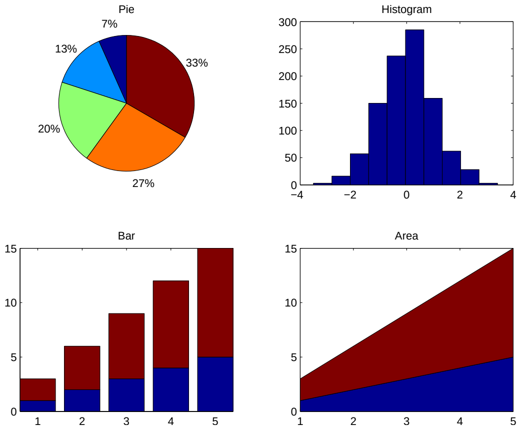


Some business graphics

Matlab can produce, bar- and area graphs. `help bar`, `help area`. There are pie charts and histograms (`help pie`, `help hist`) and a few others. Read the documentation to see the available options.

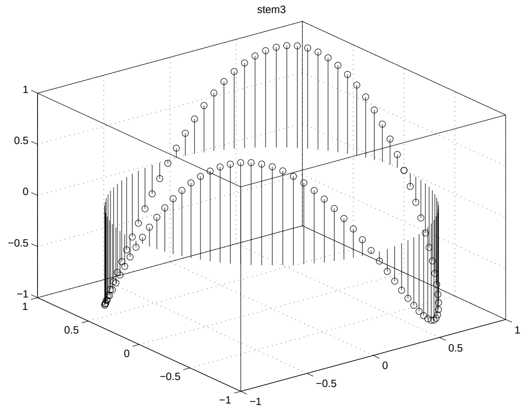
This code produces the plot on the next page:

```
figure(1)
subplot(221)
pie(1:5)
title('Pie')

subplot(222)
hist(randn(1000, 1))
title('Histogram')

subplot(223)
bar(1:5, [(1:5)', 2*(1:5)'], 'stacked')
axis tight
title('Bar')

subplot(224)
x = (1:5)';
Y = [(1:5)', 2*(1:5)'];
area(x, Y)
title('Area')
```



Here is a plot made by `stem3`:

```
>> phi = linspace(0, 2 * pi);
>> stem3(cos(phi), sin(phi), sin(2* phi))
>> title('stem3')
```



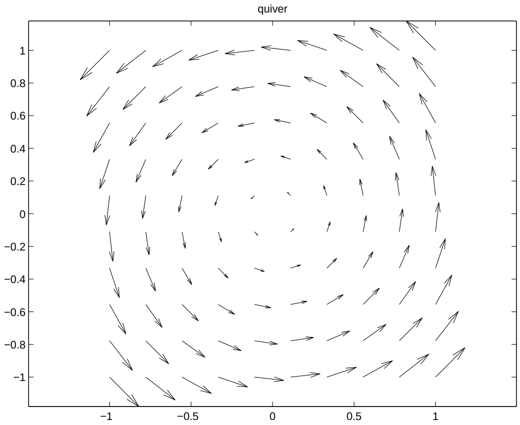
Matlab has several commands for drawing arrows, `compass`, `feather`, `quiver` and `quiver3`. These are used e.g. when drawing flow fields. Here is a `quiver`-example.

We start by creating a grid in the x-y-plane, using the `meshgrid`-command. First a word on how `meshgrid` works:

```
>> [X, Y] = meshgrid(linspace(-1, 1, 3))
X =
    -1     0     1
    -1     0     1
    -1     0     1

Y =
    -1    -1    -1
     0     0     0
     1     1     1
```

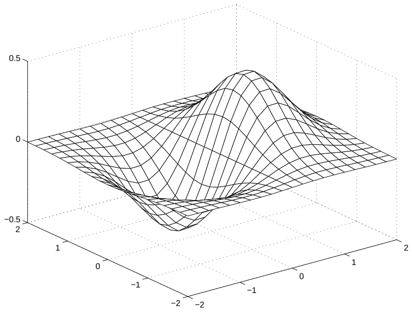
`[X, Y] = meshgrid(xvec, y_vec);` is another alternative. In this example we draw an arrow, [u, v], that is orthogonal to the vector going from the origin to [x, y]. it should have the same length as well. So one choice is taking [u, v]=[-y, x]. here is the code:

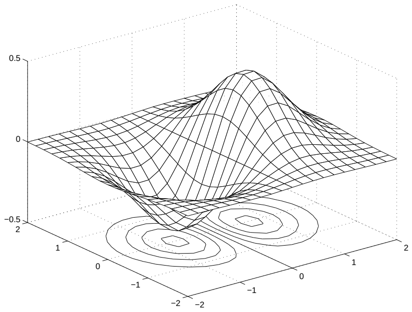
```
>> [X, Y] = meshgrid(linspace(-1, 1, 10));
>> quiver(X, Y, -Y, X)
>> axis equal
>> title('quiver')
```



The **`meshgrid`**-command is used when drawing simple surfaces as well, such as when we have a function $z = f(x, y)$. Here is an example.

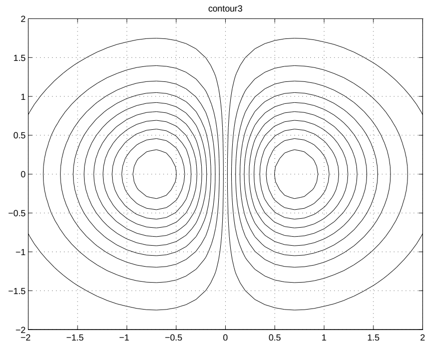
```
>> [X,Y] = meshgrid(-2:0.2:2);
>> Z = X .* exp(-X.^2 - Y.^2);  % Note elementwise
>> figure         % new plotwindow
>> mesh(X, Y, Z)
>> figure
>> meshc(X, Y, Z)  % Note the c in meshc
```



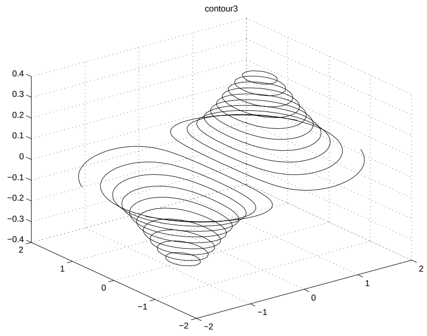


`meshc` draws the surface and contour lines, i.e. curves in the x-y-plane where $f(x, y)$ is constant. It is possible to just draw the contours using the command **`contour(X, Y, Z)`**. One can specify the number of contour lines or give the exact values where a contour line should be drawn. using **`contour3`** it is possible to put a contour line at the correct z-level.

```
>> [X,Y] = meshgrid(-2:0.1:2);
>> Z = X .* exp(-X.^2 - Y.^2);
>> contour(X, Y, Z,  20, 'k')
>> grid on
>> title('contour')
```

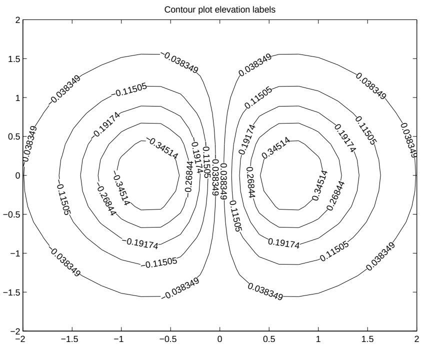


```
>> contour3(X, Y, Z,  20, 'k')
>> title('contour3')
```



One can label the contour lines

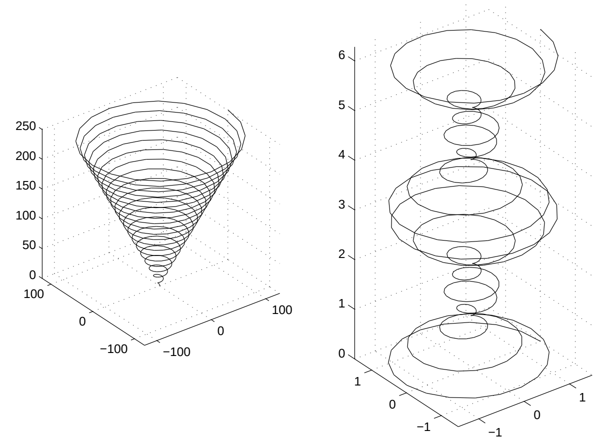
```
>> [C, h] = contour(X, Y, Z,  10, 'k');
>> clabel(C, h)
>> title('Contour plot elevation labels')
```



A rather nice contour-command is **`contourf`**, which fills the area between contour lines with different colours. Try it!

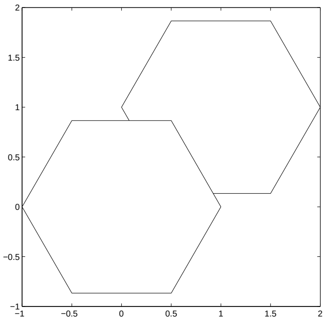
Lines in 3D:

```
>> phi = linspace(0, 40 * pi, 400);
>> subplot(121)
>> plot3(phi .* cos(phi), phi .* sin(phi), 2 * phi)
>> axis equal
>> grid on
>>
>> subplot(122)
>> r = 0.5 + cos(0.1 * phi);
>> plot3(r .* cos(phi), r .* sin(phi), phi / 20)
>> axis equal
>> grid on
```



Polygons

```
>> phi = linspace(0, 2 * pi, 7);
>> c = exp(sqrt(-1) * phi(1:end-1));  % need not close
>> x = real(c)’;
>> y = imag(c)’;
>> X = [1+x, x];  % one polygon per column
>> Y = [1+y, y];
>> fill(X, Y, ’w’)  % not to waste tuner
>> axis([-1 2 -1 2])
>> axis square
```

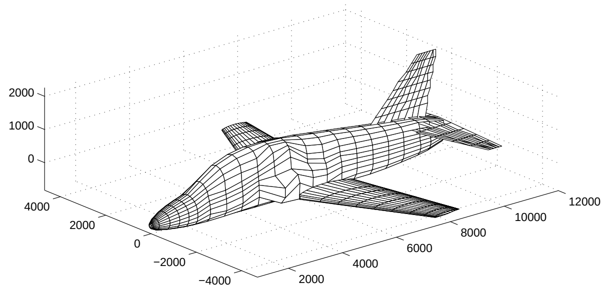


The second polygon is painted on top of the first. To see the edges we can do (much more about such things later):

```
>> h = fill(X, Y, ’w’)          % a vector of handles
>> set(h, ’FaceColor’,’None’);  % change the FaceColor-
                                % property
```

Polygons in 3D

```
>> C = Z;
>> fill3(X, Y, Z, ’w’)
>> grid on
>> axis equal
>> view([320, 20])
>> whos
  Name      Size         Bytes  Class

  C         4x1214       38848  double array
  X         4x1214       38848  double array
  Y         4x1214       38848  double array
  Z         4x1214       38848  double array

Grand total is 19424 elements using 155392 bytes
```

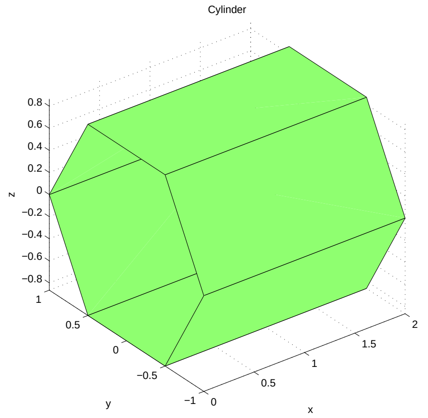


```
>> fill3(X, Y, Z, C)
```

The **surf**-command.
We have used the **mesh**-command to draw surfaces (arising from $z = f(x, y)$). When we have more general surfaces the mesh-command may not work, the surface may not be the graph of a function. The sphere is a simple example, $z = \pm\sqrt{1 - x^2 - y^2}$ does not define a function from (x, y) to z, although $z = \sqrt{1 - x^2 - y^2}$ and $z = -\sqrt{1 - x^2 - y^2}$ do. So one (perhaps not very good) way to draw a sphere is to use the mesh-command twice.

Another example is given by the following cylinder. The cylinder is centered on the x-axis and has an hexagonal cross-section.

```
>> phi = linspace(0, 2*pi, 7); % 6 corners; must close
>> surf([zeros(1,7); 2*ones(1,7)], [1;1]*cos(phi), ...
        [1;1]*sin(phi), ones(1,7)) % we could transpose
>> axis equal                      % everything
>> xlabel(’x’); ylabel(’y’); zlabel(’z’)
>> title(’Cylinder’)
```



To understand this better we can read the documentation. This is a quote from the manual:

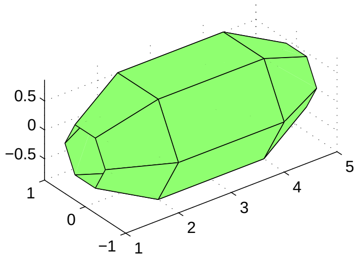
Algorithm
Abstractly, a parametric surface is parametrized by two independent variables, i and j, which vary continuously over a rectangle; for example, $1 \leq i \leq m$ and $1 \leq j \leq n$. The three functions $x(i,j)$, $y(i,j)$, and $z(i,j)$ specify the surface. When i and j are integer values, they define a rectangular grid with integer grid points. The functions $x(i,j)$, $y(i,j)$, and $z(i,j)$ become three $m \times n$matrices, X, Y, and Z. Surface color is a fourth function, $c(i,j)$, denoted by matrix C.
Each point in the rectangular grid can be thought of as connected to its four nearest neighbors.

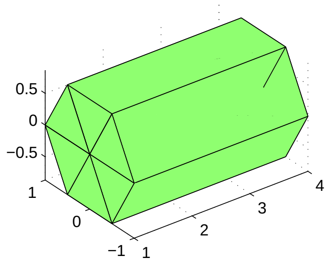
```
           i-1,j
             |
  i,j-1 - i,j - i,j+1
             |
           i+1,j
```

This underlying rectangular grid induces four-sided patches on the surface. To express this another way, $[X(:)\ \ Y(:)\ \ Z(:)]$ returns a list of triples specifying points in 3-space. Each interior point is connected to the four neighbors inherited from the matrix indexing. Points on the edge of the surface have three neighbors; the four points at the corners of the grid have only two neighbors. This defines a mesh of quadrilaterals or a quad-mesh.

Let us take the cylinder and close the ends. First an example where the ends are partially closed. Just to show that it is possible, we transpose all the arrays. Here

```
>> phi = linspace(0, 2*pi, 7)’; % Note transpose
>> z = zeros(7, 1); o = ones(7, 1);
>> c = cos(phi);    s = sin(phi);
>> subplot(211)
>> surf([o 2*o 4*o 5*o], [0.5*c c c 0.5*c], ...
        [0.5*s s s 0.5*s], ones(7,4))
>> axis equal
>> subplot(212)
>> surf([o o 4*o 4*o], [z c c z], [z s s z], ones(7,4))
>> axis equal
```





The line in the right part is not visible on the monitor.

It is possible to draw the cylinders using the **`fill3`**-command as well. That would, however, require more points.

The first cylinder we drew was defined by 14 points (two times seven edges). Using polygons we would need 24 points (6 polygons with four corners).

When we come to shading (colouring polygons with light present) we will notice a difference as well (with the normals). Six polygons are six different objects while the **`surf`**-cylinder is one object. We have 24 normals for the polygons and 14 for the **`surf`**-command.

Polygons do not have to be planar (all point in a plane). Consider the following polygon with four corners:

```
>> X = [0 1 1 0]’;
>> Y = [0 0 1 1]’;
>> Z = [0 0 0 1]’;
>> C = ones(size(X));
>> h = fill3(X, Y, Z, C)
```

Matlab breaks up till polygon in two triangles (i.e. two planar polygons). This is a special case of tessellation:

Etymology: Late Latin tessellatus, past participle of tessellare to pave with tesserae, from Latin tessella, diminutive of tessera : to form into or adorn with mosaic

There is an image-toolbox. Here I am covering a surface with an image (usually called a texture, in this context, and the process is called texture-mapping). We will be using textures in the OpnGL-lab.

```
>> B = imread(’te.jpg’, ’jpg’);
>> image(B)    % to look at the image
>> axis image  % correct scaling
>> [X, Y] = meshgrid(linspace(-1, 1, 10));
>> warp(X, Y, (X.ˆ2 - Y.ˆ2) * cos(0.1 * Y), B)
>> axis off
```

In the upper part of the windows there are buttons for zooming, rotation etc. Have a look at the Tools- and View-menus as well. Some of the remaining buttons and menus are used for editing an image (adding text, arrows etc).

There are several other plot-commands, but before we get back to those we need to have a look at Matlab’s handle graphics.

Handle graphics

Plots, windows, polygons etc. are stored in a tree structure. The windows (figure) are child-nodes to the Root (which can think of as the screen: it is created by Matlab and contains data).
The axes is a child of a figure-window and the plot data is a child of the axes etc. Each node has a set of attributes, properties that can take different values.
A "figure" has a Color-property which is the colour around the drawing area in the window. The standard value of this colour is the RGB-vector `[0.8 0.8 0.8]`. Here is the tree, there are hundreds of properties in total.

```
                  Root
                    |
UI objects   ---- Figure ---- Hidden Annotation Objects
                    |                   |
                    |               Annotation Objects
                    |
Core objects ---- Axes ---- Group objects
                    |
              Plot Objects
```

This layout is new for Matlab v7, in previous versions there were less talk about objects. A figure is a window in which the graphics is displayed. Figures contain menus, toolbars, user-interface objects (e.g. buttons and sliders), context menus (a menu bound to a curve for example), axes.
Annotation objects are things like arrows, rectangles and are usually created using the builtin plot editor.
Core objects are axes, image, light, line, patch, rectangle, surface, text.
Groups objects can be used to collectively refer to several axes, for example.
Plot Objects group together core objects. We will not look at all the objects in detail, so what follows is a simplified presentation. The manual contains more than 120 pages on the subject.

Let us look at an example. We have just started Matlab (say) and have typed the following commands:

```
>> format compact
>> x = 1:10;
>> plot(x, x.^2, 'r')
>> grid on
>> xlabel('x')
>> ylabel('y')
>> title('y = x^2')
```

The tree is linked together by handles (pointers). They are of type double and usually have many decimals. The handles of the figure windows are positive integers and the Root has handle zero. Using the function `get` we can access the value of a property for an object pointed to by the handle:
`get(handle, 'PropertyName')`
`set(handle, 'PropertyName', value)`sets the value.

Some properties are read only. `get(handle)` prints the values of all the properties and `set(handle)` displays all property names and their possible values for the object.

Let us start to inspect the Root. I have (usually) edited the output to make it shorter. My comments after `%`.
This is about a third of what is printed.

```
>> get(0)
  CurrentFigure = [1]            % figure 1
  Diary = off
  DiaryFile = diary
  FixedWidthFontName = Courier New
  Format = short                 % set with format
  FormatSpacing = compact        % set with format
  ScreenDepth = [24]
  ScreenSize = [1 1 1280 1024]
  Units = pixels
  Children = [1]                 % The figure window
```

To see the possible alternatives for `Format`, we can do

```
>> set(0, 'Format')
[ short | long | shortE | longE | shortG | longG | hex
  bank | + | rational | debug | shortEng | longEng ]
```

and to change format to `longE` we can

```
>> set(0, 'Format', 'longE')
>> pi
ans =
     3.141592653589793e+00
```

Usually we would instead type the shorter:

```
>> format long e
```

Property names are not case sensitive and we can shorten the name as long as it becomes unique.

```
>> set(0, 'uNiTs', 'centimeters')
>> set(0, 'units', 'centimeters')
>> set(0, 'uni', 'centimeters')
>> set(0, 'u', 'centimeters')
??? Error using ==> set
Ambiguous root property: 'u'.
```

Let us now look at the Children of the Root. We have only one child, the figure window. Since the handle is an integer we need not fetch it, but I have done so just to show how `get` works.

```
>> hf = get(0, 'Children')   % hf for handle to figure
hf =
     1
```

Here are a few of the properties:

```
>> get(hf)
  Color = [0.8 0.8 0.8]     % border colour
  Colormap = [ (64 by 3) double array]
  CurrentAxes = [153.009]   % handle to the axes
  DoubleBuffer = on         % for animation
  IntegerHandle = on        % figure 1 has handle 1
```

```
  NumberTitle = on          % Figure 1, 2 ...
  Renderer = painters       % Hidden lines removal
  Resize = on              % Can freeze the size
  WindowButtonDownFcn =     % A Callback
  WindowButtonMotionFcn =   % Another
  WindowButtonUpFcn =       % Another
  ButtonDownFcn =           % Click over an object
  Children = [153.009]      % Same as axes
  CreateFcn =               % More callbacks
  DeleteFcn =
  Parent = [0]              % Root
  Tag =                    % For us
  UserData = []             % For us
  Visible = on              % Can hide the window
>> set(1, 'Color', [1 0 0]) % change to red
```

Let us now look at the axes. Sometime it is inconvenient to go down in the tree this way so there are functions that gives the handles directly.

```
 gca  Get handle to current axis.
 gcf  Get handle to current figure.
 gcbo Get handle to current callback object.
 gco  Get handle to current object.
 gcbf Get handle to current callback figure.
```

So

```
>> gcf
ans = 1

>> get(get(0, 'Child'), 'Child')
ans = 1.530087890625000e+02

>> gca
ans = 1.530087890625000e+02

>> ha = get(1, 'Children')
ha = 1.530087890625000e+02  % Don't write the decimals
```

```
>> get(ha)
  AmbientLightColor = [1 1 1]
  Box = on
  CameraPosition = [5.5 50 17.3205]
  CameraUpVector = [0 1 0]
  CLim = [0 1]
  FontAngle = normal
  FontName = Helvetica
  FontSize = [10]
  FontUnits = points
  FontWeight = normal
  GridLineStyle = :
  LineWidth = [0.5]
  NextPlot = replace
  Projection = orthographic
  Position = [0.13 0.11 0.775 0.815] % llx, lly, w, h
  Title = [160.016]         % made with title command
  XLabel = [155.018]        % made with xlabel
  XTick = [ (1 by 10) double array]
  XTickLabel = 1 2 3 4 5 6 7 8 9 10
  Children = [154.088]      % the plot data
   ...
```

Let us make the grid- and axle lines wider (not the curve), use a larger font for the ticks

```
>> set(ha, ’LineWidth’, 2, ’FontSize’, 16, ...
           ’FontWeight’, ’Bold’)
```

The title is hardly readable so lets make that larger as well:

```
>> set(get(ha, ’Title’))
  FontAngle: [ {normal} | italic | oblique ]
  FontName
  FontSize
  FontWeight: [ light | {normal} | demi | bold ]
  HorizontalAlignment: [ {left} | center | right ]
```

```
>> get(get(ha, ’Title’), ’String’)
ans =
y = x^2

>> set(get(ha, ’Title’), ’Fontsize’, 16)
```

This is not so convenient, so many commands can set the properties directly.

```
>> title(’y = x^2’, ’Fontsize’,16, ’Fontweight’,’Bold’)
```

We have one level left in the tree. Let us look at a leaf (terminal node), the child to the axes.

```
>> hp = get(ha, ’Children’)
hp =
    1.540881347656250e+02
>> get(hp)
           Color: [1 0 0]
       LineStyle: ’-’
       LineWidth: 5.000000000000000e-01
          Marker: ’none’
      MarkerSize: 6                  % useful
           XData: [1 2 3 4 5 6 7 8 9 10]
           YData: [1 4 9 16 25 36 49 64 81 100]
           ZData: [1x0 double]       % empty
   ButtonDownFcn: []
        Children: [0x1 double]       % no child
            Type: ’line’
   UIContextMenu: []
        UserData: []
         Visible: ’on’
          Parent: 1.530087890625000e+02

>> set(hp)
ans =
             LineStyle: {5x1 cell}
                Marker: {14x1 cell}
...
```

```
>> set(hp, ’LineStyle’)
[ {-} | -- | : | -. | none ]  % {-} the current
>> set(hp, ’Marker’)
[ + | o | * | . | x | square | diamond | v | ^ | > | <
 pentagram | hexagram | {none} ]
```

I can change one point on the curve by typing:

```
>> y = get(hp, ’Ydata’)
y =
  1  4  9 16 25 36 49 64 81 100
>> y(3) = 100;
>> set(hp, ’Ydata’, y)
```

I can change the line width, but I would usually do it using the plot-command. The curve replaces the old one. The plot-function returns the handle.

```
>> hp = plot(x, x.^2, ’r’, ’LineWidth’, 2)
hp =
     1.540887451171875e+02

>> get(gca, ’Child’)    % A new child
ans =
     1.540887451171875e+02

>> delete(hp)           % deletes the curve
>> get(gca, ’Child’)
ans =
   Empty matrix: 0-by-1 % no child
```

It can be convenient to use structures:

```
>> prop.LineWidth = 3;
>> prop.Color = [1 0 0]
prop =
    LineWidth: 3
        Color: [1 0 0]

>> set(h1, prop)
>> set(h2, prop)
```

`str = get(handle);`returns a structure in `str`.
The field names are the property names and the field values are the corresponding values of the properties.

Properties have default, factory, values. We can see the 589 of them by typing `get(0, ’factory’)`(only for the root). Matlab searches for a value beginning with the current object, going up in the tree until a user-defined or factory-defined value is found. We can define our own default values, which will affect objects after the change. Say that we would like to increase the font size for axes and text, say `xlabel`, in general.

```
>> diary factory            % diary filename
>> get(0, ’factory’)
>> diary off
>> !grep -i font factory  % unix; edited
              factoryAxesFontAngle: ’normal’
               factoryAxesFontName: ’Helvetica’
axes, xlabel   factoryAxesFontSize: 10
title         factoryAxesFontUnits: ’points’
             factoryAxesFontWeight: ’normal’
              factoryTextFontAngle: ’normal’
               factoryTextFontName: ’Helvetica’
in plot area   factoryTextFontSize: 10
              factoryTextFontUnits: ’points’
             factoryTextFontWeight: ’normal’
```

```
>> figure(1)
% Change Factory (in the name) to Default, to set the
% default value. This works for plots in figure 1 only.
>> set(1, 'DefaultAxesFontSize', 16,  ...
          'DefaultTextFontSize', 16)

% This works for all windows,
>> set(0, 'DefaultAxesFontSize', 16, ...
          'DefaultTextFontSize', 16)
```

To keep the defaults one can save them in `~/matlab/startup.m` which is executed when Matlab starts.
`reset(handle)` resets the values, of the object, to the factory defaults. (so `DefaultAxesFontSize` is set to 10).
To reset (remove) a specific default property, type
`set(0, 'DefaultAxesFontSize', 'remove')` for example.

Sometimes we get arrays with handles:

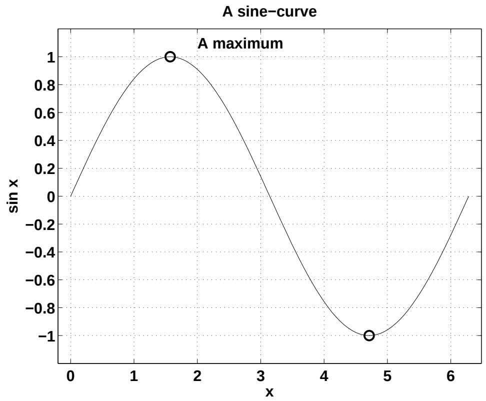
```
>> hp = plot(x, x.^2, 'k-', x, x.^2, 'ro')
hp =
   1.5400e+02
   1.5500e+02

>> get(hp, 'Type')
ans =
    'line'
    'line'

>> get(hp, 'Marker')
ans =
    'none'
    'o'

>> set(hp, 'Color', [0 1 0]) % for all the objects
```

The `fill`-command will produce patches (polygons) etc.

The above stuff is good to know when you make presentations, reports etc. Graphics does not help much if the audience cannot see it.
Here is an example. I do not claim that I have chosen the best fonts etc. An alternative is to use the builtin plot editor (see the menus and buttons in the top of the window).

```
% Say we are going to make a transparency for a lecture
figure(1)
set(1, 'DefaultAxesFontSize',      16,   ...
       'DefaultTextFontSize',      16,   ...
       'DefaultAxesFontWeight', 'Bold',  ...
       'DefaultTextFontWeight', 'Bold')

x = linspace(0, 2 * pi);
plot(x, sin(x))
hold on
grid on

% Suppose we would like to mark min and max
h = plot(0.5 * pi * [1 3], [1 -1], 'o')   % 2 handles
set(h, 'LineWidth', 2, 'MarkerSize', 10)
axis([-0.2 2*pi+0.2 -1.2 1.2])

xlabel('x')
ylabel('sin x')
title('A sine-curve')

text(2, 1.1, 'A maximum')
```

We can give the properties in the commands as well, e.g.
`text(5, -1, 'A minimum', 'Fontsize', 10)` which overrides the default.



An alternative to the above is to use the builtin plot editor (see the upper part of the window). This is convenient when you are doing an image once. I usually generate roughly the same image many times (new data, new course etc.) in which case it is more convenient to have an automatic generation in a program.

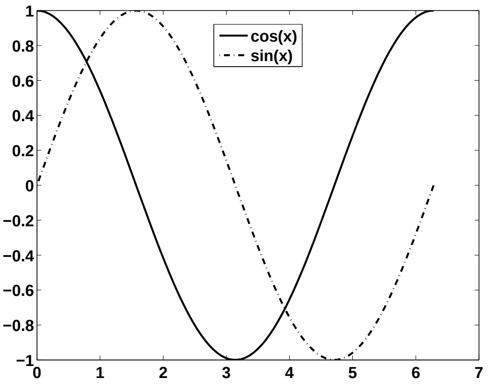
Something different: I have a command that deletes all the plot-windows (store in `~/matlab/del.m` for example)

`delete(get(0, 'Children'))`
One can use `close all` instead.

If we have several curves we should add a legend.

```
>> plot(x, cos(x), '-', x, sin(x), '-.', 'Linewidth', 2
% Default Location is NorthEast.
% Can move the legend using the mouse as well.
>> hl = legend('cos(x)', 'sin(x)', 'Location', 'Best');

>> set(hl,  'Fontsize', 16, 'Fontweight', 'Bold')
>> set(gca, 'Fontsize', 16, 'Fontweight', 'Bold')
```

Looks like this:



Annotations

Once you have produced your plot you may want to add so-called annotations, things like arrows, text, lines, rectangles etc. This can be done using the `annotation`-function, but it is easier to use the `Insert`-menu in the figure-window, since you can work with the mouse instead of typing coordinates.

A nice feature is that you can save an M-file containing the code necessary to generate the figure, `File/Generate M-file...` (otherwise you may have to redo the annotations if you want change the image).

Another way is to save the figure in `fig`-format, using the `Save As...`-menu alternative.

The following sequence together with an inserted `Text Arrow` (the text “intersection” and an arrow pointing at the intersection between the cures), produced the M-file on the next page:

```
>> x = linspace(0, 1);
>> plot(x, sin(x), 'r', x, cos(x), 'b')
>> grid
>> xlabel('x')
>> ylabel('y')
>> title('Intersection between sin x and cos x')
```

Note, on the next page, that the actual data is not included in the file. The comments are produced by Matlab.

Note the use of the `annotation`-function.

```
function createfigure(X1, YMatrix1)
%CREATEFIGURE(X1,YMATRIX1)
%  X1:  vector of x data
%  YMATRIX1:  matrix of y data

%  Auto-generated by MATLAB on 16-Sep-2010 17:39:49

% Create figure
figure1 = figure;

% Create axes
axes1 = axes('Parent',figure1);
box(axes1,'on');
grid(axes1,'on');
hold(axes1,'all');

% Create multiple lines using matrix input to plot
plot1 = plot(X1,YMatrix1);
set(plot1(1),'Color',[1 0 0]);
set(plot1(2),'Color',[0 0 1]);

% Create xlabel
xlabel('x');

% Create ylabel
ylabel('y');

% Create title
title('Intersection between sin x and cos x');

% Create textarrow
annotation(figure1,'textarrow',...
    [0.576785714285712 0.737499999999997],...
    [0.610904761904764 0.682333333333335],...
    'TextEdgeColor','none',...
    'String',{'intersection'});
```

Callbacks

It is common in Matlab-, OpenGL-, X11-programming to use callback routines. Such a routine is bound to a special event (e.g. the click of a mouse button) and the routine is called if the event occurs.
In this example a ButtonDownFcn-property of a curve, is used to change the colour of a curve. When we click close (5 pixels) to the curve it will change colour from blue (standard) to red. The value of the property (the callback) is, in this example, a Matlab-command. It will be executed if we click on the curve.

```
>> x = 0:0.1:2*pi;
>> h1 = plot(x, cos(x));
>> hold on
>> h2 = plot(x, sin(x));

>> get(h1)
....
        ButtonDownFcn =
        CreateFcn =
        DeleteFcn =

>> set(h1, 'ButtonDownFcn', ...
       'set(h1, ''Color'', [1 0 0])')

>> get(h1)
        ButtonDownFcn = set(h1, 'Color', [1 0 0])
```

This could be used to do the picking for the complex cosine function (in the introduction).

Note, a common misconception: the callback is not executed when we define it. It is executed if/when the action is performed.

Note also that the example shows unsafe programming, the variable `h1` may not exist when we click on the curve. Here is a better way, using the `gcbo`-function (get current callback object):

```
>> set(h1, 'ButtonDownFcn', ...
      'set(gcbo, ''Color'', [1 0 0])')
```

All graphics objects have three properties for which you can define callback routines:

- `ButtonDownFcn` as above.
- `CreateFcn` executes during object creation after all properties are set
- `DeleteFcn` executes just before deleting the object

User interface objects have a Callback property; more later on. Figures have the three callbacks above and (from the manual):

- `CloseRequestFcn` executes when a request is made to close the figure (by a close command, by the window manager menu, or by quitting MATLAB). Default is `closereq`.
- `KeyPressFcn` executes when users press a key while the cursor is within the figure window.
- `ResizeFcn` executes when users resize the figure window.
- `WindowButtonDownFcn` executes when users click a mouse button while the cursor is over the figure background, a disabled uicontrol, or the axes background.
- `WindowButtonMotionFcn` executes when users move the mouse within the figure window (but not over menus or title bar).
- `WindowButtonUpFcn` executes when users release the mouse button, after having pressed the mouse button within the figure.

The callback can be a Matlab command, as in the example, but also:

- a string with the name of an M-file (script or function).
- a cell array of strings (see the manual, a bit special).
- a function handle or a cell array containing a function handle and additional arguments (see the manual for the last case).

When using a function handle the callback-function must define at least two input arguments. The handle of the object generating the callback, and the event data structure (can be empty for some callbacks). Matlab passes these two arguments implicitly whenever the callback executes (it is possible to add input arguments, see the manual). Here is a simple example:

```
>> h1 = plot(x, cos(x));
>> set(h1, ’ButtonDownFcn’, @my_callback)
>> type my_callback        % list a file

function my_callback(handle, event_str)
% list input arguments (only in this example)
handle
event_str

% Can skip all. Must be true for all elements.
if all(get(handle, ’Color’) == [1 0 0])
  set(handle, ’Color’, [0 0 1])
else
  set(handle, ’Color’, [1 0 0])
end

>> handle =             % clicked on the curve
     1.540119628906250e+02
event_str =
     []
```

When we use the first alternative (a string) there are no required variables (we decide). An advantage with using function handles is that we, when making GUIs, can collect all the callbacks in one file, as in the following example. This is convenient since one tends to get many callbacks.
Here is the complex cosine-example again. We make a rectangular grid, in the complex plane, in a left subplot. Lines with constant real-parts are black, and lines with constant imaginary parts are red.
In the right subplot we plot the cosine of the points on the lines (using the same colours).
When we click on a red or black curve in either plot, the curve and the corresponding one in the other window, should become blue and twice as wide.
When we click on a blue curve in either plot, the curve and the corresponding one in the other window, should return to its original colour and get its original width.
When we click on a curve, a callback is called. In this callback we can find out the handle of the curve. The callback needs to find out the handle of the corresponding curve in the other plot. This can be solved in a number of ways.

- We can store the handles in a matrix, one row per pair of handles.
- subplot creates an axes object, so the figure has two axes-children. Each child has an array of handles to line-objects. The two handle arrays are probably ordered in the same way.
- A more general approach: use the `UserData`-property of a line to store the handle of the corresponding curve (one could store more data, e.g. a cell-array). Since the callback needs to know the original colours (can be done in several ways), I have used the `Tag`-property to store the colour as a string, `’r’` for red and `’k’` for black.

Here comes the program. The user should give intervals (real and imag) the number of lines.

```
function cos_ex(real_int, imag_int, n)
% To save space I have not included any help

figure  % New window
subplot(121); hold on % to avoid hold in the loop
subplot(122); hold on

iu = sqrt(-1);           % 50 is a bit arbitrary
im = iu * linspace(imag_int(1), imag_int(2), 50);

for re = linspace(real_int(1), real_int(2), n)
  subplot(121)
  h1 = plot([re re], imag_int, ’k’);

  subplot(122)
  c = cos(re + im);
  h2 = plot(real(c), imag(c), ’k’);

  set(h1,’UserData’,h2, ’Tag’,’k’, ’ButtonDownFcn’,@cb)
  set(h2,’UserData’,h1, ’Tag’,’k’, ’ButtonDownFcn’,@cb)
end

re = linspace(real_int(1), real_int(2), 50);
for im = linspace(imag_int(1), imag_int(2), n)
  subplot(121)
  h1 = plot(real_int, [im im], ’r’);

  subplot(122)
  c = cos(re + iu * im);
  h2 = plot(real(c), imag(c), ’r’);

  set(h1,’UserData’,h2, ’Tag’,’r’, ’ButtonDownFcn’,@cb)
  set(h2,’UserData’,h1, ’Tag’,’r’, ’ButtonDownFcn’,@cb)
end

subplot(121); axis tight
subplot(122); axis tight
```

There is a reason for:

```
  c = cos(re + im);
  h2 = plot(real(c), imag(c), ’k’);
```

If we write like this, it may not work:

```
  h2 = plot(cos(re + im), ’k’);
```

Why? Consider the following:

```
>> iu = sqrt(-1);

% draws a line from (0, 0) to (0, 1) in R^2
>> plot([0; iu])
>> hold on

% a line from (1, 0) to (2, 1). Not what we want!
>> plot([0; 1])   % imag = 0

% equivalent to
>> plot([1; 2], [0; 1])

% essentially a line from (0, 0) to (1, 0)
>> plot([0; 1] + eps * iu)
```

Here comes the callback. Note that `event` is not used so I choose to ignore it using a tilde (new in Matlab R2009b).

```
function cb(handle, ~)  % note, in the same file
blue = [0 0 1];

c = get(handle, 'Color');
if all(c == blue)  % new colours, reset
% get(handle, 'Tag') is original colour 'k' or 'r'

  set(handle, 'Color', get(handle, 'Tag'), ...
              'LineWidth', 1)
  h = get(handle, 'UserData');    % other subplot
  set(h, 'Color', get(h, 'Tag'), 'LineWidth', 1)
else
% original colours, change
  set(handle, 'Color', blue, 'LineWidth', 2)
  set(get(handle, 'UserData'), 'Color', blue, ...
                                'LineWidth', 2)
end
```

This works well in many situations. One problem is that the inverse of cos does not always exist. So there may be $z_1 \neq z_2$ with $\cos z_1 = \cos z_2$. This gives a problem with colour, clicking on z_1 may not give the same blue colour on $\cos z_1$. If $\cos z_1$ is on top of $\cos z_2$ we get a blue line, otherwise we get a mix of black and blue (or no change if we have a different line width).
A more severe problem is if we click on $\cos z_1 = \cos z_2$, only one line (not two) will become blue in the first plot. Which line reacts? Here is a short test;

```
>> v = [0 1];
>> plot(v, v, 'ButtonDownFcn', '1')  % echo 1
>> hold on
>> plot(v, v, 'ButtonDownFcn', '2')  % echo 2
>> ans =  % clicking on the line
     2
```

So the latest drawn line triggers the callback.

The following "works"; we can click on the line or on the markers.

```
>> v = linspace(0, 1, 30);
>> plot(v, v, 'ButtonDownFcn', '1')
>> hold
>> plot(v, v, 'ro', 'ButtonDownFcn', '2')
>> ans =  % clicking on a marker
     2
>> ans =  % clicking on the line
     1
```

This may be another solution in some cases:

```
>> h1 = plot(v, v, 'ButtonDownFcn', '1');
>> hold on
>> h2 = plot(v, v, 'ButtonDownFcn', '2');

>> set(h2, 'HitTest', 'Off') % cannot trigger
>> ans =                     % clicked
     1
>> set(h1, 'HitTest', 'Off') % switch this of as well
```

When both lines are "switched off" we do not get any print out (unless we have set the `ButtonDownFcn` of the current axes).

Finally an example where the event structure is not empty. Let us use the `KeyPressFcn` of a figure.

```
>> figure(1)
>> set(1, 'KeyPressFcn', @key_cb)
>> type key_cb

function key_cb(handle, event)

handle
event

>> handle = 1  % pressed the a-key with the
event =        % mouse in the window
    Character: 'a'
     Modifier: {1x0 cell}
          Key: 'a'

handle = 1     % pressed shift (part of writing A)
event =
    Character: ''
     Modifier: {1x0 cell}
          Key: 'shift'

handle = 1     % two events are generated for A
event =
    Character: 'A'
     Modifier: {'shift'}
          Key: 'a'

handle = 1     % pressed left arrow
event =
    Character: ''          % some garbage
     Modifier: {1x0 cell}  % the key sends ^[[A
          Key: 'leftarrow' % ^[ = escape
```

GUIs

It is now time to construct a more general GUI. Many things to think about when constructing a GUI, here are a few. For more references see the Diary. Some guidelines:

- No surprises! A good GUI behaves as the user expects. One should not have to hesitate when pushing a button. Nice with Undo and Cancel-alternatives.
- Consistency. Similar tasks should be done in similar ways. The user can learn principles.
- Use metaphors. A button with a magnifying glass for zooming, for example.
- Try to make the GUI self-explanatory. A user will not read manuals, perhaps not even a few lines.
- Give feedback. Did I push the button or not? Is the program running or has it crashed?
- Do not overuse strong colours, sound or movement. Keep messages readable (font, fontsize, fontweight) and clear.
- No builtin order. Modelessness. Should be able to press all buttons etc. without the program crashing. Turn off (gray out), or hide, alternatives that cannot be chosen, for example.
- Think of portability. Does the program work on another system? How does the monitor's resolution and size change the GUI? Are the sizes of buttons in pixels or cm?
- For Matlab GUIs. The users may have done other work before running your program, so be careful with using variables and windows. When your GUI quits, just clean up after your program, do not close all the windows, for example.

Matlab provides GUIDE (GUI Design Environment). You must run the GUI-mode of Matlab to use it (so do not start with `matlab -nojvm`). Then type `guide`. I will not use `guide` in this lecture.

Let us make a Quit-button. When we press the button, the window, which the button resides in, should be deleted. We make the button gray with the black text, Quit, on it. `uicontrol` is the basic tool.

```
>> figure
>> h = uicontrol;
>> set(h)
  BackgroundColor
  Callback: string -or- function handle -or- cell array
  Enable: [ {on} | off | inactive ]
  FontName
  FontSize
  ForegroundColor
  HorizontalAlignment: [ left | {center} | right ]
  KeyPressFcn: string -or- function handle -or- cell ar
  Max
  Min
  Position
  String
  Style: [ {pushbutton} | togglebutton | radiobutton |
            checkbox | edit | text | slider | frame |
            listbox | popupmenu ]
  TooltipString
  Units: [ inches | centimeters | normalized | points |
          {pixels} | characters ]
  Value
...
  Visible: [ {on} | off ]
```

We can choose between the following types:

- pushbutton, button with no memory
- togglebutton, on-off-button
- radiobutton, to choose the station on a radio (mutually exclusive)
- checkbox, tick choices
- edit, text that can be edited
- text, above a button. for example
- slider
- frame, rectangles that provide a visual enclosure for regions of a figure window (obsolete)
- listbox, scrollable list with alternatives
- popupmenu (does not work with `-nojvm`)

Some of the buttons only differ in appearance; we have to fix the functionality. A suitable button in our example is a pushbutton, which is the default. In this example we could use a string instead of a function.

```
>> type Quit_ex

function Quit_ex

hf = figure;
set(hf, 'Name',          'My GUI',       ...
        'NumberTitle',   'Off',          ...
        'MenuBar',       'None',         ...
        'Units',         'centimeters', ...
        'Position',      [10, 10, 5, 3])
```

```
hb = uicontrol( ...
     'Style',           'pushbutton',   ...  % default
     'Units',           'centimeters',  ...
     'Position',        [0.5 0.5 2 1],  ...
     'String',          'Quit',         ...
     'TooltipString',   'Close this window', ...
     'BackgroundColor', [0.7 0.7 0.7],  ...
     'ForegroundColor', [0 0 0],        ...
     'Callback',        @Quit_cb );

function Quit_cb(handle, event)

% gcbf: Get handle to current callback figure.
%  fig = gcbf returns the handle of the figure
%       that contains the object whose callback
%       is currently executing.

delete(gcbf)
```

Position is lower left x, lower left y, width, height.

Here comes a toggle button. The string, on the button, should alternate between On and Off. The button has a Value-property. Matlab will automatically alternate the value of Value between 0 and 1.

```
>> type Toggle_ex
function Toggle_ex

hf = figure;
set(hf, 'Name',          'My GUI',       ...
        'NumberTitle',   'Off',          ...
        'MenuBar',       'None',         ...
        'Units',         'centimeters', ...
        'Position',      [10, 10, 5, 3])

% Toggle buttons set Value to Max (default 1) when
% they are down (selected) and Min (default 0)
% when up (not selected).
hb = uicontrol( ...
     'Style',           'togglebutton', ...
     'Units',           'centimeters',  ...
     'Position',        [0.5 0.5 2 1],  ...
     'String',          'Off',          ...
     'BackgroundColor', [0.7 0.7 0.7],  ...
     'ForegroundColor', [0 0 0],        ...
     'Value',           0,              ... % Initially
     'Callback',        @Toggle_cb );       % Off

function Toggle_cb(handle, event)

% If Value = 1 when we clicked, then Value = 0
% in this callback.
if get(handle, 'Value')
  set(handle, 'String', 'On')  % used to be Off
else
  set(handle, 'String', 'Off') % used to be On
end
```

A shorter version:

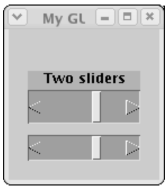
```
function Toggle_cb(handle, event)
str = {’Off’, ’On’};
set(handle, ’String’, str{1 + get(handle, ’Value’)})
```

Note that `str = [’Off’, ’On’];`gives one string, `’OffOn’`.

Here comes a slider, where we can set values continuously. We should put a text close to each slider. In the example we use the same callback. This is not necessary, nor is he use of the Tag.

```
>> type Slider_ex

function Slider_ex

hf = figure;
set(hf, ’Name’, ’My GUI’, ’NumberTitle’, ’Off’,   ...
’MenuBar’, ’None’, ’Units’, ’centimeters’,        ...
’Position’, [10, 10, 4, 4],                       ...
’DefaultUicontrolUnits’,           ’centimeters’, ...
’DefaultUicontrolBackgroundColor’, [0.7 0.7 0.7], ...
’DefaultUicontrolForegroundColor’,       [0 0 0])

uicontrol(’Style’, ’slider’,        ...
     ’Position’,   [0.5 0.5 3 0.7], ...
     ’Min’, -1,   ...  % min value of slider
     ’Max’, 2,    ...  % max value
     ’Value’, 1,  ...  % initial value
     ’Tag’, ’slider_1’,   ...
     ’Callback’, @Slider_cb );

uicontrol(’Style’, ’slider’,           ...
     ’Position’, [0.5 1.5 3 0.9],      ...
     ’Min’, -1, ’Max’, 2, ’Value’, 1,  ...
     ’Tag’, ’slider_2’,                ...
     ’Callback’, @Slider_cb );
```

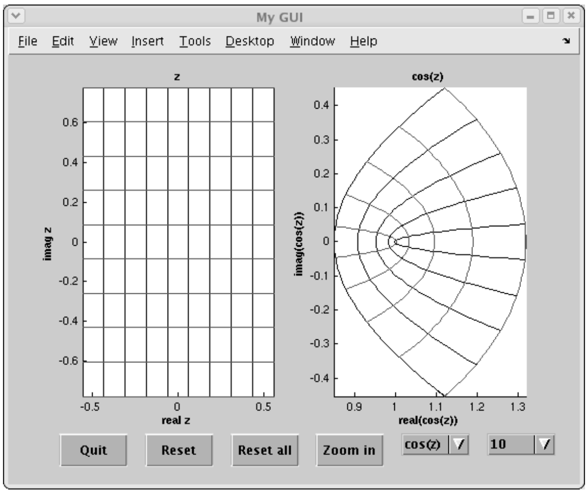
```
% Help text. May want a different BG-colour
uicontrol(’Style’, ’text’, ’String’, ’Two sliders’, ...
          ’FontWeight’, ’Bold’,...
          ’Position’, [0.5 2.4 3 0.5])

function Slider_cb(handle, event)
% Can have different callbacks for different
% sliders of course. Does not do anything
% useful.

val = get(handle, ’Value’)
if get(handle, ’Tag’) == ’slider_1’
  disp(’slider_1’)
else
  disp(’slider_2’)
end
```



This is using Matlab with Java. Turning off Java, **-nojvm**, gives a different appearance.

Notice also the area for the text (slightly darker).

Here comes a more sophisticated example. We take the old cosine-example (where we can click on the curves) and add some buttons and menus. We start the program by typing **cos_ex_gui** and get the following window:



Quit should delete the window. Reset should reset all the lines to their original colours and width. Using the left popupmenu we can choose between four functions; the plot is updated.

The next menu sets the number of grid lines; the plot is updated. Zoom in allows us the click twice in the left window to mark a smaller rectangle; the plot is updated. Reset all, resets everything (like starting over).

There should be texts above the menus.

Here is the code (> 240 lines). I have had to compress it (compared to my original). All routines in one file.

```
function cos_ex_gui
% Should have better names for the global variables
% or not use global. Can use UserData of the figure.
global ha1 ha2 hm_fun hm_n fun funcs ...
       real_int imag_int n

% default values
real_int = [-1 1]; % real interval
imag_int = [-1 1]; % imag interval
n        = 10;     % # of grid lines
fun      = 1;      % choice of function

funcs = {@(z)cos(z), @(z)sin(z), @(z)exp(z), @(z)z.ˆ2};

make_gui    % create buttons etc.
make_plots  % draws the grid and function(grid)

% ------------------ make_gui ------------------
function make_gui
global ha1 ha2 hm_fun hm_n funcs

hf = figure;

set(hf, ’Name’, ’My GUI’, ’NumberTitle’, ’Off’, ...
         ’Units’, ’centimeters’,                ...
        ’DefaultAxesUnits’,     ’centimeters’, ...
        ’DefaultUicontrolUnits’, ’centimeters’, ...
        ’DefaultUicontrolFontWeight’, ’Bold’,   ...
        ’DefaultUicontrolBackgroundColor’, ...
          [0.7 0.7 0.7],                   ...
        ’DefaultUicontrolForegroundColor’, ’k’)

ha1 = subplot(121); hold on
ha2 = subplot(122); hold on
```

```
% shrink subplots
dp = 1.2 * [0 1 0 -1];
set(ha1, ’Position’, get(ha1, ’Position’) + dp)
set(ha2, ’Position’, get(ha2, ’Position’) + dp)

% create buttons and menus
pos = [1.5 0.5 2 1]; dx = 0.5;

% Quit-button
uicontrol(’Position’, pos,           ...
’String’,          ’Quit’,          ...
’TooltipString’,   ’close window’,   ...
’Callback’,        ’delete(gcbf)’ );  % string

% Reset-button
pos(1) = pos(1) + pos(3) + dx;
uicontrol(’Position’, pos,           ...
’String’,          ’Reset’,         ...
’TooltipString’,   ’reset lines’,    ...
’Callback’,        @reset_cb);

% Reset all-button
pos(1) = pos(1) + pos(3) + dx;
uicontrol(’Position’, pos,             ...
’String’,          ’Reset all’,       ...
’TooltipString’,   ’reset everything’, ...
’Callback’,        @reset_all_cb);

% Zoom-button
pos(1) = pos(1) + pos(3) + dx;
uicontrol(’Position’, pos,              ...
’String’,          ’Zoom in’,           ...
’TooltipString’,   ’zoom in left plot’, ...
’Callback’,        @zoom_cb);
```

```
% Build menu-items
for fun = 1:length(funcs)
  t = char(funcs{fun});    % @(z)expression
  t = t(t ˜= ’.’);         % rm elementwise
  items{fun} = t(5:end);
end

% Function menu
pos(1) = pos(1) + pos(3) + dx;
hm_fun = uicontrol(’Style’, ’popupmenu’,   ...
’Position’,        pos,           ...
’TooltipString’,   ’function’,    ...
’String’,          items,         ...
’Value’,           1,             ...  % default
’Callback’,        @menu_fun_cb);

% An alternative to cell arrays
% n-menu (number of grid lines)
pos(1) = pos(1) + pos(3) + dx;
hm_n = uicontrol(’Style’, ’popupmenu’, ...
’Position’,        pos,               ...
’String’,                             ...
    ’5|6|7|8|9|10|11|12|13|14|15|16|17|18|19|20’, ...
’TooltipString’,   ’# of lines’,     ...
’Value’,           6,                ...
’Callback’,        @menu_n_cb);

% ------------------ equal ------------------
function eq = equal(s1, s2)
% Compare two strings. used by reset_cb.
% May be of unequal length (strcmp) and different
% case (i in strcmpi). Blanks are significant
% for strcmpi, so they are removed.

eq = strcmpi(s1(s1 ˜= ’ ’), s2(s2 ˜= ’ ’));
```

```
% ------------------ reset_cb ------------------
function reset_cb(handle, event)
% Could call make_plots instead
% but this shows a different technique

h  = get(handle, ’Parent’);  % i.e. the figure
hc = get(h, ’Children’);     % axes and uicontrol

for h = hc(:)’          % for all axes and uicontrols
  if equal(get(h, ’Type’), ’axes’)
    hl = get(h, ’Children’);  % lines

    for hline = hl(:)’  % for all lines
      set(hline, ’Linewidth’, 1, ...
                 ’Color’, get(hline, ’Tag’))
    end
  end
end

% ------------------ reset_all_cb ------------------
function reset_all_cb(handle, event)
global hm_fun hm_n fun real_int imag_int n

real_int = [-1 1]; % default values
imag_int = [-1 1];
fun      = 1;
n        = 10;

set(hm_fun, ’Value’, fun) % reset menus
set(hm_n,   ’Value’, 6)

make_plots                % redraw
```

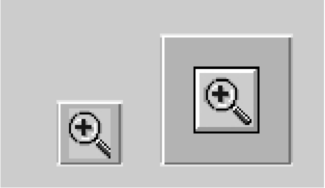
```
% ------------------ zoom_cb ------------------
function zoom_cb(handle, event)
%
% Can zoom in (but not out)
% There is builtin support for zoom (help zoom).
%
global real_int imag_int

[re, im] = ginput(2); % no conflict with
real_int = sort(re);  % clicking on lines
imag_int = sort(im);  % should check the values

make_plots            % redraw

% ------------------ menu_fun_cb ------------------
function menu_fun_cb(handle, event)
global fun

fun = get(handle, ’Value’);
make_plots          % redraw

% ------------------ menu_n_cb ------------------
function menu_n_cb(handle, event)
global n

n = 4 + get(handle, ’Value’);
make_plots          % redraw

% ------------------ make_plots ------------------
function make_plots
% almost like the old version
global ha1 ha2 fun funcs real_int imag_int n
iu = sqrt(-1);
```

```
% Remove curves. OK if empty. This is new.
delete(get(ha1, ’Children’))
delete(get(ha2, ’Children’))

im = iu * linspace(imag_int(1), imag_int(2), 50);
for re = linspace(real_int(1), real_int(2), n)
  subplot(ha1)   % subplot(121) changes position. New.
  h1 = plot([re re], imag_int, ’k’);

  subplot(ha2)
  c = funcs{fun}(re + im);   % This is new
  h2 = plot(real(c), imag(c), ’k’);

  set(h1, ’UserData’, h2, ’Tag’, ’k’, ...
          ’ButtonDownFcn’, @plot_cb)
  set(h2, ’UserData’, h1, ’Tag’, ’k’, ...
          ’ButtonDownFcn’, @plot_cb)
end

re = linspace(real_int(1), real_int(2), 50);
for im = linspace(imag_int(1), imag_int(2), n)
  subplot(ha1)
  h1 = plot(real_int, [im im], ’r’);

  subplot(ha2)
  c = funcs{fun}(re + iu* im);
  h2 = plot(real(c), imag(c), ’r’);

  set(h1, ’UserData’, h2, ’Tag’, ’r’, ...
          ’ButtonDownFcn’, @plot_cb)
  set(h2, ’UserData’, h1, ’Tag’, ’r’, ...
          ’ButtonDownFcn’, @plot_cb)
end
```

```
% This is new
subplot(ha1); axis tight
h(1) = xlabel(’real z’);
h(2) = ylabel(’imag z’);
h(3) = title(’z’);

subplot(ha2); axis tight
t = char(funcs{fun});   % something like @(z)expression
t = t(5:end);           % rm @(z)
t = t(t ˜= ’.’);        % rm dots

% xlabel should be real(cos(z)) etc.
h(4) = xlabel([’real(’, t, ’)’]);
h(5) = ylabel([’imag(’, t, ’)’]);
h(6) = title(t);
set(h, ’FontWeight’, ’Bold’)

% ------------------ plot_cb  ------------------
function plot_cb(handle, event)
% ... same as function cb in the previous example
blue = [0 0 1];

c = get(handle, ’Color’);
if all(c == blue)  % new colours, reset
% get(handle, ’Tag’) is original colour ’k’ or ’r’

  set(handle, ’Color’, get(handle, ’Tag’), ’LineWidth’,
  h = get(handle, ’UserData’);    % other subplot
  set(h, ’Color’, get(h, ’Tag’), ’LineWidth’, 1)
else
% original colours, change
  set(handle, ’Color’, blue, ’LineWidth’, 2)
  set(get(handle, ’UserData’), ’Color’, blue, ...
      ’LineWidth’, 2)
end
```

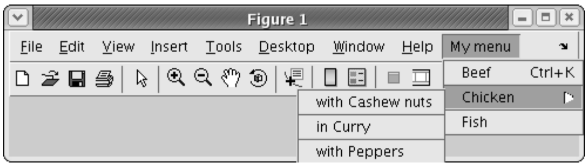
It is possible to have textures on buttons. I fetched a gif-image of a magnifying glass. Matlab requires true colour (24-bit colour) and I used the `xv`-command to convert the image and saved it as a jpeg-image (highest quality). The original image has a black border.

```
>> C = imread(’mag.jpg’, ’jpg’);  % read the file
>> image(C)                       % look at it
>> axis image                     % correct scaling
>> size(C)
ans =
    32    32     3                % a 3D-matrix

>> figure
>> uicontrol(’Style’,   ’Pushbutton’,     ...
             ’Units’,   ’pixels’,         ...  % Note
             ’Position’, [100 100 32 32], ...
             ’CData’,    C,               ...  % Note
             ’Callback’, @Zoom_cb );

>> uicontrol(’Style’,   ’Pushbutton’,     ...
             ’Units’,   ’pixels’,         ...
             ’Position’, [150 100 64 64], ...
             ’CData’,    C,               ...
             ’Callback’, @Zoom_cb );
```



We can add menus at the top of the window as well.

```
h  = figure;
hm = uimenu(h,  ’Label’, ’My menu’);
% set(h, ’MenuBar’, ’None’) removes the standard menu

% Accelerator: type CTRL-K with the mouse in the window
alt(1) = uimenu(hm, ’Label’,       ’Beef’,             ..
                    ’Callback’,    ’disp(’’Beef’’)’, ..
                    ’Accelerator’, ’K’);

% Can have callback here as well
alt(2) = uimenu(hm, ’Label’, ’Chicken’);

alt(3) = uimenu(hm, ’Label’,    ’Fish’,         ...
                    ’Callback’, ’disp(’’Fish’’)’ );

% We can do hierarchical menus. Don’t overuse!

uimenu(alt(2), ’Label’, ’with Cashew nuts’, ...
               ’Callback’, ’disp(’’Cashew’’)’);

uimenu(alt(2), ’Label’, ’in Curry’, ...
               ’Callback’, ’disp(’’Curry’’)’);

uimenu(alt(2), ’Label’, ’with Peppers’, ...
               ’Callback’, ’disp(’’Peppers’’)’);
```



A few more words about clicking on curves.

If you choose “Data Cursor”-tool (to the right of the rotate button) you can click on an object (also in 3D) to get the coordinates.

You can change the cursor to one of several predefined:

```
>> set(gcf, ’Pointer’, ’arrow’)
% or ’watch’ etc. See the manual.
```

The watch-cursor is an animation under Gnome.

You can make your own cursor, as well. Create a 16 × 16-matrix containing 1 (black), 2 (white) and NaN (transparent).
Let us make a large X.

```
>> C = eye(16); C = C + C(:, end:-1:1);
>> C = C ./ C;
Warning: Divide by zero.

>> C(6:11, 6:11)
ans =
     1   NaN   NaN   NaN   NaN     1
   NaN     1   NaN   NaN     1   NaN
   NaN   NaN     1     1   NaN   NaN
   NaN   NaN     1     1   NaN   NaN
   NaN     1   NaN   NaN     1   NaN
     1   NaN   NaN   NaN   NaN     1

>> figure(1)
>> set(1, ’Pointer’,              ’Custom’, ...
          ’PointerShapeCData’,     C,       ...
          ’PointerShapeHotSpot’,  [8.5 8.5])
```

`PointerShapeHotSpot`is the pointer location.

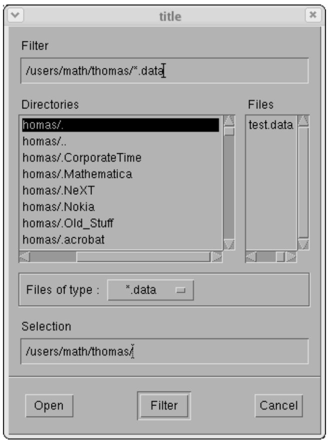
We can bind a menu (context menu) to a graphical object, e.g. a curve.

```
figure(1)

% Create a context menu
cmenu = uicontextmenu;

x = 0:0.1:1;

% and bind it to the curve
hp = plot(x, sin(x), ’UIContextMenu’, cmenu);

% Define callbacks...
cb1 = ’set(hp, ’’LineStyle’’, ’’--’’)’;
cb2 = ’set(hp, ’’LineStyle’’, ’’:’’)’;
cb3 = ’set(hp, ’’LineStyle’’, ’’-’’)’;

% Define the menu alternatives
uimenu(cmenu, ’Label’, ’dashed’, ’Callback’, cb1)
uimenu(cmenu, ’Label’, ’dotted’, ’Callback’, cb2)
uimenu(cmenu, ’Label’, ’solid’,  ’Callback’, cb3)
```

If one RIGHT-clicks on the curve, a menu appears where we can choose between dashed, dotted and solid.

Loading files

```
> type load_file.m
function load_file
uicontrol(’Style’,’PushButton’, ’Units’,’centimeters’,
          ’Position’, [1 3 2 1.5], ’String’, ’Load’,
          ’Callback’,    @load_cb )

function load_cb(handle, event)
pos    = [100 100];  % [from_left, from_top], in pixels
filter = ’*.data’;

[file_name, path_to_file] = ...
  uigetfile(filter, ’title’, ’Location’, pos);

file_name, path_to_file   % We usually don’t print

>> load_file
file_name = test.data
path_to_file = /users/math/thomas/
```



/users/math/thomas/*.data

homas/.
homas/..
homas/.CorporateTime
homas/.Mathematica
homas/.NeXT
homas/.Nokia
homas/.Old_Stuff
homas/.acrobat

test.data

*.data

/users/math/thomas/

Error messages

```
function error_msg(msg)

% Inactivate all other windows using a modal
% dialogue
figure( ’Units’,       ’centimeters’,  ...
        ’Position’,     [15 15 4 2],    ...
        ’Color’,        [1.0 0.5 0.5], ...
        ’MenuBar’,     ’None’,          ...
        ’NumberTitle’, ’Off’,           ...
        ’WindowStyle’, ’Modal’,         ...  % Note
        ’Name’,        ’Error’ );
axis(’off’)

% the error message
text(0, 0.7, msg, ’FontWeight’, ’Bold’)

% Remove the window when we have pressed OK

uicontrol( ’Style’,       ’PushButton’,    ...
           ’Units’,       ’centimeters’,   ...
           ’Position’,     [0.3, 0.3, 1, 0.7], ...
           ’String’,      ’OK’,               ...
           ’Callback’,    ’delete(gcbf)’)
```

`error_msg(’Nothing to plot’)`gives:

Nothing to plot

There is a function for this in Matlab:
`errordlg(’message’, ’title’, ’modal’.)`
This is one of several such functions. See help or the manual.

Predefined Dialog Boxes

- dialog Create and display dialog box
- errordlg Create and display error dialog box
- helpdlg Create and display help dialog box
- inputdlg Create and display input dialog box
- listdlg Create and display list selection dialog box
- msgbox Create and display message dialog box
- pagesetupdlg Display page setup dialog box
- printdlg Display print dialog box
- questdlg Display question dialog box
- uigetdir Display standard dialog box for retrieving a directory
- uigetfile Display standard dialog box for retrieving files
- uigetpref Display dialog box for retrieving preferences
- uiputfile Display standard dialog box for saving files
- uisave Display standard dialog box for saving workspace variables
- uisetcolor Display standard dialog box for setting an object’s ColorSpec
- uisetfont Display standard dialog box for setting an object’s font characteristics
- waitbar Display waitbar
- warndlg Display warning dialog box

Animation in Matlab

Example: we would like to animate a square that bounces inside a rectangle. We assume that the square always hits a wall at a 45 degree angle and that no energy is lost in the contact.
Here is a simple solution:

```
function test5
global min_x max_x min_y max_y v cont

% initial position for square
x = 3 * [0 1 1 0]’ + 15;
y = 3 * [0 0 1 1]’ + 15;

hf = figure;
set(hf, ’DeleteFcn’, @clean_up)

% plot square
h = fill(x, y, ’r’);
axis equal
min_x = 0; max_x = 90; min_y = 0; max_y = 31;

% boundingbox
axis([min_x max_x min_y max_y])
set(gca, ’xtick’, [], ’ytick’, [])

v    = [1 1];    % initial direction
cont = 1;

while cont
% drawnow        % update screen
  pause(0.01)    % pause and update screen
  update_pos(h)
end
```

drawnow or **pause** is needed to flush the queue for graphics events, otherwise all the events will accumulate and nothing is plotted.

```
function update_pos(h)
global min_x max_x min_y max_y v cont

% necessary since this routine can be called when
% we have deleted the window
if ˜cont, return, end

% fetch position
x = get(h, ’xdata’);
y = get(h, ’ydata’);

% check if squrea has hit a wall or a corner
off_y = y(3) >= max_y || y(1) <= min_y;
if x(2) >= max_x || x(1) <= min_x
  if off_y
    v = -v;
    set(h, ’Facecolor’, ’g’)  % change colour as well
  else
    v = [-v(1), v(2)];
    set(h, ’Facecolor’, ’b’)
  end
elseif off_y
  v = [v(1), -v(2)];
    set(h, ’Facecolor’, ’y’)
end

% update position
x = x + 0.2 * v(1);
y = y + 0.2 * v(2);

% update graphics data
set(h, ’xdata’, x, ’ydata’, y)
```

```
function clean_up(obj, event)
% called when we delete the window
global cont

cont = 0;
```

Another way to update the image is to use a so-called timer object. A timer object is similar to a clock that runs in parallel with ones program (a separate thread).
The clock can be set up so that it calls a callback routine at times $t_0, t_0 + \delta_t, t_0 + 2\delta_t, t_0 + 3\delta_t, \ldots$. t_0 is called start delay and δ_t period. Java must be enabled for this to work.
First some simple examples.

```
> t = timer(’TimerFcn’, ’disp(’’tic’’)’,      ...
            ’ExecutionMode’, ’fixedSpacing’, ...
            ’Period’, 1, ’TasksToExecute’, 5)

   Timer Object: timer-1

   Timer Settings
      ExecutionMode: fixedSpacing
             Period: 1
           BusyMode: drop
            Running: off

   Callbacks
           TimerFcn: ’disp(’tic’)’
           ErrorFcn: ’’
           StartFcn: ’’
            StopFcn: ’’

>> start(t)
tic
tic
tic
tic
tic
```

```
>> get(t, 'Running')
ans = off
>> delete(t)

% make a new timer
>> t = timer('TimerFcn', 'disp(''tic'')',     ...
             'ExecutionMode', 'fixedSpacing', ...
             'Period', 10, 'TasksToExecute', 5);
>> start(t)
tic
>> get(t, 'Running')
ans = on
>> stop(t)
>> delete(t)

% make a new timer
>> t = timer('TimerFcn', 'disp(''tic'')',     ...
             'ExecutionMode', 'fixedSpacing', ...
             'Period', 10, 'TasksToExecute', 5);
>> start(t)
tic
tic
>> delete(t)
Warning: One or more timer objects were stopped
         before deletion.

% make two new timers
>> t1 = timer('TimerFcn', 'disp(''tic_1'')',   ...
              'ExecutionMode', 'fixedSpacing', ...
              'Period', 1, 'TasksToExecute', 5);

>> t2 = timer('TimerFcn', 'disp(''tic_2'')',   ...
              'ExecutionMode', 'fixedSpacing', ...
              'Period', 1, 'TasksToExecute', 5);
```

```
>> start(t1)
tic_1
tic_1
>> start(t2)
tic_2
tic_1
tic_2
tic_1
tic_2
tic_1
tic_2
tic_2
>> timerfind

Timer Object Array

   Index:  ExecutionMode:  Period:  TimerFcn:
   1       fixedSpacing    1        'disp('tic_1')'
   2       fixedSpacing    1        'disp('tic_2')'

>> delete(timerfind)
>> timerfind
ans =
     []


% make a new timer
>> t = timer('TimerFcn', 'disp(''tic'')',     ...
             'ExecutionMode', 'fixedSpacing', ...
             'Period', 1, 'TasksToExecute', 5);
>> start(t); wait(t)   % block the command line
```

Possible to have `'TasksToExecute', Inf`

Here are some of the most important properties of a timer object.

- `BusyMode` Action taken when a timer has to execute `TimerFcn` before the completion of previous execution of `TimerFcn`.
  `'drop'`, do not execute the function. (default).
  `'error'`, generate an error.
  `'queue'`, execute function at next opportunity.
- `ExecutionMode` Determines how the timer object schedules timer events. `'singleShot'` (default), `'fixedDelay'`, `'fixedRat`
  `'fixedSpacing'`.
- `Period` Specifies the delay, in seconds, between executions of `TimerFcn`.
- `Running` Indicates whether the timer is currently executing.
- `StartDelay` Specifies the delay, in seconds, between the start of the timer and the first execution of the function specified in `TimerFcn`.
- `StartFcn` Function the timer calls when it starts.
- `StopFcn` Function the timer calls when it stops.
- `TasksToExecute` Specifies the number of times the timer should execute the function specified in the `TimerFcn` property.
- `TimerFcn` Timer callback function.
- `UserData` User-supplied data.

More deails about `ExecutionMode` The duration of the lag depends on what other processing Matlab happens to be doing at the time.

```
singleShot
               T i m e r  e x e c u t e s
   start          lag       TimerFcn   timer stops
 ----|---------|------+-----------------|
      start
      delay
```

Here are the other three cases:

```
    delay
 --|----|-----------------------------------------
 start

fixedSpacing:
        |lag|TimerFcn|<---- period ---->|lag|TimerFcn|

fixedDelay:
             <---- period ---->
        |lag|TimerFcn|         |lag|TimerFcn|

fixedRate:
         <---- period ---->
        |lag|TimerFcn|    |lag|TimerFcn|
```

Here is a code for the bouncing squre using timer objects.

```
function test55
global min_x max_x min_y max_y

hf = figure;
% Can stop the square by clicking in the window
% outside the plot area (stop_go).
% When we close the window clean_up is executed.

set(hf, 'ButtonDownFcn', @stop_go, ...
        'DeleteFcn', @clean_up)
hold off

x = 3 * [0 1 1 0]' + 15;  % same as before
y = 3 * [0 0 1 1]' + 15;
h = fill(x, y, 'r');
axis equal
min_x = 0; max_x = 30; min_y = 0; max_y = 51;
axis([min_x max_x min_y max_y])
set(gca, 'xtick', [], 'ytick', [])
```

```
% Create timer and define properties
t = timer;
set(t, 'TimerFcn', @my_update, 'StartDelay', 0,  ...
       'TasksToExecute', Inf,  'Period', 0.015,  ...
       'ExecutionMode', 'fixedSpacing',          ...
       'BusyMode', 'drop');

v = [1 1];
set(t, 'UserData', {h, v})  % store h and v in Userdata

set(hf, 'UserData', t) % store handle to timer in figur
start(t)
% -----------------------------------------------------
function my_update(obj, event)
global min_x max_x min_y max_y

ud = get(obj, 'UserData');  % obj = timer
h = ud{1};
if ~ishandle(h)  % just to be sure...
  disp('no handle')
  stop(t)
  delete(t)
  return
end

v = ud{2};
x = get(h, 'xdata');
y = get(h, 'ydata');

% same as before
off_y = y(3) >= max_y || y(1) <= min_y;
if x(2) >= max_x || x(1) <= min_x
  if off_y
    v = -v;
    set(h, 'Facecolor', 'g')
  else
    v = [-v(1), v(2)];
```

```
    set(h, 'Facecolor', 'b')
  end
elseif off_y
  v = [v(1), -v(2)];
    set(h, 'Facecolor', 'y')
end

x = x + 0.2 * v(1);
y = y + 0.2 * v(2);

set(h, 'xdata', x, 'ydata', y)

set(obj, 'UserData', {h, v})

% -----------------------------------------------

function clean_up(obj, event)
disp('clean_up')
t = get(obj, 'UserData');  % obj = figure
run = get(t, 'Running');
if run(1:2) == 'on' % other is off
  stop(t)
end
delete(t)

% -----------------------------------------------

function stop_go(obj, event)
t = get(obj, 'UserData');  % obj = figure
run = get(t, 'Running');
if run(1:2) == 'on' % other is off
  stop(t)
else
  start(t)
end
```

It does happen that the timer continues to run even though we have removed the window (I do not know why). Typing `^C` in the Matlab command window seems to solve the problem.

In some versions of Matlab it may be useful to switch on double buffering (on our system it is switched on). This makes for a more steady, flicker free, animation.

In this method, two graphics pages in the video memory are used. While one page is displayed by the monitor, the other is drawn. When drawing is complete, the roles of the two pages are switched, so that the previously shown page is modified, and the previously drawn page is shown.

```
>> figure(1)
>> set(1, 'DoubleBuffer')
[ {on} | off ]
```

Another property that is important is `Renderer`. It can take one of four values, only the first are of interest to us:

```
>> set(1, 'Renderer')
[ {painters} | zbuffer | OpenGL | None ]
```

The meaning of the different values will be explained later in the course. `painters` is a fast method for drawing simple graphics having no light sources. `zbuffer` and `OpenGL` are used for more complicated scenes and `OpenGL` is also the choice when we would like to use the system's graphics hardware. Matlab switches automatically (provided `RenderMode` is set to `auto`), for example:

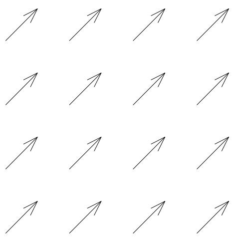
```
>> figure(1)
>> get(1, 'Renderer')
ans = None

>> plot(rand(10, 1))
>> get(1, 'Renderer')
ans = painters
```

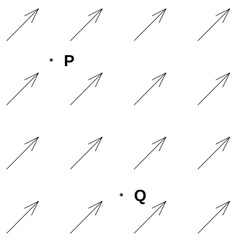
```
>> surf(rand(10))
>> get(1, 'Renderer')
ans = painters

>> shading interp
>> get(1, 'Renderer')
ans = OpenGL
```

```
>> opengl info

Version         = 3.0.0 NVIDIA 180.51
Vendor          = NVIDIA Corporation
Renderer        = GeForce 9500 GT/PCI/SSE2
MaxTextureSize  = 8192
Visual          = 0x26 (TrueColor, depth 24, RGB mask 0
Software        = false
# of Extensions = 157

Driver Bug Workarounds:
OpenGLBitmapZbufferBug    = 0
OpenGLWobbleTesselatorBug = 0
OpenGLLineSmoothingBug    = 0
OpenGLClippedImageBug     = 1
OpenGLEraseModeBug        = 0

>> opengl software
>> opengl info

Version         = 1.5 Mesa 6.0.1
Vendor          = Brian Paul
Renderer        = Mesa X11
MaxTextureSize  = 2048
Visual          = 0x21 (TrueColor, depth 24, RGB mask 0
Software        = true
# of Extensions = 96

Driver Bug Workarounds:  etc.
```

Vectors and points

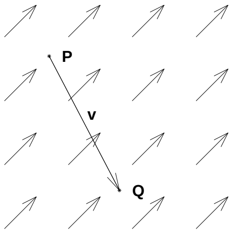
Important to distinguish between point and vectors in computer graphics, so here comes a short review. A vector is an equivalence class (think set) of directed line segments that share the same length and direction. One of the segments is a representative of the vector.
The left image shows some representatives of the vector.





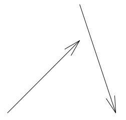
There are infinitely many representatives. A point, however, is a unique object. **P** and **Q** are two points (right image).

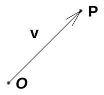
Two points define a vector: $v = Q - P$ is the vector which starts in P and goes to Q. A point and a vector define a new point: $Q = P + v$. A single point, P (or Q) does not define a vector. A vector does not define a point, either.



A basis is a set of linearly independent vectors such that all vectors (in the space) can be written as a linear combination of the basis vectors. A vector has a coordinate representation in such a system. The left image shows a basis. It is common to draw the representatives starting at the same point.
Note, however, that we still do not have an origin.

Let us now forget the basis for a while, and instead introduce a special, fix point, the origin, $\mathcal{O}$.





Given the origin we can get a 1-1 correspondence between vectors and points by using the representative starting in $\mathcal{O}$ and ending in the P (Sw. ortsvektor). In the right image v corresponds to the point P.

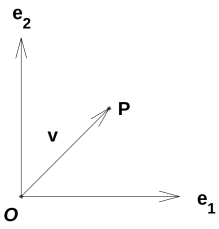
A coordinate system is an origin together with a basis. A point and a vector has a coordinate representation in such a system.
We will use ON-systems (orthogonal and normalized basis).

Let e_1, e_2 and e_3 be a basis in 3D. A point, P, can be written $P = p_x e_1 + p_y e_2 + p_z e_3 + \mathcal{O}$. $P - \mathcal{O}$ is the vector which is a linear combination of the basis. $p_x e_1 + p_y e_2 + p_z e_3$.

A vector, v, can be written $v = v_x e_1 + v_y e_2 + v_z e_3$. Formally:

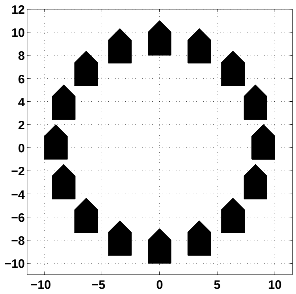
$$P = [e_1,\ e_2,\ e_3,\ \mathcal{O}] \begin{bmatrix} p_x \\ p_y \\ p_z \\ 1 \end{bmatrix} \quad \text{and} \quad v = [e_1,\ e_2,\ e_3,\ \mathcal{O}] \begin{bmatrix} v_x \\ v_y \\ v_z \\ 0 \end{bmatrix}$$

Coordinates with four components (three in 2D) are called homogeneous coordinates. This is how it looks in 2D:



One advantage with homogeneous coordinates is that a translation can be written as a matrix-vector product (i.e. not only linear mappings). This leads to a unified treatment of simple mappings. Homogeneous coordinates are also used when dealing with perspective projections.

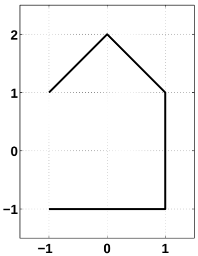
In computer graphics it is common to change coordinate systems. Suppose we would like to produce the following image (the coordinate system should not be included).



A cumbersome way is to create absolute coordinates
for the corners of the polygon.

```
x = [ 8 10 10 9 8];         % initial position
y = [-1 -1  1 2 1];
fill(x, y, 'k')             % draw one polygon
hold on
x = [7.3149 9.3149 ... ];  % new coordinates
y = [2.4442 2.4442 ... ];
fill(x, y, 'k')             % draw the next polygon
```

More convenient is to design ONE polygon in a “design coordinate system”, using so-called modeling coordinates.



We draw the polygons by translating the coordinates:

```
x = [-1  1  1  0 -1];    % nice x and y
y = [-1 -1  1  2  1];    % using modeling coordinates
for k = 1:16             % number of polygons
  dx = ...; dy = ...;    % translate
  fill(dx + x, dy + y, ’k’)
  hold on
end
```

The above is the normal way in Matlab, but in most, low level, graphics systems one would do like this instead:

```
for k = 1:16
  make a temporary translation of the coordinate
  system to where the polygon should be drawn

  draw_polygon()  % draw using modeling coordinates

  translate back
end
```

`draw_polygon` knows only about the modeling coordinates.
To move points (using `dx` and `dy`) or to move a coordinate system are two sides of the same coin.
We will look at this in more detail later on.

Note, also that we no longer talk about functions. We do not plot $y = f(x)$. Instead we create sets of points and these points can be given different interpretations.

```
plot(x, y)          % solid curve
plot(x, y, ’o’)     % separate points
fill(x, y, ’k’)     % polygon
etc.
```

Some transformations

We would like to transform points given in homogeneous coordinates. What types of transformations do we need? Scaling, rotation and translation. Linear transformations are not sufficient, since they map the origin onto the origin (which excludes translation). We need an affine transformation (linear plus translation). Using homogeneous coordinates we can write the transformation as a matrix-vector multiply, where the matrix is given by:

$$M = \begin{bmatrix} A & t \\ 0 & 1 \end{bmatrix}$$

A is a 3×3-matrix in the 3D-case, and t is a 3×1-matrix.

A point, P_2, in 2D and a point, P_3, in 3D can be written:

$$P_2 = \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}, \quad P_3 = \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Let p denote the x, y-part, the x, y, z-part in the 3D-case. Then

$$P_2 = \begin{bmatrix} p \\ 1 \end{bmatrix}, \quad P_3 = \begin{bmatrix} p \\ 1 \end{bmatrix}$$

Let us see how M transforms a point. P is a 2D- or 3D-point.

$$MP = \begin{bmatrix} A & t \\ 0 & 1 \end{bmatrix} \begin{bmatrix} p \\ 1 \end{bmatrix} = \begin{bmatrix} Ap + t \\ 1 \end{bmatrix}$$

Ap corresponds to a linear part and $+t$ gives a translation. We get a pure translation by setting $A = I$ (the identity) and a pure linear transformation (e.g. scaling, rotation) by taking $t = 0$.

Example: Show that the inverse transformation, M^{-1}, exists when A is nonsingular, and that:

$$M^{-1} = \begin{bmatrix} A & t \\ 0 & 1 \end{bmatrix}^{-1} = \begin{bmatrix} A^{-1} & -A^{-1}t \\ 0 & 1 \end{bmatrix}$$

M can be factored as:

$$\begin{bmatrix} A & t \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} I & t \\ 0 & 1 \end{bmatrix} \begin{bmatrix} A & 0 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} A & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} I & A^{-1}t \\ 0 & 1 \end{bmatrix}$$

So M can be written as a product of a linear transformation followed by a translation (which is no surprise). The reverse is true, if A^{-1} exists.

Let us see how M transforms a straight line. Use s as parameter and write the line in the following form:

$$L(s) = P + sW$$

where P is a point and W is a vector. How do we write a vector in homogeneous coordinates? We change the 1 to a 0. W can be interpreted as $w_x e_1 + w_y e_2 + w_z e_3 + 0 \cdot \mathcal{O}$, so a vector is a linear combination of the basis vectors. We wrote a point, in homogeneous coordinates, as: $w_x e_1 + w_y e_2 + w_z e_3 + 1 \cdot \mathcal{O}$. A point is a vector plus a point, in other words.

M maps a vector this way (w is the coordinate part):

$$\begin{bmatrix} A & t \\ 0 & 1 \end{bmatrix} \begin{bmatrix} w \\ 0 \end{bmatrix} = \begin{bmatrix} Aw \\ 0 \end{bmatrix}$$

Note that t is not included. It is not meaningful to translate a vector.

Our line is mapped as follows:

$$\begin{bmatrix} A & t \\ 0 & 1 \end{bmatrix} (P + sW) = \begin{bmatrix} A & t \\ 0 & 1 \end{bmatrix} \left[\begin{bmatrix} p \\ 1 \end{bmatrix} + s \begin{bmatrix} w \\ 0 \end{bmatrix} \right] =$$

$$\underbrace{\begin{bmatrix} Ap + t \\ 1 \end{bmatrix}}_{\text{point}} + s \underbrace{\begin{bmatrix} Aw \\ 0 \end{bmatrix}}_{\text{new direction}}$$

If $Aw = 0$ (A is singular and $w \in \mathcal{N}(A)$) the whole line is mapped to a single point.

Exercise: show that M maps planes onto planes and planar polygons onto planar polygons.

A translation example in 2D

We would like to translate the unit square so that the lower left corner ends up in $(1, 1)$. It is sufficient to look at how the corners are translated, since we have seen that straight lines are mapped onto straight lines. Here are the corners in homogeneous coordinates:

$$\begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}, \quad \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix}, \quad \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}, \quad \begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix}$$

We take $A = I$ and $t = [1, 1]^T$, and apply the transformation, M, on all four corners at the same time:

$$\underbrace{\begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix}}_{M} \underbrace{\begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix}}_{\text{corners}} = \underbrace{\begin{bmatrix} 1 & 2 & 2 & 1 \\ 1 & 1 & 2 & 2 \\ 1 & 1 & 1 & 1 \end{bmatrix}}_{\text{transformed corners}}$$

M^{-1} is given by taking $t = [-1, -1]^T$, which is in correspondence with our intuition (I hope).

Exercise: suppose we make a series of translations (one M-matrix for each). What is the M-matrix for the combined transformation.

Some scalings

For a pure scaling we set $t = 0$ and A to a diagonal matrix with scale factors. Let us double the width of the unit square. The matrix is (in 2D):

$$M = \begin{bmatrix} 2 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

The matrix

$$M = \begin{bmatrix} 2 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

doubles the lengths of both sides and

$$M = \begin{bmatrix} 0.5 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

halves the width and doubles the height etc.

Two sides of the same coin

Example: let us study how M transforms an arbitrary point P:

$$M = \begin{bmatrix} 1 & -1 & 1 \\ 1 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix}, \quad P = \begin{bmatrix} p_x \\ p_y \\ 1 \end{bmatrix}, \quad MP = \begin{bmatrix} p_x - p_y + 1 \\ p_x + p_y + 1 \\ 1 \end{bmatrix}$$

This can be written in the following way:

$$MP = M \left[p_x \underbrace{\begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}}_{e_1} + p_y \underbrace{\begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}}_{e_2} + 1 \underbrace{\begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}}_{\mathcal{O}} \right] =$$

$$p_x \underbrace{M \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}}_{e'_1} + p_y \underbrace{M \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}}_{e'_2} + 1 \underbrace{M \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}}_{\mathcal{O}'} = p_x e'_1 + p_y e'_2 + 1\mathcal{O}'$$

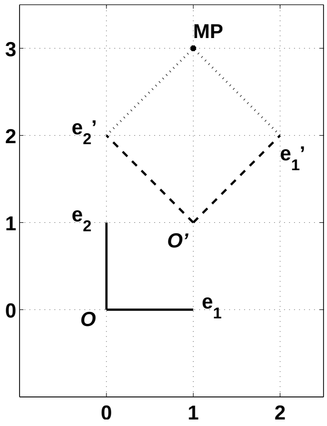
So, MP, can be interpreted as using the original coordinates for P, but in the transformed coordinate system $\{e'_1, e'_2, \mathcal{O}'\} = \{Me_1, Me_2, M\mathcal{O}\}$.

So, in the first interpretation we change the point's coordinates, but keep the original coordinate system.
The second interpretation keeps the original coordinates for the point, but we transform the coordinate system.

In this particular example the new coordinate system is given by:

$$e'_1 = \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix}, \quad e'_2 = \begin{bmatrix} -1 \\ 1 \\ 0 \end{bmatrix}, \quad \mathcal{O}' = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$$

This image shows how P, with $p_x = p_y = 1$, is transformed. MP has coordinates $(1, 3)$. The dashed lines show the transformed coordinate system.



Another example: A translation produces the new system: $(e_1, e_2, e_3, t + \mathcal{O})$ since the translation of the basis vectors gives the same basis.

Let us look at a few more complicated examples, involving rotations.

Rotations in 2D

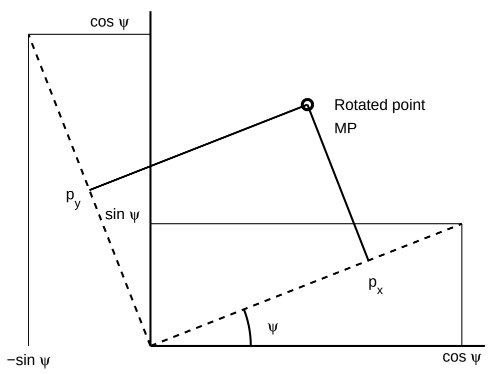
We would like to rotate points an angle ψ, ccw (counter clockwise) around the origin. Here is M:

$$M = \begin{bmatrix} \cos\psi & -\sin\psi & 0 \\ \sin\psi & \cos\psi & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

We can simplify the analysis of M by looking at how the coordinate system is transformed. Since M is linear (no translation) the origin is mapped onto the origin. So if we take a point at the end of each coordinate axis we can see how the coordinate system is transformed.

$$M = \begin{bmatrix} \cos\psi & -\sin\psi & 0 \\ \sin\psi & \cos\psi & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \end{bmatrix} = \begin{bmatrix} \cos\psi & -\sin\psi \\ \sin\psi & \cos\psi \\ 1 & 1 \end{bmatrix}$$

The dashed lines gives a rotated system. p_x, p_y are the original coordinates.



Combined transformations

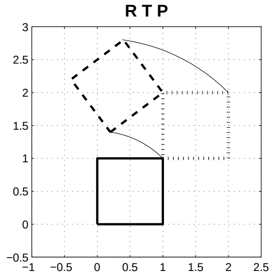
Let us study $M = TR$ and $M = RT$ where T is a translation and R is a rotation. Set $C = R(1:2, 1:2)$. We get:

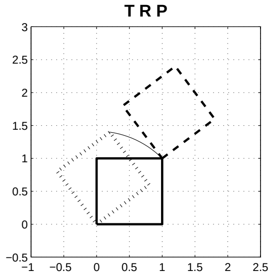
$$RT = \begin{bmatrix} C & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} I & t \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} C & Ct \\ 0 & 1 \end{bmatrix}$$

$$TR = \begin{bmatrix} I & t \\ 0 & 1 \end{bmatrix} \begin{bmatrix} C & 0 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} C & t \\ 0 & 1 \end{bmatrix}$$

Both products have the same structure, but in the RT-case, the translation vector has been multiplied by C, i.e. t has been rotated.

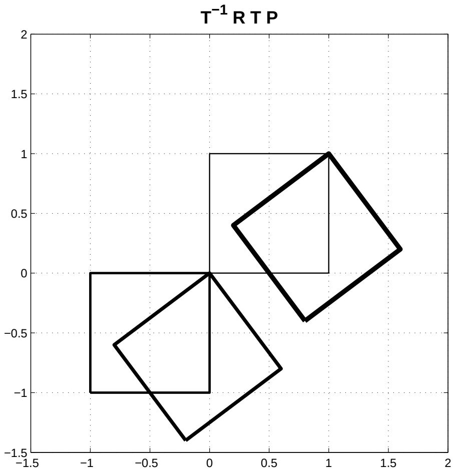
The following images show RT and TR acting on the unit square. The dotted unit squares shows the situation after the first step of the transformations has been applied. The dashed lines show the final result. As usual it is sufficient to look at the corners.





Suppose we would like to rotate the unit square around an arbitrary point (not just around the origin). It is easy to perform the transformation in three steps.
Pick $(1, 1)$ as the point (the upper right corner of the square). Let T translate this point to the origin ($t = [-1, -1]^T$). The following sequence gives the requested transformation: $M = T^{-1}\, R\, T$. In words: translate the point of rotation to the origin, rotate around the origin, translate back.
Note that T^{-1} corresponds to a translation with $t = [1, 1]^T$.

The following image shows the steps. I have increased the linewidth in each step.



OpenGL and transformations

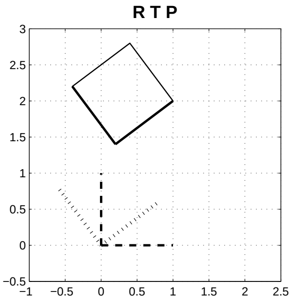
Let us return to the RT, TR-example and see how this is done in OpenGL (at least in principle, all the details will come later). Suppose we do the following function calls in OpenGL (... marks parameters that we skip for the time being):

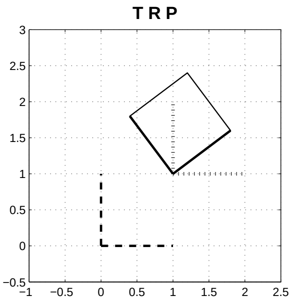
```
Call                   Matrix operation

glLoadIdentity();      M = I       // M = Model matrix
glRotatef(...);        M = M* R  // affects coming
glTranslatef(...);     M = M* T  // points

glVertex3fv(point);  // plot(M* point) (roughly)
```

First we note that OpenGL uses post-multiplication, every new transformation matrix is multiplying M from the right. Pre-multiplication would multiply M from the left. So, after the calls, $M = RT$ even though we made the Rotate first and the Translate after.

To get $M = TR$ we must first call Translate and then Rotate. So why this strange order (post and not pre)? The reason is that post is the correct order if we take the view of transforming coordinate systems rather than points.

The following image shows how the coordinate systems are transformed (the same example as before, but now with coordinate systems and not squares/points). The original system is dashed, the next one (after the first transformation) is dotted and the last is plotted using solid lines.





In the first image $M = RT$, so the order of the OpenGL-calls are Rotate, Translate. The dotted system is rotated relative the original. The solid has been translated relative the rotated system. The unit square is drawn using the last system.

When $M = TR$ Translate is called first and then Rotate. The dotted system has been translated, and the next system has been rotated relative to the newly translated system. The unit square is drawn using the last system.

A common problem

Suppose we have drawn an object (like in the house-example) in a nice coordinate system centered on the origin. We would like to place copies in different positions in the plane. Suppose the object should be placed in the four positions $(1,0,0)$, $(0,1,0)$, $(-1,0,0)$ and $(0,-1,0)$. Assume that our C-routine, `DrawObject()` draws the object.

Will the following code sequence give the required result?

```
  glLoadIdentity();
  glTranslatef(1, 0, 0);  // x = 1, y = 0 (z = 0)
  DrawObject();

  glTranslatef(0, 1, 0);  // x = 0, y = 1 (z = 0)
  DrawObject();
etc.
```

The answer is no! The second glTranslate can be interpreted relative to the translated system (made by the first Translate). The second object will be drawn in $(1,1,0)$, in other words.

It would be possible to, in the second glTranslate, correct for the first and write `glTranslatef(-1, 1, 0);` This will be rather complicated if we have many transformations.

A better alternative is to save the old coordinate system (the old M) and make a temporary change. OpenGL has three matrix stacks (for different kinds of transformations). We are going to use the stack for the `GL_MODELVIEW`-matrix (in which M is a part).

```
  glMatrixMode(GL_MODELVIEW); // Choose type of matrix
  glPushMatrix();             // Save current M
  glTranslatef(1, 0, 0);      // x = 1, y = 0 (z = 0)
  DrawObject();
  glPopMatrix();              // Fetch saved M

  glPushMatrix();             // Save current M
  glTranslatef(0, 1, 0);      // x = 0, y = 1 (z = 0)
  DrawObject();
etc.
```

Transformations in 3D

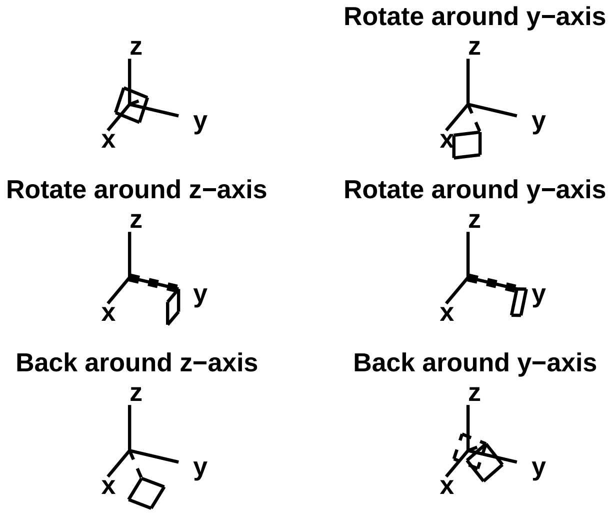
Here are the most common transformations in 3D. S = scaling, T = translation.

$$S = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \quad T = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

R_x = rotation ccw the angle ψ around the x-axis (when we look along the negative x-axis). Analogous for R_y and R_z.
Let $c = \cos\psi$ and $s = \sin\psi$.
Note that we have the number one for the axis.

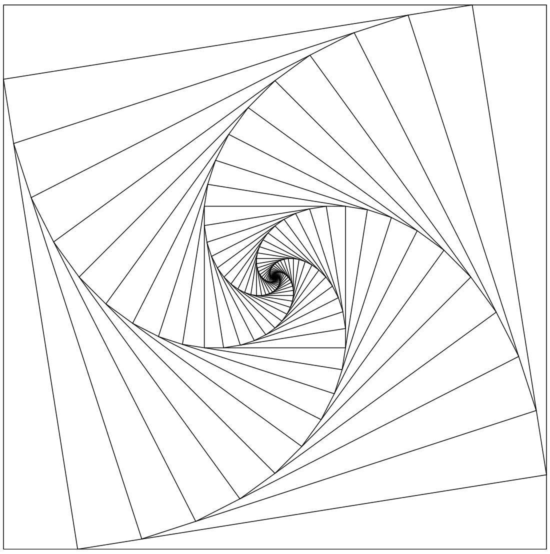
$$R_x = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & c & -s & 0 \\ 0 & s & c & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \quad R_y = \begin{bmatrix} c & 0 & s & 0 \\ 0 & 1 & 0 & 0 \\ -s & 0 & c & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \quad R_z = \begin{bmatrix} c & -s & 0 & 0 \\ s & c & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

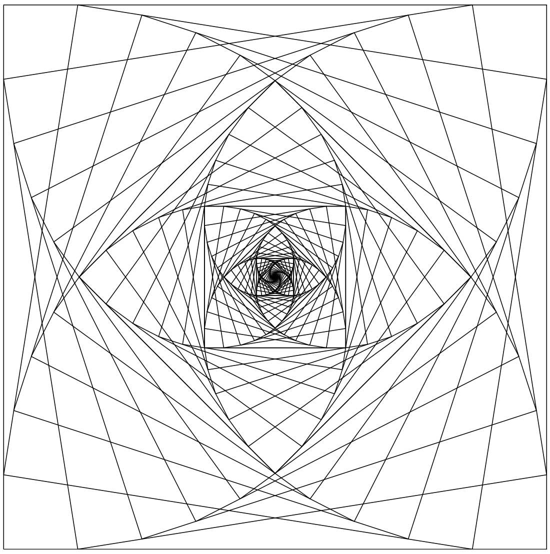
To make more complicated transformations we can combine the above. Here is an example where we want to rotate the square around the dashed axis.

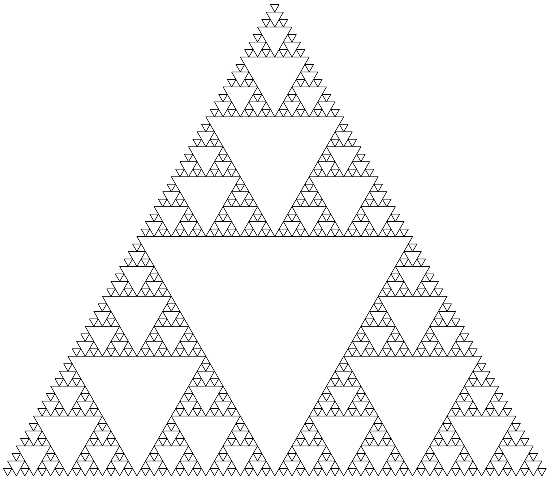


A few exercises

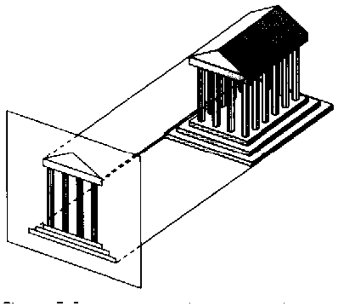
- Find the M-matrix that maps the rectangle, with corners in $(1,1)$, $(3,3)$, $(2,4)$ and $(0,2)$, onto the unit square.
- Find the M-matrix which maps the quadrilateral, with corners in $(0,0)$, $(1,0)$, $(2,1)$ and $(1,1)$, on the unit square. This is an example of a shear transformation.
- Let $R(\psi)$ be a rotation matrix in 2D. Why is it true that $R(\psi)R(\varphi) = R(\psi+\varphi)$?
  Use this equality to prove the additions laws:
  $\sin(\psi+\varphi) = \sin\psi\cos\varphi + \cos\psi\sin\varphi$ etc.
- Find the M-matrix which mirrors the plane in the y-axis (i.e. the point (x,y) is mapped onto $(-x,y)$).
- Do the same for the plane $x = c$ (c is a constant).
- Which M-matrices keep distances between (arbitrary) points?
- Which M-matrices preserve angles between vectors?
- Suppose we have two sets $\mathcal{P}$ and $\mathcal{Q}$ where each set contains three distinct points. Is there always an M-matrix which maps $\mathcal{P}$ onto $\mathcal{Q}$?
  (Hint: think in geometrical terms.)
- Write a Matlab program that creates the image on the next page. The program should start with a square and then transform it. The second image contains two images of the above kind, on with the rotation $R(\psi)$ and the other with $R(-\psi)$.
- Use recursion in Matlab to draw some type of the Sierpinski triangle (last page).



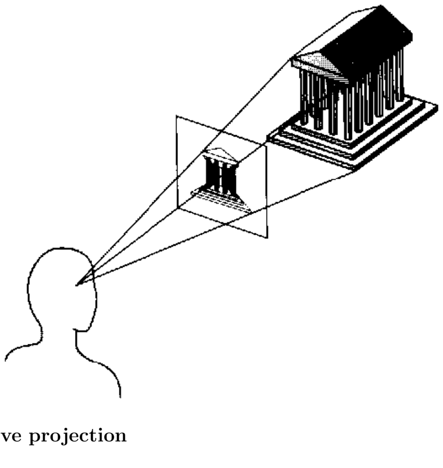




Projections

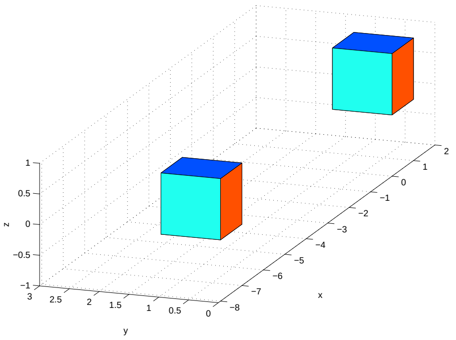


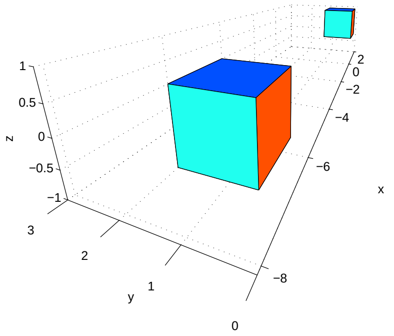
Orthografic (parallel) projection

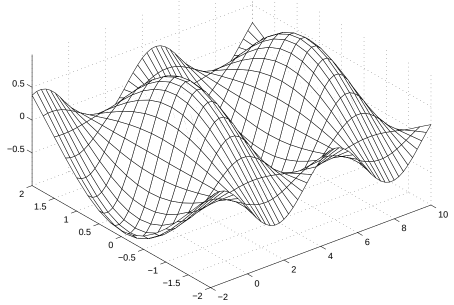


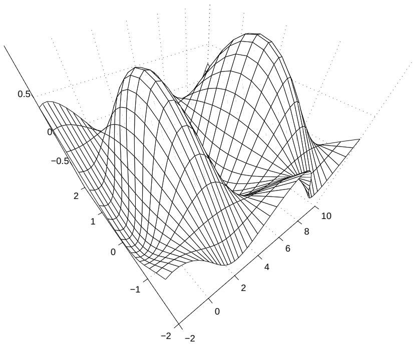
Perspective projection

```
set(gca, 'Projection', 'orthographic', ... % first plot
        'CameraPosition', [-10 -1 2])}
set(gca, 'Projection', 'perspective', ...  % second plo
        'CameraPosition', [-10 -1 2])}
```









Projections, the modelview matrix

Transformations in a simple OpenGL-program.

```
  // Define the projection
  // Orthografic in this case

  glMatrixMode(GL_PROJECTION); // Projection matrix
  glLoadIdentity();            // Matrix = I
  glOrtho(-1,3, -1,3, 0,4);    // Multiply
...
  // Place the eye (camera).
  // gluLookAt(eye_pos, look_at, up_direction)

  glMatrixMode(GL_MODELVIEW);      // Modelview matrix
  glLoadIdentity();                // Matrix = I
  gluLookAt(1,0,1, 0,0,0, 0,1,0);  // Multiply
```

later in the program

```
  // Create the transformation for coming points
  glTranslatef(2, 0, 0); // transformations
  glRotatef(...          // etc.

  // Points, affected by the above transformations
  glColor3f(1, 0, 0);    // Choose colour
  glBegin(GL_QUADS);     // Rectangle
    glVertex3f(0, 0, 0); // Define corners
    glVertex3f(1, 0, 0); // that are sent
    glVertex3f(1, 1, 0); // through the
    glVertex3f(0, 1, 0); // graphics pipeline
  glEnd();

  // We can modify CT (Current Transformation)
  glTranslatef(1, 1, -1); // multiplies M

  // and define new objects
  glBegin(GL_QUADS);
   ...
```

The eye is initially in the origin looking along the negative z-axis. The up-direction is along the positive y-axis. We can only see part of the room, the view volume, which is specified using `glOrtho` (for an orthografic projection) or by `gluPerspective` (for a perspective projection). There are other routines as well.

`glOrtho` takes the following parameters:

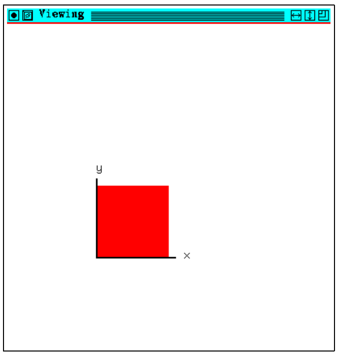
```
glOrtho(x_min, x_max, y_min, y_max, near, far)
```

The view volume is a box (rectangular parallelepiped).

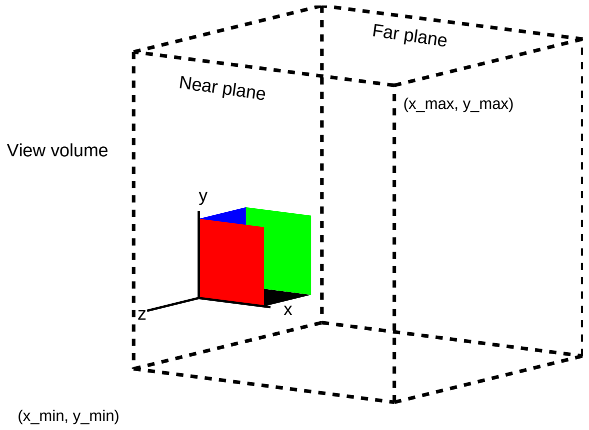
Here is an example. I have created a square window and made the call:

```
glOrtho(-1,3, -1,3, 0,4);
```

`gluLookAt` has not been called so the eye remains in the initial position. The image contains four squares and a coordinate system placed in the origin. So initially the eye is located in the origin and is looking at the red square. Here is the plot window.



This Matlab-plot shows the situation from another direction. The view volume is marked with dashed lines.



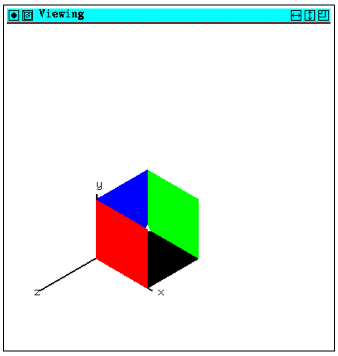
In this example the read square lies in the “near plane”. If we increase **near** the near plane is moved away from the eye (more negative z). Part of the scene will clipped away (one talks about the near clipping plane, as well). In the same way we get clipping if we move the “far plane” towards the eye (if we decrease **far**).
We can get clipping in the x- and y-directions as well.

To see something more than the red square we can move the objects or, equivalently, move the eye.

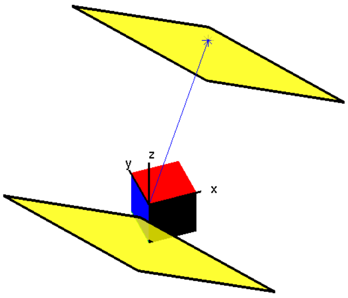
Here is the window after the following call:

```
gluLookAt(2,2,2, 0,0,0, 0,1,0);
```

Note that the view volume is “attached” to the eye (like one has glued the view volume to the front of ones head). The volume is not “deep” enough, one corner is clipped.

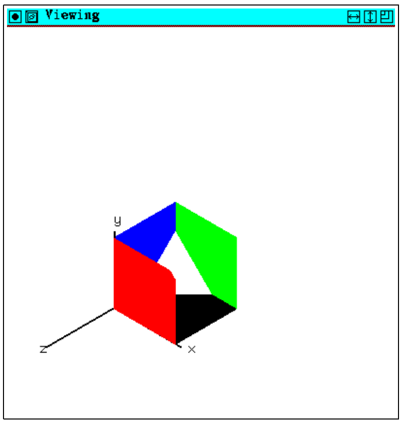


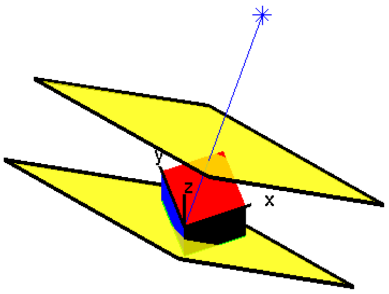
Here is a Matlab illustration, from an other angle:



Let us decrease the volume (increase near and decrease far) even further (note the use of transparency in Matlab, **help alpha**):

```
float d = 2 * sqrt(3) - 2 / sqrt(3);
glOrtho(-1,3, -1,3, d+0.05, 3.7);
```





The transformation of points correspond to matrix multiplications which generate the model matrix, M (each transformation updates M). How does **gluLookAt** work? We can change the view in two (equivalent) ways.

- We move the eye but not the points.
- We move the points but not the eye. This corresponds to extra modeling transformations.

So suppose we set the view first and then apply modeling transformations. This can be seen a matrix multiplication using a view matrix, V, computing $M = VM$. Note that only the product, VM, is stored, and we refer to the product as the modelview matrix. Let us look at two simple examples.

```
gluLookAt(0,0,1, 0,0,0, 0,1,0);
```

The eye should be placed in $(0, 0, 1)$ and look at the origin, $(0, 0, 0)$.
The up direction is $(0, 1, 0)$ (the y-axis). We can generate this view by translating all points by $(0, 0, -1)$ (one step along the negative z-axis). The view matrix, V, should consequently be:

```
1 0 0  0
0 1 0  0
0 0 1 -1
0 0 0  1
```

We can check that is the case, using the following code sequence:

```
...
  GLenum error;        // to be on the safe side
  float V[16];                   // memory for V
  char format[] = "%5.1f %5.1f %5.1f %5.1f\n";

  glMatrixMode(GL_MODELVIEW);     // choose MV-matrix
  glLoadIdentity();              // MV = I

  gluLookAt(0,0,1, 0,0,0, 0,1,0);        // multiply
  glGetFloatv(GL_MODELVIEW_MATRIX, V);   // fetch V

  // note, stored in Fortran order, column-wise
  printf(format, V[0], V[4],  V[8], V[12]);
  printf(format, V[1], V[5],  V[9], V[13]);
  printf(format, V[2], V[6], V[10], V[14]);
  printf(format, V[3], V[7], V[11], V[15]);

  error = glGetError();  // problems?
  if ( error != GL_NO_ERROR )
    printf("glGetError = %d\n", error);
...
```

```
% gcc modelview.c -lGL -lglut
% a.out
  1.0   0.0   0.0   0.0
  0.0   1.0   0.0   0.0
  0.0   0.0   1.0  -1.0
  0.0   0.0   0.0   1.0
```

I will talk more about `glGetError` when we come to the OpenGL lectures.

Let us now analyze the call:

```
  gluLookAt(1,0,1, 0,0,0, 0,1,0);
```

The eye should be placed in $(1, 0, 1)$, a translation as above, but we also make a rotation 45° around the y-axis. Moving points, we first make the rotation $-45°$ ccw (i.e. 45° cw) looking along the negative y-axis. Then we perform the translation $(0, 0, -\sqrt{2})$. Let us do this in Matlab:

```
>> T = eye(4);
>> T(3, 4) = -sqrt(2);   % translation

>> a = -pi / 4;  % angle
>> c = cos(a);
>> s = sin(a);
>> R = [c  0  s  0       % rotation
        0  1  0  0
       -s  0  c  0
        0  0  0  1];

>> V = T * R             % note the order

V =
    0.7071         0   -0.7071         0
         0    1.0000         0         0
    0.7071         0    0.7071   -1.4142
         0         0         0    1.0000
```

which is in accordance with the printout from the OpenGL-program.

In the examples above, $M = I$, so let us set both matrices.

```
  glLoadIdentity();
  gluLookAt(1,0,1, 0,0,0, 0,1,0);   % changing V
   ...

  glTranslatef(1, 1, 1);            % changing M
```

We continue using our V from the Matlab-program.

```
>> M = eye(4); M(1:3, 4) = [1 1 1]’
M =
     1     0     0     1
     0     1     0     1
     0     0     1     1
     0     0     0     1

>> V * M
ans =
    0.7071         0   -0.7071    0.0000
         0    1.0000         0    1.0000
    0.7071         0    0.7071    0.0000
         0         0         0    1.0000
```

which is OK as well.

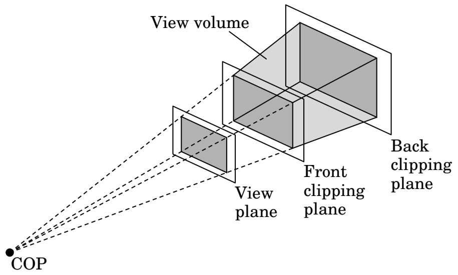
One can set the matrix as well:

```
  glMatrixMode(GL_MODELVIEW);
  glLoadMatrixf(matrix_data);  // sets MV
```

We choose perspective projection by:

```
gluPerspective(view_angle, aspect_ratio, near, far);
```

where `view_angle` is the field of view angle, in degrees, in the y direction `aspect_ratio` is the ratio of x (width) to y (height). `near` and `far` as before.



On its way to the screen a point will undergo several transformations. The point is sent through a graphics pipeline.

A somewhat simplified picture, and only for orthografic projections, looks like this:

- The point is first multiplied by the modelview matrix. Note that this matrix can be changed at a later stage.
  These, later, changes do not affect our point. The current matrix is often called CT (Current Transformation). We send a point through the pipeline by using `glVertex`.
- The next step is multiplication with the projection matrix which has been created by `glOrtho` (or `gluPerspective`).
  This matrix transforms the point so that they reside in the standard cube ((-1, 1) in each dimension).
  This step is more complicated for perspective projections. The direction of the z-axis is reversed, so that increasing values of z correspond to a larger distance from the eye.
  After this step the objects are usually deformed, but that is fixed in the last step.
- Clipping (removal of parts outside the standard cube) is the next step. The clipping has been made easier since we can cut against the sides of a cube.
- The last step is to map the standard cube onto a 3D "viewport", where x and y correspond to a rectangular part of the screen, and z lies in [0, 1]. `glViewport` sets up the viewport; more about this later.

An example of a projection matrix. Suppose we have made the call:

```
glOrtho(0,1, -1,5, 0,4);
```

The projection matrix, P, should map the view volume onto the standard cube. The first step is to make a translation (the centre of the view volume should be mapped to the origin) and then a scaling so that all sides has length two.

In our example, the view volume is defined by: $0 \leq x \leq 1$, $-1 \leq y \leq 5$ and $-4 \leq z \leq 0$. So the following transformation should work:

```
T = eye(4); T(1:3, 4) = [-0.5; -2; 2]  % translate
```

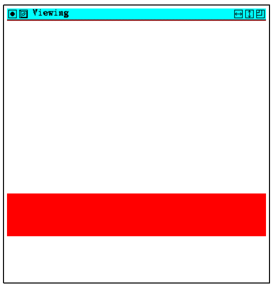
and then scale

```
S = diag([2 1/3 -1/2 1])  % -1/2, reversal of z-axis
```

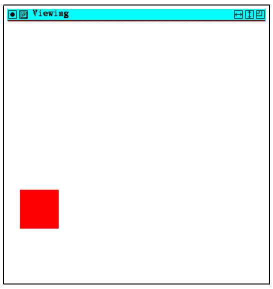
The product `S * T` is what OpenGL produces as well:

```
GL_PROJECTION_MATRIX
      2        0        0       -1
      0    0.333        0   -0.667
      0        0     -0.5       -1
      0        0        0        1
```

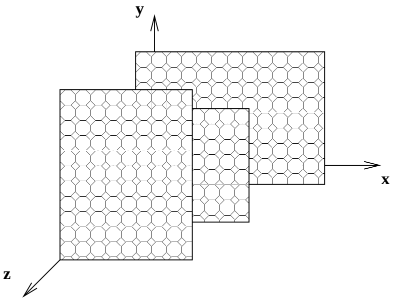
On the next page I have plotted the unit square using the view volume above and without moving the eye. The window was 400x400 pixels and the viewport had the same dimension as the window.
We can see that the square is deformed.



Let us set the viewport: `glViewport(20, 20, 60, 360);`
The lower left corner of the viewport is 20 pixels to the right and above the lower left corner of the window. The width of the viewport is 60 pixels and the height is 360. So the ratio between height and width is six, which is the same ratio we had in `glOrtho`, $0 \leq x \leq 1$, $-1 \leq y \leq 5$. This causes the square to get correctly scaled.



Removing hidden objects



we are looking along the negative z−axis

The basic painter's method:

1. compute the centre of mass (for example) for each polygon
2. sort the polygons according the z-coordinates of the centres
3. paint the polygons, in order of increasing z-coordinates

The depth buffer (z-buffer) method. We have a matrix (z-buffer) containing the distances from a point to the eye and a "framebuffer" (image memory) where we store the pixels.

```
set all element in the z-buffer to the distance to
the back clipping plane

for each polygon
  for each pixel, with coords. (x, y, z), in the polygo
    if z < z_buffer(x, y) then
      z_buffer(x, y)    = z
      framebuffer(x, y) = the colour in (x, y, z)
    end if
  end
end
```

In Matlab we can choose between several methods:

```
>> h = figure;
>> set(h, 'Renderer')
[ {painters} | zbuffer | OpenGL | None ]
```

`None` gives no rendering at all.
Here are some pros and cons with the different methods.
`painters`: fast for simple figures, users vector graphics (lineto, moveto), good for PostScript, gives high resolution. Cannot handle light, transparency or 24-bit colour surfaces. Can draw incorrect figures (example next page).

`zbuffer`: uses bitmap (raster) graphics, faster than painter's (when complex figures), can use a lot of memory, can cope with light but not transparency.

`opengl`: uses bitmap (raster) graphics, the fastest for complex scenes (tries to use the machine's graphics hardware), can handle both light and transparency, but not Phong shading (later).

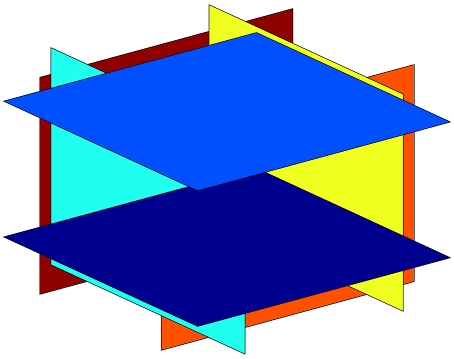
`opengl` sometimes renders images in an incorrect way.

A disadvantage with both `zbuffer` and `opengl` is that the PostScript files can be very large.

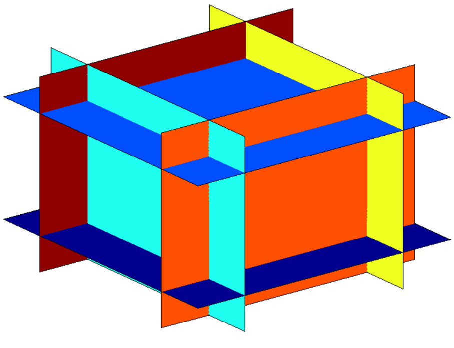
```
>> peaks  % a demo-command that draws a surface
>> set(1,'renderer')
[ {painters} | zbuffer | OpenGL | None ]
>> print -depsc peak_paint.eps

>> set(1,'renderer', 'zbuffer')
>> print -depsc peak_z.eps

>> set(1,'renderer', 'opengl')
>> print -depsc peak_ogl.eps

>> !ls -s peak*
6384 peak_ogl.eps   432 peak_paint.eps  6384 peak_z.eps
```

So the raster images require more than fifteen time as much space. It is possible to change the print-resolution (`help print`, see the `-r` option).

Note that `opengl` gives <u>much</u> faster graphics, on the math-machines. Very useful if we want to rotate a complex image, for example. The following images show one major disadvantage with the `painters` algorithm in Matlab.



`set(h, 'Renderer', 'painters')`



`set(h, 'Renderer', 'zbuffer')`

A few words about colours

The eye has two kinds of receptor cells. The cones are colour-sensitive and the rods that cannot distinguish colour nor see fine details. Each eye has $6 \cdot 10^6 - 7 \cdot 10^6$ cones, each with its on nerve cell, making it possible to see fine details.
The cones are concentrated in a small area, the fovea, in the centre of the retina. The fovea, also called the "yellow spot" is less than 1 square millimeter.

The number of rods is $75 \cdot 10^6 - 150 \cdot 10^6$, and many rods are attached to one nerve cell. The rods are spread out over the retina surrounding the fovea. The rods are use for night vision, and they will not be of interest in the following discussion.

Humans have three types of cones, sensitive to yellowish-green light (Long wavelength), bluish-green (Medium) and blue-violetish (Short) respectively. The last type is much less sensitive.
The peak wavelengths are 564 nm, 534 nm, and 420 nm respectively.

This (trichromatism) is the reason it is sufficient with three types of phosphors in a television tube and why we can use the RGB-system of colours in computer graphics.

Phosphor should not mixed up of with Phosphorus, one of the elements (symbol P). A typical phoshor is zinc sulfide with a few ppm of copper. When bombarded by electrons this phosphor will produce a green colour.
For more details: `http://en.wikipedia.org/wiki/Phosphor`

Not all animals have three types of cones, chickens have as many as 12 kinds of receptors, for example.

Not all humans have a complete set of cones; colour blindness. About 10% of males and 1% of females have some form of deficiency in their colour vision. The most common is a lack of receptors for the L-cones (protanopia) or for the M-cones (deuteranopia). This makes it hard to distinguish between red and green.
Note that even people with a full set of cones are less sensitive to blue. This is one reason why it is bad to present fine detail (e.g. small text) in blue on a black background. This is, unfortunately, not so uncommon on the web, and it makes for hard reading.

The RGB-system is the most common colour system in computer graphics. A colour is described by the amounts of the primary colours, red, green and blue. The minimum amount is zero and if the maximum amount is one, the RGB-triple [1,0,0] corresponds to red. [0,0,0] is black and [1,1,1] is white. [0.9,0.9,0.9] is light gray etc.

There are other colour systems. In the HLS-system we use hue (the type of colour from the spectrum), lightness and saturation (the intensity of the colour, the purity) instead.

Of more interest, in this course, is the CMY-system. Cyan, Magenta and Yellow are the so called complementary colours of red, green and blue. Complementary, in the sense that cyan+red=magenta+green=yellow+blue all equal white.
So the RGB-triples for cyan is [0,1,1], magenta has [1,0,1] and yellow [1,1,0].

The RGB-system is an additive colour system, we add R, G and B, to black, to get our colour. In a subtractive system, like CMY, we start with white light and remove colours (think of using a filter) to produce the colour.

To see how this works let us take the CMY-triple [0.4,0.5,0.2]. The corresponds to the RGB-triple [1,1,1]-[0.4,0.5,0.2]=[0.6,0.5,0.8]. To describe the “subtraction” we let white light pass through three filters.

The first has the CMY-triple [0.4,0,0] (corresponding to RGB [0.6,1,1], looks like light cyan). This filter will remove 0.4 of red. The next filter has CMY [0,0.5,0] (RGB [1,0.5,1], light magenta) and it removes 0.5 of green. The last filter, finally, has CMY [0,0,0.2] (RGB [1,1,0.8], light yellow) removes 0.2 of blue. The resulting colour is RGB [0.6,0.5,0.8] (a kind of grayish purple, I think it looks like).

This is interesting when we, later, are going to look at the diffuse reflection of light. Suppose white light is reflected from a non-shiny surface, having the RGB-colour [0,1,1]. The reflected light is void of red. (Reflection from shiny surfaces tend to be white, regardless of the colour of the surface.)

This is used in printing, where the CMYK-system is common. K stands for black (you can find the etymology below). Mixing cyan-coloured pigments into a colourless paint will remove the red colour component from the incoming white light and reflect green and blue. Mixing C, M and Y would, in theory, remove all the light giving a black surface. So why is a separate black ink used for printing?

There are several reasons, according to Wikipeda: the mix of CMY becomes “a dark murky color”. Using so much ink would make the paper wet, requiring longer times for drying and high quality paper. It is easier to write details (text) using black, rather than having to mix three inks. Black ink may be cheaper.

There are problems mixing colour systems, since physical devices such as a monitor or printer may have different colour ranges (usually called the colour gamut of the device). The colour gamut of a printer is usually a subset of that of a monitor. The primaries R, G and B may different on different monitors, as well. It is not uncommon that a colour image looks different on two different systems.

Even if an RGB-colour on the monitor is representable on the printer the relationship may be complicated. There are commercial systems, colour samples on paper with a unique code (like when you buy a new car or wallpaper; the systems, for printing, are not free, however). On can pick the colours one needs and tell the printer the codes. The printer should know how to produce the correct colours given the codes.

On the math-computers we have so called 24-bit colour (often called true colour). Each pixel is represented by three bytes, one each for red, green and blue. The total number of different colours is $(2^8)^3 = 2^{24} = 16\ 777\ 216$. Each byte can store an unsigned integer between 0 and 255. So white is represented by the RGB-triple [255,255,255].

On older systems a colour look-up table (CLUT) was often used. Think of the CLUT as being a matrix, with three columns, one for each of the primaries. The number of rows equals the number of colours (a power of two, so 64, 128 or 256 colours, perhaps).

The pixels in the image store a row index, into the CLUT (so this is often called indexed colour). Using 256 rows in the CLUT makes the required memory for the image smaller (only one byte per pixel instead of three).

One, very noticeable drawback is that each application (program) usually has its own CLUT. When one moves the mouse between windows, different CLUTs are used, but since a particular CLUT is used for all the windows on the screen there will be a lot of colour flashes.

Some etymology (with web-sources):

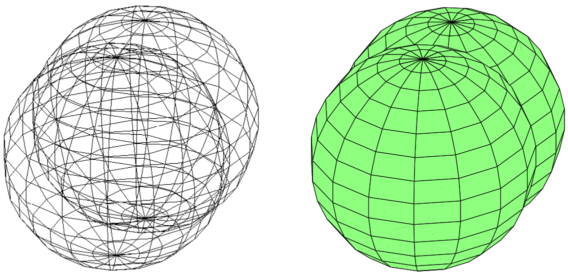
magenta: 1860, in allusion to the Battle of Magenta, in Italy, where the French and Sardinians defeated the Austrians in 1859, because the brilliant crimson aniline dye was discovered shortly after the battle.... `www.etymonline.com`

About K for black: In printing, a key plate was the plate which printed the detail in an image. When printing color images by combining multiple colors of inks, the colored inks usually did not contain much image detail. The key plate, which was usually impressed using black ink, provided the lines and/or contrast of the image... `www.wikipedia.org`

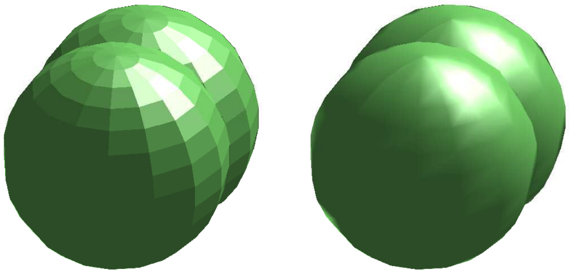
Gamut: Medieval Latin gamma, lowest note of a medieval scale (from Late Latin, 3d letter of the Greek alphabet)
1: the whole series of recognized musical notes
2: an entire range or series “ran the gamut from praise to contempt” `www.m-w.com`

Shading models

Say we want to draw a green billiard ball. Here are some example showing increasing levels of realism. Wire frame (left image), hidden surface removal (right).



Adding light: flat shading (left image), one colour for each polygon. We can smooth out the colours: Gouraud- or Phong-shading. Add highlights, “specular light” (right image).

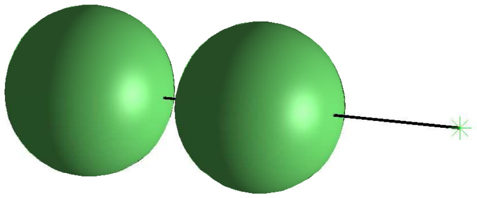


Shading does nor mean “shadows”, but it means to color so that the shades, of colour, pass gradually from one to another.

We would like to mimic different surface textures and materials: balls for billiards, tennis. Steel, copper etc. OpenGL does not support the rendering of shadows, or realistic reflection and refraction. If you have such needs look at the links (raytracing).

OpenGL does the light computation for each polygon and then each pixel at a time. Shadows, reflection take too much time, and are faked, but some physics is used.

In the following image the green * marks the light source. Note that the sphere, to the left, gets as much light as the one to the right, even though the left one is hidden.



Normal vectors will be important as will the location of the eye and the direction of the incoming light. We can make the computations for each of the primaries separately and then add the resulting components at the end.

Two types of light sources:

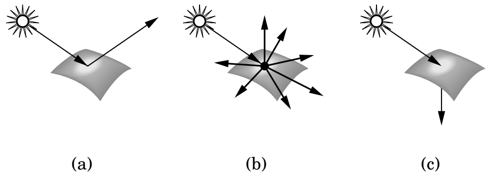
- Point sources (can shine in all directions, like the Sun, or in a limited cone, like a spotlight). We can have distant light sources (the Sun) or local (a table lamp). It is faster to do the computations for distant light sources since only direction and not actual distance has to be considered.
- Ambient light (surrounding) gives a general level of light in the scene. This light source has not position or direction; light is spread equally in all directions. Since OpenGL does not handle the reflection, refraction etc. of light in a realistic manner it must be faked. Without ambient light we get sharp contrasts in the scene. Too much ambient gives a watered down, insipid look.

Från Merriam-Webster: `www.m-w.com`
Etymology: Latin ambient-, ambiens, present participle of ambire to go around, from ambi- + ire to go – more at
ISSUE.
Date: 1596
: existing or present on all sides : ENCOMPASSING

We do not set the colour using RGB-vectors, instead there are intensities for the light sources and material properties (reflection coefficients) for the objects (points) in the scene.

The ambient light has intensity I_a, really one for each of the primaries, so så I_{ar}, I_{ag} och I_{ab}. Let I_a stand for one of them.
Each corner of each polygon has a reflection coefficient for ambient light ρ_a (or rather ρ_{ar}, ρ_{ag} och ρ_{ab}). The corner gets the light contribution $\rho_a I_a$ (for each primary). The colours of the corners will later be used to colour the whole surface of the polygon.

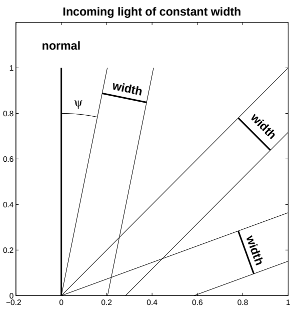
We now look at light having a direction, and we will see how much is reflected to the eye.



(a) (b) (c)

(a) shows specular reflection (billiard ball)
(b) shows diffuse reflection (tennis ball). The reflected light is spread equally in all directions.
(c) shows transparency and refraction. Transparency can be simulated in OpenGL and Matlab.

We start with diffuse reflection. Since the light is spread equally in all direction the position of the eye does not affect the light computation (as long as the eye sees the front of the polygon). The position of the polygon relative to the light source is of importance, however.



Suppose that the ray has width w_r. It should be spread out over an interval, of length $w_x = w_r / \cos\psi$, along the x-axis.

The intensity of the light, along the x-axis, is proportional to $1/w_x$ i.e. to $\cos\psi$. So if the incoming light has intensity I_d, the reflected light has intensity $\rho_d I_d \cos\psi$ (for each primary).
This is called Lambert's law.

We can use vectors to compute $\cos\psi$.
Let us only consider solid objects having outward normals. Note that OpenGL does not compute normals for us (Matlab does) so we have to fix them.

Let L be the normalized direction to the light source, and let n be the normal to the surface in the point where the ray hits, then $\cos\psi = L \cdot n$.
If $L \cdot n < 0$ the backside of the polygon is hit by the light, but according to our assumption we cannot see that side, so the intensity becomes:

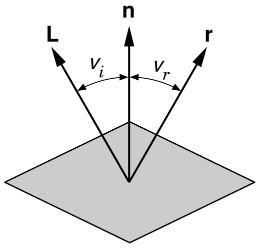
$$\rho_d I_d \max\left[L \cdot n, 0\right]$$

It is common to take $\rho_a = \rho_d$.

More etymology:

Main Entry: specular
Etymology: Latin specularis of a mirror, from speculum
Date: 1661
: of, relating to, or having the qualities of a mirror

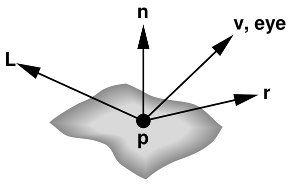
Next, specular reflection. If we have a perfectly polished surface and a spotlight is located in the L-direction, the eye will see a reflected ray only if it is located along r.



Real-life surfaces are not perfect, so a more realistic model will show light also in the vicinity of the r-direction. The amount of reflected light should decrease when we move away from r.
The Phong reflection model (Bui Tuong Phong, b. Vietnam, 19??-1998) tries to capture this behaviour. The intensity of the reflected light is

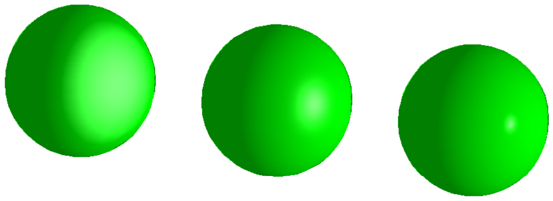
$$\rho_s I_s (r \cdot v)^f$$

r is as above and v is the normalized direction to the eye. f is the "specular reflection coefficient" and it measures how much the light is spread. A large f gives a small spread of light and a small f gives a large spread. OpenGL approximates the angle, by using the angle between n and $L + v$ (which is $\psi/2$ if all the vectors lie in the same plane). This makes it unnecessary to compute r (faster).

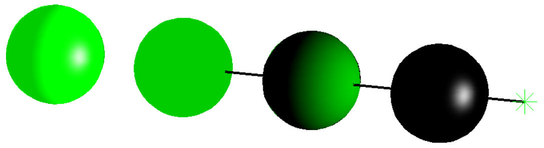


This is how the intensity varies with f:

From left to right: f = 1, 10, 100



The following image shows, from right to left, specular, diffuse and ambient. The last sphere is rendered using all three.



The colour of the light matters as well. If we use red light on a green sphere (using only diffuse), it will be black. The reason an object is green is because it reflects green light.

If we have a local light source (not the Sun, say) the distance is taken into account. The intensity of the source should decay as $1/r^2$ (where r is the distance), but this does not look realistic, so the programmer can set up a fake decay rate: $1/(a + br + cr^2)$ (a, b and c can be adjusted).

We are now ready to add together the intensities. We should add over all light sources and for the three primaries: så:

$$I = \frac{\rho_a I_a + \rho_d I_d \max\left[L \cdot n, 0\right] + \rho_s I_s \max\left[\frac{L+v}{||L+v||} \cdot n, 0\right]^f}{distance}$$

It is possible to add a general ambient source, which is not bound to any point. There is also "emissive" color; an object can glow, for example. Finally there is a factor for spotlights, which emit light in a cone.

We have now computed a colour in each corner, and it is time to colour the whole polygon, pixel by pixel. This can be done in several ways. If we use the same colour for all the pixels, one talks about flat shading. In this case we use one normal for the whole polygon. The surface gets a faceted appearance.

To create smooth shading we must create more normals (by calling `glNormal`). Suppose we have one normal in each corner. In Matlab there is support for Gouraud shading and for Phong shading. OpenGL only supports Gouraud shading.

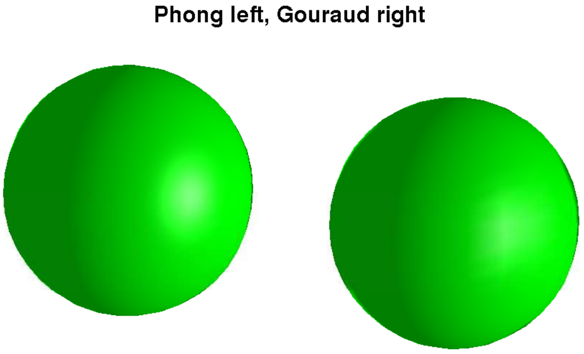
Suppose this is the polygon, with corners a-d:

```
    c-----------d
   /             \
p1/------ p ------\p2
 /                 \
a------------------b
```

When colouring the polygon OpenGL works pixel-row by pixel-row (scan lines). Suppose pixel p should be coloured. In Gouraud shading we use linear interpolation of the intensities in a and c to get a value in p1. Similarly the intensities in b and d are combined to form a value in p2. Finally, the intensities in p1 and p2 are combined to give the final value in p.

Phong shading gives a more realistic result, but it takes more time to compute. Here new normals are computed in p1 and p2 using linear interpolation (as for the intensities in Gouraud shading). Using linear interpolation we compute a new normal in p. This new normal is used for doing the light computation in pixel p.

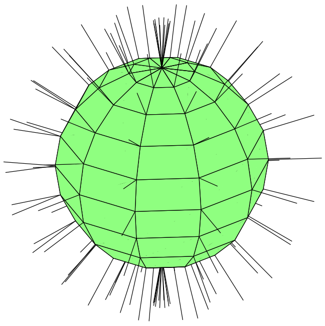
In this image one can (on the screen at least) see that Phong shading gives a less jagged highlight.



Normals in Matlab

When we create polygons and surfaces in Matlab, the normals will be created for us. Consider the following code:

```
>> [X, Y, Z] = sphere(10); % type sphere for the code
>> h = surf(X, Y, Z, ones(size(X)));
>> get(h)  % part of the output
        XData = [ (11 by 11) double array]
        YData = [ (11 by 11) double array]
        ZData = [ (11 by 11) double array]
        FaceLighting = flat
        EdgeLighting = none
        AmbientStrength = [0.3]
        DiffuseStrength = [0.6]
        SpecularStrength = [0.9]
        SpecularExponent = [10]
        SpecularColorReflectance = [1]
        VertexNormals = [ (11 by 11 by 3) double array]

% Run this code...
hold on
N = get(h, ’VertexNormals’);

d = 0.5;
for j = 1:11
  for k = 1:11
    x = X(j, k); y = Y(j, k); z = Z(j, k);
    n = [N(j, k, 1) N(j, k, 2) N(j, k, 3)];
    n = d * n / norm(n);  % not normalized
    plot3([x, x+n(1)], [y, y+n(2)], [z, z+n(3)], ’k’)
  end
end
view(-3.5, 28)
axis equal
axis off
```



Not quite the normals we would like. Matlab produces normals to the polygons (it seems) but we would like to have the normals of the sphere. Like this:

```
...
for j = 1:11
  for k = 1:11
   N(j, k, 1) = X(j, k);
   N(j, k, 2) = Y(j, k);
   N(j, k, 3) = Z(j, k);
  end
end

set(h, ’VertexNormals’, N)
```

One cannot see any difference, however.
By setting the normals to a random matrix produces differences (when light has been switched on):

```
>> set(h, ’VertexNormals’, randn(size(N)))
```

Why does the surface look smooth with Gouraud- and Phong shading? This is because we have one normal in each point, so the polygons coming together in a point share this normal. This gives a continuous variation over the edges.

This is not quite the case when we use the `fill3`-command. Here is an example. I have reused the cylinder example.

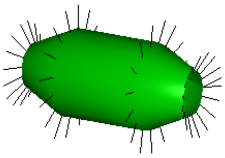
The first plot uses `surf` (and light etc). The lines are the normals (length 0.5).

The second plot uses `fill3`. I have reversed the direction of some normals. The four normals for one polygon have the same direction, so this gives something looking like flat shading.

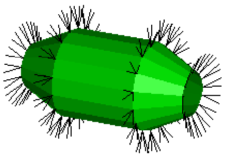
In the third plot, I use the same number of normals as in the second, but they have all been adjusted. This looks similar to the `surf`-plot.

The only problem with `surf` is where the cylinder is closed along a “seam”. The normals, on adjacent polygons along the seam, have different directions which gives rise to a difference in colour. So, to get a perfect result we should adjust the normals along the seam so that they have the same direction.

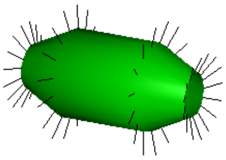
surf



fill3



fill3 + new normals



Colour and light in Matlab

Let us start without light and look at colour only. Matlab supports something like indexed colour as well as 24-bit colour. First something about indexed colour.

When we make a simple plot with mesh or surf, `surf(X,Y,Z)`, the colour is set by the height (z-value) in the following way. Each window has a `Colormap`-property (like the CMAP we discussed earlier). The default value is a 64x3 RGB-matrix. The entries are double precision numbers in [0,1] (not [0,255] in this case).

The smallest- and largest z-value are stored in a two-element vector `[cmin, cmax]`. Each axis-object stores such an array in its `CLim`-property. The index, `cmi`, into the CMAP for a specific c-value (c = z in this case), is given by the following expression:

```
cmi = fix((c-cmin) / (cmax-cmin)* cm_length) + 1
```

`cm_length` is the number of entries in the CMAP and `fix` rounds towards zero.

It is possible to change `[cmin, cmax]` using the `caxis`-command. This may be useful if we gradually add objects to a plot and would like to avoid changing colours (we must know zmin and zmax in advance). It may be useful when we have several axis (subplots) in one figure, as well. It is easy to change CMAP, `colormap(CMAP)`. CMAP does not have to have 64 entries.

`colormap(’default’)` sets the current figure’s CMAP to the default, JET. There are 13 builtin functions that generate CMAPs as well, such as `pink`, `copper`, `hot`, `summer`.
See the documentation for a complete list.

```
>> C = copper(4)  % a small CMAP
C =
           0            0            0
   4.1667e-01   2.6040e-01   1.6583e-01
   8.3333e-01   5.2080e-01   3.3167e-01
   1.0000e+00   7.8120e-01   4.9750e-01
>> colormap(C)   % changes the figure immediately

>> colormap(copper(128))  % we don’t have to use C
```

There is even a CMAP-editor, `help colormapeditor` (the Java-gui must be switched on). `brighten` is another command.

Suppose we supply an extra matrix in the surf-command: `surf(X,Y,Z,C)`. In this case `caxis` contains the min- and max of `C` and the c-values in the formula above are the `C(j, k)`-elements.

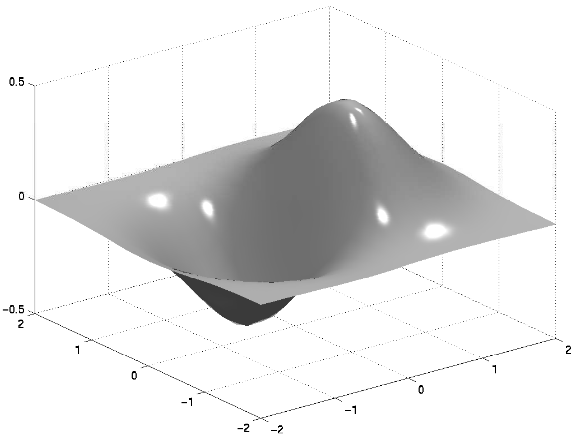
The `colorbar`-command places a colour bar in the plot. This provides a connection between colour and the numerical values in the `C`-matrix (or the z-values if no `C` is present).

Now for 24-bit colour. We can type `surf(X,Y,Z,C)`, where `C` is a 3D-matrix. `C(:,:,1)` contains the red component and the size should coincide with the coordinate data `X` etc. Here is a silly example:

```
>> [X, Y, Z] = peaks;  % get some data
>> C(:, :, 1) = ones(size(X));
>> C(:, :, 2) = ones(size(X));
>> C(:, :, 3) = zeros(size(X));
>> surf(X, Y, Z, C)  % gives a yellow surface
```

Let us finally look at light and shading in Matlab. It resembles OpenGL, but it is not always quite clear (to me) how it works. OpenGL is simple in the sense that everything is specified in the OpenGL standard.

```
[X, Y] = meshgrid(-2:0.2:2);
Z      = X .* exp(-X.^2 - Y.^2);

figure(1)
h = surf(X, Y, Z);

% Here are some of the properties.
% Matlab has ONE AmbientStrength etc. and not
% one for every primary. To a certain extent this
% can be adjusted using the colour data.
%
color = [1 1 1];
set(h,                          ...
 'FaceColor’,           color, ...
 'EdgeColor’,          'none’, ...
 'EdgeLighting’,      'phong’, ...
 'FaceLighting’,      'phong’, ...
 'AmbientStrength’,      0.23, ...
 'DiffuseStrength’,      0.28, ...
 'SpecularStrength’,     0.77, ...
 'SpecularExponent’,       90)

lh1 = light(’Position’, [-10 -4 4], ...
            ’Style’,   ’Infinite’);
lh2 = light(’Position’, [ 10  0 4], ...
            ’Style’,   ’Local’);
get(lh1)
...
        Position = [-10 -4 4]
        Color = [1 1 1]
        Style = infinite
```



There are some commands that set these properties for us. Let us look at what they do when we have plotted a surface with **`surf`** (slightly different things are done for a **`mesh`** since it does not fill the polygons, which **`surf`** does). This is what happens internally:

	`facecolor`	**`edgecolor`**
`shading flat:`	**`flat`**	**`none`**
`shading interp:`	**`interp`**	**`none`**
`shading faceted:`	**`flat`**	**`black`**

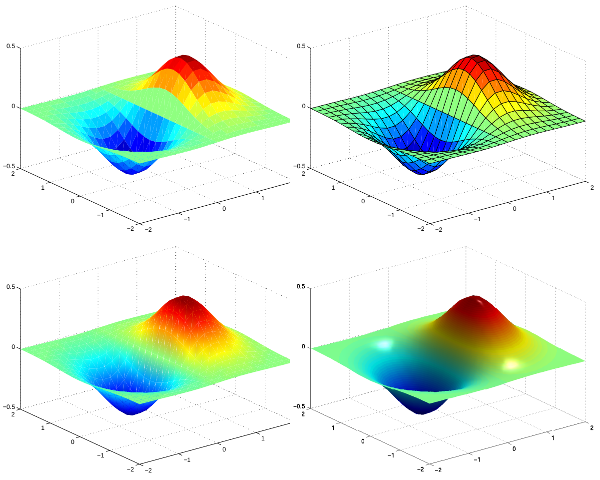
	`facelighting`	**`edgelighting`**
`lighting flat:`	**`flat`**	**`none`**
`lighting gouraud:`	**`gouraud`**	**`none`**
`lighting phong:`	**`phong`**	**`none`**
`lighting none:`	**`none`**	**`none`**

The **`material`**-command sets the reflection coefficients. It can be used in several ways, e.g.
`material shiny`
`material([ka kd ks])`
`material([ka kd ks n sc])`
and it sets (part of) AmbientStrength, DiffuseStrength, SpecularStrength, SpecularExponent and SpecularColorReflectance.

It is possible to use **`shading interp`**without using light. What it means is that a Gouraud-procedure is used to colour the inside of a polygon.

The best way to understand what happens is to try:

```
[X, Y] = meshgrid(-2:0.2:2, -2:0.2:2);
Z      = X .* exp(-X.^2 - Y.^2);

figure(1)
surf(X, Y, Z);
shading flat    % flat shading

figure(2)
surf(X, Y, Z);
shading faceted % flat shading with mesh lines

figure(3)
surf(X, Y, Z);
shading interp   % Gouraud shading

figure(4)
surf(X, Y, Z);
shading interp   % Gouraud shading

light            % default light
lighting  phong  % changes face- och edgelighting

% [rho_a rho_d rho_s spec_exp]
material([0.4 0.6 0.5 30])

% material metal
% material dull
% material shiny
% material default
```

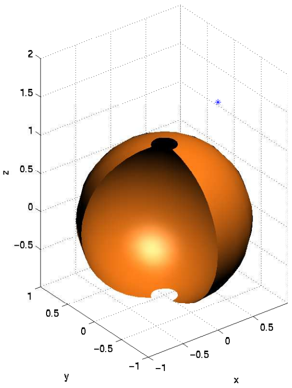


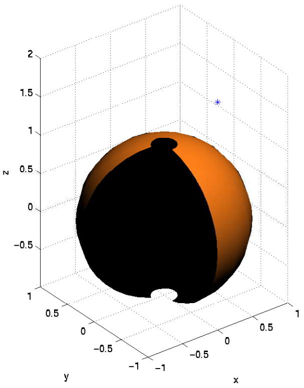
The back and front of polygons

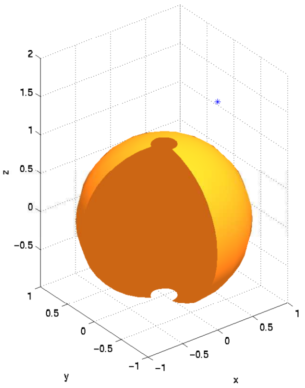
A quote from the manual:
“The default value for `BackFaceLighting`is `reverselit`. This setting reverses the direction of the vertex normals that face away from the camera, causing the interior surface to reflect light towards the camera. Setting `BackFaceLighting`to `unlit` disables lighting on faces with normals that point away from the camera.”

```
[X, Y, Z] = sphere(20);
Z(X <= 0 & Y <= 0) = NaN;                % TRICK!
color = [1 0.5 0.1];

figure(1)
hold off
h = surf(X, Y, Z);
set(h, ’AmbientStrength’,  0.0,    ...  % NOTE
       ’DiffuseStrength’,  1.0,    ...
       ’SpecularStrength’, 0.5,    ...
       ’FaceColor’,        color,  ...
       ’EdgeColor’,        color,  ...
       ’FaceLighting’,    ’phong’, ...
       ’EdgeLighting’,    ’phong’)
hold on

light_pos = [0 -1 2];
plot3(light_pos(1), light_pos(2), light_pos(3), *”)
light(’Position’, light_pos)
axis equal

figure(2)  etc.
set(h, ’AmbientStrength’,  0.0,    ...
  etc.
       ’BackFaceLighting’, ’unlit’)     % NOTE!
```

```
% Default is reverselit
figure(3) etc.
set(h, ’AmbientStrength’,  0.8,    ...  % NOTE!
  etc.
       ’BackFaceLighting’, ’unlit’)
```







More 3D plot commands

Now that we have seen how to use more fancy graphics it is time to list some of the remaining 3D-plot commands. They can, essentially, be divided into two groups.

If we have a scalar quantity, like pressure or temperature, defined in (x, y, z), we can use tools like isosurfaces or slices. If, on the other hand, a vector (velocity) is defined in each point, we would usually use some type of stream lines or arrows.

One cannot do justice to these functions using transparencies. Many of the commands require lighting, transparency, and the z-buffer. Also the description in the manual requires 45 pages. My suggestion is that you try them, which is not hard work. Almost all the commands have one or more examples at the end of the help text. So just cut-and-paste!

Here is a list taken directly from the manual:

Functions for scalar Data

`contourslice`	Draw contours in volume slice planes
`isocaps`	Compute isosurface end-cap geometry
`isocolors`	Compute the colors of isosurface vertices
`isonormals`	Compute normals of isosurface vertices
`isosurface`	Extract isosurface data from volume data
`patch`	Create a patch (multipolygon) graphics object
`reducepatch`	Reduce the number of patch faces
`reducevolume`	Reduce the number of elements in a volume data set
`shrinkfaces`	Reduce the size of each patch face
`slice`	Draw slice planes in volume
`smooth3`	Smooth 3-D data
`surf2patch`	Convert surface data to patch data
`subvolume`	Extract subset of volume data set

Functions for Vector Data

`coneplot`	Plot velocity vectors as cones in 3-D vector fields
`curl`	Compute the curl and angular velocity of a 3-D vector field
`divergence`	Compute the divergence of a 3-D vector field
`interpstreamspeed`	Interpolate streamline vertices from vector-field magnitudes
`streamline`	Draw stream lines from 2-D or 3-D vector data
`streamparticles`	Draw stream particles from vector volume data
`streamribbon`	Draw stream ribbons from vector volume data
`streamslice`	Draw well-spaced stream lines from vector volume data
`streamtube`	Draw stream tubes from vector volume data
`stream2`	Compute 2-D stream line data
`stream3`	Compute 3-D stream line data
`volumebounds`	Return coordinate and color limits for volume (scalar and vector)

To use these routines the coordinates must usually be gridded (as if produced by `meshgrid`).

About OpenGL, according to
`http://www.opengl.org/about/overview/`

OpenGL is the premier environment for developing portable, interactive 2D and 3D graphics applications. Since its introduction in 1992, OpenGL has become the industry's most widely used and supported 2D and 3D graphics application programming interface (API), bringing thousands of applications to a wide variety of computer platforms...

4.010 What is GLU? How is it different from OpenGL?

If you think of OpenGL as a low-level 3D graphics library, think of GLU as adding some higher-level functionality not provided by OpenGL. Some of GLU's features include:

... Specialty transformation matrices for creating perspective and orthographic projections, positioning a camera, and selection/picking. Rendering of disk, cylinder, and sphere primitives ...

3.010 What is GLUT? How is it different from OpenGL?

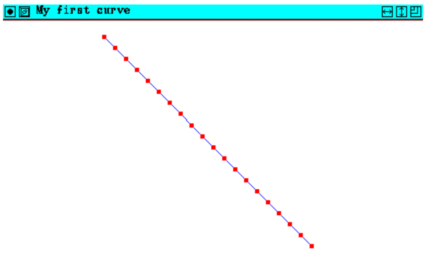
Because OpenGL doesn't provide routines for interfacing with a windowing system or input devices, an application must use a variety of other platform-specific routines for this purpose. The result is nonportable code.

Furthermore, these platform-specific routines tend to be full-featured, which complicates construction of small programs and simple demos.

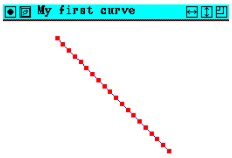
GLUT is a library that addresses these issues by providing a platform-independent interface to window management, menus, and input devices in a simple and elegant manner.

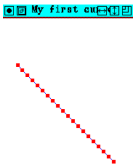
Some OpenGL-examples

How can we create the following image using OpenGL and C?



After having reshaped the window:





Here is the C-program. If you have not seen C before, see the Diary. The line numbers are not part of the program.

```
// I’m using C++-comments // in this code.
#include <GL/glut.h> // includes gl.h, glu.h as well
#include <stdlib.h>  // For void exit(int)

void Display();      // Prototypes
void MyInit();
void Reshape(int, int);
void KeyHandler(unsigned char, int, int);

// argc = argument count >= 1 (command name first)
// argv = arg vector (array of pointers to char)

int main(int argc, char *argv[])
{
  glutInit(&argc, argv);

  // use RGB-color and not indexed color
  glutInitDisplayMode(GLUT_RGB);

  // width = 500, height = 300 pixels
  glutInitWindowSize(500, 300);

  // (0, 0) upper-left corner of screen
  glutInitWindowPosition(10, 10);
  glutCreateWindow("My first curve"); // title

  // the following calls define three callbacks
  glutDisplayFunc(Display);     // at re-displays
  glutReshapeFunc(Reshape);     // change in size
  glutKeyboardFunc(KeyHandler); // keypress

  MyInit();       // my own initializations
  glutMainLoop(); // wait for events
  return 0;
}
```

```
void MyInit()
{
  glClearColor(1, 1, 1, 0); // white to erase

  // set up projection matrix
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity(); // matrix = I

  // 2D orthographic projection
  // x_min, x_max, y_min, y_max
  gluOrtho2D(0, 2, -1, 1);

  // set the modelview matrix to I
  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();
}
void Display()
{
  float x;
  // clear color buffer, i.e. erase
  glClear(GL_COLOR_BUFFER_BIT);

  glColor3f(0, 0, 1);       // blue
  glBegin(GL_LINE_STRIP);   // draw solid curve
    glVertex2f(0, 1);       // define point
    glVertex2f(1.9, -0.9);  // define point
  glEnd();                  // end of curve

  // Note that glColor is in effect for all
  // points defined by glVertex2f.
  glColor3f(1, 0, 0);       // new color
  glPointSize(5);           // larger points
  glBegin(GL_POINTS);       // draw points
    for(x = 0; x < 1.99; x += 0.1)
      glVertex2f(x, 1 - x); // define point
  glEnd();                  // end of GL_POINTS
```

```
75    glFlush();                // force drawing
76  }
77
78  void Reshape(int w, int h) // new size in pixels
79  {
80    int border = 20;   // a frame around the curve
81
82    // area where we draw the curve, positive
83    int size_of_curve;
84    int low_left_x, low_left_y;  // viewport
85
86    if ( w > h ) {
87      if ( h < 2 * border ) border = 0;
88      size_of_curve = h - 2 * border; // >= 0
89      low_left_x    = 0.5 * (w - size_of_curve);
90      low_left_y    = border;
91    } else {
92      if ( w < 2 * border ) border = 0;
93      size_of_curve = w - 2 * border;
94      low_left_x    = border;
95      low_left_y    = 0.5 * (h - size_of_curve);
96    }
97
98    glViewport(low_left_x, low_left_y,
99               size_of_curve, size_of_curve);
100 }
101
102 void KeyHandler(unsigned char key, int x, int y)
103 {
104   if (key == ’q’ || key == 27)
105     exit(0);
106 }
```

5-8: Prototypes.
13: **argv** and **argc** are not used in our case.
33: We never return from **glutMainLoop**

40: Color values are floats, but we are using the automatic conversion between int and float in this case. The last values is the alpha-value (for transparency).

54: **Display** is called to draw the image. Called after **Reshape**.

58: Fill using the color defined on line 40.

60: **3f** = three floats. There are 32 different **glColor**-routines, e.g. **glColor3fv** which takes a float vector with three elements **glColor3f(0.0, 0.0, 1.0);** is OK as well.

61: **glBegin** defines how the **glVertex**-calls should be interpreted, e.g. like points on a curve or like separate points. Compare Matlab, **plot(x, y)** and **plot(x, y, ’o’)**. There are: **GL_POINTS**, **GL_LINES**, **GL_LINE_STRIP**, **GL_LINE_LOOP**, **GL_TRIANGLES**, **GL_TRIANGLE_STRIP**, **GL_TRIANGLE_FAN**, **GL_QUADS**, **GL_QUAD_STRIP**, and **GL_POLYGON**.
See the man-page for **glBegin** for details.

78: Called when a window is created and when it is modified in size. We must rescale things so that the curve is not deformed.
A viewport is rectangular area of the window
x, y, width, height

This is the idea behind the values. We get two cases. If **w** is larger than **h**, the new width and height, of the window:

```
-----------------------------
|       border    (x2, y2)  |
|        +--------+         |
|        |        |         |
|        | curve  |         |
|        |        |         |
|        +--------+         |
|    (x1, y1)    border     |
-----------------------------

size_of_curve = h - 2 * border (size of square)
y1 = border
x1 = w / 2 - size_of_curve / 2
x2 = x1 + size_of_curve
y2 = y1 + size_of_curve
```

Similarly when **w** is less than **h**.

102: This routine is called whenever we press a key and when the mouse is placed in the window. **(x, y)** is the position of the mouse, in pixels, **(0, 0)** = upper left. We exit the program if q or escape is pressed. escape has character code 27.

A typical OpenGL manual page:

```
% man glvertex (in edited form)

Misc. Reference Manual Pages                GLVERTEX()

NAME
glVertex2d,  glVertex2f,  glVertex2i,  glVertex2s,
glVertex3d,  glVertex3f,  glVertex3i,  glVertex3s,
glVertex4d,  glVertex4f,  glVertex4i,  glVertex4s,
glVertex2dv, glVertex2fv, glVertex2iv, glVertex2sv,
glVertex3dv, glVertex3fv, glVertex3iv, glVertex3sv,
glVertex4dv, glVertex4fv, glVertex4iv, glVertex4sv
- specify a vertex


C SPECIFICATION
 void glVertex2d( GLdouble x, GLdouble y )
 void glVertex2f( GLfloat  x, GLfloat  y )
 void glVertex2i( GLint    x, GLint    y )
...
 void glVertex3d( GLdouble x, GLdouble y, GLdouble z )
 void glVertex3f( GLfloat  x, GLfloat  y, GLfloat  z )
...
PARAMETERS
x, y, z, w Specify x, y, z, and w coordinates of a
           vertex. Not all parameters are present
           in all forms of the command.

C SPECIFICATION
 void glVertex2dv( const GLdouble*v )
 void glVertex2fv( const GLfloat *v )
...
 void glVertex3dv( const GLdouble*v )
 void glVertex3fv( const GLfloat *v )
...
```

`const` protects the elements in the array from change.
TE’s comment.

```
PARAMETERS
v Specifies a pointer to an array of two, three, or
  four elements. The elements of a two-element array
  are x and y; of a three-element array, x, y, and z;
  and of a four-element array, x, y, z, and w.

DESCRIPTION
glVertex commands are used within glBegin/glEnd pairs
to specify  point, line, and polygon vertices. The
current color, normal, and texture coordinates are
associated  with the vertex when glVertex is called.

When only x and y are specified, z defaults to 0.0 and
w defaults to 1.0. When x, y, and z are specified, w
defaults to 1.0.

NOTES
Invoking glVertex outside of a glBegin/glEnd pair
results in undefined behavior.

SEE ALSO
glBegin, glCallList, glColor, glEdgeFlag, glEvalCoord,
glIndex, glMaterial, glNormal, glRect, glTexCoord
```

A careful OpenGL programmer uses the OpenGL types (I have not), e.g.:

```
void Display(void)
{
  GLfloat  color[3] = {0, 0, 1}, x;

  glClear(GL_COLOR_BUFFER_BIT);
  glColor3fv(color);

  glBegin(GL_LINE_STRIP);
    for(x = 0; x < 1.99; x += 0.1)
      glVertex2f(x, 1 - x);
  glEnd();
...
```

However, looking in `/usr/include/GL/gl.h`one sees that:

```
typedef unsigned int GLenum;
typedef unsigned char GLboolean;
typedef unsigned int GLbitfield;
typedef signed char GLbyte;
typedef short GLshort;
typedef int GLint;
typedef int GLsizei;
typedef unsigned char GLubyte;
typedef unsigned short GLushort;
typedef unsigned int GLuint;
typedef float GLfloat;
typedef float GLclampf;
typedef double GLdouble;
typedef double GLclampd;
typedef void GLvoid;
```

Here is a simple 3D-example. `Reshape` and `KeyHandler` are unchanged from the previous example (and are not included).

```
#include <GL/glut.h>
#include <stdlib.h>

void Display();
void MyInit();
void Reshape(int, int);
void KeyHandler(unsigned char, int, int);
void DrawCoordSys();
void DrawSquares();

int main(int argc, char *argv[])
{
  glutInit(&argc, argv);
  // switch on Z-buffer: GLUT_DEPTH
  glutInitDisplayMode(GLUT_RGB | GLUT_DEPTH);
  glutInitWindowSize(300, 300);
  glutInitWindowPosition(10, 10);
  glutCreateWindow("A 3D-example");
  glutDisplayFunc(Display);
  glutReshapeFunc(Reshape);
  glutKeyboardFunc(KeyHandler);
  MyInit();
  glutMainLoop();
  return 0;
}


void Display()
{
  // clear color- and Z-buffer (depth buffer)
  glClear(GL_COLOR_BUFFER_BIT |  // NOT ||
          GL_DEPTH_BUFFER_BIT);



```

```
  DrawSquares();  // draw the squares
  DrawCoordSys(); // draw a coord syst.

  glFlush();
}

void MyInit()
{
  glClearColor(1, 1, 1, 0);
  glEnable(GL_DEPTH_TEST); // enable Z-buffer

  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  glOrtho(-2, 2,  -2, 2,  0, 3); // view volume

  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();
  gluLookAt(1,1,1, 0,0,0, 0,1,0); // place eye
}

void DrawCoordSys()
{
  float  color[] = {0, 0, 0}, p[] = {0, 0, 0};
  char   xyz[]   = {’x’, ’y’, ’z’};
  int    axis;

  glLineWidth(2);
  for(axis = 0; axis <= 2; axis++) {
    color[axis] = 1;
    glColor3fv(color);
    color[axis] = 0;   // back to black

    glBegin(GL_LINE_STRIP);
      glVertex3fv(p);
      p[axis] = 1;  glVertex3fv(p);
    glEnd();

```

```
    glColor3fv(color);
    p[axis] = 1.1;  glRasterPos3fv(p);
    glutBitmapCharacter(GLUT_BITMAP_9_BY_15,
                        xyz[axis]);
    p[axis] = 0;
  }
}
void DrawSquares()
{
  // red unit square at z = 0.5
  glColor3f(1, 0, 0);
  glBegin(GL_POLYGON);
    glVertex3f(0, 0, 0.5);
    glVertex3f(1, 0, 0.5);
    glVertex3f(1, 1, 0.5);
    glVertex3f(0, 1, 0.5);
  glEnd();

  // blue unit square at z = -0.5
  glColor3f(0, 0, 1);
  glBegin(GL_POLYGON);
    glVertex3f(0, 0, -0.5);
    glVertex3f(1, 0, -0.5);
    glVertex3f(1, 1, -0.5);
    glVertex3f(0, 1, -0.5);
  glEnd();
}
```

It is possible to call `glColor` once for every `glVertex`. The polygon is then coloured using interpolation, provided smooth shading is on, which is the default (`glShadeModel(GLSMOOTH)`). If one has switched on flat shading (`glShadeModel(GLFLAT)`) the colour of the first vertex in the polygon is used to colour the whole polygon.

Handling the mouse

```
...
void MouseHandler(int, int, int, int);

int main(int argc, char *argv[])
{
 ...
  glutMouseFunc(MouseHandler);
 ...
}

void MouseHandler(int button, int state, int x, int y)
{
/*
  button: one of GLUT_LEFT_BUTTON, GLUT_MIDDLE_BUTTON,
  or GLUT_RIGHT_BUTTON. state is either GLUT_UP or
  GLUT_DOWN indicating whether the callback was due to
  a release or press respectively.

  If a menu is attached to a button for a window,
  mouse callbacks will not be generated for that
  button.  (x, y) = (0, 0) upper-left
*/
  ....
}
```

If the display should be redrawn call `glutPostRedisplay();`
Do not call `Display();` directly.

The next page shows how rotations work. `main` and `DrawCoordSys` have not been included.

We create a square window: `glutInitWindowSize(300, 300);`

```
1  #include <GL/glut.h>
2  void MouseHandler(int, int, int, int);
3  void Display();
4  void MyInit();
5  void DrawCoordSys();
6
7  void MyInit()
8  {
9    glClearColor(1, 1, 1, 0);
10   glEnable(GL_DEPTH_TEST);
11
12   glMatrixMode(GL_PROJECTION);
13   glLoadIdentity();
14   gluPerspective(20, 1, 1, 10);
15
16   glMatrixMode(GL_MODELVIEW);
17   glLoadIdentity();
18   gluLookAt(7,3,5, 0,0,0, 0,1,0);
19 }
20 void
21 MouseHandler(int button, int state, int x, int y)
22 {
23   if ( state == GLUT_UP ) {
24     switch ( button ) {         // new statement
25       case GLUT_LEFT_BUTTON :
26         glRotatef(90, 1, 0, 0); // Rx
27         break;                  // NOTE!
28       case GLUT_MIDDLE_BUTTON :
29         glRotatef(90, 0, 1, 0); // Ry
30         break;
31       case GLUT_RIGHT_BUTTON :
32         glRotatef(90, 0, 0, 1); // Rz
33         break;
34     }
35     glutPostRedisplay();
36   }
37 }
```

```
38
39 void Display()
40 {
41   glClear(GL_COLOR_BUFFER_BIT |
42           GL_DEPTH_BUFFER_BIT);
43   DrawCoordSys();
44   glFlush();
45 }
```

14: The `gluPerspective` arguments are:
“field of view angle” (in degrees) in the y-direction.
“aspect ratio” that determines the field of view in the x-direction.
The aspect ratio is the ratio of x (width) to y (height).
“distance from the viewer” to the near clipping plane (> 0).
“distance from the viewer” to the far clipping plane (> 0).

21: When clicking on the mouse we get the following coordinate systems:

```
    y                    x
    |                    |
    |--- x    --- x      |--- z      --- z
   /         /|         /           /|
  z        y |         y          x |
             z                      y

 Initially   After Rx    After Ry    After Rz
```

The next program contains several new OpenGL-constructs. Double buffering, lighting and materials.

The program draws two spheres (radius one), a red centered on the origin and a green centered on (2, 0, 0). A light is placed at (5, 0, 0). When + is pressed the spheres rotate around the origin in a ccw fashion, and when - is pressed they rotate the other way. By using a menu we can make the light follow the spheres or to be stationary.

```
#include <GL/glut.h>
#include <stdlib.h>

void  Display();
void  MyInit();
void  KeyHandler(unsigned char, int, int);
void  MenuHandler(int); // For menus
void  CreateObject();
int   rotating_light = 0; // global variable

int main(int argc, char *argv[])
{
  glutInit(&argc, argv);

  // GLUT_DOUBLE = double buffering
  glutInitDisplayMode(GLUT_RGB | GLUT_DEPTH |
                      GLUT_DOUBLE);

  glutInitWindowSize(500, 500);
  glutCreateWindow("Spheres");
  glutKeyboardFunc(KeyHandler);
  glutCreateMenu(MenuHandler);     // Menu
  glutAddMenuEntry("Rotating light",   1);
  glutAddMenuEntry("Stationary light", 2);
  glutAddMenuEntry("Quit",             3);
  glutAttachMenu(GLUT_RIGHT_BUTTON); // for example

  glutDisplayFunc(Display);
```

```
  MyInit();
  glutMainLoop();
  return 0;
}
void Display()
{
  glClear(GL_COLOR_BUFFER_BIT |
          GL_DEPTH_BUFFER_BIT);

  CreateObject();    // my own routine
  glutSwapBuffers(); // double buffering
}
void MyInit()
{
  float
    light_pos[]       = {5, 0, 0, 0},
    light_ambient[]   = {0.2, 0.2, 0.2, 1},
    light_diffuse[]   = {1, 1, 1, 1},
    light_specular[]  = {1, 1, 1, 1};

  glClearColor(1, 1, 1, 0);
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  gluPerspective(45, 1, 1, 100);

  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();
  gluLookAt(0,0,10, 0,0,0, 0,1,0);

  // set up ambient, diffuse, and specular
  // components for light 0

  glLightfv(GL_LIGHT0, GL_AMBIENT, light_ambient);
  glLightfv(GL_LIGHT0, GL_DIFFUSE, light_diffuse);
  glLightfv(GL_LIGHT0, GL_SPECULAR, light_specular);

  glEnable(GL_LIGHTING); // switch on lighting
```

```
  glEnable(GL_LIGHT0);   // at least 8 lamps

  // set the position of light0
  glLightfv(GL_LIGHT0, GL_POSITION, light_pos);

  // switch on smooth shading; the other
  // alternative is GL_FLAT
  glShadeModel(GL_SMOOTH);

  glEnable(GL_DEPTH_TEST);
}
void CreateObject()
{
  float // material properties (refl. coeff.)
    white_rc[]  = {1, 1, 1, 1},
    red_rc[]    = {1, 0, 0, 1},
    green_rc[]  = {0, 1, 0, 1},
    spec_exp    = 100;

  // define material properties for front face
  glMaterialfv(GL_FRONT, GL_AMBIENT,   white_rc);
  glMaterialfv(GL_FRONT, GL_DIFFUSE,   red_rc);
  glMaterialfv(GL_FRONT, GL_SPECULAR,  white_rc);
  glMaterialf (GL_FRONT, GL_SHININESS, spec_exp);

  // create the polygons and normals for a
  // sphere; radius, resolution along
  // longitudes and latidudes

  glutSolidSphere(1, 20, 20);

  // the translate should be temporary
  glPushMatrix();
    glTranslatef(2, 0, 0);
    glMaterialfv(GL_FRONT, GL_DIFFUSE, green_rc);
    glutSolidSphere(1, 20, 20);
  glPopMatrix();
```

```
}
void KeyHandler(unsigned char key, int x, int y)
{
  float  light_pos[] = {5, 0, 0, 0};

  if (key == ’q’)
    exit(0);
  else if (key == ’+’)
    glRotatef(3, 0, 1, 0);   // Ry, 3 degrees
  else if (key == ’-’)
    glRotatef(-3, 0, 1, 0);  // Ry, -3 degrees
  else
    return;

  // The position of a light is affected by M, so...
  if ( rotating_light ) // Transform by M
    glLightfv(GL_LIGHT0, GL_POSITION, light_pos);
  else {                // Stationary light
    glPushMatrix();
    glLoadIdentity();   // Do NOT multiply by M
    glLightfv(GL_LIGHT0, GL_POSITION, light_pos);
    glPopMatrix();
  }

  glutPostRedisplay(); // update image
}
void MenuHandler(int id) // id = menu alternative
{
  if (id == 1)
    rotating_light = 1;    // global variable
  else if (id == 2)
    rotating_light = 0;
  else if (id == 3)
    exit(0);               // Quit
}
```

45-etc: Define light properties.
If last element in `light_pos` = 0, skip the actual distance to the light source, just look at the direction. If the sphere is centered on `(8, 0, 0)` the light still comes from the right. If the component is 1 the position is taken into account and a sphere centered on `(8, 0, 0)` is lit from the left.

The fourth element in `light_ambient` etc. is for transparent materials.

121-: If we do not move the light, it will always come from the right.

More on animation

In the previous example we used double buffering to get a smooth animation (line 17, 41). This should be used in the planet-lab as well, but a difference is that the planets should move on their own, we should not have to press any buttons.

To fix that we define an “idle-callback”, a callback that OpenGL executes when it is idle.
We set the callback by `glutIdleFunc(idlecallback)`, where `idle_callback`, is our callback routine. In this routine one updates the positions of the Earth and Moon and then calls `glutPostRedisplay()`

It is possible to solve the updating problem in several ways. In some solutions it is necessary for the callback to “remember” values between calls. We can do that by using global variables. Another alternative is to use static variables. Here are two silly examples.

```
#include <stdio.h>

void idle_func();

int remember_me = 0; // global variable (in this file)

int main(int argc, char *argv[])
{
  idle_func();   idle_func();   idle_func();
  return 0;
}

void idle_func()
{
  remember_me++;
  printf("remember_me = %d\n", remember_me);
}
```

Here is another way:

```
#include <stdio.h>

void idle_func();

int main(int argc, char *argv[])
{
  idle_func();   idle_func();   idle_func();
  return 0;
}

void idle_func()
{
  static int remember_me = 0;   // NOTE static

  remember_me++;
  printf("remember_me = %d\n", remember_me);
}
```

Both solutions will produce the following printout:

```
remember_me = 1
remember_me = 2
remember_me = 3
```

One difference between these programs is the `remember_me` is local to the function in the second case, but accessible to all functions in the first program.

A hint on debugging

```
 ...

void Display()
{
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    ... draw stuff

  // You find CheckErr.c in the directory:
  // /chalmers/groups/thomas_math/VIS/OpenGL/
  CheckErr();
}

void KeyHandler(unsigned char key, int x, int y)
{
  if (key == 'q' || key == 27) {
    exit(0);
  } else {
    glPushMatrix(); // mistake,  no matching Pop

    glTranslatef(0.1, 0, 0);

    glutPostRedisplay();
  }
}
```

After 32 calls we get `GL_STACK_OVERFLOW`.
Changing to `glPopMatrix();` gives `GL_STACK_UNDERFLOW` after the first call.

Here comes `CheckErr`:

```
void CheckErr()
{
  int err;
  char errors[7][21] =
      { "GL_INVALID_ENUM",      "GL_INVALID_VALUE",
        "GL_INVALID_OPERATION", "GL_STACK_OVERFLOW",
        "GL_STACK_UNDERFLOW",   "GL_OUT_OF_MEMORY",
        "GL_TABLE_TOO_LARGE"  };

  err = -1;
  switch (glGetError()) {
  case GL_NO_ERROR:
    break;                        // do nothing
  case GL_INVALID_ENUM:
    err = 0;
    break;
  case GL_INVALID_VALUE:
    err = 1;
    break;
  case GL_INVALID_OPERATION:
    err = 2;
    break;
  case GL_STACK_OVERFLOW:
    err = 3;
    break;
  case GL_STACK_UNDERFLOW:
    err = 4;
    break;
  case GL_OUT_OF_MEMORY:
    err = 5;
    break;
  case GL_TABLE_TOO_LARGE:
    err = 6;
  }

  if (err >= 0) printf("%s\n", errors[err]);
}
```

OpenDX och ParaView

We end the course with two visualization systems that have more advanced graphics than Matlab. These systems have no support for computations (apart from very simple ones), and the user has to supply the plot-data using files.
In previous versions of the course the focus was on OpenDX, but this year we will use ParaView. See the old PDF-file from the Diary for more about OpenDX.
Let us have a look at OpenDX before we start with ParaView. OpenDX, `www.opendx.org` is an open version of IBM's "Visualization Data Explorer".

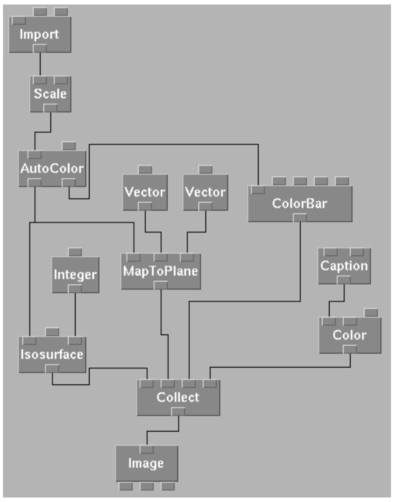
Some, but not all, important points:

- Advanced tools for visualization of 3D-data.
- Takes longer to learn than Matlab, but you can do more. Often faster.
- Modules are connected using a GUI, graphical programming. Visual Program Editor, VPE.
- Input from files (not variables as in Matlab).
- The modules transform the input and sends it to the next module.
- Supports several input formats. Using the "Data Prompter"-program simple inputs can be handled (e.g. uniform, gridded input).
- Lots of documentation. Many demo programs. Few simple examples. Should read a book (or take this course :-)
  David Thompson, Jeff Braun, Ray Ford,
  OpenDX: Paths to Visualization. Consists of a sequence of solved visualization problems.
  `http://www.vizsolutions.com`

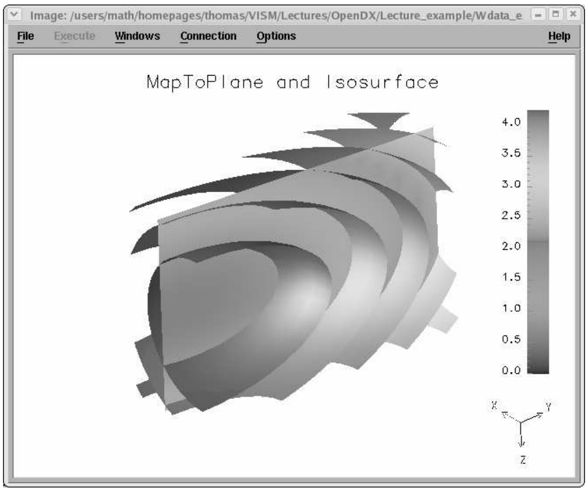
Here is a short example (an extract from the old course) to give you an idea of how one uses OpenDX.

We would like to visualize data of the form $w = f(x, y, z)$.
It is possible to remove part of the data (everything on one side of a plane). We use the module ClipPlane. It takes the data, a point in the plane and a normal defining the clip plane.
Everything on the side of the plane (in the direction of the normal) is removed.

Here is a related construct. The MapToPlane-module creates an arbitrary cutting plane through 3D-space and interpolates data values onto it. The plane is defined by a point a normal, just as the ClipPlane. Using the Vector interactors we can move to plane.
I have combined MapToPlane with Isosurface. I have also added Colorbar which draws a scale (as in Matlab). Finally there is Caption which corresponds to Matlab's title.
Here is the program



and here is a (bad) version of the resulting image.



For more details see the old handouts. The rest of the chapter deals with ParaView.

ParaView

Here are a few sentences from **`www.paraview.org`**

Overview:
ParaView is an open-source, multi-platform application designed to visualize data sets of size varying from small to very large.

The goals of the ParaView project include the following:

- Develop an open-source, multi-platform visualization application.
- Support distributed computation models to process large data sets.
- Create an open, flexible, and intuitive user interface.
- Develop an extensible architecture based on open standards.

ParaView runs on distributed and shared memory parallel as well as single processor systems and has been successfully tested on Windows, Mac OS X, Linux and various Unix workstations, clusters and supercomputers. Under the hood, ParaView uses the Visualization Toolkit as the data processing and rendering engine and has a user interface written using Qt®.

The ParaView project started started in 2000 as a collaborative effort between Kitware Inc. and Los Alamos National Laboratory. The initial funding was provided by a three year contract with the US Department of Energy ASCI Views program. Today, ParaView development continues as a collaboration between Kitware, Sandia National Labs, CSimSoft, Los Alamos National Lab, Army Research Lab and others.

There is a set of books available from Kitware Inc. providing details about VTK and ParaView. In this course it is sufficient to study the "ParaView 3 tutorial for Supercomputing 07" (used in the labs), and the "VTK file formats documentation" (see the home page for links).

The hardest part with using OpenDX and ParaView is the creation of the input files and this chapter will show you some examples.

VTK supports many styles of file formats. In this course we will use two, the legacy VTK formats and the XML formats.

From the dictionary:

> Legacy: Designating software or hardware which, although outdated or limiting, is an integral part of a computer system and difficult to replace.

Suppose we want implement the following Matlab-program in ParaView:

```
[X, Y] = meshgrid(linspace( 0, 2, 30), ...
                  linspace(-1, 1, 30));
surf(X, Y, X.^2 + sin(3* Y))
```

Here is a first step, the file **`ex1.vtk`** (the line numbers are not part of the file). For more details see the formats-manual.

```
1  # vtk DataFile Version 2.0
2  Data for z = f(x, y).
3  ASCII
4  DATASET STRUCTURED_POINTS
5  DIMENSIONS 3 3 1
6  ORIGIN     0 0 0
7  SPACING    1 1 1
8
9  POINT_DATA 9
10 SCALARS name_1 float
11 LOOKUP_TABLE default
12 1 2 3 4 5 6 7 8 9
```

Line 1 is a header, and line 2 a title (comment). Line 3 gives the data format for numbers (coordinates etc), see the documentation for binary formats.

Lines 4-7 describe the dataset structure (also called the geometry or the topology) of the data. In our case we have grid points in the x-y-plane. The points are (j, k), j, $k = 0, 1, 2$.
Finally, on lines 9-12, we have the dataset attributes, the values of the function on the grid (the values are 1-9).

We have **`POINT_DATA`**, i.e. we have defined a scalar value in each grid point. The value is a scalar-float (i.e not a vector for example) and we have named it, **`name_1`**. Choose meaningful names e.g. pressure, temperature etc.

We can have several quantities, by having several groups like 10-13. Using the name, we can later pick the relevant quantity in ParaView. On line 11 we define a colour lookup table (here the default). One should be able to define ones own, but this seems buggy in the present ParaView-version.

I have not included any images in the handouts, since the PDF-files become so large. Some of the **`vtk`**-files (and corresponding images) are available on the student computer system, so you can try them yourself see
`/chalmers/groups/thomasmath/VIS/Handoutsex_ParaView`.

The following line denotes a missing image.
[Image]

This is how I made the [Image]. I loaded the file, choose the "Glyph-filter", changed the "Glyph Type" to "Sphere", increased the "Radius" and "Theta Resolution". I pressed the "Toggle Color Legend Visibility"-button. Not to waste printer-toner, I changed the background colour (so the background text is not very visible).

So what is a glyph?

(Glyph from from Greek Glyphe, carved work, from
glyphein to carve.
1: an ornamental vertical groove especially in a Doric frieze
2: a symbolic figure or a character (as in the Mayan system of writing) usually incised or carved in relief
3: a symbol (as a curved arrow on a road sign) that conveys information nonverbally).

Looking at the image we can see that the point data is ordered the following way:

$(x_1, y_1)\ (x_2, y_1)\ (x_3, y_1)\ (x_1, y_2)\ (x_2, y_2)\ (x_3, y_2)\ (x_1, y_3)\ (x_2, y_3)\ (x_3, y_3)$

How can we generate data in that order from Matlab? Here is a short example:

```
>> [X, Y] = meshgrid(-1:1, -1:1)

X =  -1     0     1
     -1     0     1
     -1     0     1

Y =  -1    -1    -1
      0     0     0
      1     1     1

>> [X(:), Y(:)]

ans =
    -1    -1  % (x_min,          y_min)
    -1     0  % (x_min,          y_min +   dy)
    -1     1  % (x_min,          y_min + 2 dy)
     0    -1  % (x_min +   dx,   y_min)
     0     0  % (x_min +   dx,   y_min +   dy)
     0     1  % (x_min +   dx,   y_min + 2 dy)
     1    -1  % (x_min + 2 dx,   y_min)
     1     0  % (x_min + 2 dx,   y_min +   dy)
     1     1  % (x_min + 2 dx,   y_min + 2 dy)

>> X = X';   % Not what we want, so transpose
>> Y = Y';
>> [X(:), Y(:)]
ans =
    -1    -1  % (x_min,          y_min)
     0    -1  % (x_min +   dx,   y_min)
     1    -1  % (x_min + 2 dx,   y_min)
    -1     0  % (x_min,          y_min +   dy)
     0     0  % (x_min +   dx,   y_min +   dy)
     1     0  % (x_min + 2 dx,   y_min +   dy)
    -1     1  % (x_min,          y_min + 2 dy)
     0     1  % (x_min +   dx,   y_min + 2 dy)
     1     1  % (x_min + 2 dx,   y_min + 2 dy)
```

Here comes a Matlab-program that produces a suitable datafile for ParaView. In a real application, we may have a Fortran/C/C++-code that produces the data.

```
% Make surface data for ParaView
n = 30;
[X, Y] = meshgrid(linspace( 0, 2, 30), ...
                  linspace(-1, 1, 30));
Z = X.^2 + sin(3 * Y);

% Open output file
fid = fopen('surf_ex.vtk', 'w');

% Write a header and a comment
fprintf(fid, '# vtk DataFile Version 2.0\n');
fprintf(fid, 'z = x^2 + sin(3 y)\n');

% Data type and type of grid
fprintf(fid, 'ASCII\n');
fprintf(fid, 'DATASET STRUCTURED_POINTS\n');

% Here comes the data. First the nodes.
fprintf(fid, 'DIMENSIONS %d %d 1\n', n, n);  % z = 1
fprintf(fid, 'ORIGIN 0 -1 0\n');

% spacing not used for z
spacing = X(1, 2) - X(1, 1); % i.e. 2 / (n - 1)
fprintf(fid, 'SPACING %e %e %e\n', ...
              spacing, spacing, spacing);

fprintf(fid, 'POINT_DATA %d\n', n* n);
fprintf(fid, 'SCALARS z float\n');
fprintf(fid, 'LOOKUP_TABLE default\n');
fprintf(fid, '%e\n', Z'); % Note, transpose

fclose(fid);  % close file
```

In ParaView the data will show up as a flat coloured plane (where the colours correspond to the Z-values). To produce heights from the Z-values we use two filters, "Clean to Grid" followed by "Warp(scalar)". The first filter (quoting the help):

> It also converts the data set to an unstructured grid. You may wish to do this if you want to apply a filter to your data set that is available for unstructured grids but not for the initial type of your data set (e.g., applying warp vector to volumetric data).

and the second

> The Warp (scalar) filter translates the points of the input data set along a vector by a distance determined by the specified scalars. This filter operates on polygonal, curvilinear, and unstructured grid data sets containing single-component scalar arrays.

The vector is $(0, 0, 1)$ in this case. The warp-filter has a "Scale Factor" so one can exaggerate (scale) the z-direction.

Another filter, which we can apply directly on the data, is "Contour".

[Image]

If we make a mistake in the VTK-file, we get an error message in a separate window "Output Message". As an example, if we give two, instead of three, numbers in the `DIMENSIONS`-statement we get the following error message:

```
ERROR: In /home/berk/Work/ReleaseBuilds/ParaView3/
VTK/IO/vtkStructuredPointsReader.cxx, line 131
vtkStructuredPointsReader (0x8ca9d18):
Error reading dimensions!
```

I have fetched a pre-compiled binary, that is the reason for the absolute path.
It may be instructive to look at the source, to see the origin of the message. Fetching and unpacking `vtk-5.2.0.tar.gz` from `http://www.vtk.org/get-software.php` we look at the C++-file `VTK/IO/vtkStructuredPointsReader.cxx`

```
% wc -l vtkStructuredPointsReader.cxx
 533 vtkStructuredPointsReader.cxx

if ( ! strncmp(this->LowerCase(line), "dimensions",10)
  {
  int dim[3];
  if (!(this->Read(dim) &&
        this->Read(dim+1) &&
        this->Read(dim+2)))
    {
    vtkErrorMacro(<<"Error reading dimensions!");
    this->CloseVTKFile ();
    this->SetErrorCode( vtkErrorCode::FileFormatError )
    return 1;
    }
```

`vtkErrorMacro` is line 131.

In the following VTK-file we construct a tiny vector field in 3D. You should use more points in a real application. You could have `SCALARS`-data as well.

```
# vtk DataFile Version 2.0
Vector field in 3D.
ASCII
DATASET STRUCTURED_POINTS
DIMENSIONS 3 3 3
ORIGIN     0 0 0
SPACING    1 1 1

POINT_DATA 27
VECTORS vec float
1 2 3
  etc.
```

One could use the “Stream Tracer” and “Generate Tubes” filters to visualize the flow. [Image]
In the previous examples every node (point) has a quantity (scalar or vector) associated with it. In some applications it is more natural to associate a value with an area or volume (a so-called cell). A biologist may count the number of bugs, plants etc. per km^2 or number of fish per km^3.
Here comes a 2D-example using cell-data with scalar values.

```
1   # vtk DataFile Version 2.0
2   A 2D cell example
3   ASCII
4   DATASET STRUCTURED_POINTS
5   DIMENSIONS 4 4 1
6   ORIGIN     0 0 0
7   SPACING    1 1 1
8
9   CELL_DATA 9
10  SCALARS name float
11  LOOKUP_TABLE default
12  1 2 3 4 5 6 7 8 9
```

Line 5 defines a 4 × 4 point grid so with 3 × 3 cells, i.e. nine values on line 9. A plot gives a checkerboard pattern (in color). [Image]
Here is a 3D-example with $3 \times 3 \times 3$ cells, i.e cubes. The central cube is number 14, having the value -1 on line 13. The data can be inspected using the Clip filter, for example. [Image]

```
1   # vtk DataFile Version 2.0
2   A 3D cell example
3   ASCII
4   DATASET STRUCTURED_POINTS
5   DIMENSIONS 4 4 4
6   ORIGIN     0 0 0
7   SPACING    1 1 1
8
9   CELL_DATA 27
10  SCALARS name float
11  LOOKUP_TABLE default
12   1  2  3  4  5  6  7  8  9
13  10 11 12 13 -1 15 16 17 18
14  19 20 21 22 23 24 25 26 27
```

Here is a 2D cell example where we associate a vector with each cell. Using the “Cell Centers” and “Glyph”-filters, we get arrows starting in the center of each cell (square). [Image]

```
# vtk DataFile Version 2.0
A 2D cell, vector, example
ASCII
DATASET STRUCTURED_POINTS
DIMENSIONS 4 4 1
ORIGIN     0 0 0
SPACING    1 1 1

CELL_DATA 9
VECTORS vec float
1 0 0   1 0 0   1 0 0
0 1 0   0 1 0   0 1 0
0 0 1   0 0 1   0 0 1
```

In the next example we create a more complicated geometry which is not quite so regular. Let us make a simple model of the surface of a house. We use triangles to construct the surface (compare the surface mesh in a finite element computation).
We use point data from now on.
This primitive drawing shows the numbering of the points.

```
8  -------- 9     Top of roof
|\          |\
  6 --------- 7
|/          |/    Roof level
4 --------- 5

  2 --------- 3
 /          /    Ground level
0 --------- 1
```

```
1   # vtk DataFile Version 2.0
2   A house
3   ASCII
4   DATASET POLYDATA
5
6   POINTS 10 float
7   0 0 0   2 0 0   0 1 0    2 1 0
8   0 0 1   2 0 1   0 1 1    2 1 1
9   0 0.5 1.5  2 0.5 1.5
10
11  TRIANGLE_STRIPS 2 20
12  10    0 4 1 5 3 7 2 6 0 4
13   8    4 8 5 9 7 8 6 4
14
15  POINT_DATA 10
16  SCALARS name float
17  LOOKUP_TABLE default
18  11 12 13 14 15 16 17 18 19 20
```

On lines 6-9 we list the 10 coordinates for the points that define the corners of the triangles. The walls are made using one triangle strip (saves space compared to separate triangles), line 12. The points are numbered in a zig-zag-order, the first point having index zero. The roof is defined on line 13. The numbers on line 11 denote number of strips and number of integers in lines 12, 13. The first number on line 12 (13) denotes the numbers of points in the strip.

By using “Surface With Edges”, using “Glyph” with “Glyph Type = Sphere”, “Scalar Mode=scalar”, clicking in “Edit” and setting “Set Scale factor=0.01” we get the following [Image].

Here comes an example of an unstructured grid composed by tetrahedrons. We reuse the points from the house example.
I used Matlab to construct the tetrahedrons, here is a code segment. `x`, `y` and `z`, contain the coordinates from the house.

```
...
% Tesselate the volume using tetrahedrons. T is an
% n_tetra x 4 matrix with indices into x, y and z.

T = delaunay3(x, y, z, [])

n_tetra = size(T, 1);
C       = jet(n_tetra);  % some colours

% Explode the view by moving the tetrahedrons
% from the centre
xm = mean(x);
ym = mean(y);
zm = mean(z);
d = 0.1;        % scale factor

% P is used to extract corners in the four faces
% of a tetrahedron
P = [1 2 3; 1 2 4; 1 3 4; 2 3 4];
```

```
for k = 1:n_tetra
  xmT = mean(x(T(k, :)));  % centre of tetrahedron
  ymT = mean(y(T(k, :)));
  zmT = mean(z(T(k, :)));
  vx = d * (xmT - xm);     % translation
  vy = d * (ymT - ym);
  vz = d * (zmT - zm);

  for j = 1:4              % plot all four faces
    t = T(k, P(j, :));
    fill3(x(t) + vx, y(t) + vy, z(t) + vz, C(k, :))
  end
end
...
```

Here is the `T`-matrix

```
T =
     7     3     2     1
     7     2     5     1
     7     3     4     2
     7     4     8     2
     7     6     5     2
     7     8     6     2
    10     8     6     5
    10     7     5     9
    10     7     8     5
```

Here is a sequence of images each with a different `d`-value, showing an “exploded view” [Image].

Boris Nikolaevich Delaunay or Delone, 1890-1980, was one of the first Russian mountain climbers and a Soviet/Russian mathematician (according to Wikipedia).

What is the difference, with respect to visualization, between the two houses (the first and the second)?

Filter	First house	Second house
none	surface	volume
contour	curves	surfaces
clip	surface	volume
slice	curve	surface

Here comes the vtk-file:

```
1  # vtk DataFile Version 2.0
2  A tesselated house
3  ASCII
4  DATASET UNSTRUCTURED_GRID
5
6  POINTS 10 float
7  0 0 0   2 0 0   0 1 0    2 1 0
8  0 0 1   2 0 1   0 1 1    2 1 1
9  0 0.5 1.5  2 0.5 1.5
10
11 CELLS 9 45
12      4     6     2     1     0
13      4     6     1     4     0
14      4     6     2     3     1
15      4     6     3     7     1
16      4     6     5     4     1
17      4     6     7     5     1
18      4     9     7     5     4
19      4     9     6     4     8
20      4     9     6     7     4
21
22 CELL_TYPES 9
23 10 10 10 10 10 10 10 10 10
24
25 POINT_DATA 10
26 SCALARS name float
27 LOOKUP_TABLE default
28 11 12 13 14 15 16 17 18 19 20
```

Line 4 has been changed from the first version. Lines 6-9 are unchanged. I have replaced the 2D triangle strips with 3D tetrahedrons, lines 11-23. The rest of the file is unchanged.

Line 11 starts the description of the corners of the tetrahedrons, there are nine tetrahedrons and 45 ($9 \cdot 5$) numbers are required to describe them. Line 12, `4 6 2 1 0`, says that the coordinates of the four (the first 4) corners are given by 6:th, 2:d, 1:th and 0:th point (indices start at zero). To get the correct indices I had to subtract one from the `T`-matrix produced by `delaunay3`.

Lines 22-23 describe the type of cells. We have nine tetrahedrons, which are identified by number ten (see the formats-manual for the numbers). [Image]

Suppose you want to visualize data produced by $w = f(x, y, z)$, and where you are using Matlab to produce the w-values. It is better to use `ndgrid` instead of `meshgrid` as will be explained below.

```
>> [X, Y, Z] = ndgrid(0:0.1:1, 10:2:40, -1:0.1:1);
>> W = X.^2 + (0.05 * (Y - 10)).^2 + Z.^2;
>> w = W(:);
>> save -ascii wdata w  % for example
```

To understand how the values are stored in the file we look at a much smaller example.

```
>> [X, Y, Z] = ndgrid(0.1:0.1:0.3, -1:1, 20:10:40)

X(:,:,1) =
    0.1000    0.1000    0.1000
    0.2000    0.2000    0.2000
    0.3000    0.3000    0.3000
X(:,:,2) =
    0.1000    0.1000    0.1000
    0.2000    0.2000    0.2000
    0.3000    0.3000    0.3000
```

```
X(:,:,3) =
    0.1000    0.1000    0.1000
    0.2000    0.2000    0.2000
    0.3000    0.3000    0.3000

Y(:,:,1) =
    -1     0     1
    -1     0     1
    -1     0     1
Y(:,:,2) =
    -1     0     1
    -1     0     1
    -1     0     1
Y(:,:,3) =
    -1     0     1
    -1     0     1
    -1     0     1

Z(:,:,1) =
    20    20    20
    20    20    20
    20    20    20
Z(:,:,2) =
    30    30    30
    30    30    30
    30    30    30
Z(:,:,3) =
    40    40    40
    40    40    40
    40    40    40
```

So we get 3D-matrices and from the next page we see that when W is computed, x varies faster than y which changes faster than z. Had I used `meshgrid` the order would have been y, x, z, which is less regular.

```
>> [X(:), Y(:), Z(:)] % I have added blank lines
ans =
    0.1000   -1.0000   20.0000  % (x1, y1, z1)
    0.2000   -1.0000   20.0000  % (x2, y1, z1)
    0.3000   -1.0000   20.0000  % (x3, y1, z1)

    0.1000         0   20.0000  % (x1, y2, z1)
    0.2000         0   20.0000  % (x2, y2, z1)
    0.3000         0   20.0000  % (x3, y2, z1)

    0.1000    1.0000   20.0000  % (x1, y3, z1)
    0.2000    1.0000   20.0000  % (x2, y3, z1)
    0.3000    1.0000   20.0000  % (x3, y3, z1)

    0.1000   -1.0000   30.0000  % (x1, y1, z2)
    0.2000   -1.0000   30.0000  % etc.
    0.3000   -1.0000   30.0000

    0.1000         0   30.0000
    0.2000         0   30.0000
    0.3000         0   30.0000

    0.1000    1.0000   30.0000
    0.2000    1.0000   30.0000
    0.3000    1.0000   30.0000

    0.1000   -1.0000   40.0000
    0.2000   -1.0000   40.0000
    0.3000   -1.0000   40.0000

    0.1000         0   40.0000
    0.2000         0   40.0000
    0.3000         0   40.0000

    0.1000    1.0000   40.0000
    0.2000    1.0000   40.0000
    0.3000    1.0000   40.0000
```

A more general format, using XML

XML, “Extensible Markup Language”, is a language which can be used to transport and store data. It can be used to create markup languages, such as HTML (a language defining how text and images should be displayed). Note that HTML was designed to display data defining the size and position of text for example. XML does not know about layout.

In this XML-example we define our own tags to structure some data.

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<!-- This is a comment. -->
<!-- The first line is an XML declaration, defining
     version and encoding -->
<course>
<student>
Thomas Ericsson
</student>
<student>
Karin Andersson
</student>
</course>
```

XML is case sensitive, `<student>Thomas Ericsson</Student>` is illegal.
In the following example we use our own attributes as well:

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<course>
<student sex="male">   <!-- " " or ’ ’ -->
Thomas Ericsson
</student>
<student sex="female">
Karin Andersson
</student>
</course>
```

This is all we need to know about XML (but one can learn more). Here comes a simple data file, `xml_ex.vtr` (note `vtr`), in XML-format. The line numbers are <u>not</u> part of the file.
The file describes a `RectilinearGrid`(line 2), like the following Matlab example:

```
x = [-1 2]; y = [2 4]; z = [0 1 4];
[X, Y, Z] = ndgrid(x, y, z);
```

```
1   <?xml version="1.0" encoding="iso-8859-1" ?>
2   <VTKFile type="RectilinearGrid" version="0.1">
3
4    <RectilinearGrid WholeExtent="0 1 0 1 0 2">
5     <Piece Extent="0 1 0 1 0 2">
6
7      <PointData>
8       <DataArray type="Float32" Name="temperature">
9         1 2 3 4 5 6 7 8 9 10 11 12
10      </DataArray>
11     </PointData>
12
13     <Coordinates>
14      <DataArray type="Float32" NumberOfComponents="1"
15        -1 2
16      </DataArray>
17      <DataArray type="Float32" NumberOfComponents="1"
18         2 4
19      </DataArray>
20      <DataArray type="Float32" NumberOfComponents="1"
21        0 1 4
22      </DataArray>
23     </Coordinates>
24
25    </Piece>
26   </RectilinearGrid>
27  </VTKFile>
```

Here is an [Image].

`WholeExtent`, on line 4, gives indices (corresponds to indices in the `x`, `y` and `z`-arrays in the Matlab example). It is possible to have more than one piece, so one particular `Piece Extent` can be a subset of `WholeExtent`.

Lines 7-11 define a temperature, say. Lines 13-23 define what corresponds to the Matlab arrays. It is possible to use additional keywords, and `NumberOfComponets`is not necessary (default one). The indentation is not necessary.

In the following example comes an input file for a structured grid. Think of producing a grid using Matlab’s `ndgrid`, and then perturbing the points, but not so much that cells overlap or
intersect. In this file I have reproduced the rectilinear grid using a structured grid (which is a waste).

```
<?xml version="1.0" encoding="iso-8859-1" ?>

<VTKFile type="StructuredGrid" version="0.1">
 <StructuredGrid WholeExtent="0 1 0 1 0 2">
  <Piece Extent="0 1 0 1 0 2">
   <PointData>
    <DataArray type="Float32" Name="temp">
     1 2 3 4 5 6 7 8 9 10 11 12
    </DataArray>
   </PointData>

   <!-- x varies fastest, then y and last z -->
   <Points>
    <DataArray type="Float32" NumberOfComponents="3"
    -1     2     0
     2     2     0
    -1     4     0
     2     4     0
    -1     2     1
     2     2     1
    -1     4     1
```

```
        2     4     1
       -1     2     4
        2     2     4
       -1     4     4
        2     4     4
      </DataArray>
     </Points>
    </Piece>
   </StructuredGrid>
  </VTKFile>
```

Note that the keyword is `StructuredGrid`, that `NumberOfComponents="3"`and that filename ends in `.vts`. One can perturb the coordinates and still have a structured grid.

If we perturb the point sufficiently we do not get a structured grid, the points can be in arbitrary positions and we get an unstructured grid. Here comes the house-example again, but this time in XML-format.

```
<?xml version="1.0" encoding="iso-8859-1" ?>

<VTKFile type="UnstructuredGrid" version="0.1">
 <UnstructuredGrid>
  <Piece NumberOfPoints="10" NumberOfCells="9">

   <PointData>
    <DataArray type="Float32" Name="temperature">
        11 12 13 14 15 16 17 18 19 20
    </DataArray>
   </PointData>

   <Cells>
    <DataArray type="Int32" Name="connectivity">
        6  2  1  0   6  1  4  0   6  2  3  1
        6  3  7  1   6  5  4  1   6  7  5  1
        9  7  5  4   9  6  4  8   9  6  7  4
    </DataArray>
```

```

    <DataArray type="Int32" Name="offsets">
     4   8  12  16  20  24  28  32  36
    </DataArray>

    <DataArray type="UInt32" Name="types">
       10 10 10 10 10 10 10 10 10
    </DataArray>
   </Cells>

   <Points>
    <DataArray type="Float32" NumberOfComponents="3"
       0 0 0   2 0 0   0 1 0    2 1 0
       0 0 1   2 0 1   0 1 1    2 1 1
       0 0.5 1.5  2 0.5 1.5
    </DataArray>
   </Points>

  </Piece>
 </UnstructuredGrid>
</VTKFile>
```

The `offsets`-array contains the indices into the `connectivity`-array for the *end* of each cell. For some reason the offsets start at one (and not zero).

Finally an animation example, a cube that moves to the right changing colour at the same time. We store a sequence of frames, using a rectilinear format, one frame in each file. The file `animation.pvd`is a main file referring to the frame-files. In a real application we would probably have more frames (50-100 say).

```
<?xml version="1.0"?>
<VTKFile type="Collection" version="0.1">
 <Collection>
  <DataSet timestep="1" file="1.vtr"/>
  <DataSet timestep="2" file="2.vtr"/>
  <DataSet timestep="3" file="3.vtr"/>
  <DataSet timestep="4" file="4.vtr"/>
  <DataSet timestep="5" file="5.vtr"/>
 </Collection>
</VTKFile>
```

Here is the first frame-file, `1.vtr`:

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<VTKFile type="RectilinearGrid" version="0.1">

 <RectilinearGrid WholeExtent="0 1 0 1 0 1">
  <Piece Extent="0 1 0 1 0 1">
   <PointData>
    <DataArray type="Float32" Name="temp">
     1 1 1 1 1 1 1 1
    </DataArray>
   </PointData>
   <Coordinates>
    <DataArray type="Float32"> 0 1 </DataArray>
    <DataArray type="Float32"> 0 1 </DataArray>
    <DataArray type="Float32"> 0 1 </DataArray>
   </Coordinates>
  </Piece>
 </RectilinearGrid>
</VTKFile>
```

in the next frame frame, I change `temp` and the x-coordinates.

Even easier is to create files having names like `a1.vtr`, `a2.vtr`, `a3.vtr` etc. (must be a letter first) and then just mark the group of files when using the Open-alternative in the File-menu.

Textures

Sometimes one can increase the level of realism by using textures. A texture is a matrix with colour values, e.g. an image. In one lab you are going to simulate the Sun-Earth-Moon system, using textures for the Earth and Moon. Textures are common in computer games, e.g. a brick wall in a castle would be drawn using a texture instead of drawing brick by brick. A texture could be the result of a computation as well, a procedural texture. Graphics cards have support for working with textures.

The default behaviour (can be changed) is that the colour of the texture will be mixed with the colour of the pixels in a polygon.
An image is made up by a finite set of pixels (often called texels in this context) but using some form of interpolation OpenGL will provide the colour in an arbitrary point in the texture: texture(s, t). s and t are two coordinates, $0 \le s, t \le 1$ (usually).

We need to map the texture onto a surface, e.g. a rectangle. In the lab we will map a texture onto a sphere. We do this by giving an (s, t)-pair for every (x, y, z) on the surface. So the code may look something like

```
... compute s, t, x, y and z

      glTexCoord2f(s, t);
      glVertex3f(x, y, z);
```

OpenGL must be able to change the size of the texture, e.g. if we change the size of the window. More about that later on.

To create the texture we need to know how it should be stored. My examples assume that every texel is represented by an RGB-triple, each colour consisting of an unsigned byte. The datatype in OpenGL is `GLubyte`. In the GL-header file, `gl.h`, it says `typedef unsigned char GLubyte;`

In the manual page for **glTexImage2D** it says:

The first element corresponds to the lower left corner of the texture image. Subsequent elements progress left-to-right through the remaining texels in the lowest row of the texture image, and then in successively higher rows of the texture image. The final element corresponds to the upper right corner of the texture image.
Here is the order if the width is 3 and the height is 2.

```
 3  4  5
 0  1  2
```

If we store the RGB-triples in sequence in an one-dimensional array it would look like this.

```
  r(0) g(0) b(0)    texel 0
  r(1) g(1) b(1)    texel 1
  r(2) g(2) b(2)    texel 2
  r(3) g(3) b(3)    texel 3
  r(4) g(4) b(4)    texel 4
  r(5) g(5) b(5)    texel 5
```

The colours are stored in byte order in memory, so an array `Glubyte vec[2 *3 * 3];` would work like this:

```
  vec[0]  <-> red(0)
  vec[1]  <-> green(0)
  vec[2]  <-> blue(0)
  vec[3]  <-> red(1)
   etc.
```

Another way is to use a matrix. In C the rightmost dimension varies fastest then comes the columns and last the rows, so like this:

```
  Glubyte mat[2][3][3];

  mat[0][0][0]   // red
  mat[0][0][1]   // green
  mat[0][0][2]   // blue
  mat[0][1][0]   // red
  mat[0][1][1]
  mat[0][1][2]
  mat[0][2][0]
  mat[0][2][1]
  mat[0][2][2]

  mat[1][0][0]  Next row
  mat[1][0][1]
  mat[1][0][2]
  mat[1][1][0]
  mat[1][1][1]
  mat[1][1][2]
  mat[1][2][0]
  mat[1][2][1]
  mat[1][2][2]
```

Usually we would have much larger textures than this. Small textures may, in fact, lead to problems. It used to be that the width and height had to be powers of two. Some implementations require even numbers and perhaps a minimum size. One reason for this is performance. Some machines have hardware that is far more efficient at moving data to and from the framebuffer if the data is aligned on two-byte, four-byte, or eight-byte boundaries in processor memory.
The default alignment is four, and in our example one row occupies $3 \cdot 3 = 9$ bytes, leading to misaligned rows (and an incorrect image on the screen). If we pad the matrix

```
  Glubyte tex[2][4][3];
```

keeping the values of height and width, it works. Another way is to change the alignment by the following calls:

```
  glPixelStorei(GL_PACK_ALIGNMENT, 1);
  glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
```

Here comes a small example where we construct the textures using a function. First a routine `MakeTexture` which is called from `main` (before `glutMainLoop` is called).

```
void MakeTexture()
{
  int     width = 3, height = 2;
  GLubyte mat[height][width][3],
          vec[3 * width * height];

  // loops are an alternative :-)
  mat[0][0][0] = mat[0][0][1] = mat[0][0][2] = 50;
  mat[0][1][0] = mat[0][1][1] = mat[0][1][2] = 100;
  mat[0][2][0] = mat[0][2][1] = mat[0][2][2] = 150;

  mat[1][0][0] = mat[1][0][1] = mat[1][0][2] = 250;
  mat[1][1][0] = mat[1][1][1] = mat[1][1][2] = 200;
  mat[1][2][0] = mat[1][2][1] = mat[1][2][2] = 150;

  vec[0]  = vec[1]  = vec[2]  = 150;
  vec[3]  = vec[4]  = vec[5]  = 200;
  vec[6]  = vec[7]  = vec[8]  = 250;

  vec[9]  = vec[10] = vec[11] = 150;
  vec[12] = vec[13] = vec[14] = 100;
  vec[15] = vec[16] = vec[17] =  50;

  // For all future pixel operations
  glPixelStorei(GL_PACK_ALIGNMENT,   1);
  glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
```

```
  glBindTexture(GL_TEXTURE_2D, 100);
  // Done for each texture
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
                  GL_NEAREST);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
                  GL_NEAREST);
  glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height,
               0, GL_RGB, GL_UNSIGNED_BYTE, mat);

  glBindTexture(GL_TEXTURE_2D, 200);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
                  GL_NEAREST);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
                  GL_NEAREST);
  glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height,
               0, GL_RGB, GL_UNSIGNED_BYTE, vec);

  glEnable(GL_TEXTURE_2D);
  glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE,
            GL_MODULATE);
}
```

Note that we normally would not change the alignment.
`glBindTexture` gives the texture, to be defined, a name (a positive integer, 100 in this case).

We do not usually have an image that contains the same number of texels as the number of pixels in the rectangle (polygon). `glTexParameteri` is used to define what should happen if the rectangle is smaller or larger than the texture.
GL_TEXTURE_MIN_FILTER defines the function which is used when the texture must be minified. **GL_TEXTURE_MAG_FILTER** defines the function which is used when the texture must be magnified.

When **texture(s, t)** is needed, **GL_NEAREST** tells OpenGL to use colour from the nearest pixel (in $|| \, ||_1$) in the original image. Another choice is **GL_LINEAR**. This uses a weighted average of the four texture elements that are closest to the center of the pixel being textured.

GL_NEAREST is generally faster than **GL_LINEAR**, but can produce textured images with sharper edges because the transition between texture elements is not as smooth.

In **glTexImage2D** we finally make the image data available to the OpenGL-system. The parameters are: **GL_TEXTURE_2D** defines the type of the texture, level specifies the level of detail. Level 0 is the base level.

GL_RGB specifies the number of colours in the texture (we could have written 3). **width** and **height** obvious. It is possible to have a border around the texture, we say that its width is zero. This **GL_RGB** specifies the format of the data (**mat** and **vec** contain RGB-triples), and **GL_UNSIGNED_BYTE** is the type. Finally comes an address to the data.

glEnable enables texturing.
The last call (which is unnecessary, since I have chosen the default value) says that the colour of the textures should be mixed with the colour of the object.

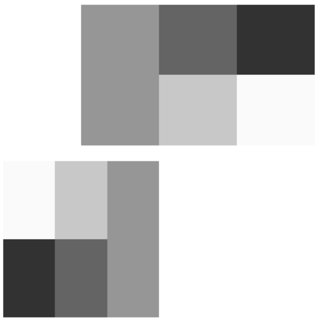
So the resulting red (ambient + diffuse + specular) component, for example, in a pixel becomes $r_s \cdot r_t$, where r_s is the red component originating from the ordinary shading computation and r_t is the red component from the texture.

In the **Display**-routine below we bind the two textures to two rectangles. In this simple program lighting is not used, so the textures will modulate the colour white, set by
glColor3f(1, 1, 1);

The call of **glBindTexture** picks the 100-texture. The pairs of calls to **glTexCoord2f** and **glVertex3f** defines the mapping between image and rectangle. Note that we can deform the image by changing the mapping.

```
void Display()
{
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  glColor3f(1, 1, 1);

  glBindTexture(GL_TEXTURE_2D, 100);
  glBegin(GL_POLYGON);
    glTexCoord2f(0.0, 0.0); glVertex3f(0.0, 0.0, 0.5);
    glTexCoord2f(1.0, 0.0); glVertex3f(1.0, 0.0, 0.5);
    glTexCoord2f(1.0, 1.0); glVertex3f(1.0, 1.0, 0.5);
    glTexCoord2f(0.0, 1.0); glVertex3f(0.0, 1.0, 0.5);
  glEnd();

  glBindTexture(GL_TEXTURE_2D, 200);
  glBegin(GL_POLYGON);
    glTexCoord2f(0.0, 0.0); glVertex3f(0.5, 1.1, 0.5);
    glTexCoord2f(1.0, 0.0); glVertex3f(2.0, 1.1, 0.5);
    glTexCoord2f(1.0, 1.0); glVertex3f(2.0, 2.0, 0.5);
    glTexCoord2f(0.0, 1.0); glVertex3f(0.5, 2.0, 0.5);
  glEnd();

  glFlush();
}
```

Here is part of the window (since I used grayscale in the images it is easy to interpret the result). The origin is in the lower left corner of the leftmost black rectangle.



Let us try a harder example. We are going to wrap an OpenGL-logo on a cylinder. The cylinder is symmetric around the y-axis. An additional problem is that we are going to use light, so the program has to compute normals. Just to see that I have produced the image in the correct way the program puts the image on a rectangle as well. I used **xv** to transform the image, from gif to **PBM/PGM/PPM (ascii)** (as it says in **xv**). I named the file **opengl.ppm** and the first lines look like:

```
P3
# CREATOR: XV version 3.10a-jumboFix+Enh of 20050501
220 97
255
255 255 255 255 255 255 255 255 255 255 255 255 255 255
255 255 255 255 255 255 255 255 255 255 255 255 255 255
```

220 97 is the dimension (which I could have read in). It is hard-coded in the code. As it turns out I have to reverse the rows when reading the lines (or the logo will be upside-down). First comes the resulting image and then parts of the program.

```
void MakeTexture()
{
  int     r, g, b, row, col, width = 220, height = 97;
  char    c;
  GLubyte logo[height][width][3];
  FILE    *fp;

  if ((fp = fopen("opengl.ppm", "r")) == NULL) {
    printf("Problems opening opengl.ppm.\n");
    exit(1);
  }

  row = 0;
  do {                         // skip the header
    fscanf(fp, "%c", &c);
    if ( c == '\n' ) row++;
  } while ( row < 4 );
```

```
  for (row = height - 1; row >= 0; row--)  // reverse
    for (col = 0; col < width; col++) {
      fscanf(fp, "%d %d %d", &r, &g, &b);
      logo[row][col][0] = r;
      logo[row][col][1] = g;
      logo[row][col][2] = b;
    }

  fclose(fp);

  glBindTexture(GL_TEXTURE_2D, 100);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
                  GL_NEAREST);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
                  GL_NEAREST);
  glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height,
               0, GL_RGB, GL_UNSIGNED_BYTE, logo);
  glEnable(GL_TEXTURE_2D);
}
```

The following routine is called from `Display` (as is a routine drawing a coordinate system).

```
void CreateObject()
{
  int    k;
  double r, c, s, phi, d_phi, TWO_PI = 2.0* M_PI, seg;
  float  white_rc[]  = {1, 1, 1, 1}, spec_exp = 100;

  glMaterialfv(GL_FRONT, GL_AMBIENT,   white_rc);
  glMaterialfv(GL_FRONT, GL_DIFFUSE,   white_rc);
  glMaterialfv(GL_FRONT, GL_SPECULAR,  white_rc);
  glMaterialf (GL_FRONT, GL_SHININESS, spec_exp);

  glBindTexture(GL_TEXTURE_2D, 100);
```

```
  // Draw a rectangle
  glNormal3f(1, 0, 0);  // Note
  glBegin(GL_POLYGON);
    glTexCoord2f(0.0, 0.0); glVertex3f(0.0, 1.5,  2.0);
    glTexCoord2f(1.0, 0.0); glVertex3f(0.0, 1.5, -2.0);
    glTexCoord2f(1.0, 1.0); glVertex3f(0.0, 3.5, -2.0);
    glTexCoord2f(0.0, 1.0); glVertex3f(0.0, 3.5,  2.0);
  glEnd();

  // Draw a cylinder
  seg   = 10;
  d_phi = TWO_PI / seg;
  r     = 2;

  glBegin(GL_QUAD_STRIP);
  for (k = 0; k <= seg; k++) {
    phi = k * d_phi;
    c   = cos(phi);
    s   = sin(phi);
    glNormal3f(s, 0, c);  // Note
    c *= r;
    s *= r;
    glTexCoord2f(k / seg, 0.0); glVertex3f(s, 0, c);
    glTexCoord2f(k / seg, 1.0); glVertex3f(s, 2, c);
  }
  glEnd();
}
```

In order to understand the last loop we first read the manual page for `glBegin`. It says the following about `GL_QUAD_STRIP`:

`GL_QUAD_STRIP` Draws a connected group of quadrilaterals. One quadrilateral is defined for each pair of vertices presented after the first pair. Vertices 2n-1, 2n, 2n+2, and 2n+1 define quadrilateral n. N/2-1 quadrilaterals are drawn. ...

So if we have vertices numbered 1, 2, 3, etc., this is the way they are used to define the quadrilaterals.

```
   2       4       6
   o-------o-------o---
   |       |       |
   |       |       |
   o-------o-------o---
   1       3       5
```

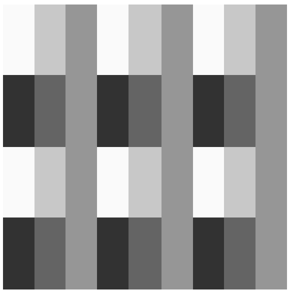
So the first quadrilateral (n = 1) is defined by vertices 1, 2, 4, 3 (2n-1, 2n, 2n+2, and 2n+1).

Now to the cylinder. $[\sin\varphi, 0, \cos\varphi]$ describes a circle in the x-z-plane. $[\sin\varphi, 2, \cos\varphi]$ is another circle at $y = 2$. Since we are alternating between $y = 0$ and $y = 2$, we get the correct order for using `GL_QUAD_STRIP`.

When textures are used in computer games, for example, it may be interesting to repeat a texture. To put a wallpaper on a wall it may be sufficient to define a small part of the pattern. The repetition happens automatically if we use texture coordinates outside [0, 1], texture(1.2, 3.4) becomes texture(0.2, 0.4) (leaving the fractions). To change this behaviour we can ask for clamping instead; using one image but stretching the pixels on the edges. The following code

```
  glBindTexture(GL_TEXTURE_2D, 100);
  glBegin(GL_POLYGON);
    glTexCoord2f(0.0, 0.0); glVertex3f(0.0, 0.0, 0.5);
    glTexCoord2f(3.0, 0.0); glVertex3f(1.0, 0.0, 0.5);
    glTexCoord2f(3.0, 2.0); glVertex3f(1.0, 1.0, 0.5);
    glTexCoord2f(0.0, 2.0); glVertex3f(0.0, 1.0, 0.5);
  glEnd();
```

will produce two image-rows with three image-columns (so our original image occurs six times).



Another way (mipmapping) to solve the minification problem is to let OpenGL build a sequence of images in decreasing sizes. This must be used in the planet-lab, otherwise the Moon-texture will flicker (it looks like small electric flashes).

“mip” is an acronym for *multum in parvo*, which is Latin for something like “much in little”.

This is what it may look like in the lab:

```
glBindTexture(GL_TEXTURE_2D, 100);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
                GL_NEAREST);

// New
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
                GL_LINEAR_MIPMAP_NEAREST);

// You have to set width, height and texture
gluBuild2DMipmaps ( GL_TEXTURE_2D, GL_RGB,
                    width, height,
                    GL_RGB, GL_UNSIGNED_BYTE,
                    texture );
```

`GL_LINEAR_MIPMAP_NEAREST` (looks best, I think) picks the mipmap that most closely matches the size of the pixel being textured and uses the `GL_LINEAR` criterion to produce a texture value.