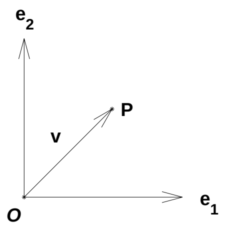
Let e_1, e_2 and e_3 be a basis in 3D. A point, P, can be written $P = p_x e_1 + p_y e_2 + p_z e_3 + \mathcal{O}$. $P - \mathcal{O}$ is the vector which is a linear combination of the basis. $p_x e_1 + p_y e_2 + p_z e_3$.

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

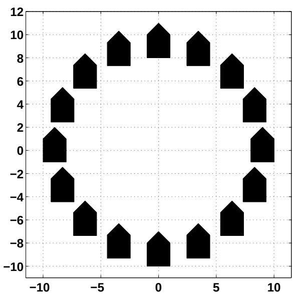
$$P = [e_1,\ e_2,\ e_3,\ \mathcal{O}] \begin{bmatrix} p_x \\ p_y \\ p_z \\ 1 \end{bmatrix} \quad \text{and} \quad v = [e_1,\ e_2,\ e_3,\ \mathcal{O}] \begin{bmatrix} v_x \\ v_y \\ v_z \\ 0 \end{bmatrix}$$

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.

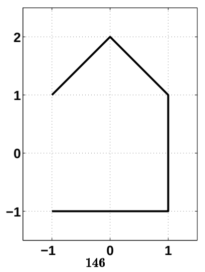
In computer graphics it is common to change coordinate systems. Suppose we would like to produce the following image (the coordinate system should not be included).



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:

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

$$\underbrace{p_x M \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}}_{e_1'} + \underbrace{p_y M \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}}_{e_2'} + \underbrace{1 M \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}}_{\mathcal{O}'} = p_x e_1' + p_y e_2' + 1\mathcal{O}'$$

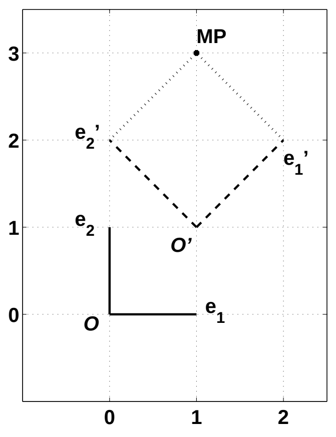
So, MP, can be interpreted as using the <u>original</u> coordinates for P, but in the <u>transformed</u> coordinate system $\{e_1', e_2', \mathcal{O}'\} = \{Me_1, Me_2, M\mathcal{O}\}$.

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

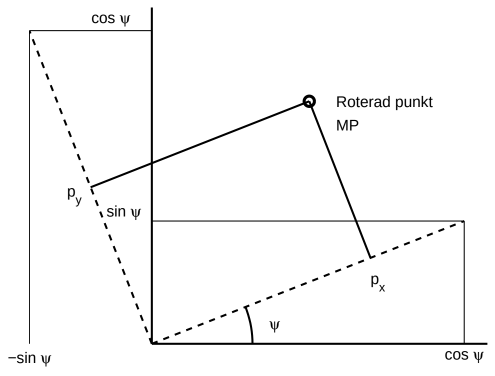
We would like to rotate points an angle ψ, ccw (counter clockwise) around the origin. Here is M:

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

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

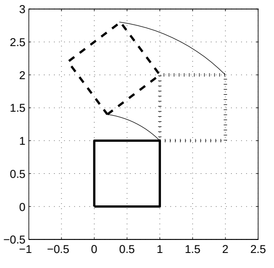
Let us study $M = TR$ and $M = RT$ where T is a translation and R is a rotation. Set $C = R(1:2, 1:2)$. We get:

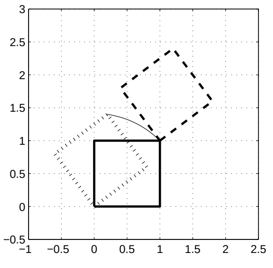
$$RT = \begin{bmatrix} C & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} I & t \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} C & Ct \\ 0 & 1 \end{bmatrix}$$

$$TR = \begin{bmatrix} I & t \\ 0 & 1 \end{bmatrix} \begin{bmatrix} C & 0 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} C & t \\ 0 & 1 \end{bmatrix}$$

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.

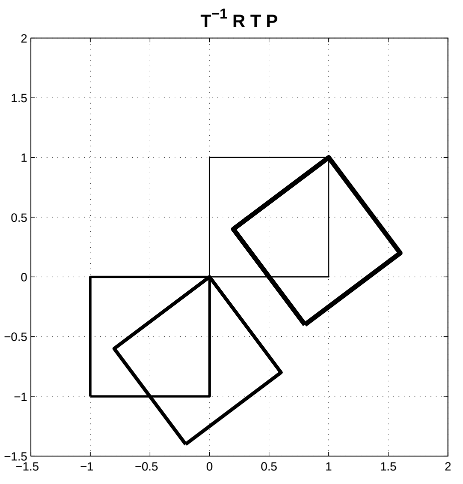
The following images show RT and TR acting on the unit square. The dotted unit squares shows the situation after the first step of the transformations have been applied. The dashed lines show the final result. As usual it is sufficient to look at the corners.





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

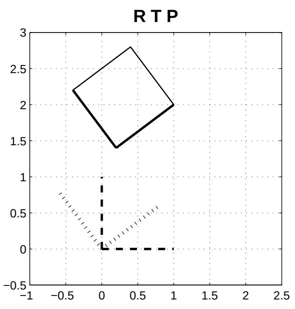
Let us return to the $\boldsymbol{RT}$, $\boldsymbol{TR}$-example and see how this is done in OpenGL (at least in principle, all the details will come later). Suppose we do the following function calls in OpenGL (. . . marks parameters that we skip for the time being):

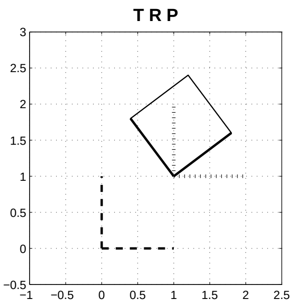
```
Call                   Matrix operation

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, $M = \boldsymbol{RT}$ even though we made the Rotate first and the Translate after.

To get $M = \boldsymbol{TR}$ 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 $M = \boldsymbol{RT}$, 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 $M = \boldsymbol{TR}$ 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

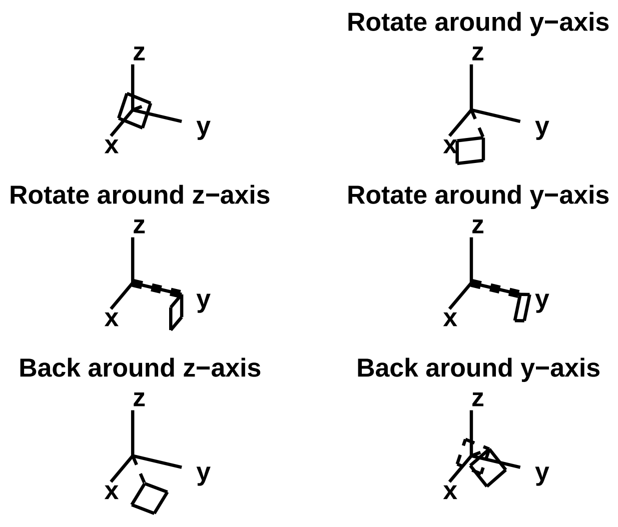
Here are the most common transformations in 3D. $\boldsymbol{S}$ = scaling, $\boldsymbol{T}$ = translation.

$$\boldsymbol{S} = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \quad \boldsymbol{T} = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$\boldsymbol{R_x}$ = rotation ccw the angle ψ around the x-axis (when we look along the negative x-axis). Analogous for $\boldsymbol{R_y}$ and $\boldsymbol{R_z}$.
Let $c = \cos\psi$ and $s = \sin\psi$.
Note that we have the number one for the axis.

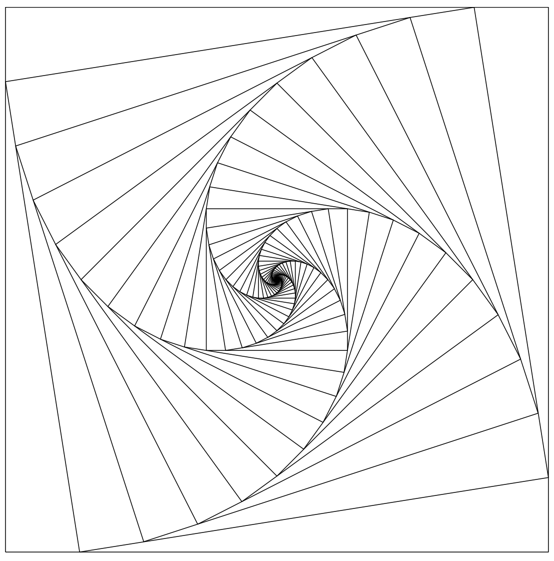
$$\boldsymbol{R_x} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & c & -s & 0 \\ 0 & s & c & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \quad \boldsymbol{R_y} = \begin{bmatrix} c & 0 & s & 0 \\ 0 & 1 & 0 & 0 \\ -s & 0 & c & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \quad \boldsymbol{R_z} = \begin{bmatrix} c & -s & 0 & 0 \\ s & c & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

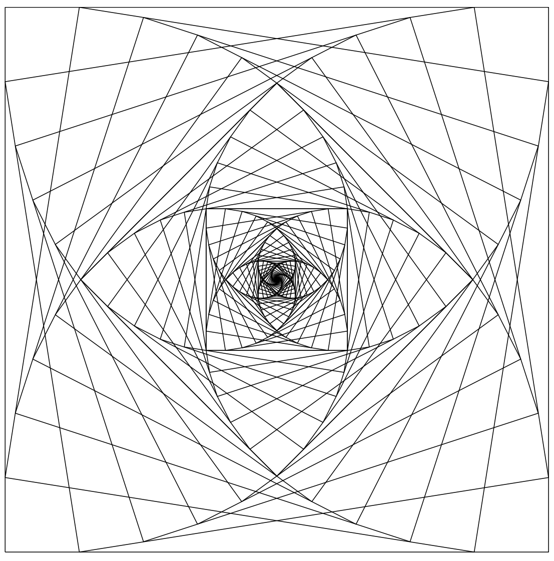
To make more complicated transformations we can combine the above. Here is an example where we want to rotate the square around the dashed axis.

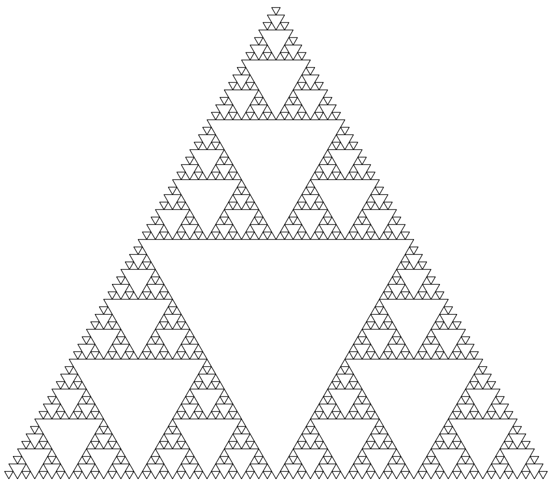


A few exercises

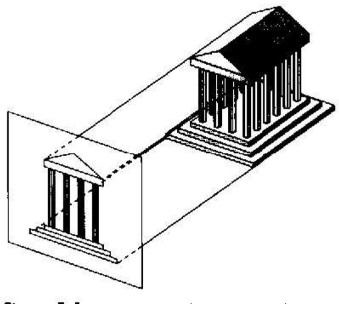
- Find the M-matrix that maps the rectangle, with corners in $(1,1)$, $(3,3)$, $(2,4)$ and $(0,2)$, onto the unit square.
- Find the M-matrix which maps the quadrilateral, with corners in $(0,0)$, $(1,0)$, $(2,1)$ and $(1,1)$, on the unit square. This is an example of a shear transformation.
- Let $R(\psi)$ 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).



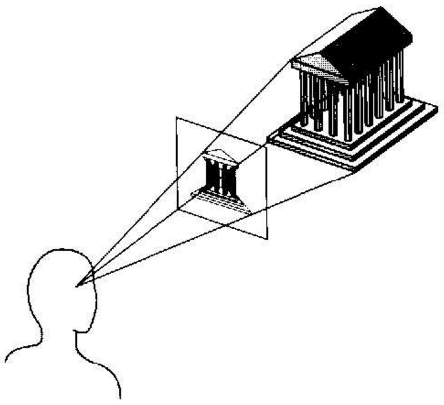




Projections

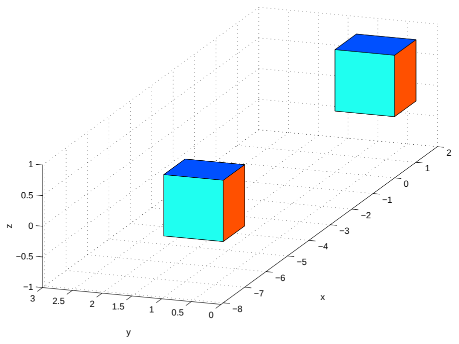


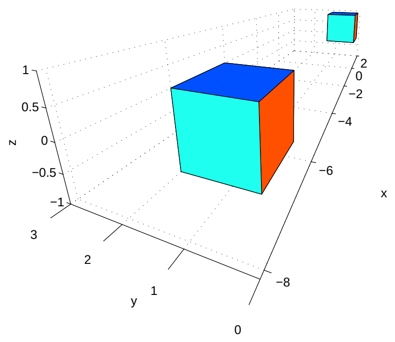
Orthografic (parallel) projection

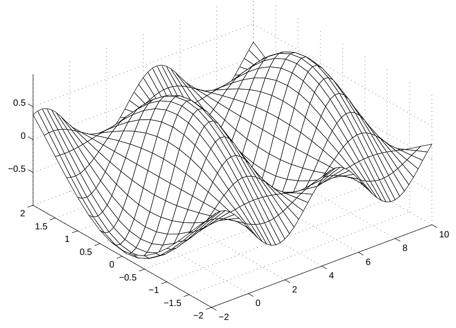


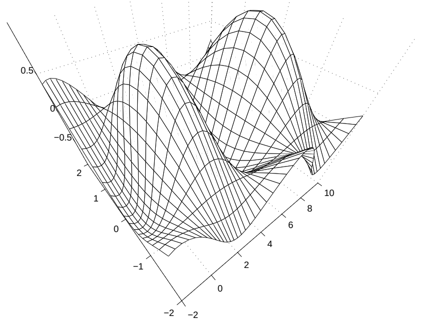
Perspective projection

```
set(gca, 'Projection', 'orthographic', ... % first plot
         'CameraPosition', [-10 -1 2])}
set(gca, 'Projection', 'perspective', ...  % second plot
         '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:

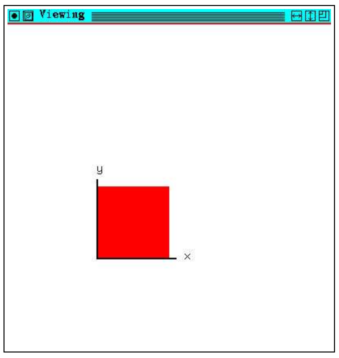
```
  glOrtho(x_min, x_max, y_min, y_max, near, far)
```

The view volume is a box (rectangular parallelepiped).

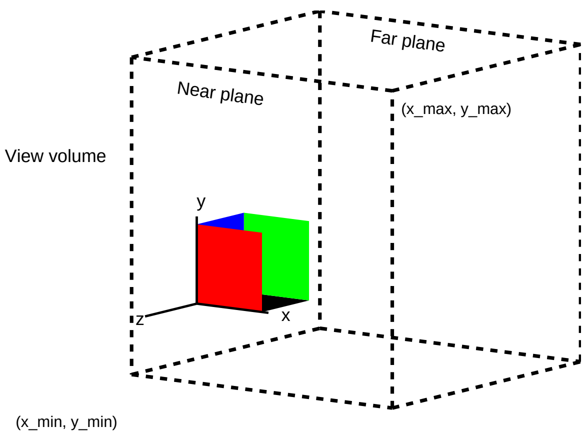
Here is an example. I have created a square window and made the call:

```
  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.



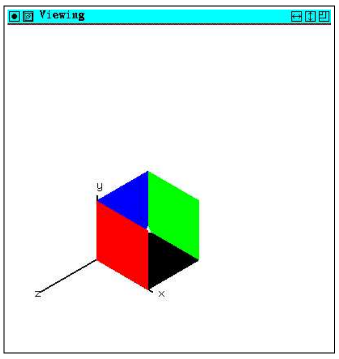
In this example the read square lies in the "near plane". If we increase **near** the near plane is moved away from the eye (more negative z). Part of the scene will clipped away (one talks about the near clipping plane, as well). In the same way we get clipping if we move the "far plane" towards the 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.

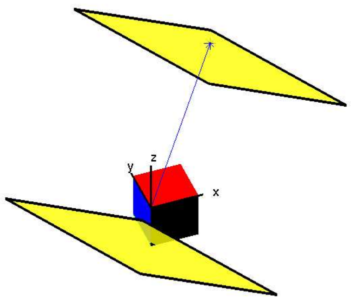
Here is the widow after the following call:

```
  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.

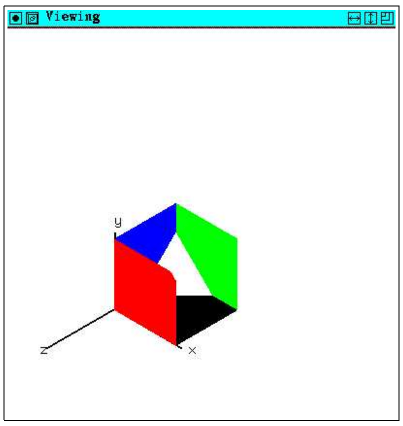


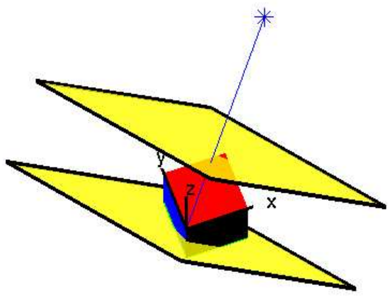
Here is a Matlab illustration, from an other angle:



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
```

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.

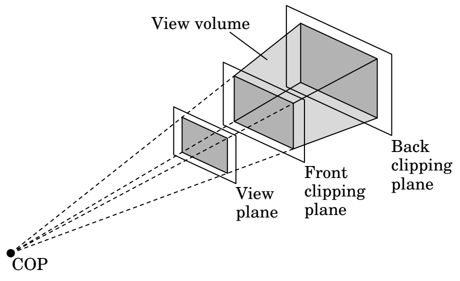
One can set the matrix as well:

```
  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 "view-port", 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: $0 \le x \le 1$, $-1 \le y \le 5$ and $-4 \le z \le 0$. So the following transformation should work:

```
  T = eye(4); T(1:3, 4) = [-0.5; -2; 2]   % translate
```

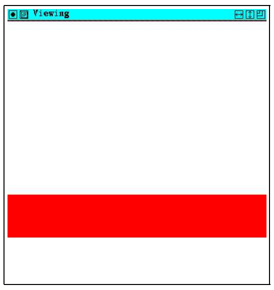
and then scale

```
  S = diag([2 1/3 -1/2 1])  % -1/2, reversal of z-axis
```

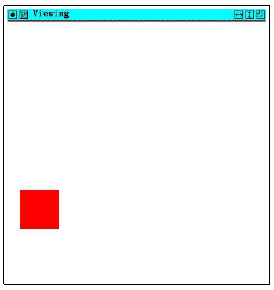
The product `S * T` is what OpenGL produces as well:

```
GL_PROJECTION_MATRIX
      2         0         0        -1
      0     0.333         0    -0.667
      0         0      -0.5        -1
      0         0         0         1
```

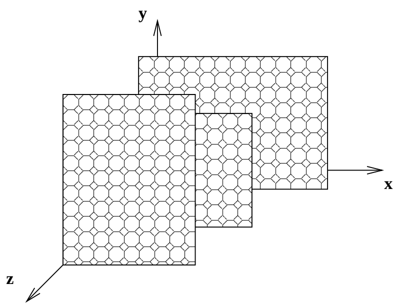
On the next page I have plotted the unit square using the view volume above and without moving the eye. The window was 400x400 pixels and the viewport had the same dimension as the window.
We can see that the square is deformed.



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 polygon
    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).

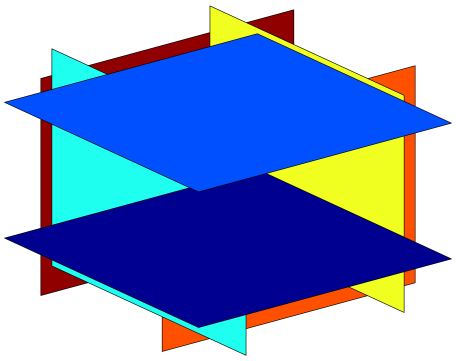
`opengl` sometimes renders images in an incorrect way.

A disadvantage with both `zbuffer` and `opengl` is that the PostScript files can be very large.

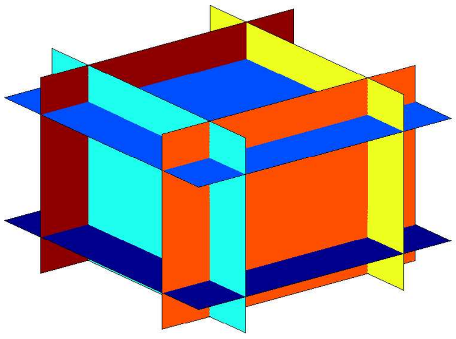
```
>> peaks  % a demo-command that draws a surface
>> set(1,'renderer')
[ {painters} | zbuffer | OpenGL | None ]
>> print -depsc peak_paint.eps

>> 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):

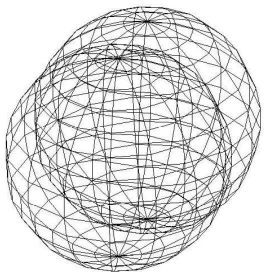
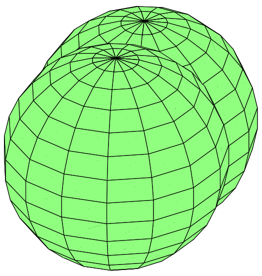
magenta: 1860, in allusion to the Battle of Magenta, in Italy, where the French and Sardinians defeated the Austrians in 1859, because the brilliant crimson aniline dye was discovered shortly after the battle.... `www.etymonline.com`

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`

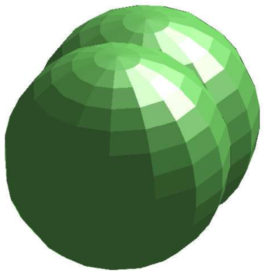
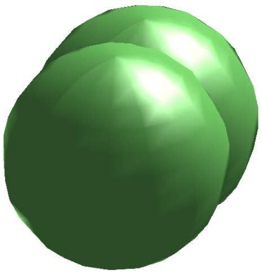
Gamut: Medieval Latin gamma, lowest note of a medieval scale (from Late Latin, 3d letter of the Greek alphabet)
1: the whole series of recognized musical notes
2: an entire range or series “ran the gamut from praise to contempt” `www.m-w.com`

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).

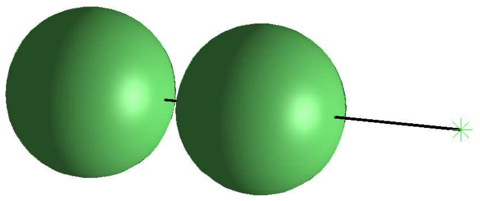



Shading does nor mean “shadows”, but it means to color so that the shades, of colour, pass gradually from one to another.

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:

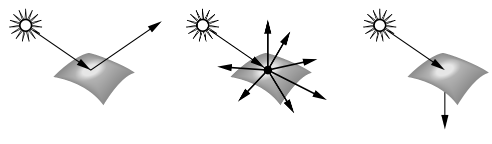
- Point sources (can shine in all directions, like the Sun, or in a limited cone, like a spotlight). We can have distant light sources (the Sun) or local (a table lamp). It is faster to do the computations for distant light sources since only direction and not actual distance has to be considered.
- 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.

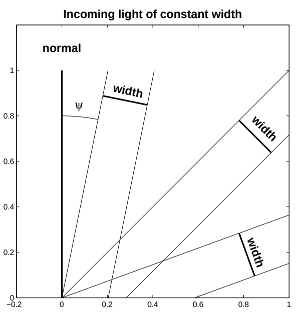
We now look at light having a direction, and we will see how much is reflected to the eye.



(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 $\boldsymbol{L}$ be the normalized direction to the light source, and let $\boldsymbol{n}$ be the normal to the surface in the point where the ray hits, then $\cos\psi = \boldsymbol{L} \cdot \boldsymbol{n}$.
If $\boldsymbol{L} \cdot \boldsymbol{n} < 0$ the backside of the polygon is hit by the light, but according to our assumption we cannot so that side, so the intensity becomes:

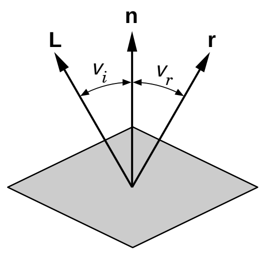
$$\rho_d I_d \max\left[\boldsymbol{L} \cdot \boldsymbol{n}, 0\right]$$

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

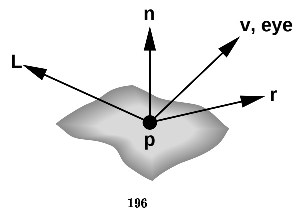
Next, specular reflection. If we have a perfectly polished surface and a spotlight is located in the $\boldsymbol{L}$-direction, the eye will see a reflected ray only if it is located along $\boldsymbol{r}$.



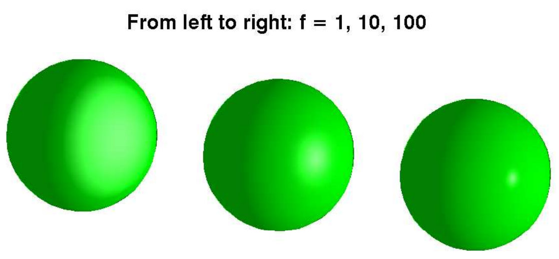
Real-life surfaces are not perfect, so a more realistic model will show light also in the vicinity of the $\boldsymbol{r}$-direction. The amount of reflected light should decrease when we move away from $\boldsymbol{r}$.
The Phong reflection model (Bui Tuong Phong, b. Vietnam, 19??-1998) tries to capture this behaviour. The intensity of the reflected light is

$$\rho_s I_s (\boldsymbol{r} \cdot \boldsymbol{v})^f$$

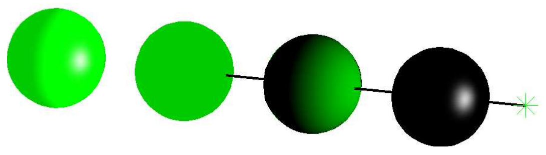
$\boldsymbol{r}$ is as above and $\boldsymbol{v}$ is the normalized direction to the eye. f is the "specular reflection coefficient" and it measures how much the light is spread. A large f gives a small spread of light and a small f gives a large spread. OpenGL approximates the angle, by using the angle between $\boldsymbol{n}$ and $\boldsymbol{L} + \boldsymbol{v}$ (which is $\psi/2$ if all the vectors lie in the same plane). This makes it unnecessary to compute $\boldsymbol{r}$ (faster).



This is how the intensity varies with f:



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 <u>one</u> 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.

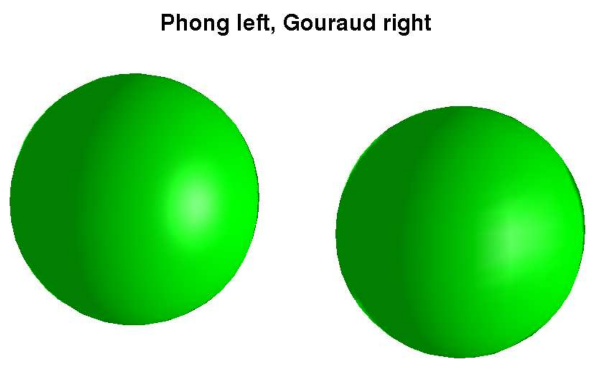
Suppose this is the polygon, with corners a-d:

```
     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.

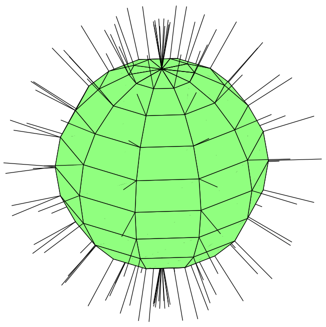
In this image one can (on the screen at least) see that Phong shading gives a less jagged highlight.



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 quit 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.

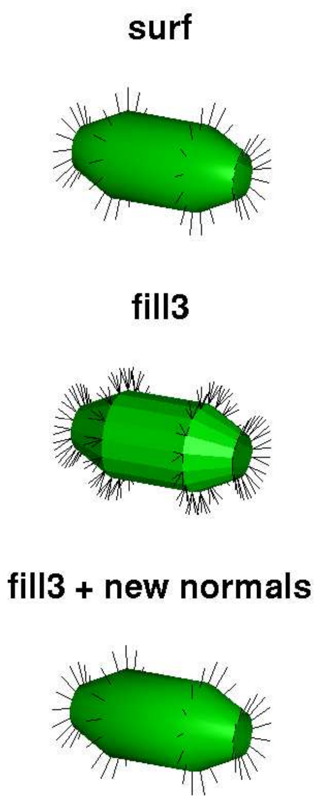
This is not quite the case when we use the `fill3`-command. Here is an example. I have reused the cylinder example.

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.



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.

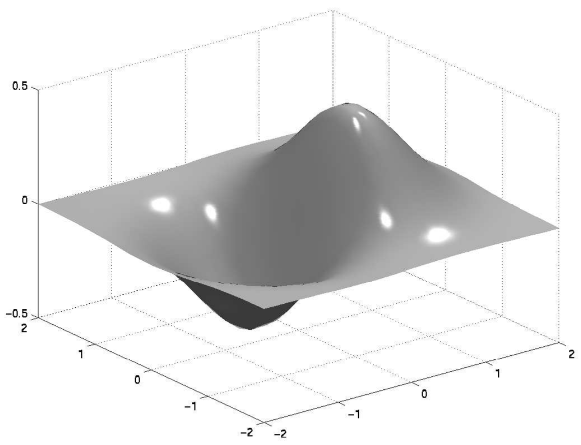
The `colorbar`-command places a colour bar in the plot. This provides a connection between colour and the numerical values in the `C`-matrix (or the z-values if no `C` is present).

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:

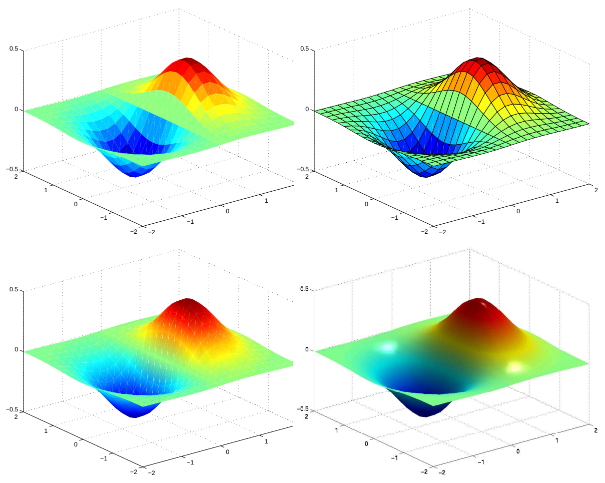
```
                       facecolor         edgecolor
shading flat:          flat              none
shading interp:        interp            none
shading faceted:       flat              black

                       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
```

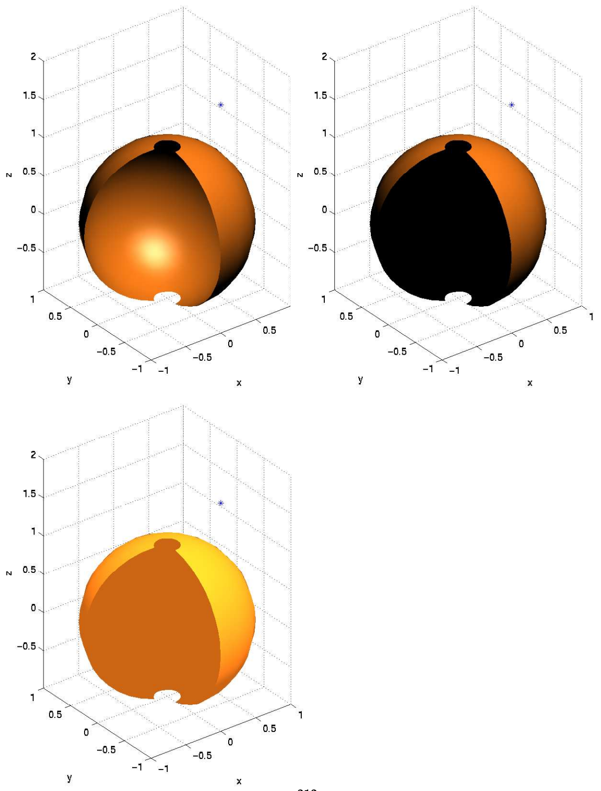


The back and front of polygons

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?

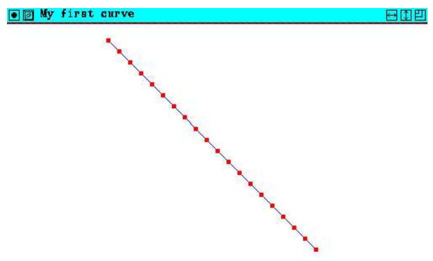
Because OpenGL doesn't provide routines for interfacing with a windowing system or input devices, an application must use a variety of other platform-specific routines for this purpose. The result is nonportable code.

Furthermore, these platform-specific routines tend to be full-featured, which complicates construction of small programs and simple demos.

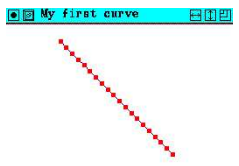
GLUT is a library that addresses these issues by providing a platform-independent interface to window management, menus, and input devices in a simple and elegant manner.

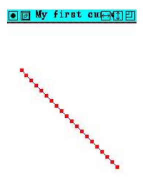
Some OpenGL-examples

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.

```
1   // I'm using C++-comments // in this code.
2   #include <GL/glut.h> // includes gl.h, glu.h as well
3   #include <stdlib.h>  // For void exit(int)
4
5   void Display();      // Prototypes
6   void MyInit();
7   void Reshape(int, int);
8   void KeyHandler(unsigned char, int, int);
9
10  // argc = argument count >= 1 (command name first)
11  // argv = arg vector (array of pointers to char)
12
13  int main(int argc, char *argv[])
14  {
15    glutInit(&argc, argv);
16
17    // use RGB-color and not indexed color
18    glutInitDisplayMode(GLUT_RGB);
19
20    // width = 500, height = 300 pixels
21    glutInitWindowSize(500, 300);
22
23    // (0, 0) upper-left corner of screen
24    glutInitWindowPosition(10, 10);
25    glutCreateWindow("My first curve"); // title
26
27    // the following calls define three callbacks
28    glutDisplayFunc(Display);      // at re-displays
29    glutReshapeFunc(Reshape);      // change in size
30    glutKeyboardFunc(KeyHandler); // keypress
31
32    MyInit();       // my own initializations
33    glutMainLoop(); // wait for events
34    return 0;
35  }
36
37
```

```
38  void MyInit()
39  {
40    glClearColor(1, 1, 1, 0); // white to erase
41
42    // set up projection matrix
43    glMatrixMode(GL_PROJECTION);
44    glLoadIdentity(); // matrix = I
45
46    // 2D orthographic projection
47    // x_min, x_max, y_min, y_max
48    gluOrtho2D(0, 2, -1, 1);
49
50    // set the modelview matrix to I
51    glMatrixMode(GL_MODELVIEW);
52    glLoadIdentity();
53  }
54  void Display()
55  {
56    float x;
57    // clear color buffer, i.e. erase
58    glClear(GL_COLOR_BUFFER_BIT);
59
60    glColor3f(0, 0, 1);         // blue
61    glBegin(GL_LINE_STRIP);     // draw solid curve
62      glVertex2f(0, 1);         // define point
63      glVertex2f(1.9, -0.9);    // define point
64    glEnd();                    // end of curve
65
66    // Note that glColor is in effect for all
67    // points defined by glVertex2f.
68    glColor3f(1, 0, 0);         // new color
69    glPointSize(5);             // larger points
70    glBegin(GL_POINTS);         // draw points
71      for(x = 0; x < 1.99; x += 0.1)
72        glVertex2f(x, 1 - x); // define point
73    glEnd();                    // end of GL_POINTS
74
```

```
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(GL_SMOOTH)`). If one has switched on flat shading (`glShadeModel(GL_FLAT)`) 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
  glFlush();
  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);

```

```
66    glEnable(GL_LIGHTING); // switch on lighting
67    glEnable(GL_LIGHT0);   // at least 8 lamps
68
69    // set the position of light0
70    glLightfv(GL_LIGHT0, GL_POSITION, light_pos);
71
72    // switch on smooth shading; the other
73    // alternative is GL_FLAT
74    glShadeModel(GL_SMOOTH);
75
76    glEnable(GL_DEPTH_TEST);
77  }
78  void CreateObject()
79  {
80    float // material properties (refl. coeff.)
81      white_rc[]  = {1, 1, 1, 1},
82      red_rc[]    = {1, 0, 0, 1},
83      green_rc[]  = {0, 1, 0, 1},
84      spec_exp    = 100;
85
86    // define material properties for front face
87    glMaterialfv(GL_FRONT, GL_AMBIENT,   white_rc);
88    glMaterialfv(GL_FRONT, GL_DIFFUSE,   red_rc);
89    glMaterialfv(GL_FRONT, GL_SPECULAR,  white_rc);
90    glMaterialf (GL_FRONT, GL_SHININESS, spec_exp);
91
92    // create the polygons and normals for a
93    // sphere; radius, resolution along
94    // longitudes and latidudes
95
96    glutSolidSphere(1, 20, 20);
97
98    // the translate should be temporary
99    glPushMatrix();
100     glTranslatef(2, 0, 0);
101     glMaterialfv(GL_FRONT, GL_DIFFUSE, green_rc);
102     glutSolidSphere(1, 20, 20);
```

```
103     glPopMatrix();
104 }
105 void KeyHandler(unsigned char key, int x, int y)
106 {
107   float  light_pos[] = {5, 0, 0, 0};
108
109   if (key == 'q')
110     exit(0);
111   else if (key == '+')
112     glRotatef(3, 0, 1, 0);   // Ry, 3 degrees
113   else if (key == '-')
114     glRotatef(-3, 0, 1, 0);  // Ry, -3 degrees
115   else
116     return;
117
118   // The position of a light is affected by M, so...
119   if ( rotating_light ) // Transform by M
120     glLightfv(GL_LIGHT0, GL_POSITION, light_pos);
121   else {                // Stationary light
122     glPushMatrix();
123     glLoadIdentity();   // Do NOT multiply by M
124     glLightfv(GL_LIGHT0, GL_POSITION, light_pos);
125     glPopMatrix();
126   }
127
128   glutPostRedisplay(); // update image
129 }
130 void MenuHandler(int id) // id = menu alternative
131 {
132   if (id == 1)
133     rotating_light = 1;    // global variable
134   else if (id == 2)
135     rotating_light = 0;
136   else if (id == 3)
137     exit(0);               // Quit
138 }
```

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.

122-: 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(idle_callback)`,
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.

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 (colour) 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.

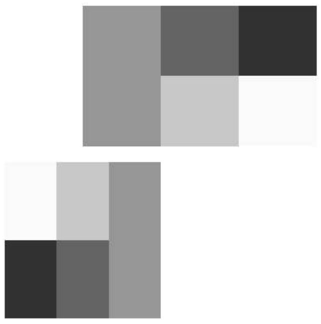
So the resulting red (ambient + diffuse) component, for example, in a pixel becomes $r_s \cdot r_t$, where r_s is the red component originating from the ordinary shading computation and r_t is the red component from the texture.

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();
}
```

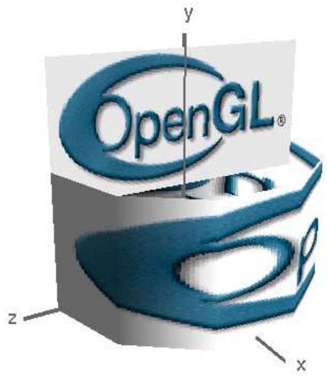
Here is part of the window (since I used grayscale in the images it is easy to interpret the result). The origin is in the lower left corner of the leftmost black rectangle.



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 2
255 255 255 255 255 255 255 255 255 255 255 255 255 255 2
```

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
```

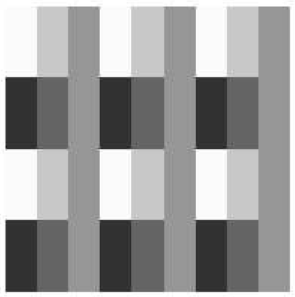
So the first quadrilateral (n = 1) is defined by vertices 1, 2, 4, 3 (2n-1, 2n, 2n+2, and 2n+1).

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.