3D.MP : 3-DIMENSIONAL REPRESENTATIONS AND ANIMATIONS IN METAPOST

version 1.0, 8 April 1998 **Denis Roegel** (`roegel@loria.fr`)

This package provides definitions enabling the manipulation and animation of 3-dimensional objects. Such objects can be included in a TEX file or used on web pages for instance. See the documentation enclosed in the distribution for more details.

Thanks to John Hobby and Ulrik Vieth for helpful hints.

LIMITATIONS:

− an object can not include TEX labels; to overcome this limitation would need to wrap the metapost output with dvips so that the necessary fonts are loaded (easy to do, but not done here)

PROJECTS FOR THE FUTURE:

− take light sources into account and show shadows and darker faces

− handle overlapping of objects (*obj_name* can be used when going through all faces)

if known *three_d_version*:
    expandafter endinput% avoids loading this package twice
fi;

message `"*** 3d, version 1.0 (c) D. Roegel, 8 April 1998 ***"`;
numeric *three_d_version*;
three_d_version = 1.0;

Vector operations

% definition of vector i by its coordinates
def *vect_def*(**expr** i, xi, yi, zi) = $vect[i]x := xi$; $vect[i]y := yi$; $vect[i]z := zi$; **enddef**;

% a point is stored as a vector
let *set_point* = *vect_def*;

% *set_obj_point* is like *set_point*, but the first parameter
% is a local point number
def *set_obj_point*(**expr** i, xi, yi, zi) = *set_point*(*pnt*(i), xi, yi, zi) **enddef**;

% definition of a vector by an other vector
def *vect_def_vect*(**expr** i, j) =
  $vect[i]x := vect[j]x$; $vect[i]y := vect[j]y$; $vect[i]z := vect[j]z$;
enddef;

% vector sum: $vect[k] \leftarrow vect[i]+vect[j]$
def *vect_sum*(**expr** k, i, j) =
  $vect[k]x := vect[i]x + vect[j]x$;
  $vect[k]y := vect[i]y + vect[j]y$;
  $vect[k]z := vect[i]z + vect[j]z$;
enddef;

% vector translation: $vect[i] \leftarrow vect[i]+vect[v]$
def *vect_translate*(**expr** i, v) = *vect_sum*(i, i, v) **enddef**;

% vector difference: $vect[k] \leftarrow vect[i]-vect[j]$
def *vect_diff*(**expr** k, i, j) =
  $vect[k]x := vect[i]x - vect[j]x$;
  $vect[k]y := vect[i]y - vect[j]y$;
  $vect[k]z := vect[i]z - vect[j]z$;
enddef;

% dot product of $vect[i]$ and $vect[j]$
vardef *vect_dprod*(**expr** i, j) =
  $(vect[i]x * vect[j]x + vect[i]y * vect[j]y + vect[i]z * vect[j]z)$
enddef;

% modulus of $vect[i]$
vardef *vect_mod*(**expr** i) = sqrt(*vect_dprod*(i, i)) **enddef**;

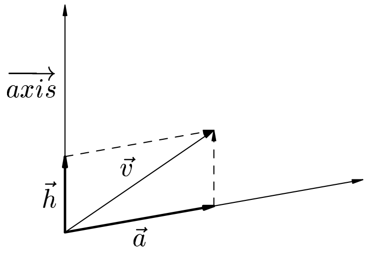
% vector product: $vect[k] \leftarrow vect[i] \wedge vect[j]$
def *vect_prod*(**expr** k, i, j) =
  $vect[k]x := vect[i]y * vect[j]z - vect[i]z * vect[j]y$;
  $vect[k]y := vect[i]z * vect[j]x - vect[i]x * vect[j]z$;
  $vect[k]z := vect[i]x * vect[j]y - vect[i]y * vect[j]x$;
enddef;

% scalar multiplication: $vect[j] \leftarrow vect[i] * v$
def *vect_mult*(**expr** j, i, v) =
  $vect[j]x := v * vect[i]x$; $vect[j]y := v * vect[i]y$; $vect[j]z := v * vect[i]z$;
enddef;

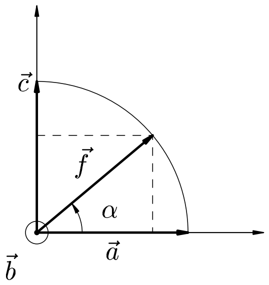
% middle of two points
def *mid_point*(**expr** k, i, j) = *vect_sum*(k, i, j); *vect_mult*(k, k, .5); **enddef**;

% Rotation of *vect*[*v*] around *vect*[*axis*] by an angle *alpha*

The vector $\vec{v}$ is first projected on the axis giving vectors $\vec{a}$ and $\vec{h}$:



If we set $\vec{b} = \frac{\overrightarrow{axis}}{\|\overrightarrow{axis}\|}$, the rotated vector $\vec{v'}$ is equal to $\vec{h} + \vec{f}$ where $\vec{f} = \cos\alpha \cdot \vec{a} + \sin\alpha \cdot \vec{c}$. and $\vec{h} = (\vec{v} \cdot \vec{b})\vec{b}$

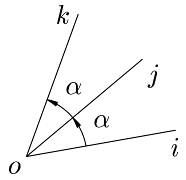



```
% The rotation is independent of vect[axis]'s module.
% v = old and new vector
% axis = rotation axis
% alpha = rotation angle
%
vardef vect_rotate(expr v, axis, alpha) =
    save v_a, v_b, v_c, v_d, v_e, v_f;
    v_a := new_vect;  v_b := new_vect;  v_c := new_vect;
    v_d := new_vect;  v_e := new_vect;  v_f := new_vect;
    v_g := new_vect;  v_h := new_vect;
    vect_mult(v_b, axis, 1/vect_mod(axis));
    vect_mult(v_h, v_b, vect_dprod(v_b, v)); % projection of v on axis
    vect_diff(v_a, v, v_h);
    vect_prod(v_c, v_b, v_a);
    vect_mult(v_d, v_a, cosd(alpha));
    vect_mult(v_e, v_c, sind(alpha));
    vect_sum(v_f, v_d, v_e);
    vect_sum(v, v_f, v_h);
    free_vect(v_h);  free_vect(v_g);
    free_vect(v_f);  free_vect(v_e);  free_vect(v_d);
    free_vect(v_c);  free_vect(v_b);  free_vect(v_a);
enddef;
```

```
% iname is the handler for an instance of an object of class name
% iname must be a letter string
vardef assign_obj(expr iname, name) =
    save tmpdef;
    string tmpdef; % we need to add double quotes (char 34)
    tmpdef = "def " & iname & "_class=" & ditto & name & ditto & " enddef";
    scantokens tmpdef;
    def_obj(iname);
enddef;

% name is the the name of an object instance
% It must be made only of letters (or underscores), but no digits.
def def_obj(expr name) =
    scantokens begingroup
        save tmpdef; string tmpdef;
        tmpdef = "def_" & obj_class_(name) & "(" & ditto & name & ditto & ")";
        tmpdef
    endgroup
enddef;

% This macro puts an object back where it was right at the beginning.
% iname is the name of an object instance.
vardef reset_obj(expr iname) =
    save tmpdef;
    string tmpdef;
    define_current_point_offset_(iname);
    tmpdef = "set_" & obj_class_(iname) & "_points";
    scantokens tmpdef(iname);
enddef;

% Put an object at position given by pos (a vector) and
% with orientations given by angles psi, theta, phi.
% The object is scaled by scale.
% iname is the name of an object instance.
vardef put_obj(expr iname, pos, scale, psi, theta, phi) =
    reset_obj(iname); scale_obj(iname, scale);
    v_x := new_vect; v_y := new_vect; v_z := new_vect;
    vect_def_vect(v_x, vect_I);
    vect_def_vect(v_y, vect_J);
    vect_def_vect(v_z, vect_K);
    rotate_obj_abs_pv(iname, point_null, v_z, psi);
    vect_rotate(v_x, v_z, psi); vect_rotate(v_y, v_z, psi);
    rotate_obj_abs_pv(iname, point_null, v_y, theta);
    vect_rotate(v_x, v_y, theta); vect_rotate(v_z, v_y, theta);
    rotate_obj_abs_pv(iname, point_null, v_x, phi);
    vect_rotate(v_y, v_x, phi); vect_rotate(v_z, v_x, phi);
    free_vect(v_z); free_vect(v_y); free_vect(v_x);
    translate_obj(iname, pos);
enddef;
```

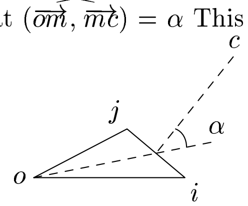
```
% Rotation of an object instance name around an axis
% going through a point p (local to the object)
% and directed by vector vect[v]. The angle of rotation is a.
vardef rotate_obj_pv(expr name, p, v, a) =
    define_current_point_offset_(name);
    rotate_obj_abs_pv(name, pnt(p), v, a);
enddef;

vardef rotate_obj_abs_pv(expr name, p, v, a) =
    save v_a;
    define_current_point_offset_(name);
    v_a := new_vect;
    for i := 1 upto obj_points_(name):
        vect_diff(v_a, pnt(i), p);
        vect_rotate(v_a, v, a);
        vect_sum(pnt(i), v_a, p);
    endfor;
    free_vect(v_a);
enddef;

% Rotation of an object instance name around an axis
% going through a point p (local to the object)
% and directed by vector pq. The angle of rotation is a.
vardef rotate_obj_pp(expr name, p, q, a) =
    save v_a, axis;
    define_current_point_offset_(name);
    v_a := new_vect; axis := new_vect;
    vect_diff(axis, pnt(q), pnt(p));
    for i := 1 upto obj_points_(name):
        vect_diff(v_a, pnt(i), pnt(p));
        vect_rotate(v_a, axis, a);
        vect_sum(pnt(i), v_a, pnt(p));
    endfor;
    free_vect(axis); free_vect(v_a);
enddef;

% Translation of an object instance name by a vector vect[v].
vardef translate_obj(expr name, v) =
    define_current_point_offset_(name);
    for i := 1 upto obj_points_(name):
        vect_sum(pnt(i), pnt(i), v);
    endfor;
enddef;

% Scalar multiplication of an object instance name by a scalar v.
vardef scale_obj(expr name, v) =
    define_current_point_offset_(name);
    for i := 1 upto obj_points_(name):
        vect_mult(pnt(i), pnt(i), v);
    endfor;
enddef;
```

% Rotation in a plane: this is useful to define a regular polygon.
% k is a new point obtained from point j by rotation around o
% by a angle α equal to the angle from i to j.




```
vardef rotate_in_plane_(expr k, o, i, j) =
    save cosalpha, sinalpha, alpha, v_a, v_b, v_c;
    v_a := new_vect; v_b := new_vect; v_c := new_vect;
    vect_diff(v_a, i, o); vect_diff(v_b, j, o); vect_prod(v_c, v_a, v_b);
    cosalpha = vect_dprod(v_a, v_b)/vect_mod(v_a)/vect_mod(v_b);
    sinalpha = sqrt(1 − cosalpha ** 2);
    alpha = angle((cosalpha, sinalpha));
    vect_rotate(v_b, v_c, alpha);
    vect_sum(k, o, v_b);
    free_vect(v_c); free_vect(v_b); free_vect(v_a);
enddef;

vardef rotate_in_plane(expr k, o, i, j) =
    rotate_in_plane_(pnt(k), o, pnt(i), pnt(j))
enddef;
```

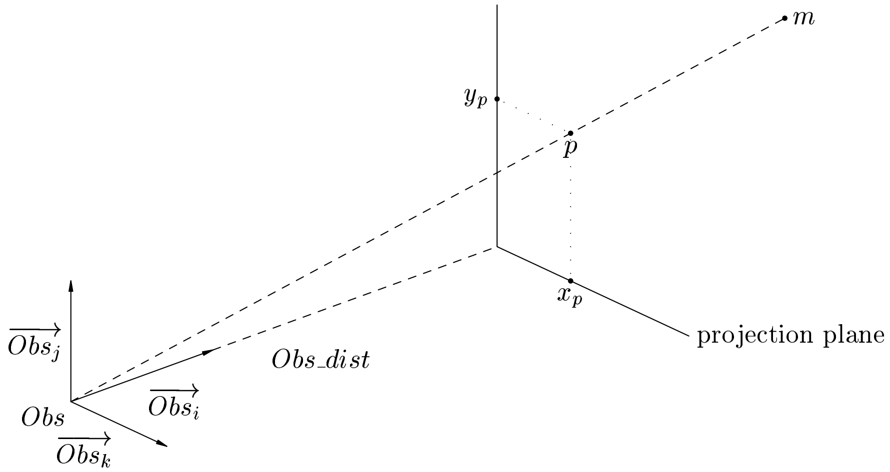
% Build a point on a adjacent face.
The middle m of points i and j is such that $(\widehat{\overrightarrow{om}, \overrightarrow{mc}}) = \alpha$ This is useful to define regular polyhedra




```
vardef new_face_point_(expr c, o, i, j, alpha) =
    save v_a, v_b, v_c, v_d, v_e;
    v_a := new_vect; v_b := new_vect; v_c := new_vect; v_d := new_vect; v_e := new_vect;
    vect_diff(v_a, i, o); vect_diff(v_b, j, o);
    vect_sum(v_c, v_a, v_b);
    vect_mult(v_d, v_c, .5);
    vect_diff(v_e, i, j);
    vect_sum(c, v_d, o);
    vect_rotate(v_d, v_e, alpha);
    vect_sum(c, v_d, c);
    free_vect(v_e); free_vect(v_d); free_vect(v_c); free_vect(v_b); free_vect(v_a);
enddef;

vardef new_face_point(expr c, o, i, j, alpha) =
    new_face_point_(pnt(c), pnt(o), pnt(i), pnt(j), alpha)
enddef;

vardef new_abs_face_point(expr c, o, i, j, alpha) =
    new_face_point_(c, o, pnt(i), pnt(j), alpha)
enddef;
```

Computation of the projection of a point on the “screen”

% p is the projection of m
% m = point in space (3 coordinates)
% p = point of the intersection plane



vardef *project_point*(**expr** p, m) =
    save *tmpalpha*, *v_a*, *v_b*;
    v_a := *new_vect*; *v_b* := *new_vect*;
    vect_diff (*v_b*, *m*, *Obs*); % vector *Obs-m*
        % *vect*[*v_a*] is *vect*[*v_b*] expressed in (*ObsI*,*ObsJ*,*ObsK*)
        % coordinates.
    $vect[v\_a]x := vect[IObsI\_]x * vect[v\_b]x$
    $+\ vect[IObsJ\_]x * vect[v\_b]y + vect[IObsK\_]x * vect[v\_b]z;$
    $vect[v\_a]y := vect[IObsI\_]y * vect[v\_b]x$
    $+\ vect[IObsJ\_]y * vect[v\_b]y + vect[IObsK\_]y * vect[v\_b]z;$
    $vect[v\_a]z := vect[IObsI\_]z * vect[v\_b]x$
    $+\ vect[IObsJ\_]z * vect[v\_b]y + vect[IObsK\_]z * vect[v\_b]z;$
    if $vect[v\_a]x < Obs\_dist$: % then, point m is too close
        message `"Point "` & decimal m & `" too close -> not drawn"`;
        $x[p] := too\_big\_;\ \ y[p] = too\_big\_;$
    else:
        if(angle($vect[v\_a]x$, $vect[v\_a]z$) $> h\_field/2$)
            or (angle($vect[v\_a]x$, $vect[v\_a]y$) $> v\_field/2$):
            message `"Point "` & decimal m & `" out of screen -> not drawn"`;
            $x[p] := too\_big\_;\ \ y[p] = too\_big\_;$
        else:
            $tmpalpha := Obs\_dist/vect[v\_a]x;$
            $y[p] := drawing\_scale * tmpalpha * vect[v\_a]y;$
            $x[p] := drawing\_scale * tmpalpha * vect[v\_a]z;$
        fi;
    fi;
    free_vect(*v_b*); *free_vect*(*v_a*);
enddef;

% Object projection
% This is a mere iteration on *project_point*
def *project_obj*(**expr** *name*) =
    *define_current_point_offset_*(*name*);
    for i := 1 **upto** *obj_points_*(*name*): *project_point*(*ipnt_*(i), *pnt*(i)); **endfor**;
enddef;

Draw one face, hiding it if it is hidden

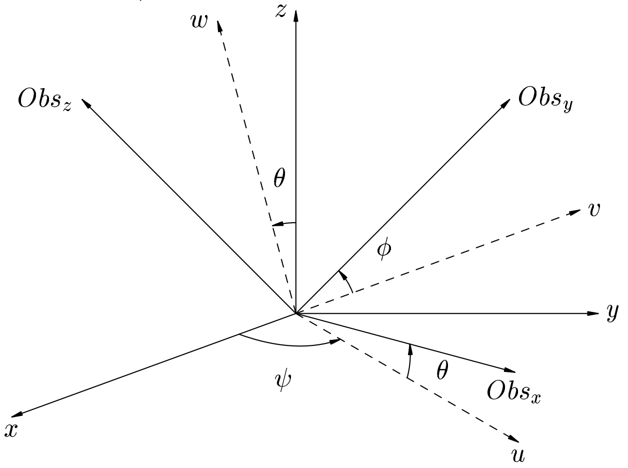
```
% The order of the vertices determines what is the visible side
% of the face. The order must be clockwise when the face is seen.
% drawhidden is a boolean; if true only hidden faces are drawn; if false,
% only visible faces are drawn. Therefore, draw_face is called twice
% by draw_faces.
vardef draw_face(text vertices)(expr col, drawhidden) =
    save p, num, normal_vect, v_a, v_b, v_c, overflow;
    path p; boolean overflow;
    overflow = false;
    forsuffixes $ = vertices:
        if z[ipnt_($)] = (too_big_, too_big_): overflow := true; fi;
        exitif overflow;
    endfor;
    if overflow: message "Face can not be drawn, due to overflow";
    else:
        p = forsuffixes $ = vertices: z[ipnt_($)] -- endfor cycle;
        num0 := 0; num1 := 0; num2 := 0;
            % get the three last suffixes:
        forsuffixes $ = vertices: num0 := num1; num1 := num2; num2 := $; endfor;
        normal_vect := new_vect; v_a := new_vect; v_b := new_vect;
        v_c := new_vect;
        vect_diff(v_a, pnt(num1), pnt(num0));
        vect_diff(v_b, pnt(num2), pnt(num1));
        vect_prod(normal_vect, v_a, v_b);
        vect_diff(v_c, pnt(num1), Obs);
        if filled_faces:
            if vect_dprod(normal_vect, v_c) < 0:
                fill p withcolor col; if draw_contours: drawcontour(p, contourwidth); fi;
            else: % draw p dashed evenly; if this is done, you must ensure
                            % that hidden faces are (re)drawn at the end
            fi;
        else:
            if vect_dprod(normal_vect, v_c) < 0: %visible
                if not drawhidden: draw p; fi;
            else: % hidden
                if drawhidden:
                    draw p withcolor background; % avoid strange overlapping dashes
                    draw p dashed evenly;
                fi;
            fi;
        fi;
        free_vect(v_c); free_vect(v_b); free_vect(v_a); free_vect(normal_vect);
    fi;
enddef;

% p is the path to draw (a face contour) and thickness is the pen width
def drawcontour(expr p, thickness) =
    pickup pencircle scaled thickness;
    draw p;
    pickup pencircle scaled .4pt;
enddef;
```

```
% Variables for face handling. First, we have an array for lists of vertices
% corresponding to faces.
string face_points_[]; % analogous to vect arrays

% Then, we have an array of colors. A color needs to be a string
% representing an hexadecimal RGB coding of a color.
string face_color_[];

% name is the name of an object instance
vardef draw_faces(expr name) =
    save tmpdef; string tmpdef;
    define_current_face_offset_(name);
        % first the hidden faces (dashes must be drawn first):
    for i := 1 upto obj_faces_(name):
        tmpdef := "draw_face(" & face_points_[face(i)]
            & ")(hexcolor(" & ditto & face_color_[face(i)] & ditto
            & "),true)"; scantokens tmpdef;
    endfor;
        % then, the visible faces:
    for i := 1 upto obj_faces_(name):
        tmpdef := "draw_face(" & face_points_[face(i)]
            & ")(hexcolor(" & ditto & face_color_[face(i)] & ditto
            & "),false)"; scantokens tmpdef;
    endfor;
enddef;

% Draw point n of object instance name
vardef draw_point(expr name, n) =
    define_current_point_offset_(name);
    project_point(ipnt_(n), pnt(n));
    if z[ipnt_(n)] ≠ (too_big_, too_big_):
        pickup pencircle scaled 5pt;
        drawdot(z[ipnt_(n)]);
        pickup pencircle scaled .4pt;
    fi;
enddef;

vardef draw_axes(expr r, g, b) =
    project_point(1, vect_null);
    project_point(2, vect_I);
    project_point(3, vect_J);
    project_point(4, vect_K);
    if(z1 ≠ (too_big_, too_big_)):
        if(z2 ≠ (too_big_, too_big_)):
            drawarrow z1 -- z2 dashed evenly withcolor r;
        fi;
        if(z3 ≠ (too_big_, too_big_)):
            drawarrow z1 -- z3 dashed evenly withcolor g;
        fi;
        if(z4 ≠ (too_big_, too_big_)):
            drawarrow z1 -- z4 dashed evenly withcolor b;
        fi;
    fi;
enddef;
```

```
% Definition of a macro obj_name returning an object name
% when given an absolute
% face number. This definition is built incrementally through a string,
% everytime a new object is defined.
% obj_name is defined by redefine_obj_name_.

% Initial definition
string index_to_name_;
index_to_name_ = "def obj_name(expr i)=if i<1:";

% name is the name of an object instance
% n is the absolute index of its last face
def redefine_obj_name_(expr name, n) =
  index_to_name_ := index_to_name_ & "elseif i<=" & decimal n & ":" & ditto
        & name & ditto;
  scantokens begingroup index_to_name_ & "fi;enddef;" endgroup;
enddef;

% i is an absolute face number
% vertices is a string representing a list of vertices
% rgbcolor is a string representing a color in rgb hexadecimal
def set_face(expr i, vertices, rgbcolor) =
  face_points_[i] := vertices; face_color_[i] := rgbcolor;
enddef;

% i is a local face number
%vertices is a string representing a list of vertices
%rgbcolor is a string representing a color in rgb hexadecimal
def set_obj_face(expr i, vertices, rgbcolor) = set_face(face(i), vertices, rgbcolor)
enddef;
```

Compute the vectors corresponding to the observer's viewpoint

% (vectors *ObsI_*,*ObsJ_* and *ObsK_* in the *vect_I*,*vect_J*,
% *vect_K* reference; and vectors *IObsI_*,*IObsJ_* and *IObsK_*
% which are *vect_I*,*vect_J*,*vect_K*
% in the *ObsI_*,*ObsJ_*,*ObsK_* reference)



(here, $\psi > 0$, $\theta < 0$ and $\phi > 0$; moreover, $|\theta| \leq 90°$)

def *compute_reference*(**expr** *psi*, *theta*, *phi*) =
  % *ObsI_* defines the direction of observation;
  % *ObsJ_* and *ObsK_* the orientation
  % (but one of these two vectors is enough,
  % since *ObsK_* = *ObsI_* ∧ *ObsJ_*)
  % The vectors are found by rotations of *vect_I*,*vect_J*,*vect_K*.
 vect_def_vect(*ObsI_*, *vect_I*); *vect_def_vect*(*ObsJ_*, *vect_J*);
 vect_def_vect(*ObsK_*, *vect_K*);
 vect_rotate(*ObsI_*, *ObsK_*, *psi*);
 vect_rotate(*ObsJ_*, *ObsK_*, *psi*); % gives (u,v,z)
 vect_rotate(*ObsI_*, *ObsJ_*, *theta*);
 vect_rotate(*ObsK_*, *ObsJ_*, *theta*); % gives (Obs_x,v,w)
 vect_rotate(*ObsJ_*, *ObsI_*, *phi*);
 vect_rotate(*ObsK_*, *ObsI_*, *phi*); % gives (Obs_x,Obs_y,Obs_z)
  % The passage matrix P from *vect_I*,*vect_J*,*vect_K*
  % to *ObsI_*,*ObsJ_*,*ObsK_* is the matrix
  % composed of the vectors *ObsI_*,*ObsJ_* and *ObsK_* expressed
  % in the base *vect_I*,*vect_J*,*vect_K*.
  % We have $X = PX'$ where X are the coordinates of a point
  % in *vect_I*,*vect_J*,*vect_K*
  % and X' the coordinates of the same point in *ObsI_*,*ObsJ_*,*ObsK_*.
  % In order to get P^{-1}, it suffices to build vectors using
  % the previous rotations in the inverse order.
 vect_def_vect(*IObsI_*, *vect_I*); *vect_def_vect*(*IObsJ_*, *vect_J*);
 vect_def_vect(*IObsK_*, *vect_K*);
 vect_rotate(*IObsK_*, *IObsI_*, −*phi*); *vect_rotate*(*IObsJ_*, *IObsI_*, −*phi*);
 vect_rotate(*IObsK_*, *IObsJ_*, −*theta*); *vect_rotate*(*IObsI_*, *IObsJ_*, −*theta*);
 vect_rotate(*IObsJ_*, *IObsK_*, −*psi*); *vect_rotate*(*IObsI_*, *IObsK_*, −*psi*);
enddef;

```
% This macro computes the three angles necessary for compute_reference
% name = name of an instance of an object
% target = target point (local to object name)
% phi = angle
vardef point_of_view_obj(expr name, target, phi) =
    define_current_point_offset_(name); % enables pnt
    point_of_view_abs(pnt(target), phi);
enddef;

% Compute absolute perspective. target is an absolute point number
vardef point_of_view_abs(expr target, phi) =
    save v_a, psi, theta;
    v_a := new_vect;
    vect_diff(v_a, target, Obs);
    vect_mult(v_a, v_a, 1/vect_mod(v_a));
    psi = angle((vect[v_a]x, vect[v_a]y));
    theta = - angle((vect[v_a]x ++ vect[v_a]y, vect[v_a]z));
    compute_reference(psi, theta, phi);
    free_vect(v_a);
enddef;

% Distance between the observer and point n of object name
% Result is put in dist
vardef obs_distance(text dist)(expr name, n) =
    save v_a;
    v_a := new_vect;
    define_current_point_offset_(name); % enables pnt
    dist := vect_mod(v_a, pnt(n), Obs);
    free_vect(v_a);
enddef;
```

Vector and point allocation

```
% Allocation is done through a stack of vectors
numeric last_vect_;
last_vect_ = 0;

% vector allocation
def new_vect = incr(last_vect_)
               % message "Vector " & decimal last_vect_ & " allocated";
enddef;

let new_point = new_vect;

def new_points(text p)(expr n) =
    save p;
    numeric p[];
    for i := 1 upto n: p[i] := new_point; endfor;
enddef;

% Free a vector
% A vector can only be freed safely when it was the last vector created.
def free_vect(expr i) =
    if i = last_vect_: last_vect_ := last_vect_ - 1;
    else: errmessage("Vector " & decimal i & " can't be freed!");
    fi;
enddef;

let free_point = free_vect;

def free_points(text p)(expr n) =
    for i := 10 step -1 until 1: free_point(p[i]); endfor;
enddef;
```

Debugging

```
def show_vect(expr t, i) =
    message "Vector " & t & "="
     & "(" & decimal vect[i]x & "," & decimal vect[i]y & ","
         & decimal vect[i]z & ")";
enddef;

let show_point = show_vect;

def show_pair(expr t, zz) =
    message t & "=(" & decimal xpart(zz) & "," & decimal ypart(zz) & ")";
enddef;
```

Access to object features

% a must be a string representing an class name, such as "dodecahedron".
% b is the tail of a macro name.

def *obj_*(**expr** a, b, i) =
    scantokens
    begingroup save n; **string** n; $n = a \mathbin{\&} b \mathbin{\&} i$; n
    endgroup
enddef;

def *obj_points_*(**expr** *name*) =
    *obj_*(*obj_class_*(*name*), "_points", *name*)
enddef;

def *obj_faces_*(**expr** *name*) =
    *obj_*(*obj_class_*(*name*), "_faces", *name*)
enddef;

vardef *obj_point_offset_*(**expr** *name*) =
    *obj_*(*obj_class_*(*name*), "_point_offset", *name*)
enddef;

vardef *obj_face_offset_*(**expr** *name*) =
    *obj_*(*obj_class_*(*name*), "_face_offset", *name*)
enddef;

def *obj_class_*(**expr** *name*) = *obj_*(*name*, "_class", "") **enddef**;

```
def define_point_offset_(expr name, o) =
    begingroup save n, tmpdef;
        string n, tmpdef;
        n = obj_class_(name) & "_point_offset" & name;
        expandafter numeric scantokens n;
        scantokens n := last_point_offset_;
        last_point_offset_ := last_point_offset_ + o;
        tmpdef = "def " & obj_class_(name) & "_points" & name&
            "=" & decimal o & " enddef";
        scantokens tmpdef;
    endgroup
enddef;

def define_face_offset_(expr name, o) =
    begingroup save n, tmpdef;
        string n, tmpdef;
        n = obj_class_(name) & "_face_offset" & name;
        expandafter numeric scantokens n;
        scantokens n := last_face_offset_;
        last_face_offset_ := last_face_offset_ + o;
        tmpdef = "def " & obj_class_(name) & "_faces" & name&
            "=" & decimal o & " enddef";
        scantokens tmpdef;
    endgroup
enddef;

def define_current_point_offset_(expr name) =
    save current_point_offset_;
    numeric current_point_offset_;
    current_point_offset_ := obj_point_offset_(name);
enddef;

def define_current_face_offset_(expr name) =
    save current_face_offset_;
    numeric current_face_offset_;
    current_face_offset_ := obj_face_offset_(name);
enddef;
```

Drawing an object

% *name* is an object instance
def *draw_obj*(**expr** *name*) = *project_obj*(*name*); *draw_faces*(*name*); **enddef**;

Normalization of an object

% This macro translates an object so that a list of vertices is centered
% on the origin, and the last vertex is put on a sphere whose radius is 1.
% *name* is the name of the object and *vertices* is a list
% of points whose barycenter will define the center of the object.
% (*vertices* need not be the list of all vertices)
vardef *normalize_obj*(**expr** *name*)(**text** *vertices*) =
    save *v_a*, *nvertices*, *last*;
    numeric *v_a*, *last*;
    nvertices = 0;
    v_a = *new_vect*; *vect_def*(*v_a*, 0, 0, 0)
    forsuffixes $ = *vertices*:
        vect_sum(*v_a*, *v_a*, *pnt*($));
        nvertices := *nvertices* + 1;
        last := $;
    endfor;
    vect_mult(*v_a*, *v_a*, −1/*nvertices*);
    translate_obj(*name*, *v_a*); % object centered on the origin
    scale_obj(*name*, 1/*vect_mod*(*pnt*(*last*)));
    free_vect(*v_a*);
enddef;

General definitions

```
% Vector arrays
numeric vect[]x, vect[]y, vect[]z;

% Observer
numeric Obs;
Obs = new_point;
% default value:
set_point(Obs, 0, 0, 20);

% Observer's vectors
ObsI_ = new_vect; ObsJ_ = new_vect; ObsK_ = new_vect;
IObsI_ = new_vect; IObsJ_ = new_vect; IObsK_ = new_vect;

% distance observer/plane (must be > 0)
numeric Obs_dist; % represents Obs_dist × drawing_scale
% default value:
Obs_dist = 2; % means Obs_dist × drawing_scale

% Screen Size
% The screen size is defined through two angles: the horizontal field
% and the vertical field
numeric h_field, v_field;
h_field = 100; % degrees
v_field = 70; % degrees

% Reference vectors 0⃗, i⃗, j⃗ and k⃗
% and their definition
numeric vect_null, vect_I, vect_J, vect_K;
vect_null = new_vect; vect_I = new_vect; vect_J = new_vect; vect_K = new_vect;
vect_def(vect_null, 0, 0, 0);
vect_def(vect_I, 1, 0, 0); vect_def(vect_J, 0, 1, 0); vect_def(vect_K, 0, 0, 1);
numeric point_null;
point_null = vect_null;

% Observer's orientation, defined by three angles
numeric Obs_psi, Obs_theta, Obs_phi;
% default value:
Obs_psi = 0; Obs_theta = 90; Obs_phi = 0;

% Points for the figures
numeric points_[];

% name is the name of an object instance
% npoints is its number of defining points
def new_obj_points(expr name, npoints) =
    define_point_offset_(name, npoints); define_current_point_offset_(name);
    for i := 1 upto obj_points_(name): pnt(i) := new_point; endfor;
enddef;

% name is the name of an object instance
% nfaces is its number of defining faces
def new_obj_faces(expr name, nfaces) =
    define_face_offset_(name, nfaces); define_current_face_offset_(name);
    redefine_obj_name_(name, current_face_offset_ + nfaces);
enddef;
```

```
% Absolute point number corresponding to object point number i
% This macro must only be used within the function defining an object
% (such as def_cube) or the function drawing an object (such as
% draw_cube).
def ipnt_(expr i) = i + current_point_offset_ enddef;
def pnt(expr i) = points_[ipnt_(i)] enddef;

def face(expr i) = (i + current_face_offset_) enddef;

% Absolute point number corresponding to local point n
% in object instance name
vardef pnt_obj(expr name, n) =
    hide(define_current_point_offset_(name);  )pnt(n)
enddef;

% Absolute face number corresponding to local face n
% in object instance name
vardef face_obj(expr name, n) =
    hide(define_current_face_offset_(name);  )face(n)
enddef;


% Scale
numeric drawing_scale;
drawing_scale = 2cm;

% Color
% This function is useful when a color is expressed in hexadecimal.
def hexcolor(expr s) =
    (hex(substring(0, 2) of s)/255, hex(substring(2, 4) of s)/255,
        hex(substring(4, 6) of s)/255)
enddef;

% Filling and contours
boolean filled_faces, draw_contours;
filled_faces = true;
draw_contours = true;
numeric contourwidth; % thickness of contours
contourwidth = 1pt;

% Overflow control
% An overflow can occur when an object is too close from the observer
% or if an object is out of sight. We use a special value to mark
% coordinates which would lead to an overflow.
numeric too_big_;
too_big_ = 4000;


% Object offset (the points defining an object are arranged
% in a single array, and the objects are easier to manipulate
% if the point numbers are divided into a number and an offset).
numeric last_point_offset_, last_face_offset_;
last_point_offset_ = 0;  last_face_offset_ = 0;
```

Computation of field parameters of an animation

```
numeric xmin_, ymin_, xmax_, ymax_;

def compute_bbox =
    if known xmin_:
        xmin_ := min(xmin_, xpart(llcorner(currentpicture)));
        ymin_ := min(ymin_, ypart(llcorner(currentpicture)));
        xmax_ := max(xmax_, xpart(urcorner(currentpicture)));
        ymax_ := max(ymax_, ypart(urcorner(currentpicture)));
    else:
        xmin_ = xpart(llcorner(currentpicture));
        ymin_ = ypart(llcorner(currentpicture));
        xmax_ = xpart(urcorner(currentpicture));
        ymax_ = ypart(urcorner(currentpicture));
    fi;
enddef;

boolean show_animation_parameters;
show_animation_parameters = false;
numeric paper_height;
paper_height = 29.7; % paper height in cm

% show bounding box of an animation, in PostScript points
% and parameters for animation script
vardef show_animation_bbox =
    save trx, try, h, w, delta, pnmx, pnmy, pnmw, pnmh, res;
    res = 36; % 36 dots per inch in bitmap
    w = xmax_ - xmin_;  h = ymax_ - ymin_;
    if show_animation_parameters:
        message "animation bbox: (llx=" & decimal round(xmin_)
            & ",lly=" & decimal round(ymin_)
            & ",w=" & decimal round(w) & ",h=" & decimal round(h) & ")";
    fi;
    if xmin_ <= 20: trx = 50 - xmin_;  else: trx = 0;  fi;
    if ymin_ <= 20: try = 50 - ymin_;  else: try = 0;  fi;
    if show_animation_parameters:
        message "translate parameters: "
         & decimal round(trx) & " " & decimal round(try);
    fi;
    xmin_ := xmin_ + trx;  ymin_ := ymin_ + try;
    delta = 10;
    pnmx = round(xmin_ * (res/72) - delta);
    pnmy = round((paper_height/2.54 * 72 - ymin_ - h) * (res/72) - delta);
    pnmw = round(w * (res/72) + 2 * delta);
    pnmh = round(h * (res/72) + 2 * delta);
    if show_animation_parameters:
        message "pnmcut parameters (with -r36): "
         & decimal pnmx & " " & decimal pnmy & " "
         & decimal pnmw & " " & decimal pnmh;
    fi;
    write_script(round(trx), round(try),
        pnmx, pnmy, pnmw, pnmh, res, jobname, "create_animation.sh");
enddef;
```

Creation of a shell script to automate the animation

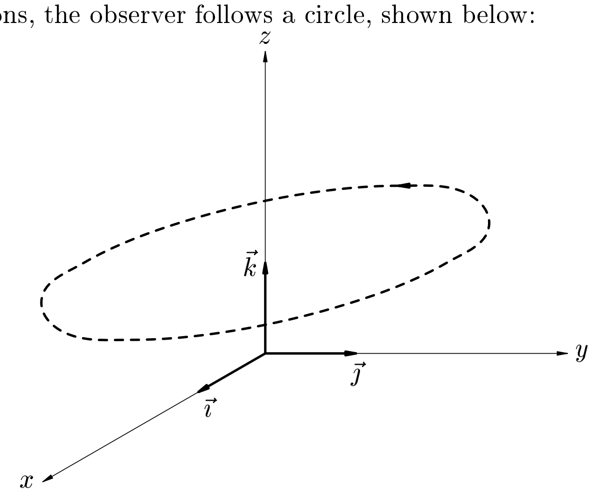
```
% This is UNIX targetted and may need to be customized.

vardef write_script(expr trx, try, xmin, ymin, w, h, res, output, file) =
    save s;
    string s;
    def write_to_file(text arg) = write arg to file; enddef;
    write_to_file("#! /bin/sh");
    write_to_file("");
    write_to_file("/bin/rm -f " & output & ".log");
    write_to_file("for i in `ls " & output & ".*| grep '" & output & ".[0-9]'`;do");
        if false: "endfor" fi% indentation hack for meta-mode.el
    write_to_file("echo $i");
    write_to_file("echo '============='");
    s := "awk < $i '{print} /^%%Page: /{print " & ditto;
    s := s & decimal trx & " " & decimal try & " translate\n" & ditto & "}' > $i.ps";
    write_to_file(s);
        % ghostscript PostScript into ppm
    s := "gs -sDEVICE=ppmraw -sPAPERSIZE=a4 -dNOPAUSE ";
    s := s & "-r" & decimal res & " -sOutputFile=$i.ppm -q -- $i.ps";
    write_to_file(s);
    write_to_file("/bin/rm -f $i.ps");
        % possible alternative:
        % s := "mogrify -compress -crop " & decimal(w) & "x" & decimal(h);
        % s := s & "+" & decimal(xmin) & "+" & decimal(ymin);
        % s := s & " -colors 32 -format gif $i.ppm";
    s := "ppmquant 32 $i.ppm | pnmcut " & decimal(xmin) & " " & decimal(ymin);
    s := s & " " & decimal(w) & " " & decimal(h) & " | ";
    s := s & "ppmtogif > `expr $i.ppm : '\(.*\)ppm'`gif";
    write_to_file(s);
    write_to_file("/bin/rm -f $i.ppm");
    write_to_file("done");
    write_to_file("/bin/rm -f " & output & ".gif");
    s := "gifmerge -10 -l1000 ";
    s := s & output & ".*.gif > " & output & ".gif";
    write_to_file(s);
    write_to_file("/bin/rm -f " & output & ".*.gif");
    write_to_file(EOF); % end of file
enddef;
```

Standard animation definitions

% These definitions produce *one* image of some kind.

extra_endfig := "compute_bbox";

% In the standard animations, the observer follows a circle, shown below:




```
% Standard image 1: this is an example and may be adapted.
% name is an object instance
def one_image(expr name, i, a, rd, ang) =
    beginfig(i);
        set_point(Obs, −rd ∗ cosd(a ∗ ang), −rd ∗ sind(a ∗ ang), 1);
        Obs_phi := 90; Obs_dist := 2;
        point_of_view_obj(name, 1, Obs_phi); % fix point 1 of object name
        draw_obj(name);
        rotate_obj_pv(name, 1, vect_I, ang);
        draw_point(name, 1); % show the rotation point
        draw_axes(red, green, blue);
    endfig;
enddef;
```

```
% Standard image 2: this is an example and may be adapted.
% name_a and name_b are object instances.
def one_image_two_objects(expr name_a, name_b, i, a, rd, ang) =
    beginfig(i);
        set_point(Obs, −rd ∗ cosd(a ∗ ang), −rd ∗ sind(a ∗ ang), 1);
        Obs_phi := 90; Obs_dist := 2;
        point_of_view_obj(name_a, 1, Obs_phi); % fix point 1 of object name_a
        draw_obj(name_a); draw_obj(name_b);
        rotate_obj_pv(name_a, 1, vect_I, ang);
        rotate_obj_pv(name_b, 13, vect_J, −ang);
            %rotate_obj_pp(name_b, 13, 7, −ang);
        draw_point(name_a, 1); % show the rotation point
        draw_axes(red, green, blue);
    endfig;
enddef;
```

```
% Standard image 3: this is an example and may be adapted.
% name_a, name_b and name_c are object instances.
def one_image_three_objects(expr name_a, name_b, name_c, i, a, rd, ang) =
    beginfig(i);
        set_point(Obs, -rd * cosd(a * ang), -rd * sind(a * ang), 1);
        Obs_phi := 90;  Obs_dist := 2;  h_field := 100;  v_field := 150;
        point_of_view_obj(name_a, 1, Obs_phi); % fix point 1 of object name_a
        draw_obj(name_a);  draw_obj(name_b);  draw_obj(name_c);
        v_a := new_vect;
        vect_def(v_a, .03 * cosd(-a * ang + 90), .03 * sind(-a * ang + 90), 0);
        translate_obj(name_c, v_a);
        free_vect(v_a);
        rotate_obj_pv(name_a, 1, vect_I, ang);
        rotate_obj_pv(name_b, 13, vect_J, -ang);
            %rotate_obj_pp(name_b, 13, 7, -ang);
        draw_point(name_a, 1); % show the rotation point
        draw_axes(red, green, blue);
    endfig;
enddef;

% Standard image 4: this is an example and may be adapted.
% name_a and name_b are object instances.
def one_image_two_identical_objects(expr name_a, name_b, i, a, rd, ang) =
    beginfig(i);
        set_point(Obs, -rd * cosd(a * ang), -rd * sind(a * ang), 2);
        Obs_phi := 90;  Obs_dist := 2;
        point_of_view_obj(name_a, 1, Obs_phi); % fix point 1 of object name_a
        draw_obj(name_a);  draw_obj(name_b);
        rotate_obj_pv(name_a, 1, vect_I, ang);
        rotate_obj_pv(name_b, 13, vect_J, -ang);
            %rotate_obj_pp(name_a, 13, 7, -ang);
        draw_point(name_a, 1); % show the rotation point
        draw_axes(red, green, blue);
    endfig;
enddef;
```

```
% An animation is a series of images, and these series are produced here.

% Standard animation 1
% name is a class name
def animate_object(expr name, imin, imax, index) =
    numeric ang; ang = 360/(imax − imin + 1);
    assign_obj("obj", name);
    for i := imin upto imax: one_image("obj", i + index, i, 5, ang); endfor;
    show_animation_bbox;
enddef;

% Standard animation 2
% name_a and name_b are class names
def animate_two_objects(expr name_a, name_b, imin, imax, index) =
    numeric ang; ang = 360/(imax − imin + 1);
    assign_obj("obja", name_a); assign_obj("objb", name_b);
    translate_obj("objb", vect_K); translate_obj("objb", vect_K);
    for i := imin upto imax:
        one_image_two_objects("obja", "objb", i + index, i, 5, ang);
    endfor;
    show_animation_bbox;
enddef;

% Standard animation 3
% name_a, name_b and name_c are class names
vardef animate_three_objects(expr name_a, name_b, name_c, imin, imax, index) =
    numeric ang; ang = 360/(imax − imin + 1);
    assign_obj("obja", name_a); assign_obj("objb", name_b);
    assign_obj("objc", name_c);
    scale_obj("objb", .7);
    numeric v_a; v_a := new_vect;
    vect_def_vect(v_a, vect_K); vect_mult(v_a, v_a, 4); put_obj("objb", v_a, 1, 0, 0, 0);
    free_vect(v_a);
    scale_obj("objc", .5);
    translate_obj("objc", vect_K); translate_obj("objc", vect_K);
    for i := imin upto imax:
        one_image_three_objects("obja", "objb", "objc", i + index, i, 7, ang);
    endfor;
    show_animation_bbox;
enddef;

% Standard animation 4
% name is a class name
def animate_two_identical_objects(expr name, imin, imax, index) =
    numeric ang; ang = 360/(imax − imin + 1);
    assign_obj("obja", name); assign_obj("objb", name);
    translate_obj("objb", vect_K); translate_obj("objb", vect_K);
    for i := imin upto imax:
        one_image_two_identical_objects("obja", "objb", i + index, i, 10, ang);
    endfor;
    show_animation_bbox;
enddef;

endinput
```