Graphics

Space geometry with METAPOST*

Denis Roegel

Abstract

METAPOST is a tool especially well-suited for the inclusion of technical drawings in a document. In this article, we show how METAPOST can be used to represent objects in space and especially how it can be used for drawing geometric constructions involving lines, planes, as well as their intersections, orthogonal planes, etc. All the features belong to a new METAPOST package aimed at all those who teach and study geometry.

This article is dedicated to Donald Knuth whose PhD dissertation was on projective geometry.

1 Introduction

METAPOST (Hobby, 1992; Goossens, Rahtz, and Mittelbach, 1997; Hoenig, 1998; Hagen, 2002) is a graphical description language created by John Hobby from the METAFONT system (Knuth, 1986).

* Translated from "La géométrie dans l'espace avec METAPOST," *Cahiers GUTenberg* **39–40**, May 2001, pages 107-138, with permission.

A two-dimensional drawing is represented as a program which is compiled into a PostScript file. A drawing can be described in a very precise and compact fashion by taking advantage of the declarative nature of the language. For instance, linear constraints between the coordinates of several points, such as those of a central symmetry, are expressed very naturally by equations. Furthermore, it is possible to manipulate equations involving values that are not completely known. For instance, in order to express that p_3 is the middle of $[p_1, p_2]$, it suffices to write: `p3-p1=p2-p3`, or `p3=.5[p1,p2]`.

When this equation is given, some and possibly all of the coordinates of the three points may be unknown. Taking the equation into account represents the addition of a constraint. Constraints are added until the values involved are precisely known. In the previous example, the three points can be completely determined by positioning p_1 and p_2. A value can remain indetermined as long as it is not involved in a drawing. Finally, METAPOST alerts the user if there are redundant or inconsistent equations.

2 A first example in plane geometry

In order to get a good understanding of how METAPOST can naturally express a geometric problem, let us study the representation of a triangle property, such as the existence of the *nine points circle* (first stated by Poncelet and Brianchon in 1821). Figure 1 shows the result produced by METAPOST. This example will also serve as an introduction to METAPOST for the reader discovering the language here.

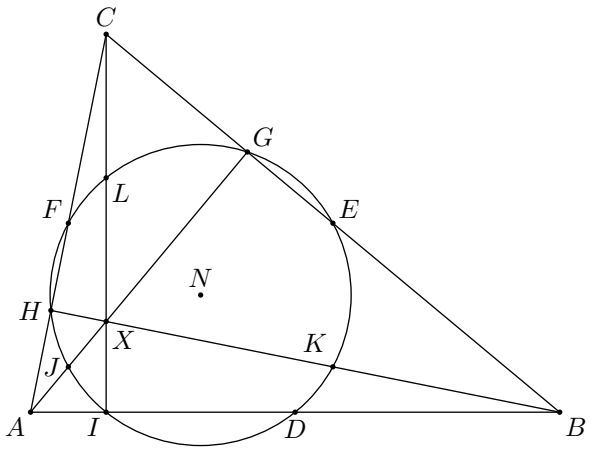


Figure 1: The nine points circle.

In this figure, we have first defined the points, then we set the three vertices of the triangle as functions of the origin (`origin`) and an arbitrary unit u enabling us to easily change the size of the graphics later on:[1]

```
numeric u; u=1cm;
pair A,B,C,D,E,F,G,H,I,J,K,L,N,X;
A=origin; B-A=(7u,0); C-A=(u,5u);
```

Then, the middles D, E and F of the triangle's sides are determined by equations:

```
D=.5[A,B]; E=.5[B,C]; F=.5[A,C];
```

G, H and I are the feet of the triangle's heights. They can be obtained easily by computing the intersection of a side with a segment starting at the opposite vertex and directed toward a direction at right angle with the opposite side. The `whatever` definition is especially useful in this case, since it represents an anonymous unknown (so that several occurrences of `whatever` do not represent the same unknown!). We have thus:

```
G=whatever[B,C]
 =whatever[A,A+((C-B) rotated 90)];
```

This means that G is somewhere on (BC) and also somewhere on (AP), where P is a point on the height. METAPOST gives a value to *both* unknowns in order to fulfill this equation. Similarly,

```
H=whatever[A,C]
 =whatever[B,B+((C-A) rotated 90)];
I=whatever[A,B]
 =whatever[C,C+((B-A) rotated 90)];
```

The orthocenter (intersection of the heights) is obtained with `intersectionpoint`. The `A--G` construction represents the $[AG]$ segment:

```
X=(A--G) intersectionpoint (C--I);
```

The middles J, K and L of $[AX]$, $[BX]$ and $[CX]$ are obtained as were D, E and F previously:

```
J=.5[A,X]; K=.5[B,X]; L=.5[C,X];
```

Finally, in order to find the center N of the nine points circle (assuming its existence), it is sufficient to compute the intersection of two perpendicular bisectors, for instance those of $[ID]$ and $[DH]$:

```
N=whatever[.5[D,I],
           (.5[D,I]+((D-I) rotated 90))]
 =whatever[.5[D,H],
           (.5[D,H]+((D-H) rotated 90))];
```

The circle's radius is found with:

```
r=arclength(I--N);
```

The triangle as well as the heights and the circle (centered on N and of diameter $2r$) are drawn with:

```
draw A--B--C--cycle;
draw A--G; draw B--H; draw C--I;
draw fullcircle scaled 2r shifted N;
```

The points are marked with `drawdot` after the line width has been increased. Finally, the annotations are all obtained on the model of:

[1] For the points, we could also have used `z0`, `z1`, etc., which are predefined variables.

```
label.top(btex $C$ etex,C);
```

The `label` instruction allows for the inclusion of TEX labels.

This example reveals the natural expression of geometric constraints for problems in plane geometry. All the constructions we have used are absolutely standard in METAPOST. Of course, if we had many such figures, we would introduce functions for the computation of the heights, the perpendicular bisectors, etc.

3 METAPOST extensions

METAPOST is an extensible system. At the basis, it is a program which loads an initial set of macros. It is then possible to add new domain-specific definitions. For instance, when we worked on the plane representation of objects in space, we developed a `3d` package, initially in order to manipulate polyhedra (Roegel, 1997). We have recently developed other extensions resting on the `3d` package. We view these extensions as "modules" of the `3d` package. In particular, we wanted to manipulate objects other than polyhedra, such as curves defined by equations, or given by a sequence of points. Among the extensions created, we have created a module providing various functionalities adapted to space geometry. This module, introduced here, is the `3dgeom` module[2] (see figure 2).

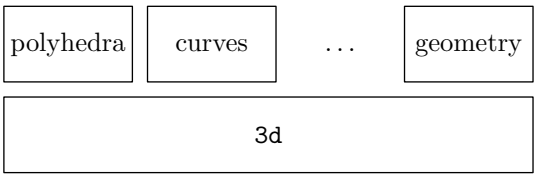


Figure 2: Structure of the `3d` *package* and of modules.

Some of the modules automatically load other modules. The `3dgeom` module loads for instance `3d`. A module is loaded only once.

A program using `3dgeom` will therefore start with `input 3dgeom`.

4 Space geometry

4.1 A simple example

We will start by representing an elementary object, the cube (figure 3). For that, we will give the coordinates of its eight vertices.

The `3d` package defines a concept of point or vector as a triple of numerical values. The points

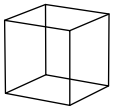


Figure 3: A cube shown in linear perspective.

must be defined by an allocation mechanism and must be freed when they are no longer used. The allocation of a point (resp. a vector) is done with `new_point` (resp. `new_vec`). This is a macro taking a point (resp. vector) name and allocating a memory area to store it. Freeing a point or a vector is done with `free_point` or `free_vec`, by giving the point or vector identification as a parameter. Hence, a program that wishes to use a vector `v` will look like this:

```
new_vec(v);
...
free_vec(v);
```

The set of all points and vectors is stored internally in a stack. Allocating or freeing a vector merely changes the stack pointer. As a consequence, points and vectors must be freed in the reverse order of their allocation. If that order is not respected, an unallocation error is raised.

```
new_point(pa); new_point(pb);
...
free_point(pb); free_point(pa);
```

It is not compulsory to free vectors or points, but not doing so will often have dramatic consequences if the allocations are within loops.

In order to ease the manipulation of sets of points, arrays can be allocated with `new_points` and freed with `free_points`. An array defined in that way has a name and a number of elements n. The elements are numbered from 1 to n. In our example, in order to create a cube, we declare a `vertex` array of eight points:

```
new_points(vertex)(8);
...
free_points(vertex)(8);
```

Each vertex is declared with the `set_point_` command, for instance:

```
set_point_(vertex1)(0,0,0);
```

By default, the perspective is linear (or central) and we have a camera witnessing the scene (figure 4). The camera must also be set. It corresponds to the predefined point *Obs*. Its position can be defined in space with `set_point_`. Moving the camera is done by expressing its coordinates in a parametric way, for instance:[3]

[2] This module is available on CTAN under `graphics/metapost/macros/3d`.

[3] In the code, `cosd` and `sind` represent the trigonometric functions with arguments in degrees.

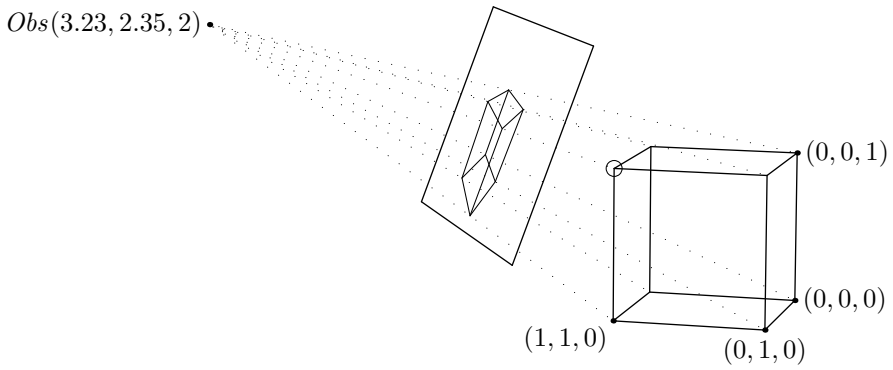


Figure 4: A cube and its projection on the screen. The focused point is circled.

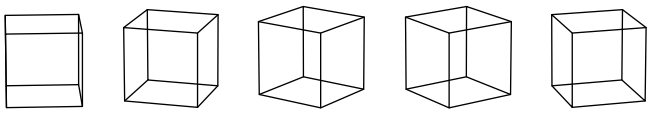


Figure 5: Five views of the cube "animation."

```
set_point_(Obs)(20*cosd(3.6*i),20*sind(3.6*i),6);
```

By varying i, the camera goes through points of the circle $\mathcal{C}(t) = (20\cos(3.6t), 20\sin(3.6t), 6)$. Figure 5 shows five views of that sequence, for $i = 0, 5, 10, 15$ and 20. An animation can be obtained by creating a sufficient amount of close views and by transforming the METAPOST outputs into GIF files. The procedure is explained in detail in our first article (Roegel, 1997).

Besides the camera position, it is necessary to define its orientation. It can be specified by three angles, but also in a simpler way by indicating a point towards which the camera is oriented and an angle. In order for the camera to be constantly focused on vertex 8, with the angle 90 (this angle corresponds to the degree of freedom of a rotation around the direction of view), it suffices to write:

```
Obs_phi:=90;
point_of_view_abs(vertex8,Obs_phi);
```

The ":=" assignment is used because it makes it possible to change the value of a variable when this variable already has a value. Writing `Obs_phi=90` can produce an error if `Obs_phi` is already set, and in particular if it has a value different from 90.

Finally, in order to define the view completely, the position of the screen must be given. In linear perspective, the screen is a plane orthogonal to the viewing direction. It is on the screen that the points in space are projected. Figure 4 shows that the focused point lies in the middle of the screen. The screen is determined by its distance to the camera. This distance should also be that from which the computed scene is looked at (provided the scene is not scaled). We take for instance:

```
Obs_dist:=2;
```

At the time of projection, this value as well as the other coordinate values are multiplied by the value of `drawing_scale`, which defaults to 2 cm. The above value of `Obs_dist` therefore corresponds to a camera located at a distance of 4 cm from the screen.

Once the cube's vertices and the camera are in place, the points must be projected on the screen with the `project_point` command. To each point in space corresponds a point in the plane. With `project_point(3,vertex3)`, point z_3 of the plane is associated to vertex 3 of the cube. Once the points have been projected, the edges can be drawn:

```
draw z1--z2--z4--z3--cycle;
draw z5--z6--z8--z7--cycle;
draw z1--z5; draw z2--z6;
draw z3--z7; draw z4--z8;
```

The complete program, with a few more initializations as well as the loop creating the animation, is given in figure 6. (This example didn't make use of the geometry module.)

4.2 Improvements to the previous example

The source code producing the cube is rather simple, but it is easy to come up with drawings that

```
input 3danim; drawing_scale:=10cm;
new_points(vertex)(8);
for i:=0 upto 20:
  beginfig(100+i);
    % Cube
    set_point_(vertex1)(0,0,0); set_point_(vertex2)(0,0,1);
    set_point_(vertex3)(0,1,0); set_point_(vertex4)(0,1,1);
    set_point_(vertex5)(1,0,0); set_point_(vertex6)(1,0,1);
    set_point_(vertex7)(1,1,0); set_point_(vertex8)(1,1,1);
    % Observer/camera
    set_point_(Obs)(20*cosd(3.6*i),20*sind(3.6*i),6);
    Obs_phi:=90; Obs_dist:=2; point_of_view_abs(vertex8,Obs_phi);
    % Projections
    for j:=1 upto 8:
      project_point(j,vertex[j]);
    endfor;
    % Lines
    draw z1--z2--z4--z3--cycle; draw z5--z6--z8--z7--cycle;
    draw z1--z5; draw z2--z6; draw z3--z7; draw z4--z8;
  endfig;
endfor;
free_points(vertex)(8);
end.
```

Figure 6: Program producing the cube animation.

are more complex and very difficult to handle, be it only because of the large amount of points involved. Moreover, the points do not necessarily belong to the same objects (we can have a cube, a tetrahedron, or other objects all present at the same time) and they may be subject to different treatments. It is in this spirit that we have introduced in the `3d` package notions of *classes* and *objects* making it possible to group a number of points, in order to manipulate them globally, or in order to instantiate certain classes several times (see (Meyer, 1997) for more details on object-oriented programming). We will therefore redefine the cube, as a class, and then instantiate it.

The new code (figure 7) shows the definition of a "C" class. All the objects of that class are cubes. The class definition is split in three parts, which have to be called `def_C`, `set_C_points` and `draw_C`:

- The general definition function is `def_C`: This function takes as a parameter an object name and instantiates it. Specifically, this function defines the number n of points making up the object (they will be numbered 1 to n) and calls the function defining the points. Depending on the nature of the defined objects, it may perform other initializations; in particular, though it is not the case here, the initialization can depend on the object name, that is, on the parameter.

- `set_C_points`, the function defining the points, takes the calls to `set_point_` but replaces them with `set_point`. Calling `set_point(1)(0,0,0)` means that point 1 of that object is defined. Thus, we have now a *local* point notion.

- Finally, a drawing function `draw_C` indicating how the object must be drawn.

The instantiation itself, that is the operation associating an object to a class, is done with a call to `assign_obj("cube","C")`. The latter operation defines the *cube* object as an instance of the C class. It leads in particular to the call of the `def_C` function, hence to the computation of the cube's points. It should be noted that in this example the points of the cube are set *before* the camera is set. In more complex drawings (like in figure 21), it is sometimes necessary to have points of an object depending on the position of the camera. In that case, besides the call to the `assign_obj` function (which must only occur once per object), the object positions can be recomputed with `reset_obj` (`set_C_points` cannot be used directly since this function doesn't state how the absolute point numbers should be computed).

```
input 3danim; drawing_scale:=10cm;

vardef def_C(expr inst)=
  new_obj_points(inst,8); set_C_points(inst);
enddef;

vardef set_C_points(expr inst)=
  set_point(1)(0,0,0); set_point(2)(0,0,1); set_point(3)(0,1,0); set_point(4)(0,1,1);
  set_point(5)(1,0,0); set_point(6)(1,0,1); set_point(7)(1,1,0); set_point(8)(1,1,1);
enddef;

vardef draw_C(expr inst)=
  draw_lines(1,2,4,3,1); draw_lines(5,6,8,7,5); draw_line(1,5);
  draw_line(2,6); draw_line(3,7); draw_line(4,8);
enddef;

assign_obj("cube","C");

for i:=0 upto 20:
  beginfig(100+i);
    % Camera
    set_point_(Obs)(20*cosd(3.6*i),20*sind(3.6*i),6);
    Obs_phi:=90; Obs_dist:=2; point_of_view_obj("cube",8,Obs_phi);
    draw_obj("cube");
  endfig;
endfor;

end.
```

Figure 7: "3d object" code of the cube.

The main loop is now almost empty. The definitions concerning the camera have not been modified, except that concerning the focused point. The `point_of_view_obj` function is now used to aim an object point, here the cube point 8, and not an absolute point.

Finally, the cube is drawn with `draw_obj`. This command does both the projection and calls the `draw_C` command (with the `"cube"` parameter) and it is therefore not sufficient to call `draw_C("cube")`. The projection imposes a correlation between an object's local points and those of the plane. It is no longer possible to automatically associate point 7 of an object to z_2, for instance.

In the sequel, we will always encapsulate our constructions in classes, even if (like here) we instantiate the class only once. The table below summarizes the main functions acting on objects.

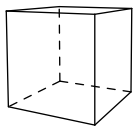


Figure 8: The cube with a camera five times closer as in figure 3.

Definition of C class	`def_C` et `set_C_points`
Drawing definition of C class	`draw_C`
Object instantiation	`assign_obj`
Operations	`translate_obj`
	`rotate_obj`
	`scale_obj`
	`reset_obj`
Object drawing	`draw_obj`

4.3 Perspective

By default, all representations are done in linear (or central) perspective, that is the perspective corresponding to what a camera sees when its field of view is projected on a plane orthogonal to the viewing direction (Le Goff, 2000). The legitimate construction

rules of a linear perspective drawing have been codified by the Quattrocento painters and architects. This perspective is not very apparent on figure 3, because the camera is far from the cube (we are more than 2 meters away from a cube with a 10 cm side). If the camera gets close (and if `drawing_scale` is decreased to prevent the drawing from becoming too large) we obtain (with no changes to the cube) figure 8. The linear perspective shows clearly the vanishing lines.

In this example, the hidden edges have been dashed. The cube being defined as in the figure 7, it is not possible to determine automatically what is visible and what is not, since nothing has been said about the faces. But if the cube is defined as a polyhedron with the `3dpoly` extension (Roegel, 1997), the removal of hidden faces can be done automatically. However, this removal is for the moment only implemented for isolated convex objects. In the present article, all dashed lines are inserted manually.

Other perspectives are provided when the value of `projection_type` is changed. 0 corresponds to the linear perspective. 1 corresponds to a parallel perspective, where all projections are done parallel to the viewing direction and orthogonally to the projection plane. This perspective is different from the cavalier drawing. It corresponds to a camera set at an infinite distance, but looking at the scene with a telescope of infinite power. Since this perspective doesn't change the sizes, it is usually necessary to reduce the size of the projection by decreasing `drawing_scale`.

A first category of parallel projections are the isometric (also called military) perspective, dimetric and trimetric projections, all usually grouped under the axonometric perspectives. However, according to Krikke (Krikke, 2000), these projections are misnamed. Whereas the isometric perspective was invented by William Farish in 1822 to fulfill needs created by the industrial revolution, the real axonometric perspective is a perspective that originated in China and Japan, in particular because it is well suited to a presentation in rolls. In all parallel perspectives, the appearance of an object depends only on its orientation, not on its distance. A distant object doesn't appear smaller than a close object.

A value of 2 for `projection_type` corresponds to the oblique perspectives (which are parallel perspectives), but where the projection plane is not necessarily orthogonal to the projection direction. In general, the projection plane is chosen parallel to one of the object's faces. The most common oblique perspectives are the cavalier drawing and the cabinet drawing. The *asian axonometry*, where an horizontal axis is orthogonal to the viewing direction, also seems to be an oblique perspective.

The `3d` package makes it possible to obtain any of these perspectives. In the case of parallel perspectives, the camera position is only used to find the viewing direction, not how close the view is. In the case of oblique projections, the projection plane being distinct from the camera, it is necessary to give it explicitly.

Figure 9 summarizes the various parallel perspectives.

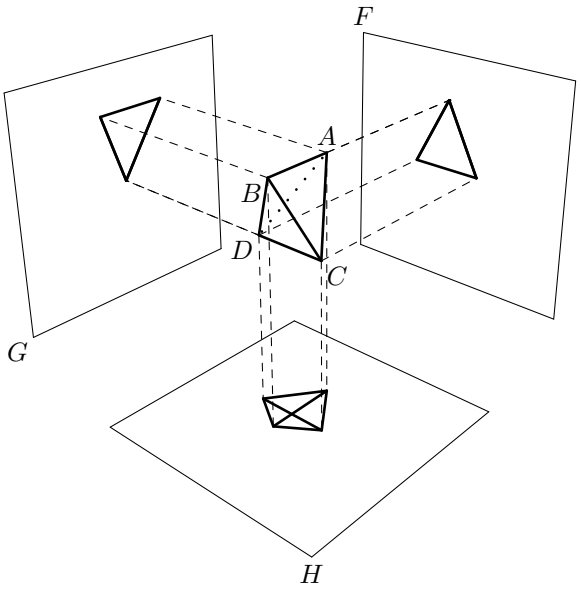


Figure 10: Orthogonal projections.

Another type of representation uses orthogonal (or orthographic) projections that are complementary. An object is described by several orthogonal projections on planes which are themselves orthogonal (see figure 10). The use of several orthogonal views and their crosschecking is known since antiquity, but have been expounded geometrically for the first time by Piero della Francesca in the Renaissance (cf. (Le Goff, 2000, p. 70)). The crosschecking of orthogonal views can be used to build a linear perspective view. For instance, in figure 10, if the two projections on planes F and G are represented in the same plane, the central projection on the plane H can be determined by crosscheckings. Albrecht Dürer used this technique to build the intersection of a cone and a plane in a famous engraving (Dürer, 1525).

Later on, Gaspard Monge systematically studied the method of double projection and codified it

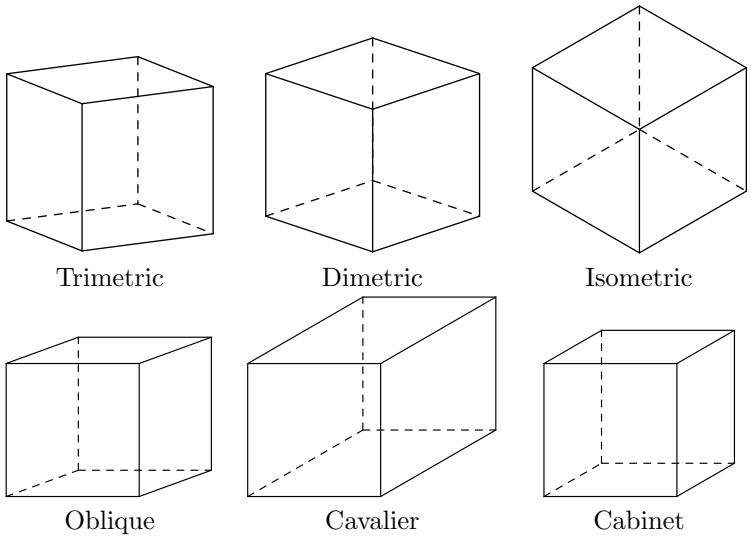


Figure 9: Parallel or perspective projections. The projections in the first row are projections where the projection plane is orthogonal to the projection direction. These projections are also named axonometric by certain authors (Gourret, 1994). The projections in the second row are oblique projections, where the projection plane is not orthogonal to the projection direction. The trimetric projection is the usual case of axonometric projection. In the dimetric projection, two of the axes are at the same scale and in the isometric projection, the three axes are represented at the same scale. In the oblique projections, one of the faces of the object is in general parallel to the projection plane and appears therefore without deformation. In the three cases shown, one of the faces of the projected cube is indeed a square. The angle of the vanishing lines varies. In the cavalier projection, the vanishing lines are represented at the same scale as the lines in the projection plane, which confers a non-natural aspect to that projection (even though it is a genuine oblique projection with no deformation). In order to get a natural feeling, one should observe the drawing from the right. In the cabinet projection, the vanishing lines are represented at 1/2 scale which gives a more natural representation, even though there is no vanishing point. Certain oblique projections are also called planometric, for instance when the face parallel to the projection plane represents a floor plan and when the vertical lines appear vertical in projection.

---

in his *Géométrie descriptive* (1799). His construction method is sometimes called Monge construction or Monge projection. The developments of this technique led to the standard representation in technical drawings in France, where a front face of an object is displayed, to its left the view on the right, to its right the view on the left, below the view seen from the top, etc. (In the U.S, the order is reversed, and the left-side view is on the left, the right-side view is on the right, etc.) The `3d` package currently does not provide an automatic means of representing these perspectives, but they can be simulated easily.

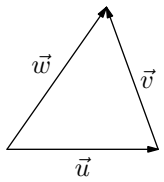
4.4 Local and absolute point indices

We have seen that each point or vector defined with `new_point` or `new_vec` corresponds to a stack element. A point is then given by its index in that stack. The local index of a point is that point's number as it appears in the object definition, if it has been defined as an object point. When we defined the point 2 of our cube with `set_point`, 2 was not this point's index in the stack, but an index relative to the beginning of that object in the stack.

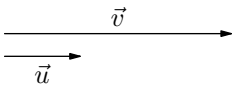
In certain cases, it is necessary to know the absolute index of a point, for instance because certain functions need it. This absolute index is obtained by the `pnt` function. For instance, in order to compute the middle of two points 1 and 2 and put the value in point 3, all these points being local points of an object, one way is to write:

```
vec_sum_(pnt(3),pnt(1),pnt(2));
vec_mult_(pnt(3),pnt(3),.5);
```

This is because `vec_sum_` (sum of two vectors) and `vec_mult_` (scalar multiplication of a vector)



vec_sum_(w,u,v)



vec_mult_(v,u,3)

Figure 11: Two vector operations.

(see figure 11) take as parameters absolute indices. It is not sufficient to create variants of these functions for the cases where these points (vectors) are local (these variants do exist and are called here `vec_sum` and `vec_mult`) because there are often intermediate cases involving local points from one or several objects, as well as points given by their absolute index (like for instance `Obs`). As a consequence, we have provided variants for the most common functions, but not for all of them. In certain cases, one has to resort to the `pnt` function. However, for the previous example, the `mid_point` function can be used:

```
mid_point(3,1,2);
```

The non-local (absolute) variant of `mid_point` is `mid_point_`. The more "internal" functions from the `3d` package have this final "`_`." Thus, making the call `mid_point(3,1,2)` is equivalent to calling `mid_point_(pnt(3),pnt(1),pnt(2))`.

If we had wanted to define a non-local point p (defined outside an object) as being the middle of two local points, we would have written:

```
mid_point_(p,pnt(1),pnt(2));
```

In all cases, we should be careful to use `pnt` only inside an object (this makes it possible not to mention the object explicitly in the above examples) and more precisely only in the functions `def` and `draw` of that object.

4.5 Space structures

The objects definable with the `3d` package are in general rather complex objects which will end up being projected and drawn. These objects are not specially suited for a mathematical treatment. However, geometrical constructions, be it in the plane or in space, involve simple concepts such as lines, planes and other mathematically defined surfaces or volumes. These concepts, which we call here structures, are often used as intermediates for finding new points or new curves. The structures are seldom drawn. We will never draw a line, but only a segment of a line. We will never draw a plane, but only for instance a rectangle in that plane, or merely a few points in that plane.

In order to facilitate the manipulation of these structures, we have created them in a different and simpler fashion than the objects. These structures bear some similarities to the primitive types of Java. Each structure could be wrapped in an object or associated with an object, but we don't do it here.

The structures are defined in the `3dgeom` module. They must also be allocated and freed.

4.5.1 Lines

The simplest of the structures we define is the line. A line is defined with two points. For instance, in order to define the line l going through local points 4 and 6, we write:

```
new_line(l)(4,6);
```

This function (abs. vers. `new_line_`) memorizes the two points, so that a later modification of these points does not modify the line. The line is therefore only initially attached to the points. However, this is seldom a problem for the structures are often introduced locally in a construction. Moreover, it is possible to create a version of `new_line` that does not duplicate the points.

Sometimes we want to define a line whose points have not yet been computed. Sometimes also, we would like to define a line using another pair of points, in order to start a different construction. The `set_line` (or `set_line_`) command can then be used:

```
set_line(l)(4,8);
```

Finally, when a line is no longer needed, it can be freed with `free_line` (which has only one version):

```
free_line(l);
```

It should be observed that the structures defined in an object must always be freed within that object and, as with the points, in the reverse order of their allocation.

4.5.2 Planes

A plane is defined in a manner analogous to a line, but using three points:

```
new_plane(p)(i,j,k);
set_plane(p)(i,j,k);
free_plane(p);
```

`new_plane_` and `set_plane_` are the absolute versions of these functions.

4.5.3 Other structures

Other structures (in the plane or not, such as circles, spheres, etc.) are defined in `3dgeom`, but they have not yet all been developed. It is very easy to add new structures and functions manipulating them.

4.6 Elementary constructions

4.6.1 Plane definitions

The use of structures tremendously simplifies geometric constructions in space. For instance, in order to draw the projections of the tetrahedron vertices in figure 10, we have defined the three projection planes with `new_plane`:

```
new_plane(f)(9,10,11);
new_plane(g)(13,14,15);
new_plane(h)(5,6,7);
```

4.6.2 Perpendiculars of a plane

We have then obtained the perpendiculars of these planes going through the object points, using the `def_vert_pl` function:

```
def_vert_pl(17)(1)(h);
def_vert_pl(18)(2)(h);
def_vert_pl(19)(3)(h);
def_vert_pl(20)(4)(h);
...
```

This function takes a point and a plane and determines the foot of the perpendicular to the plane going through the point given as parameter (here the second parameter).

4.6.3 Intersections between lines and planes

One of `3dgeom`'s functions computes the intersection of a line and a plane:

```
boolean b;
b:=def_inter_p_l_pl(i)(l)(p);
```

The intersection of the line l and of the plane p, if it exists, is computed. If there is an intersection reduced to a point, the function returns `true`, otherwise `false`. The returned point is i (local index).

This function will be illustrated with a high school plane geometry problem: $ABCD$ is a tetrahedron such that $AB = 3$, $AC = 6$, $AD = 4.5$. I is the point of $[AB]$ such that $AI = 1$ and J is the point of $[AC]$ such that $AJ = 4$. We must determine the intersection of the line (IJ) with the plane (BCD). We start by constructing the tetrahedron (figure 12).

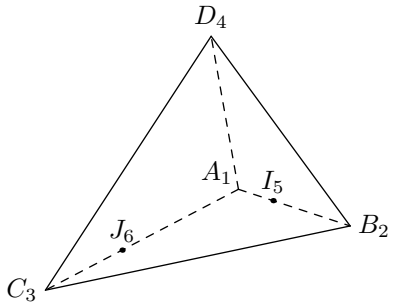


Figure 12: Tetrahedron: first construction.

We should first notice that several tetrahedra are fulfilling the requirements: B, C and D are independent. In order to obtain a rather general construction, we must parameterize it. A can for instance be set in $(0, 0, 0)$, B in $(3\cos\beta, 3\sin\beta, 0)$, C in $(6\cos\gamma, 6\sin\gamma, 0)$ and D obtained by the means of two rotations, one around $\vec{k}$, the other around a vector orthogonal to $\vec{k}$. In METAPOST, this is done with the commands given in figure 13.

On figure 12, the numbers of the points have been added as indices. This figure is of course not well suited to this problem, because (BCD) is facing the observer. We will therefore move the observer, for instance to $C+5\overrightarrow{DC}+5\overrightarrow{BC}+3\overrightarrow{CA}$ (figure 14). In order to achieve this shift, we have reached to tetrahedra points outside of the tetrahedron with the `pnt_obj` function (this function makes it also possible to define points of an object using points from another object):

```
new_vec(v_a);
new_vec(v_b);
new_vec(v_c);
vec_diff_(v_a,pnt_obj("tetra",3),
          pnt_obj("tetra",4)); % DC→
vec_mult_(v_a,v_a,5);          % 5·DC→
vec_diff_(v_b,pnt_obj("tetra",3),
          pnt_obj("tetra",2)); % BC→
vec_mult_(v_b,v_b,5);          % 5·BC→
vec_diff_(v_c,pnt_obj("tetra",1),
          pnt_obj("tetra",3)); % CA→
vec_mult_(v_c,v_c,3);          % 3·CA→
% C+5DC→:
vec_sum_(Obs,pnt_obj("tetra",3),v_a);
% C+5·DC→+5·BC→:
vec_sum_(Obs,Obs,v_b);
% C+5·DC→+5·BC→+3·CA→:
vec_sum_(Obs,Obs,v_c);
free_vec(v_c);
free_vec(v_b);
free_vec(v_a);
```

```
set_point(1)(0,0,0); % A
set_point(2)(3*cosd(b),3*sind(b),0); % B
set_point(3)(6*cosd(c),6*sind(c),0); % C
new_vec(v_a); new_vec(v_b);
vec_def_vec_(v_a,vec_I);  % v_a ← i
vec_rotate_(v_a,vec_K,d); % rot. of v_a around k by an angle d
vec_prod_(v_b,v_a,vec_K); % v_b ← v_a ∧ k
vec_rotate_(v_a,v_b,e);   % rot. of v_a around v_b by an angle e
vec_mult_(v_a,v_a,4.5);
vec_sum_(pnt(4),pnt(1),v_a); % D
free_vec(v_b); free_vec(v_a);
% Determination of I and J:
% I = A + AB→/‖AB→‖
vec_diff(5,2,1); % V5→ ← AB→
vec_unit(5,5);   % V5→ ← AB→/‖AB→‖
vec_sum(5,5,1);  % I ← A + AB→/‖AB→‖
% J = A + 4 · AC→/‖AC→‖
vec_diff(6,3,1); % V6→ ← AC→
vec_unit(6,6);   % V6→ ← AC→/‖AC→‖
vec_mult(6,6,4); % V6→ ← 4 · AC→/‖AC→‖
vec_sum(6,6,1);  % J ← A + 4 · AC→/‖AC→‖
```

Figure 13: Code for figure 12.

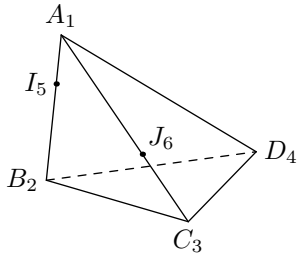


Figure 14: Tetrahedron: second construction.

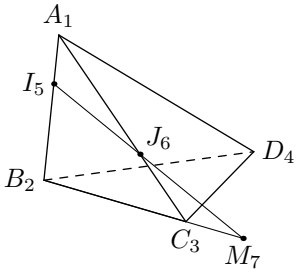


Figure 15: Tetrahedron: third construction.

We now come to the computation of the intersection of the (IJ) line with the (BCD) plane (figure 15).

```
new_plane(bcd)(2,3,4);
new_line(ij)(5,6);
boolean b;
b:=def_inter_p_l_pl(7)(ij)(bcd):
if not b: message "no intersection"; fi;
free_line(ij);
free_plane(bcd);
```

Two other intersections can be computed using K, the middle of $[AD]$ (figure 16). The three intersections are then aligned. (We can see that L, M and N are as a matter of fact on the intersection of the planes (IJK) and (BCD).) We have shown the alignment with a segment slightly extending on each side, using

```
draw_line_extra(9,10)(-0.1,1.1);
```

The second pair of parameters indicates how much the segment extends on each side. $(0, 1)$ corresponds to no extension and a smaller (or larger) value for the first (or second) parameter produces an extension.

Figure 16 also illustrates the famous theorem by Girard Desargues (1639), which is the cornerstone of projective geometry. According to this theorem, two triangles BCD and IJK being given in the plane, the lines (BI), (CJ), (DK) have an intersection if and only if the intersections of (IK) and (BD), of (IJ) and (BC), and of (KJ) and (DC) are aligned. In our case, the lines (BI), (CJ), (DK) have indeed an intersection, namely A, and the intersections L,

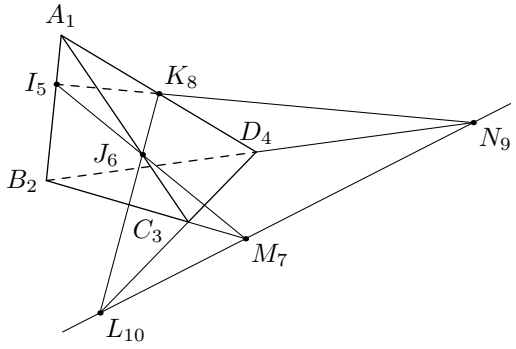


Figure 16: Tetrahedron: fourth construction.

M and N are aligned. The theorem states that the converse also holds. Seen in space, the theorem is very simple, but a purely plane proof is difficult.

Another function of `3dgeom` allows for the direct computation of the intersection between two planes:

```
b:=def_inter_l_pl_pl(l)(p)(q);
```

where `b` is a `boolean`. If the intersection between the planes p and q is a line, the function yields `true` and the line is stored in l. Otherwise, the function yields `false`. Let us see on an example how this function can be used. Consider the drawing of a tetrahedron $SABC$ whose edges SA, SB et SC are known, as well as the angles $\widehat{ASC}$, $\widehat{ASB}$ and $\widehat{BSC}$. Figure 17 shows such a tetrahedron with $SA = 9$, $SB = 8$, $SC = 4$, $\widehat{ASC} = 60°$, $\widehat{ASB} = 40°$ and $\widehat{BSC} = 30°$. Contrary to the previous example, here we do not have a wide margin to place the points. We can of course start to construct the SAC triangle. It then only remains to place the B vertex.

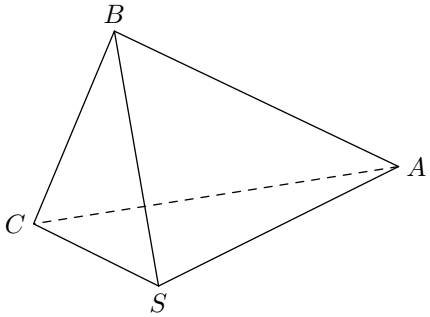


Figure 17: A tetrahedron specified by three lengths and three angles.

The angles $\widehat{ASB}$ and $\widehat{BSC}$ being given, B is obviously located on two cones: the cone of axis (SA), of apex S and of angle $\widehat{ASB}$ and the cone of axis (SC), of apex S and of angle $\widehat{CSB}$. These two cones in general intersect in two lines and B is on one of these intersections at the distance SB of S.

With a well-chosen implementation of a cone structure, the intersection can of course be automatically obtained. But it is also possible to use more restricted means by observing that we can draw the heights $[SH]$ and $[SK]$ stemming from B for each of the triangles SAB and SBC. For instance, $SH = SB \cdot \cos(\widehat{ASB})$. We can then define the planes orthogonal to the lines (SA) and (SC) going through the two heights' feet. The function

```
def_orth_pl_l_p(p)(l)(i);
```

constructs the plane p orthogonal to the line l and going through the local point i.

The intersection between the two constructed planes can then be computed. This intersection is a line orthogonal to the plane of the triangle SAC. On this line, we look for a point B at a given distance from S. The function call

```
b:=def_point_at(i)(d)(j)(l);
```

where `b` is a `boolean` variable, defines the local point i as being a point of the line l at a distance $|d|$ from the local point j if such a point can be found. In that case, the return value of the function is `true`, otherwise it is `false`. In general, two points satisfy the condition and the function will return either one depending on the sign of d.

Essentially, the figure is thus produced by the commands in figure 18.

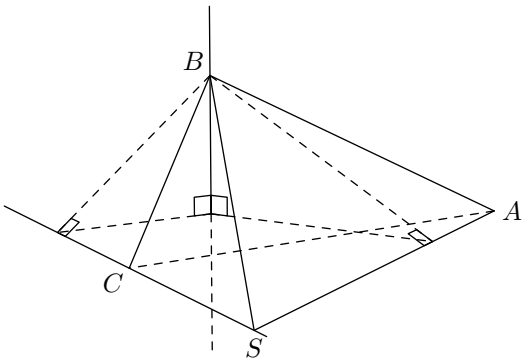


Figure 19: A tetrahedron specified by three lengths and three angles (construction).

The whole construction is given in figure 19. In order to draw it, we have "unlocalized" points such as the heights' feet and the intersection between the perpendicular in B to the (SAC) plane. In order to produce the right angles, we have used the commands `def_right_angle` for the definition and `draw_double_right_angle` for the drawing. Each

```
new_point(h); new_point(k);
set_point(1)(0,0,0);   % S
set_point(2)(lsa,0,0); % A
set_point(4)(lsc*cosd(aasc),lsc*sind(aasc),0); % C
vec_diff_(h,pnt(2),pnt(1));
vec_unit_(h,h);
vec_mult_(h,h,lsb*cosd(aasb));
vec_sum_(h,h,pnt(1));  % H
vec_diff_(k,pnt(4),pnt(1));
vec_unit_(k,k);
vec_mult_(k,k,lsb*cosd(absc));
vec_sum_(k,k,pnt(1));  % K
new_plane(hp)(1,1,1);  % initialization to three points
new_plane(kp)(1,1,1);  % ditto
new_line(sa)(1,2);     % (SA)
new_line(sc)(1,4);     % (SC)
new_line(inter)(1,1);  % intersection line of the two planes
def_orth_pl_l_p_(hp)(sa)(h); % plane orthogonal to (SA) in H
def_orth_pl_l_p_(kp)(sc)(k); % plane orthogonal to (SC) in K
if def_inter_l_pl_pl(inter)(hp)(kp): % there is an intersection
  if not def_point_at(3)(-lsb,1)(inter): % B
    message "Should not happen";
  fi;
else:
  message "PROBLEM (probably the angle ASC too small)";
  set_point(3)(1,1,1);
fi;
free_line(inter); free_line(sc); free_line(sa);
free_plane(kp); free_plane(hp);
free_point(k); free_point(h);
```

Figure 18: Code for figure 19.

right angle is made of two segments, defined using three points. These three points are new object points. For instance, one of the right angles is created with

```
def_right_angle(7,8,9,5,1,3);
```

This means that three points (numbered locally 7, 8, and 9) are introduced and that they are set according to the angle determined by the triangle of points (5,1,3).

The drawing, on the other hand, is simpler:

```
draw_double_right_angle(7,8,9,5);
```

4.7 Visual complements

4.7.1 Representation of planes

A plane is often represented using four particular points making a rectangle. These points must be defined. The drawing of an horizontal rectangle corresponding to the (SAC) plane in figure 19 can be obtained as follows (see figure 20):

```
set_point(14)(-2,-2,0); % p1
set_point(15)(11,-2,0); % p2
set_point(16)(11,10,0); % p3
set_point(17)(-2,10,0); % p4
```

The points are connected with `draw_lines`.

4.7.2 Hidden parts and visual intersections

As shown on figure 20, there is (currently) no automatic hidden parts removal or a special treatment of those parts. It is necessary to handle the dashed lines by hand and this is only practical in the case of non-moving images. With animations, the perspective can be subjected to such variations that it is advisable to have an automatic solution of the problem.

However, on a non-moving image, the problem of removing (or handling in a special way) hidden parts is rather simple to express and solve. It is actually a matter of determining "apparent" intersections between two curves. In the previous example,

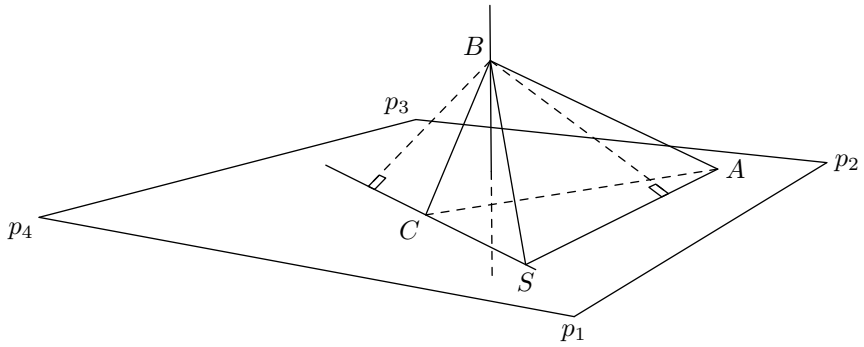


Figure 20: The addition of a plane to the drawing in figure 19.

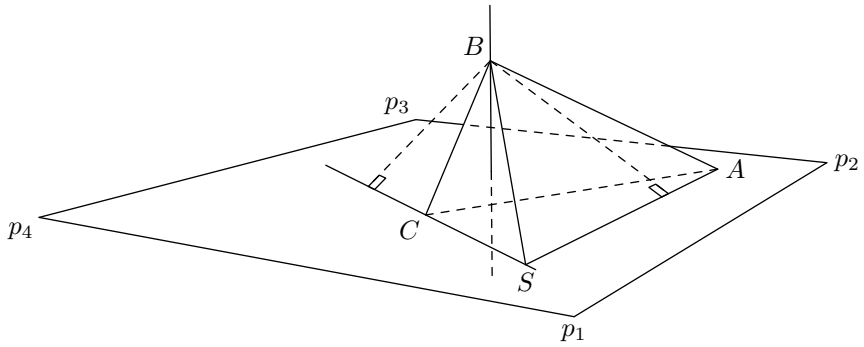


Figure 21: The interruption of a plane.

sides of the tetrahedron seem to meet sides of the rectangle representing the plane, even though they do not meet in space.

If the plane had to be represented in a clearer way, the apparent (or visual) intersections of the sides $[BA]$ and $[BC]$ with the most distant side of the rectangle would have to be determined. How can this be done?

One way is to construct a point of the segment $[p_2p_3]$, but as a function of the observer's position. The intersection between (Obs, p_2, p_3) and (Obs, A, B) is a line going through the observer and through the point of (p_2p_3) of interest to us. It is then sufficient to determine the intersection between this line and the (p_2p_3) line. This intersection being computed in space, care must be taken that rounding errors can prevent the intersection of two lines which should otherwise intersect. The `def_inter_p_l_l` function finds the middle of the two points of each line where the line is closest to the other line. This function also returns the distance between the two points. This makes it possible to find a point of (p_2p_3) corresponding to the visual interruption caused by the $[BA]$ segment. Similarly, it is possible to find the point corresponding to the visual interruption caused by the $[BC]$ segment. These two points, with p_2 and p_3, make it possible to draw a more natural plane. This is what is done in figure 21.

The `def_visual_inter` function takes care of this procedure. It takes four local points and computes a fifth one:

```
boolean b;
b:=def_visual_inter(i)(j,k,l,m);
```

If the function returns `true`, point i is located on (jk) at the apparent intersection of (jk) and (lm).

It should be noted that in a central perspective, the computed intersections depend on the observer's position. If this position changes, the intersections must be recomputed. As a consequence, as we already indicated it, it is necessary to call the `reset_obj` function after the redefinition of the observer's position.

In a parallel projection, the observer is not used in the computation of the visual intersection, but the interface remains the same. The intersections

are almost computed in the same way, except that the planes whose intersection is computed are not determined by the observer and a segment, but by the projection direction and a segment.

4.7.3 Vanishing points

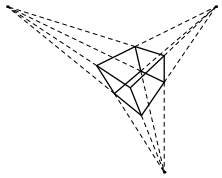


Figure 22: The three classical vanishing points of a cube.

The representation of an object containing parallel segments in a central perspective shows vanishing lines and points. The projected lines are usually no longer parallel and intersect. An example of a cube representation with three vanishing points is given in figure 22. The classical representations often distinguish the drawings with one, two or three vanishing points. The drawings with one or two vanishing points are special cases corresponding to lines which are parallel to the projection plane. In figure 22, none of the cube's sides are parallel to the projection plane, and this leads to vanishing points. If the projection plane had been parallel to the vertical segments of the cube, these segments wouldn't have exhibited vanishing points. If it is a whole face which is parallel to the projection plane, there is only one vanishing point left.

The vanishing points correspond to points located at the infinite in space, along a direction going through the observer and directed by a vector corresponding to the object's segment. Very often, some of the vanishing points will be quite distant on the drawing, and possibly outside the drawing.

The classical representations in architecture or in painting put two vanishing points on an horizontal line and a third vanishing point corresponding to the vertical vanishing lines. Figure 22 differs from that representation because the observer is not oriented along a vertical axis.

A different number of vanishing points do not correspond to different projections, but on the one hand to objects which are positioned differently in space with respect to the projection plane, and on the other hand to objects of different nature. A sphere will of course have no vanishing point! The number of vanishing points can actually be any number, including a number greater than three. It suffices to choose a pair of parallel lines on the object. Figure 23 shows that besides the three classical vanishing points, the six vanishing points stemming from the diagonal segments of the cube can be considered. One of those is represented on the figure. (It should be remarked that vanishing points being very sensitive to the location of the observer, it is rather difficult to find by hand a location where the nine vanishing points are simultaneously visible in a restricted space.) More complex objects would have even more vanishing points.

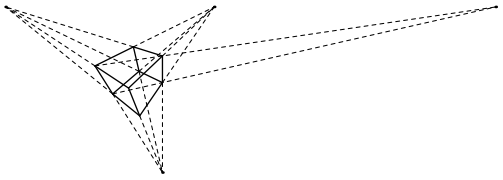


Figure 23: A fourth vanishing point.

A vanishing point can be determined in a simple way by computing the intersection between the projection plane and the line going through the observer and oriented by a vector of the object. The following lines find the vanishing point of the segment connecting two points numbered 1 and 5:

```
% defines the projection plane
def_screen_pl(screen);
new_line(l)(1,5);
if not
  def_vanishing_point_p_l_pl(11)(l)(screen):
  message "no vanishing point";
  set_point(11)(0,0,0);
fi;
...
```

Like for the visual intersections, the vanishing points depend on the observer's position and each time the point of view changes (either because the observer moves, or because the object moves), the object points must be recomputed with `reset_obj`.

It would also be possible to recompute the vanishing points directly in the plane. This would be a mere application of `whatever` (see section 2).

4.7.4 Shadows

The shadows corresponding to projections are simulated by shading the projected part. An example is given in figure 24. In that example, a triangle is projected on a plane. We have merely computed the projections of the three vertices of the triangle and we have then shaded the projection. This technique works for each projection, as long as the projection is made along a line and on a plane or a set of planes.

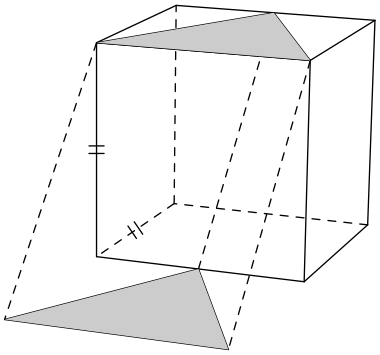


Figure 24: The cube with a shadow.

5 Modifications with respect to the first distribution

When we worked on the new version of the `3d` package, we made various changes which seemed to go in the direction of a greater homogeneity. A number of elementary functions have been renamed in order to respect our new function naming conventions. Hence, our first article (Roegel, 1997) on that matter is now no longer strictly correct because functions such as `vect_sum` must be replaced by `vec_sum_`. All the "`vect`" have been replaced by "`vec`." Finally, the creation of vectors and points has slightly changed. The differences are not especially important, but in order to make the transition easier for the reader, we have put on CTAN a corrected version of the original article, with differences highlighted.

6 Conclusion and limits

This package contains numerous features which have not been mentioned and it is easy to add new ones, in particular concerning the manipulation of new structures. We silently omitted drawing curves such as circles which involve another module that will be described elsewhere. More comprehensive documentation comes with the package.

A study of this article also reveals that the coordinates of a point have only explicitly been used for a few points constructed directly with `set_point_` or `set_point`. All other points have been obtained through various geometric operations. It doesn't mean that the coordinates are not accessible! They are given by the functions `xval`, `yval` and `zval` applied to a vector or point reference.

This package is of course not perfect and has limits. Besides METAPOST's limits, in particular in numerical capacity (the dimensions are bounded to 4096 PostScript points, at least for a normal use), our package lacks automatic computations. Still many manual interventions are needed. Other limitations concern (currently) the absence of a decent and automatic hidden parts removal algorithm. Finally, error handling in METAPOST will not be simple for who is not somewhat accustomed with the language.

This work can of course be compared to other works going in the same direction. First, it should be clear that we do not claim to compete with professional CAD or computer algebra tools. We want above all to provide a light and powerful system helping the creation of geometric constructions, in particular suited for a geometry class in high school. In the TEX world, there are to our knowledge only few works integrating space. METAGRAF (`http://w3.mecanica.upm.es/metapost`), also based on METAPOST, is an interactive system with a notion of space, but which doesn't seem to provide possibilities of geometric constructions, animations, changes of perspective, etc. The PSTricks system has a 3D module, but it is relatively undeveloped. The computations are done with TEX and extending the system is tedious. Outside the TEX world, various 3D languages are available, in particular OpenGL, which goes much beyond our system with respect to rendering. As a teaching tool for geometry, we should also mention the *Cabri-Géomètre* software (cf. `http://www.cabri.net`).

7 Acknowledgements

I would like to thank Nicolas Kisselhoff who led me to develop the `3dgeom` module by providing me with figures from his geometry course, Sami Alex Zaimi who developed the notion of an object and has indirectly influenced this work and Pablo Argon who convinced me many years ago of the usefulness of METAPOST for the geometry of the ruler and the compass. Hans Hagen proofread the French version of the article and pushed me to clarify it even further, in particular through the introduction of new figures. Finally, Jean-Michel Hufflen and Damien Wyart have made corrections and suggested various improvements.

References

Dürer, Albrecht. *Underweysung der messung mit dem zirckel und richtscheyt in Linien ebnen unnd gantzen corporen durch Albrecht Dürer zu samen getzogen und zu nutz aller kunstliebhabenden mit zu gehörigen figuren in truck gebracht im jar. M.D.X.X.V.* 1525. Facsimile (Portland, Or.: Collegium Graphicum, c1972.).

Goossens, Michel, S. Rahtz, and F. Mittelbach. *The LaTeX Graphics Companion*. Reading, MA, USA : Addison-Wesley, 1997.

Gourret, Jean-Paul. *Modélisation d'images fixes et animées*. Paris : Masson, 1994.

Hagen, Hans. *MetaFun*, 2002. http://www.pragma-ade.com.

Hobby, John D. "A User's Manual for MetaPost". Technical Report 162, AT&T Bell Laboratories, Murray Hill, New Jersey, 1992. http://cm.bell-labs.com/who/hobby/MetaPost.html.

Hoenig, Alan. *TeX unbound. LaTeX & TeX Strategies for Fonts, Graphics, & More*. Oxford, New York : Oxford University Press, 1998.

Knuth, Donald E. *Computers & Typesetting, volume C: The METAFONTbook*. Reading, MA : Addison-Wesley Publishing Company, 1986.

Krikke, Jan. "Axonometry: A Matter of Perspective". *IEEE Computer Graphics and Applications* **20**(4), 7–11, 2000. http://www.computer.org/cga/cg2000/pdf/g4007.pdf.

Le Goff, Jean-Pierre. "De la perspective à l'infini géométrique". *Pour la Science* (278), 66–72, 2000.

Meyer, Bertrand. *Object-oriented software construction*. Upper Saddle River, N.J.: Prentice Hall, 1997.

Roegel, Denis. "Creating 3D animations with METAPOST". *TUGboat* **18**(4), 274–283, 1997. ctan:graphics/metapost/macros/3d/tugboat/tb57roeg.pdf. An updated version is located at the same place.

◇ Denis Roegel
LORIA
BP 239
54506 Vandœuvre-lès-Nancy
FRANCE
roegel@loria.fr
http://www.loria.fr/~roegel