MP2GL: prototyping 3D objects with METAPOST and OpenGL

Denis Roegel
LORIA, Nancy (France)

7 March 2005

Abstract

METAPOST was created with 2D graphics in mind, and in spite of various extensions added during the last few years, it doesn't seem well adapted for 3D technical graphics. However, there are cases where simple but realistic 3D graphics are needed, for instance for inclusion in an article, and there are also cases where 3D objects are mere 2D objects with added depth. In such cases, an approach combining METAPOST with an OpenGL environment proves very useful and allows for interesting applications, in particular the prototyping of 3D objects for use independently from METAPOST. Such an approach is also a smooth way to get introduced to OpenGL. MP2GL is our first attempt towards this direction.

Contents

1 Introduction **2**
1.1 Limitations of METAPOST 2
1.2 Motivations of this work 3

2 MP2GL **3**
2.1 OpenGL 4
2.2 Overview and scope of MP2GL 5

3 Elements of OpenGL **6**
3.1 Objects 6
3.2 Projections 7

4 The MP2GL interface to OpenGL **7**
4.1 Coordinates and coordinate transformations 7
4.2 Basic objects 8
4.3 Objects constructed from paths 10
4.3.1 Simple cyclic paths 10
4.3.2 Sets of paths 11
4.3.3 Other path-constructed objects 11
4.4 Low-level constructions 11
4.5 More on wire frames 13
4.6 More complex surfaces — NURBS 13
4.7 Lights and colors 14

5 3D Equations **14**

6 Text support **18**
6.1 Basic text 18
6.2 More complex marks 18

7 Animations **20**

8 Limitations: textures, blending, ... and other advanced features **20**

9 Related work **22**
9.1 Our own METAPOST 3d (aka 'mp3d') package 22
9.2 PSTricks-3D 23
9.3 m3dplain 23
9.4 MetaGraph3D 23
9.5 FEATPOST 23
9.6 3DLDF 24

10 Conclusion **25**

11 Acknowledgments **25**

1 Introduction

METAPOST is a language aimed at the description of technical drawings, and is adapted from METAFONT [8, 7]. With METAPOST, one describes a two-dimensional drawing with primitive objects such as points and paths connecting these points. The language of METAPOST is very rich and a number of extensions have been written, in particular for handling graphs, or objects.

1.1 Limitations of METAPOST

However, METAPOST is not a 3D engine. 3D objects can be defined in the language, but doing so is tedious. Moreover, the representation of a 3-dimensional scene requires an algorithm for hidden faces removal, which is time-consuming. Currently, the few existing 3D extensions to METAPOST only handle special cases. For instance, our own 3d extension was able to handle convex polyhedra, but not always when they were overlapping [11]. But even if a general hidden faces removal algorithm were implemented, it wouldn't be the end of the story. METAPOST has known numerical limitations and such an algorithm, as well as other features like shading, would stretch it to its limits.

It appears therefore desirable, either to extend the core METAPOST and include native 3D support, or to use an external processor for the 3D computations. With the first approach, one is led to rebuild all the 3D algorithms which are already available elsewhere. It may be a sound approach, and it may provide inherent advantages, but there is a long way to go.

With the second approach, on the contrary, an existing 3D engine is used and combined with METAPOST. This may lead to some inefficiencies, for instance

when the 3D engine doesn't know about internal METAPOST parameters, but the advantages seem to far outweigh the limitations, at least for the time being.

The second approach, however, leads us to a crucial question: if we use METAPOST with a 3D engine, do we need METAPOST at all? In other words, why would we want to use METAPOST for 3D drawings? We come to the motivations of our work.

1.2 Motivations of this work

Among the main reasons for using 2D-METAPOST in a TEX environment are that:

- it is a natural partner of TEX;
- it can produce high-quality and accurate technical drawings;
- it produces vector graphics;
- it has a nice declarative approach, where equations can be used to specify points;
- it has nice types, especially the `path` type;
- and it is fun to use!

The main reasons for using 3D-METAPOST should at least include the previous ones. In particular, we are looking for a way to produce 3D vector graphics, while at the same time retaining the declarative approach and the ability to use types such as paths.

But we also want more. With 3D come more needs. Consider first a planar drawing. The coordinates of the origin of such a drawing are actually irrelevant. Whether a square is drawn with its lower left corner at $(0, 0)$ or at $(1, 1)$ doesn't make a difference, as long as the bounding box of the visible part of the drawing is used for insertion.

With 3D, we have a camera (or an observer), and its location determines the point of view. But with 3D, there is also the legitimate desire to have motion, or animations. Such was actually the motivation of the `3d` METAPOST package we wrote in 1997 [11]. With it, small GIF animations could be produced.

But these animations were not interactive. Now, we want interaction, so that a 3D scene created with METAPOST can be animated, and the best point of view chosen, something that is often best done interactively. Once the point of view is found, a vector snapshot should be produced.

Since 3D vector graphics can be obtained by other tools than METAPOST, the answer to our question on the need of 3D-METAPOST at all is therefore that it is only needed if we want to retain an homogeneous framework common with 2D-METAPOST, and if we want to be able to have a smooth integration with TEX.

2 MP2GL

MP2GL is the bridge we have developped between METAPOST on one side, and OpenGL on the other. Such a bridge is interesting not only for producing 3D

figures for inclusion in articles, but also for 3D objects independent of articles. We will see that linking METAPOST to OpenGL isn't that catastrophic in terms of loss of critical information, because METAPOST could reuse later outputs.

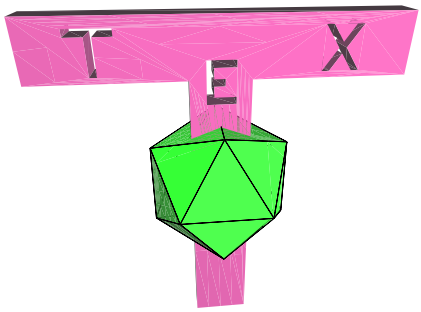

Figure 1: A TEX logo produced with MP2GL. This scene combines an object built from METAPOST paths (the 'T' with the three holes) and a predefined OpenGL icosahedron in solid output, to which a slightly larger wire frame icosahedron was superimposed.

2.1 OpenGL

OpenGL is an API for 3D graphics, which has become a standard in the industry [18]. There are OpenGL libraries for many languages and many applications are using such a library for their advanced graphics. Various games (for instance *Quake*) also use OpenGL.

OpenGL provides functions for the rendering of 3D scenes, and these can include lights, shading, and various other effects. The scene is drawn on a screen and adapted to its resolution. An OpenGL frame is a *bitmap*. OpenGL uses the z-buffer algorithm for hidden parts removal and this is done seamlessly.

We can therefore envision producing 3D bitmaps, but fortunately there are also ways to save a 3D scene, not as a bitmap, but as a vector graphics, which is independent of the screen size. This is made possible by the GL2PS library [2] (figure 2) [1].

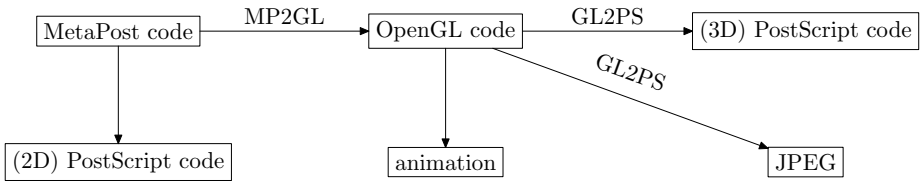

Figure 2: From METAPOST to 3D-PostScript.

[1] However, GL2PS has several limitations, and this will have as a consequence that there are 3D scenes which are constructible with MP2GL, but which cannot be saved in PostScript.

2.2 Overview and scope of MP2GL

MP2GL currently provides a limited interface to OpenGL, and tries to be faithful to the motivations given above [2].

Our aim so far has been to show the feasibility of this approach, and we have therefore only coded a few representative features. It is hoped that this approach will be extended if it proves useful.

More specifically, it occurred to us that most 3D objects that one wants to build are:

- either geometrically very simple (for instance a cube, a sphere, ...),
- or obtained simply from 2D objects (a cube, a prism, ...),
- or composed of simpler 3D objects.

In other words, we think that the greatest amount of time when building 3D objects will normally be spent on adding depth to 2D shapes. The reason behind this observation lies of course in either the manufacturing process of the objects we want to reconstruct, or in the way they function. For instance, almost every moving object is actually moving in translation or around an axis (consider a car, a steering wheel, etc.), and this usually is reflected in the shape of the object.

MP2GL is not limited to such objects, but it has facilities for handling them. In particular, MP2GL makes it possible to use paths for specifying shapes used in 3D objects.

The main features of MP2GL (in this experimental version) are:

- METAPOST input language;
- structures for points and homogeneous coordinates;
- interface to OpenGL objects (for instance polyhedra);
- ability to build low-level objects made of faces;
- ability to use METAPOST paths as a basis for prisms, including non-convex prisms with holes;
- equations can be used for positioning objects in space;
- C code using the OpenGL library is produced, with a minimal animation interface;
- from a given point of view, either a bitmap or a PS output can be produced;
- TEX labels can be added and adjusted after the PS production;
- the C output can be edited and extended;
- the objects created with MP2GL can be used without METAPOST;
- objects can also be created in OpenGL for use by MP2GL.

[2] MP2GL has only been tested on linux, but should be easy to adapt to other platforms, provided GL2PS is ported there.

The animation produced by MP2GL provides a standard set of lights and a standard position for the observer. These can easily be changed, but not yet from MP2GL in this preliminary version [3].

For some of the objects, the METAPOST interface is very simple. This is, for instance, the case for the regular polyhedra. Other objects require more work.

The reasons why an object should be coded in METAPOST or in OpenGL have to do with the need to use a METAPOST structure (for instance a `path` value), or with a greater familiarity of the user with one of these two languages. MP2GL should actually be seen as a gateway from METAPOST to OpenGL, both for the code which is translated in OpenGL, and for the user who can seamlessly learn about OpenGL and try to make changes to the produced code.

3 Elements of OpenGL

3.1 Objects

The scene itself is made of objects, which are themselves made of faces. For instance, a cube is made of six faces. Objects or faces are built using points in a given referential. OpenGL provides means to change the referential. The basic transformations in OpenGL are:

- translation: `glTranslatef(`x,y,z`)`
- rotation: `glRotatef(`α,x,y,z`)`
- scaling: `glScalef(`x,y,z`)`
- saving a position: `glPushMatrix()`
- restoring a position: `glPopMatrix()`

Building an object amounts to moving to where the object should be set, and then calling the appropriate function. For instance, building a tetrahedron at coordinates $(-4.1, 5, 12.3)$ is done as follows:

```
glTranslatef(-4.1,5,12.3);
glutSolidTetrahedron();
```

Building a square face at coordinates $(0, 0, 0)$, $(1, 0, 0)$, $(1, 1, 0)$ and $(0, 1, 0)$ is done as follows:

```
glBegin(GL_POLYGON);
  glNormal3f(0,0,1);
  glVertex3f(0,0,0);
  glVertex3f(1,0,0);
  glVertex3f(1,1,0);
  glVertex3f(0,1,0);
glEnd();
```

The `glVertex3f` calls specify the vertices and the `glNormal3f` call specifies a normal to the face, which is necessary for correct shading with lights.

Such a square can be put anywhere in space by the appropriate use of transformations before `glBegin`.

[3] It should be stressed that many features are very easy to add, since they are a mere interface to OpenGL. The main difficulty is to get the file organization right.

3.2 Projections

The main OpenGL projections are the perspective and orthographic projections. In both cases a frustum is defined and all elements outside the frustum are not drawn. In the perspective projection, a camera has to be defined.

4 The MP2GL interface to OpenGL

MP2GL can be used either to create 3D objects using the high-level OpenGL objects library, or it can create objects by a low-level approach.

We distinguish therefore an OpenGL objects library (OL) and a METAPOST objects library (ML).

In either case, objects can be set anywhere in space, and the handling of coordinates is done by mimicking the OpenGL operations. Although in many cases we do not really need to know where we are, but only how we came there, we provide means to obtain an exact location from relative motions (translations and rotations).

4.1 Coordinates and coordinate transformations

Coordinates are seen as triples, but are actually manipulated as vectors in a 4-dimensional space. Nevertheless, we use a 3-dimensional structure to store points, namely the `color` structure, renamed `Point`.

```
def Point = color enddef;
def Xpart = redpart enddef;
def Ypart = greenpart enddef;
def Zpart = bluepart enddef;
```

Internal transformations involve 4×4 matrices which are stored as 2-dimensional arrays of numerics.

The user need not care about matrices, even less in METAPOST than in OpenGL. The only transformations needed are the following:

- `ResetPosition`: return to the origin.
- `ResetLocalPosition`: return to the *local* origin.
- `PushPosition`: save the current position.
- `PopPosition`: restore the formerly saved position (if there was one).
- `Translate(`x,y,z`)`: move by (x, y, z)
- `TranslateV(`v`)`: move by (v_x, v_y, v_z)
- `RotateX(`a`)`: rotate a around the X axis
- `RotateY(`a`)`: rotate a around the Y axis
- `RotateZ(`a`)`: rotate a around the Z axis
- `Scale(`s_x,s_y,s_z`)`: scale coordinate X by s_x, Y by s_y, Z by s_z. (a scale with one negative value does a reflection)

- `CurrentPosition`: returns the absolute position of the origin of the current referential.

We also have functions to compute the usual vector operations, and in particular the normal of a surface given three points.

4.2 Basic objects

The current version of MP2GL provides only a few basic objects, among them the regular polyhedra, the sphere, the cylinder, etc. All these objects either exist in OpenGL, or can easily be constructed. Although these objects seem to be special cases, they can actually be transformed. For instance, a sphere can be used to obtain an ellipsoid, by changing the scales of the axes.

A number of basic objects are provided in two versions (solid and wire frame):

- Regular tetrahedron: `solid_tetrahedron`, `wire_tetrahedron`; in addition, there is a `new_tetrahedron` macro building a solid tetrahedron from four points;
- Cube: `solid_cube(`*size*`)`, `wire_cube(`*size*`)`;
- Octahedron: `solid_octahedron`, `wire_octahedron`;
- Dodecahedron: `solid_dodecahedron`, `wire_dodecahedron`;
- Icosahedron: `solid_icosahedron`, `wire_icosahedron`;
- Sphere: `solid_sphere(`*radius*,*slices*,*stacks*`)`, `wire_sphere(`*radius*,*slices*,*stacks*`)`;
- Cone: `solid_cone(`*radius*,*height*,*slices*,*stacks*`)`, `wire_cone(`*radius*,*height*,*slices*,*stacks*`)`;
- Torus: `solid_torus(`*r*,*R*,*nsides*,*rings*`)`, `wire_torus(`*r*,*R*,*nsides*,*rings*`)`;
- Teapot: `solid_teapot(`*size*`)`, `wire_teapot(`*size*`)`.

The dimensions will be the METAPOST dimensions interpreted as OpenGL dimensions. So, if the object `solid_cube(1cm)` is drawn, it will actually be a cube of edge $\frac{72}{2.54}$ OpenGL units, because METAPOST uses PostScript points and 2.54cm corresponds to 72 PostScript points. How big OpenGL units are is irrelevant, since it all depends on the position of the camera.

The first example in figure 3 was obtained with the following commands:

```
begin_scene;
  disable_lighting;
  setwirecolor(0,0,0); % black
  setlinewidth(0.5);
  wire_tetrahedron;
end_scene;
```

and the second is a mere call to `solid_tetrahedron` with no other settings.

The following objects are based on quadrics and there is only one version of each:

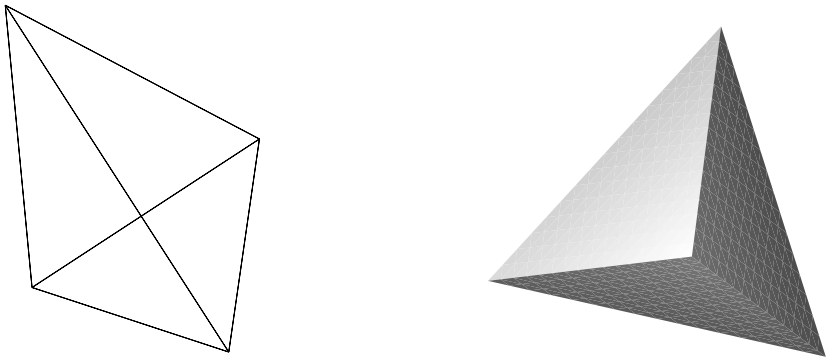

Figure 3: A regular tetrahedron in wire frame and another in solid mode.

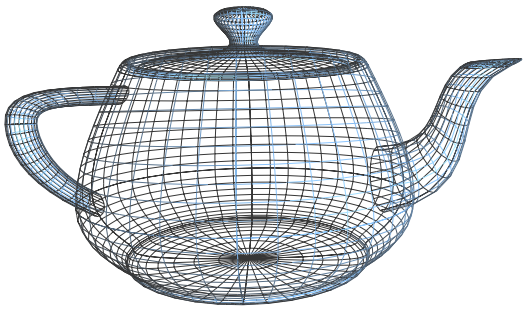

Figure 4: The standard teapot in wireframe.

- Disk: `disk(`*name*,*r*,*R*,*slices*,*rings*`)`;
- Partial disk: `partialdisk(`*name*,*r*,*R*,*slices*,*rings*,*startangle*,*sweepangle*`)`
- Sphere: `sphere(`*name*,*r*,*slices*,*stacks*`)`
- Cylinder: `cylinder(`*name*,*rbase*,*rtop*,*height*,*slices*,*stacks*`)`

4.3 Objects constructed from paths

Paths are a very convenient structure in METAPOST. Although METAPOST paths are limited in 2D, we can readily build 3D objects using them.

4.3.1 Simple cyclic paths

The first step is to evaluate a path and store it in a way appropriate for OpenGL. Currently, we merely store the points used to build the path, not the control points. In other words, only three points are used in a path such as `z1..z2..z3`. Future versions may offer options to save control points in order to take advantage of NURBS. Possibly, the computation of control points could also be added to the OpenGL library, mimicking the METAFONT/METAPOST construction of control points.

```
path p;
p=......--cycle;
storepath(p,"Path_P");
```

Once the path is stored (which saves its points in a C array), an object built on path p will use the name `"Path_P"` of the path. For instance, a prism can be built as follows [4]:

```
new_prism("Prism1","Path_P",3cm);
```

The last parameter is the height of the prism.

MP2GL functions prefixed with "`new_`" create new C functions representing the objects. Calls to these functions can be produced within METAPOST with `use_object`. For example, the scene involving the previous object can be constructed as follows:

```
begin_scene;
  use_object("Prism1");
end_scene;
```

The 2D elements involved in the construction can be examined in the same METAPOST run, by the help of an ordinary figure environment:

```
beginfig(1);
  draw p;
endfig;
```

In other words, the compilation of the METAPOST file will have two (or more) outputs: a C file which will be postprocessed by a C compiler, and one or more ordinary PS figures.

[4] The current version of the prism function works only for convex paths, because it relies on the polygon construction of OpenGL. For more general cyclic paths, the "multipath prisms" should be used.

4.3.2 Sets of paths

Sets of paths can be used to create shapes with holes. These shapes will be constructed in OpenGL using a tessellation algorithm.

The MP2GL library provides a function for multipath prisms. In that case, paths need to be split in "positive paths" (outside paths) and "negative paths" (holes). These paths do not need to be convex, and this function can therefore be used to draw simple non convex prisms.

The user creates two arrays of paths and hands them to the `storepaths` function.

```
path pospaths[],negpaths[];
numeric np,nn;
np=2;
nn=3;
pospaths0=...
pospaths1=...
negpaths0=...
negpaths1=...
negpaths2=...
storepaths(pospaths,np,negpaths,nn,"Paths_1");
```

Then the multipath prism is obtained with

```
new_multipath_prism("Prism2","Paths_1",3cm);
```

After that, the object is used like any other to compose a scene.

Again, these paths can be drawn in an ordinary METAPOST figure environment.

The TEX logo in figure 1 was obtained by this process. First, three ordinary METAPOST paths have been defined, one for 'T', one for 'E' and one for 'X'. These paths were reduced and fed as holes of the non-reduced 'T' into the `new_multipath_prism` definition. A solid and a wire icosahedron were drawn over the TEX logo.

4.3.3 Other path-constructed objects

Other objects based on paths can be imagined, and it suffices to either add them to the OpenGL library or to the METAPOST library.

4.4 Low-level constructions

Objects can be constructed using low-level components. For instance, a cube can be constructed from its six faces as follows (figure 5):

```
% builds a square on (0,X,Y)
def build_cube_face=
  begin_convex_polygon;
    normal(0,0,-1);
    vertex(0,0,0);
    vertex(0,1,0);
    vertex(1,1,0);
```

```
    vertex(1,0,0);
  end_convex_polygon;
enddef;

beginobject("Cube");
  set_diffuse_color(1.0,0.0,0.0);
  build_cube_face; % bottom face
  PushPosition;
    Translate(1,0,0);RotateY(-90);
    set_diffuse_color(0.0,1.0,0.0);
    build_cube_face;
    Translate(1,0,0);RotateY(-90);
    set_diffuse_color(0.0,0.0,1.0);
    build_cube_face;
    Translate(1,0,0);RotateY(-90);
    set_diffuse_color(1.0,1.0,0.0);
    build_cube_face;
  PopPosition;
  PushPosition;
    RotateX(-90);Translate(0,-1,0);
    set_diffuse_color(1.0,0.0,1.0);
    build_cube_face;
  PopPosition;
  PushPosition;
    Translate(0,1,0);RotateX(90);
    set_diffuse_color(0.0,1.0,1.0);
    build_cube_face;
  PopPosition;
endobject;
```

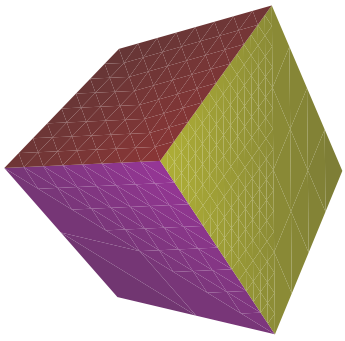

Figure 5: The cube entirely built within MP2GL.

This, however, creates an object which is not parametric. This isn't very problematic for the cube, as we can still scale it afterwards. However, other objects benefit from being parametric. There is currently no extra support for such objects, as most of these can either be obtained from paths, or can be integrated in the OL library with a minimal effort.

For instance, assume we want to add a general pyramid function having a regular polygonal base. We can build any instanciation, and this can be enough, but we could also define the OpenGL function:

```
void create_pyramid(GLint n,GLfloat r,GLfloat h) {
  ...
}
```

and then call this function from within ML by adding:

```
def new_pyramid(expr name,n,r,h) =
  ...
enddef;
```

4.5 More on wire frames

As it was shown previously, a number of objects have a wireframe variant. Other such objects can be created.

Moreover, whenever an object is made of lines, we can set the line width, and also the stippling properties (figure 6).

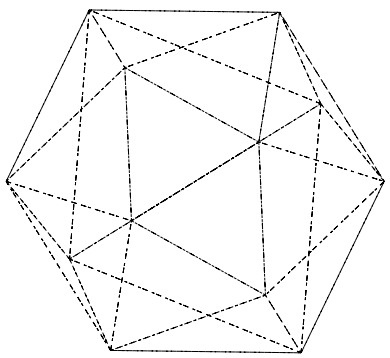

Figure 6: An icosahedron with line stippling.

4.6 More complex surfaces — NURBS

NURBS (*Non-Uniform Rational B-Splines*) provide a way to have 3D Bezier curves or surfaces. We haven't implemented an interface to them yet, but there isn't any problem doing it.

4.7 Lights and colors

There is currently a very simple support for colors in MP2GL. We can set the color when there is no lighting and these colors are used for wire frames.

When lighting is on, we can define the material properties of the next surfaces. These material properties follow the OpenGL model, and cover the emission properties, the diffusion properties (which account for the main color effect), the specular properties, the ambient properties and the shininess.

Similarly, light support is very primitive on the METAPOST side. A scene is currently lit by two lights, one at a fixed position, another on the camera. There is no support to add lights from within METAPOST, but it could easily be added. It is also easy to change the lights within the OpenGL code.

5 3D Equations

METAPOST comes with a very convenient resolution of linear equations. This feature makes it possible to use a declarative approach for a number of drawings. For instance, in order to compute the intersections of the medians of a triangle (figure 7), one can do:

```
beginfig(1);
  pair A,B,C,D,E,F,I;
  A=origin;
  B=(6u,0);
  C=(2u,4u);
  D=.5[B,C];E=.5[C,A];F=.5[A,B];
  I=whatever[B,E]=whatever[A,D];
  draw A--B--C--cycle;
  draw A--D;draw B--E;draw C--F;
  pickup pencircle scaled 3pt;
  drawdot I;
endfig;
```

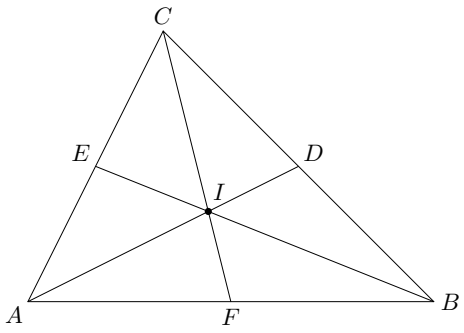

Figure 7: A triangle and its medians.

Imagine now that we want to do the same in space, say with a tetrahedron. Can we do that?

The answer is yes. And we can use the `Point` structure for the equations! Like above, we first do the computations, before drawing anything. We start with four points, the vertices of the tetrahedron.

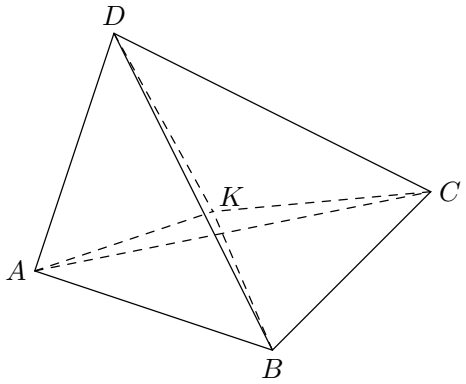

Figure 8: A tetrahedron and its center.

```
Point A,B,C,D; % vertices
Point E,F,G,H,I,J; % edge middles
Point K; % tetrahedron middle

A=(0,0,0);
B=(1,0,0);
C=(0.3,1,0);
D=(0.5,0.5,1);

E=.5[B,C];
F=.5[C,D];
G=.5[B,D];
H=.5[A,D];
I=.5[A,B];
J=.5[A,C];

K=whatever[G,J]=whatever[H,E];
```

Now, we have the center of the tetrahedron, and we can split it in four smaller tetrahedra (A, B, C, K), (C, B, D, K), (A, C, D, K) and (B, A, D, K). We can use the predefined function for tetrahedra:

```
new_tetrahedron("t1",A,B,C,K);
new_tetrahedron("t2",C,B,D,K);
new_tetrahedron("t3",A,C,D,K);
new_tetrahedron("t4",B,A,D,K);
```

and later:

```
begin_scene;
```

```
  use_object("t1");
  use_object("t2");
  use_object("t3");
  use_object("t4");
end_scene;
```

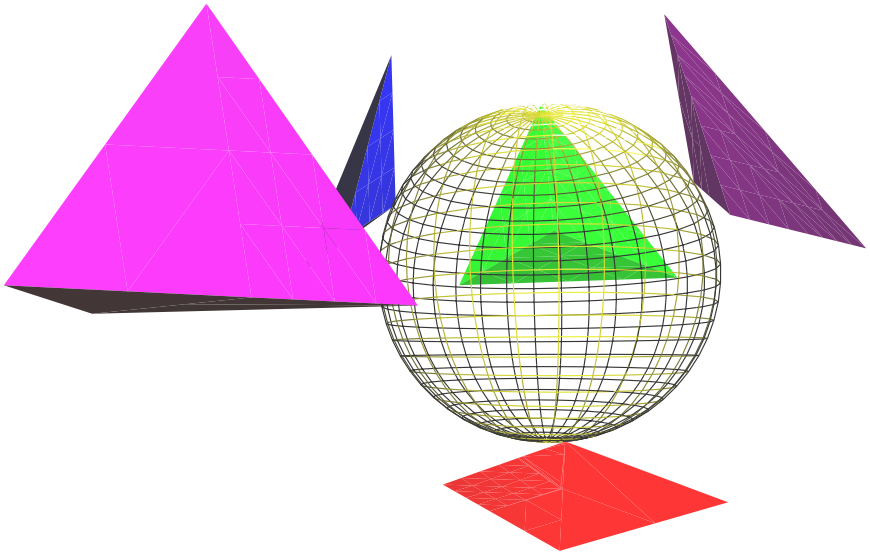

Figure 9: A scene where the tetrahedron shown left has been split in the four tetrahedra at the right, and the sphere is going through one vertex of each tetrahedron.

The tetrahedra are still bound together. In order to split them, we introduce translations from the center:

```
begin_scene;
  PushPosition;
    TranslateV(K-D);
    use_object("t1");
  PopPosition;
  PushPosition;
    TranslateV(K-A);
    use_object("t2");
  PopPosition;
  PushPosition;
    TranslateV(K-B);
    use_object("t3");
  PopPosition;
  PushPosition;
    TranslateV(K-C);
    use_object("t4");
```

```
  PopPosition;
end_scene;
```

The initial point K has now been split into four points. As an interesting 3D geometry exercise, we can use these four points in order to find the sphere going through these points, and draw it. This is done as follows. Let W_1, W_2, W_3 and W_4 be the four vertices stemming from K:

```
Point W[];
W1=K+(K-A);
W2=K+(K-B);
W3=K+(K-C);
W4=K+(K-D);
```

The center of the sphere is obtained by first finding a median plane on which the center lies, then a line, then the center itself, by a total of three plane intersections, each being handled with two `whatever`s each. This is summarized below.

```
Point V[];
V1=.5[W1,W2];
V2=crossproduct(W2-W1,W3-W2);
V3=crossproduct(W1-W2,V2);
V4=.5[W2,W3];
V5=crossproduct(W3-W2,V2);
V6=.5[W3,W4];
V7=crossproduct(W3-W2,W4-W3);
V8=crossproduct(W4-W3,V7);
V9=V1+whatever*V2+whatever*V3
  =V4+whatever*V2+whatever*V5
  =V6+whatever*V7+whatever*V8;
```

In the three previous lines, it should be noted that the six `whatever`s represent six different values!

The last point, `V9`, is the center of the sphere. The sphere itself can be drawn by inserting the following code in the scene:

```
TranslateV(V9);
wire_sphere(norm(W1-V9),30,30);
```

where `norm(W1-V9)` is the radius of the sphere, and 30 is both the number of parallels and meridians.

Equations also allow us to set some coordinates separately:

```
Xpart A=0.4;
```

or we can define equations on certain components only:

```
Xpart Q = Ypart U;
```

As usual in METAPOST, a value can be refreshed with `whatever` (this time with an assignment):

```
Xpart A:=whatever;
```

A `Point` can be refreshed like this:

```
A:=(whatever,whatever,whatever);
```

(like above, the three `whatever`s are different unknowns)

6 Text support

6.1 Basic text

Text can be added to a 3D object, but we currently have to provide a 3D position. The position of the text will depend on the point of view. An example is shown in figure 10. TEX labels are obtained by generating two files: a PostScript file containing only the drawings, and a LATEX picture environment with the labels. Labels are obtained via GL2PS and can be positioned in different ways.

When doing 2D graphics, an incremental approach can be used, whereby the final drawing is obtained by a number of modifications to the original source. This can also be achieved in 3D, but it is sometimes necessary or useful to work on the output, and therefore to be able to modify labels and their positions within the TEX file. For instance, it can be necessary to replace a label by a shifted label and an arrow. The current version of MP2GL does not yet support this feature, but a future version may produce an additional TEX/METAPOST file, which would be superimposed to the PostScript output and the labels, perhaps by the use of a simple postprocessor.

6.2 More complex marks

The usual 2-dimensional drawings have both curves, labels, but also additional elements such as dots, arrows, angle marks, textured text, etc.

These elements can all be added to a 3-dimensional drawing, provided we know the final 2-dimensional positions. However, this is often not the case. GL2PS only provides for labels at specific positions, not for more complex elements such as arrows connecting two points.

There are two simple solutions to this problem.

1. the point of view can be frozen from within MP2GL, in which case MP2GL would know exactly where a point is projected; this would require adding support for the projection transformation, something that wouldn't be difficult;

2. a less constraining solution would be to introduce fake labels which would make it to the PostScript or TEX output, and these labels could then in turn be used to specify paths (possibly in METAPOST), arrows, etc.; we would typically need two runs of METAPOST, the first for the main (fake) labels, the second for the second layer; possibly there could even be obfuscated cases needing more than two runs.

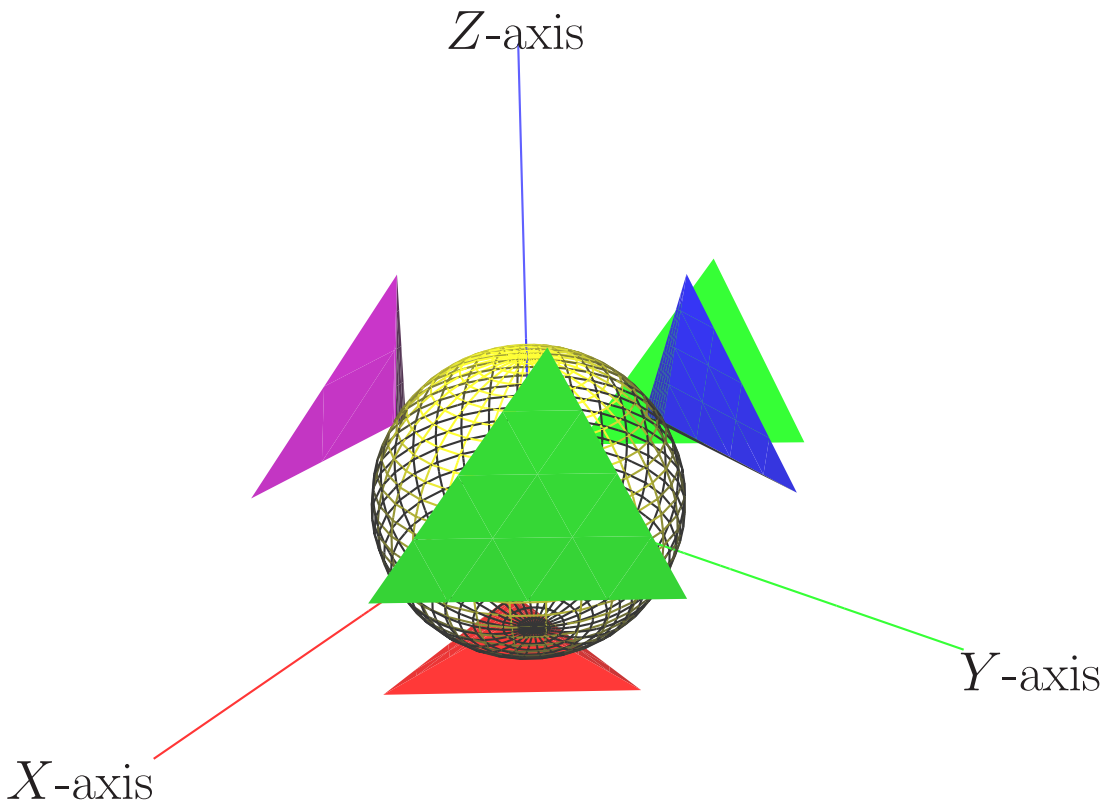

Figure 10: A scene with TEX labels. Each label is positionned differently with respect to the ends of the segments. We didn't draw arrows, but a solution for the representation of vectors in space could be to use small cones.

7 Animations

3-dimensional objects appear differently when seen from different points of view, and therefore a 3D scene is deprived of a lot of interest if it cannot be animated in some way.

The most elementary form of animation is obtained by choosing the point of view at the same time as the scene is constructed. When we need different points of view, this then requires either a tedious repetition of the processing, or some automation taking advantage of the camera path. This approach was the one used in our first `3d` package, where a series of PostScript files could be produced, by slight changes of the camera parameters (position and orientation). A similar approach is also used by the FEATPOST package.

Although such approaches are most useful in certain cases, a more natural approach is also needed, where the scene is merely animated by a high-level interface such as a mouse or a keyboard.

MP2GL provides this kind of interface, allowing the user to examine the scene under various angles and choosing the best configuration for a screenshot (PS or JPEG). Specifically, the user can:

- get closer (`PgUp`) or farther (`PgDn`) from the scene;
- change the orientation (roll: `<`, `>`, pitch: `Up`, `Down`, yaw: `Left`, `Right`);
- rotate the whole scene around the center with the mouse;
- change the field of view (‘`.`’ and ‘`,`’);
- and save the scene as a PS file (‘`s`’) or a JPEG file (‘`j`’).

The default animation uses a perspective projection, but this could easily be changed, and even toggled through a key. It could also be selected from METAPOST in the future.

This is, of course, a minimal interface.

We could, and actually can, go further, in that the animation would not only be that of the user, but an internal animation. The objects which have been created can actually interact in various ways. For instance, we can have a wheel turn, and this can produce a piston translation. Such animations are not directly supported, but it is currently sufficient to create the objects, and then to make slight modifications to the OpenGL code in order to obtain the internal animations.

The OpenGL code can also be easily extended to produce a motion of the camera and at the same time generating a bitmap at regular intervals. Various tools make it then possible to produce MPEGs (for instance `ppm2mpeg`), and other tools can be used to edit them (for instance `cinelerra`).

8 Limitations: textures, blending, ... and other advanced features

A number of METAPOST features are not handled, at least not outside METAPOST. For instance, there is no support for special pen shapes [5]. However,

[5] It should be remarked here that the author of 3DLDF (see § 9.6) intends to define 3D pens, such as sphere and cube balls.

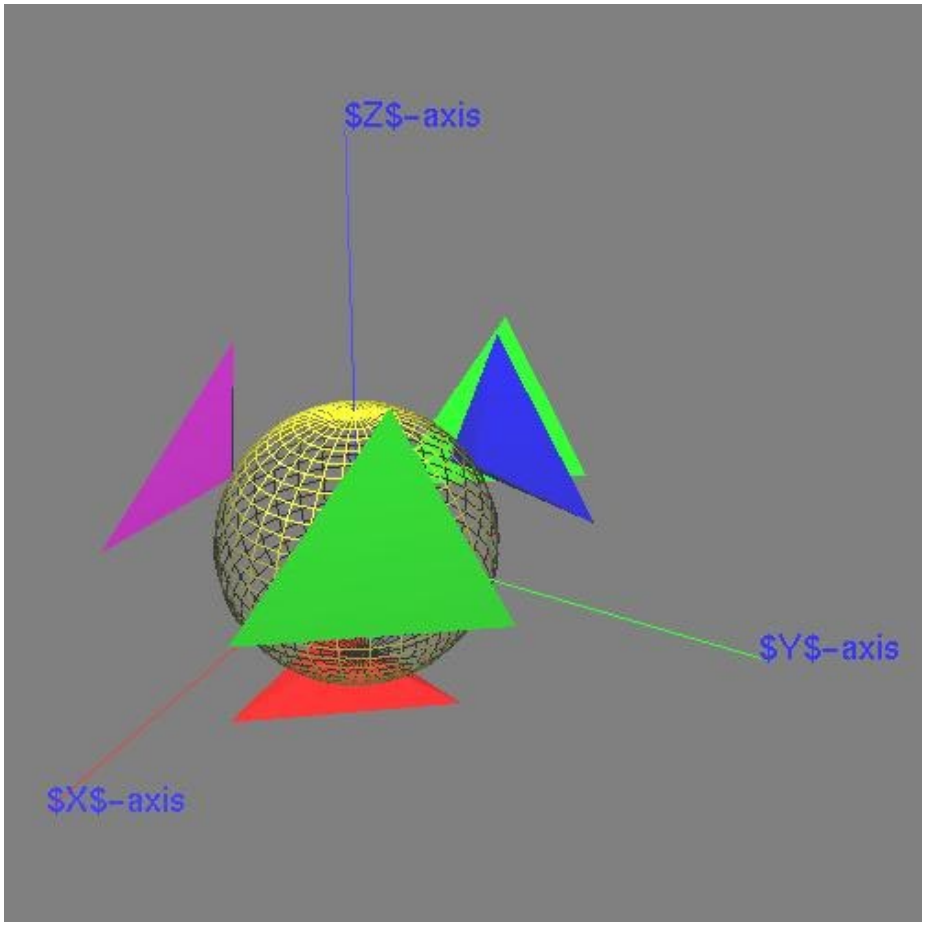

Figure 11: JPEG output of the scene shown previously. The labels are the 'raw' TEX labels, shown as Helvetica strings in orthographic projection.

if MP2GL is extended to the point where a METAPOST run can create an additional layer to a 3D drawing, then such METAPOST features could come into play.

The main intrinsic 2D feature in use by MP2GL is the `path` type, but even in this case, we are only making an elementary use of it. So far, paths are considered as polygons, but this could change if we start generating NURBS.

There is no support for textures in GL2PS, and anyway textures may be thought of inherently non-vectorial. As a consequence, there is no support for textures in MP2GL.

In principle, transparency or blending is supported by GL2PS when its output is PDF. This feature, however, is not yet supported by MP2GL.

Since shadows are by default not handled in OpenGL, they are also not covered by MP2GL. They could be handled, if implemented in OpenGL.

A 3-dimensional version of METAPOST should also be able to compute intersections, both real and projected. This is, of course, difficult, and we haven't made any attempt to solve this problem. This doesn't mean that the problem cannot be solved, but it remains to be done, either by a representation of the objects in OpenGL, or in METAPOST.

Finally, there is currently no support for a CSG [6] construction of objects, but a CSG OpenGL library could be linked to MP2GL and offer tremendous new possibilities.

9 Related work

Our work is not isolated and there have been a number of attempts at handling 3D with or around METAPOST over the years. One of the concerns of these packages has been to obtain a smooth integration with TEX, and in particular to retain the vectorial characteristics of the output.

A number of commercial products, and other products independent of TEX, produce vector output, but we will only survey the main TEX-related tools [7]. As it will be readily visible, these packages all have many features which are currently not present in our system. However, many of these features could be added to MP2GL.

9.1 Our own METAPOST `3d` (aka 'mp3d') package

This package was described in TUGboat [11] and was later extended with features for space geometry [12]. The package provided the basic tools for the construction of 3D scenes made of segments and faces. Projections could be either pespective or parallel, but there was no support for hidden faces removal. The `3d` package was created with 3D animations in mind, and in particular (but not exclusively) the animation of regular polyhedra.

The package was used by Denis Barbier and Sami Alex Zaimi for some experiments. We have also implemented extensions for handling parametric

[6] CSG (Constructive Solid Geometry) refers to booleans operations (union, intersection, etc.) used for the construction of objects.

[7] A review of several tools has already be given recently by the author of FEATPOST [5]. We would like to remark that this review oversimplied the features of our `3d` package and focused mainly on the *application* to convex polyhedra.

curves, surfaces and in particular revolution objects. These extensions were not released because they were limited by the absence of a hidden parts removal algorithm, which we were reluctant to code in METAPOST.

9.2 PSTricks-3D

PSTricks was written by Timothy van Zandt and its maintenance was stopped around 1993, but over the years many extensions were added, in particular by Denis Girou. The number of these extensions shows that the PSTricks community is very active and that the PSTricks model appears very fruitful. Several extensions handle 3D objects [3, 4, 15, 16, 9, 17].

9.3 m3dplain

Anthony Phan made a number of very interesting experiments over the years. He borrowed our ideas for the creation of animations, but the rest of his package is different and aimed at the manipulation of various mathematical objects, such as polyhedra, molecules, fractals, etc. In part of his code, he tries to implement a limited z-buffer, as well as PHIGS syntax. Phan has also worked on simulating transparency in METAPOST.

Sadly, there is no documentation available, other than the source code.

9.4 MetaGraph3D

This package is a `Java` interface for constructing simple 3D scenes interactively. The documentation available is very scarce or in a form which is unfortunately not easily searchable [8].

9.5 FEATPOST

The FEATPOST package by Luís Nobre Gonçalves is written in METAPOST and provides 3D functionality [5, 6]. As the author describes it, he wrote it in METAPOST because he wanted to keep the METAPOST machinery.

Like MP2GL, FEATPOST uses the `color` type for points [9]. Several projections are provided, including a fish-eye projection [10]. The hidden object removal is done by sorting the objects by distance from the point of view. There are therefore also cases where the objects will not be drawn correctly.

The package comes with a large number of macros for various physical applications. For instance, it has a provision to draw triangular or hexagonal meshes. It can also produce non-interactive animations, by a procedure similar in spirit to the one used in our `3d` package: METAPOST outputs are transformed in bitmaps with `netpbm` and these bitmaps are merged into an MPEG file.

The author of FEATPOST is now contemplating a reimplementation of his macros into 3DLDF (see below), because of the numerical advantages the latter provides.

[8] In spite of our efforts, we were not able to locate an independent documentation file. It is hoped that future version of MetaGraph3D will contain *separately* (that is, not hidden within an archive file) a PDF file describing the whole system.

[9] Our `3d` package doesn't use that type, although it was suggested to us in 1997 [11].

[10] In MP2GL, such a projection is naturally obtained when a large field of view is chosen.

9.6 3DLDF

3DLDF is a 3-dimensional drawing software with a METAPOST output created by Laurence D. Finston [1]. The program is written in C++ using CWEB. As of version 1.1.5 (January 2004), the input code had to be written in C++ and then compiled [11].

In 3DLDF, there is a `Point` class and a `Point` is subject to the usual operations (translations, rotations, etc.). Similarly, there is a `Transform` class for storing 4×4 transformation matrices on homogeneous coordinates. Transforms can be applied to `Point`s, inverted, etc.

3DLDF has provisions for drawing `Point`s and labelling `Point`s. It also has a 3D `Path` class, similar to METAPOST's `path` but in 3D. These paths can be drawn and filled.

The system provides a number of plane geometric figures (polygon, rectangle, circle, etc.). There are also a number of solid figures: cuboids, polyhedra,

It allows the specification of the projection. For the perspective projection, 3DLDF needs the specification of a camera [12].

3DLDF's hidden surface algorithm currently doesn't work for intersecting surfaces. 3DLDF has four different ways to sort objects. It can also find the intersection of a few non-arbitrary paths, such as two polygons, a line and a polygon, two ellipses, etc. So far, 3DLDF doesn't have support for lights, shading, etc.

3DLDF does not have linear equations solving in the METAPOST style, although there are plans to add equations. It also doesn't have macros.

Current plans are to implement NURBS in 3DLDF, but NURBS do exist in OpenGL and could easily be interfaced in our work.

3DLDF appears as an interesting approach, somewhat symmetrical to ours. It does however remind us of our two major previous METAPOST packages, `3d` [11] and METAOBJ [13, 14]. 3DLDF recodes a number of objects, such as polyhedra, as we did (modestly) in our `3d` package in 1997. 3DLDF also has an object-oriented approach, not unlike what we did for plane objects in METAOBJ.

3DLDF is bound to code a lot of features which are already present in OpenGL, but it may also produce code which can be reused in our OpenGL library. For instance, the author of 3DLDF plans to code a number of exotic polyhedra, which should be easy to recode in C. Actually, if our C part were written in C++, we could even output 3DLDF code!

3DLDF is also a very interesting approach in that its author is trying to extend to the third dimension features that may not be that easy to transfer, such as pen shapes. In a way, 3DLDF is aiming at the most orthodox extension of METAPOST, albeit by using an external processing stage.

Finally, one problem in METAPOST is the problem of the numerical values. This is one reason why the author of 3DLDF created his package. That is, by working in C++, the various computations associated to intersections, projections, etc., are easier and less constrained.

[11] The use of an external processor has also recently been applied to 2D plots by Brook Moses [10].

[12] In MP2GL, this is not done at the METAPOST level, but at the OpenGL level. We could do it at the METAPOST level, but we assume the user wants to animate the object and find the most convenient location. Nothing prevents him/her of setting the projection as perspective, parallel, isometric or axonometric.

The MP2GL approach, on the contrary, uses very little computational features of METAPOST, apart from a few matrix operations, often confined to very small values. Moreover, objects can always be created at a small size, and then scaled within OpenGL. In the rare cases where an overflow would occur, a special mode could be selected which would turn off the matrix computations on the METAPOST side. In most cases, these matrix operations are not needed in METAPOST.

10 Conclusion

Our package is still in its infancy and our purpose has only been to explore its feasability. In particular, it should be noted that most of the features presented here are still unstable, and that function names are likely to change. Technical problems can arise, either because OpenGL is not always correctly implemented (GL2PS uses the feedback buffer of OpenGL and this buffer may lack some of the elements it is supposed to contain), or because GL2PS is still in development, or for other reasons.

But it does anyway seem to us that we have achieved our goal and that it is or will be very easy to produce a great variety of high quality 3D graphics with MP2GL. Moreover, there are several possible future directions, which could be developped, in particular through a collaborative effort:

- The MP2GL code could be extended on the METAPOST side in order to cover a larger subset of OpenGL;
- the OpenGL library could be extended with various objects useful from a METAPOST perspective.

But, for many specific applications, some of the reviewed tools may both be more adapted, more complete, and maybe easier to use than MP2GL, especially with respect to text handling when all of the control lies within METAPOST. Our tool should therefore be seen as a new possibility to produce not only certain 3D graphics, but also to *prototype* objects which can be used beyond an article, for instance as part of a complex animation.

11 Acknowledgments

The author would like to thank Christophe Geuzaine for his kind help during the development of MP2GL.

References

[1] Laurence D. Finston. 3DLDF User and Reference Manual. Manual edition 1.1.5.1, January 2004.

[2] Christophe Geuzaine. GL2PS: an OpenGL to PostScript printing library, 2004.

[3] Denis Girou. The 'pst-ob3d' package: A PSTricks package for three dimensional basic objects, 2002.

[4] Denis Girou. The 'pst-gr3d' package: A PSTricks package for three dimensional grids, 2004.

[5] Luís Nobre Gonçalves. FEATPOST and a Review of 3D METAPOST Packages. In *Proceedings of TUG 2004: TEX, XML, and Digital Typography*, volume 3130 of *Lecture Notes in Computer Science (LNCS)*, pages 112–124, Xanthi, Greece, 2004.

[6] Luís Nobre Gonçalves. *FEATPOST macros*, 2004.

[7] John D. Hobby. A User's manual for MetaPost. Technical report, AT&T Bell Laboratories, Murray Hill, New Jersey, 1992. Computing Science Technical Report 162.

[8] Donald E. Knuth. *The METAFONTbook*. Reading, MA: Addison-Wesley, 1986.

[9] Manuel Luque and Herbert Voß. 3D views with `pst-vue3d`, 2005.

[10] Brook Moses. MetaPlot, MetaContour, and Other Collaborations with METAPOST. In *Proceedings of Practical TEX2004*, 2004.

[11] Denis Roegel. Creating 3D animations with METAPOST. *TUGboat*, 18(4):274–283, 1997.

[12] Denis Roegel. Space geometry with METAPOST. *TUGboat*, 22(4):298–314, 2001.

[13] Denis Roegel. *The METAOBJ tutorial and reference manual*, 2001.

[14] Denis Roegel. METAOBJ: Very High-Level Objects in METAPOST. In *Proceedings of TUG 2002*, Trivandrum, India, 2002.

[15] Herbert Voß. Three dimensional plots with `pst-3dplot`. *TUGboat*, 22(1):319–329, 2001.

[16] Herbert Voß. 3D plots: PST-3dplot v1.63, 2005.

[17] Herbert Voß. *PSTricks. Grafik mit PostScript für TEX und LATEX*. Lehmanns, 2005. [2nd edition].

[18] Mason Woo, Jackie Neider, Tom Davis, and Dave Shreiner. *OpenGL Programming Guide, Fourth Edition*. Addison-Wesley, 2004.