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.

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.

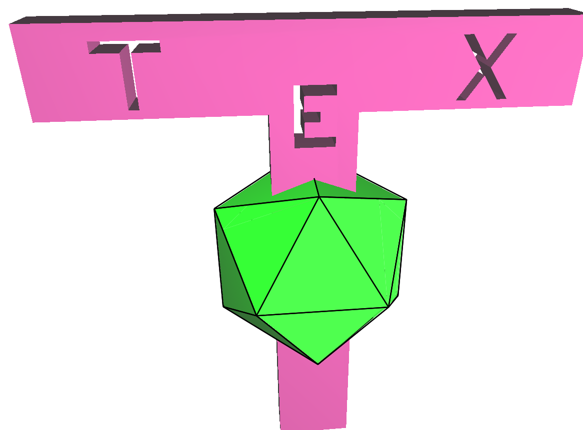

Figure 1: A TEX logo produced with MP2GL. This scene combines an object built from METAPOST paths (the 'T' with the three holes) and a predefined OpenGL icosahedron in solid output, to which a slightly larger wire frame icosahedron was superimposed.

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].

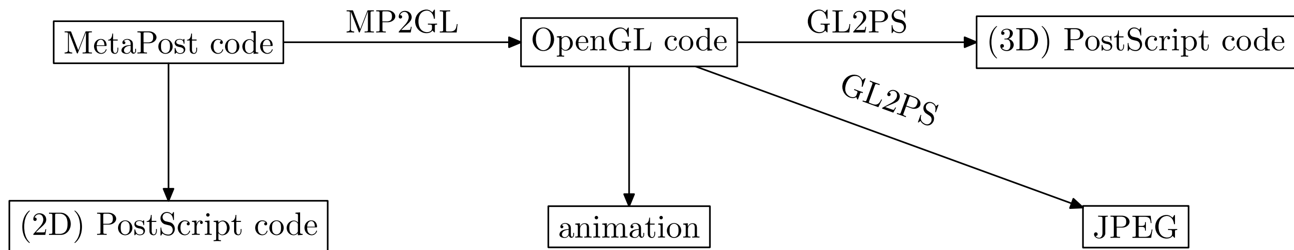

Figure 2: From METAPOST to 3D-PostScript.

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;

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

- 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.

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.