LaTeX in 3D: OpenDX annotations

J. P. Hagon

Abstract

We present a system, *DXfontutils*, for adding high-quality annotation to OpenDX objects using LaTeX as the typesetting engine. The system utilizes native OpenDX fonts converted from original outlines (TrueType, OpenType or PostScript) using the author's font2dx translator. Also we demonstrate how OpenDX can be used as a tool for producing special effects with OpenDX text elements which have been typeset by LaTeX.

1 Introduction

OpenDX [2] is a general purpose data visualization system similar to Khoros, IDL, AVS, Amira and others. As its name implies it is open source software and freely available. It was formerly a product from IBM known as Data Explorer. IBM released the Data Explorer source code for public use under a special licence in 1999.

OpenDX has an extremely versatile data model and an excellent visual programming interface. Figure 1 shows the output of a simple example. This output was produced with the visual program illustrated in figure 2. The program consists of an *Import* module which reads in the data, and an *Image* module which displays the data.

The modules contain input and output tags. In this case, the output tab from *Import* is connected to the input tab of *Image*. The connection is made simply by clicking and dragging with a mouse. Clicking on the *Import* icon produces an entry box in which the name of the import file is typed. The appropriate tab then appears in the closed form shown in figure 2 within the *visual programming editor* (VPE) indicating that the parameter has explicitly been set. In fact, I/O tabs can be hidden to simplify the layout — *Image* has many more input tabs than the one shown here. Note also that there can be more than one output tab — the three output tabs from *Image* provide information about the rendered object, the viewing camera and the viewing position. Here is the program:

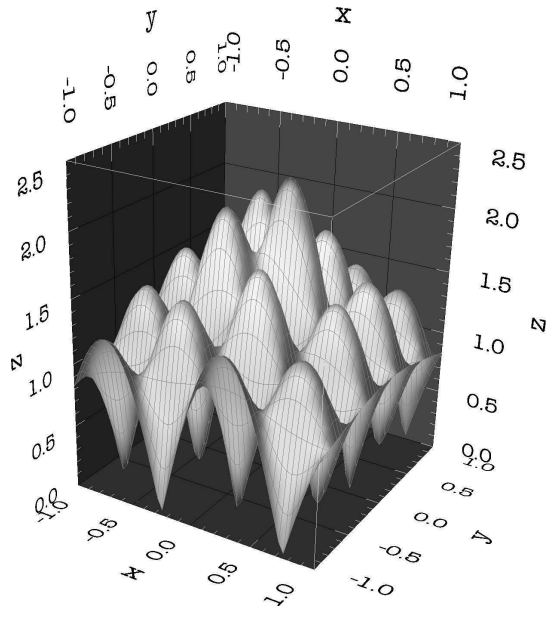

Figure 1: A simple 2D function plot in OpenDX.

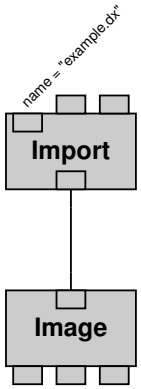

Figure 2: The OpenDX visual program which produced figure 1.

The VPE can be used to build large-scale interactive GUIs for specialized data analysis. Furthermore, the user can write custom *modules* (plugins usually written in C) and *macros* (a visual program combining other modules and macros).

The writing of modules is facilitated by a *Module Builder* interface and all of the standard OpenDX modules are available via a set of C libraries for skilled programmers. In fact, it is possible to produce an application using an external GUI library combined with the OpenDX graphics and rendering libraries. OpenDX can even be run in *script mode* using its own scripting language. Visual programs created through the VPE are stored in this scripting language.

Although OpenDX is a rather intimidating piece of software, there is extensive documentation, active user forums, commercial third party support and an introductory, tutorial based book [9]. Many useful third party macro and module libraries are available [2] for fields as wide-ranging as geophysics, medical imaging, quantum chemistry, biology, astronomy, social science, finance and engineering.

2 OpenDX Font Format

Two types of font are supported by OpenDX — 'line' fonts and 'area' fonts. The former are similar to the fonts that were common on pen-plotter output devices some years ago. Such fonts are still useful for screen display where hard copy quality is not important since they can be rendered very quickly. They are not our concern here and will not be discussed further. 'Area' fonts rely on filled polygons and are therefore capable of much higher quality than line fonts. Unfortunately there is just one such font supplied with the standard OpenDX release — the *Pitman* monospaced font.

2.1 Area Font Structure

Most outline fonts are fairly simple in concept — inner and outer boundary lines (often defined in terms of cubic splines) define an area to be filled. The spline defining the inner outline is opposite in direction (clockwise/anti-clockwise) to a spline defining an outer boundary. PostScript and TrueType fonts have opposite conventions in this regard.

Things are more complicated with an OpenDX area font. First, polygons rather than splines are used to define the outlines. Second, areas to be filled are not defined with clockwise/anti-clockwise polygons; instead, the required area must be *triangulated* to create an *area mesh*. These concepts are illustrated in figure 3.

In an OpenDX font file, the boundary polygons are defined through a set of positions and the connections defining the triangulated mesh are defined as a set of integer triples, each integer referring to a particular position. For example, a simple hyphen (essentially just a rectangle) might be defined in an OpenDX font file as shown in figure 4.

OpenDX fonts have exactly 256 entries, making them equivalent to *8-bit* fonts commonly used today. There is no flexibility in the format to allow for larger (or smaller) fonts. The files themselves adhere to the OpenDX data model and can be in text or binary format. The binary format is generally more compact. The official description of the font format can be found in the *OpenDX User's Guide* [1].

3 The Font Conversion Method

In order to get from, say, a Type 1 PostScript outline to an OpenDX font in the form illustrated in figure 4 requires roughly the following steps:

1. Obtain the boundary points corresponding to all inner and outer lines for each character in a font.

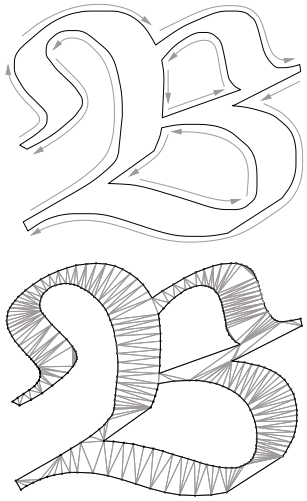

Figure 3: Comparing methodologies for PostScript/TrueType and OpenDX fonts. The letter $\mathfrak{B}$ of the AMS Euler Bold Fraktur font as represented in PostScript form (upper figure) and OpenDX form (lower). In the upper diagram there are three boundaries defined, the outer one going clockwise and the two inner ones going anti-clockwise. The lower diagram shows how a filled area is represented in a OpenDX font using a triangulated mesh bounded by the same outlines as in the upper diagram.

```
object "positions_hyphen" class array type
float rank 1 shape 3 items 4 data follows
 0.276  0.187 0.0
 0.011  0.187 0.0
 0.011  0.245 0.0
 0.276  0.245 0.0

attribute "dep" string "positions"
#
object "connections_hyphen" class array type
int rank 1 shape 3 items 2 data follows
2 1 0
0 3 2

attribute "ref" string "positions"
attribute "element type" string "triangles"
attribute "dep" string "connections"
#
object "hyphen" class field
component "positions" value "positions_hyphen"
component "connections" value "connections_hyphen"
attribute "name" string "hyphen"
attribute "char width" number 0.333
attribute "series position" number 45.000000
```

Figure 4: An entry for the hyphen character from a native OpenDX font file. Note the '`char width`' and '`series position`' attributes.

2. Triangulate the appropriate regions and obtain a set of connections for each character.
3. Output positions, connections and width information for each character in OpenDX font format.

To perform this task, we make use of three software packages, all of which are freely available. The packages are fontforge [10], pstoedit [6] and Triangle [8]. A brief description of each package follows, along with an explanation of its contribution to the OpenDX font conversion process.

3.1 fontforge

This remarkable application, by George Williams, is an outline font editor capable of creating and editing both PostScript and TrueType fonts. It is similar to commercial font editors such as Fontlab or Fontographer and provides much of the same functionality. It is available for multiple platforms and can be compiled from source if required. Further details may be obtained from the fontforge web site [10].

For all its many features, only limited use is made of fontforge in the OpenDX font production procedure. In particular it is used to obtain the following vital font information:

- The official name of the font.
- The name, ASCII code and widths of each font character. This is stored temporarily in one file for each font.
- An Encapsulated PostScript (EPS) rendering of each character in the font for subsequent processing by pstoedit.

The above procedure can be automated via fontforge's own scripting language.

3.2 pstoedit

Written by Wolfgang Glunz, this is a well-established and very useful package which converts PostScript (and PDF) files into a variety of vector formats.

pstoedit is used to extract the boundary point information for each character by converting the eps files generated by fontforge into gnuplot [4] commands. The gnuplot driver was chosen because its output is in a very convenient form for subsequent processing — the boundary points being returned as a column of (x, y) pairs. When a full closed curve is completed, this is indicated by a blank line and the next set of points started, if there is more than a single closed curve for the given character. The generated output file can then be loaded into gnuplot and viewed via the gnuplot command `plot <file>` or alternatively `plot <file> with lines` if you want to see the points joined up. Some gnuplot output for the Euler Fraktur character discussed earlier is shown in figure 5. The remaining task is to add the connection information.

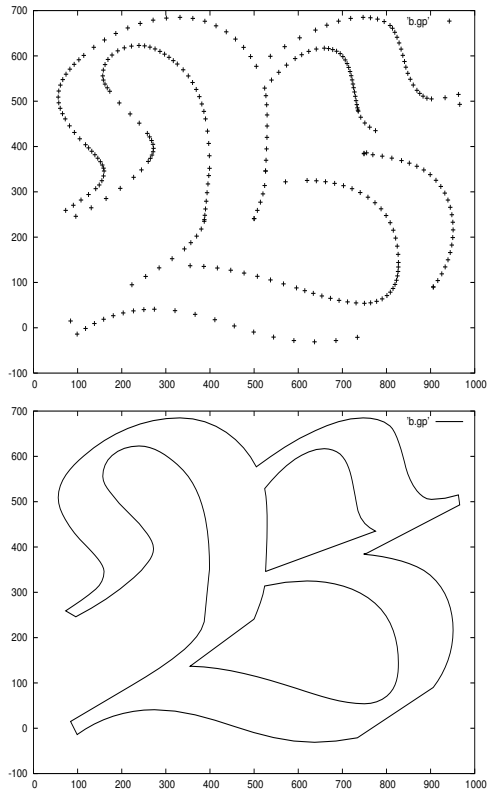

Figure 5: gnuplot rendering of the Euler Fraktur $\mathfrak{B}$. The upper diagram shows a points plot, the lower shows a line plot (each point is connected to the next point (as ordered in the input file created by pstoedit).

3.3 Triangle

This program is the work of Jonathan Shewchuk. It produces a triangulated mesh, given a set of input points and constrained segments — i.e. the boundary outlines of each font character. Triangle is a very efficient program and makes the task of triangulation relatively straightforward. The input required is a simple text file (referred to as a `.poly` file — see figure 6) with entries supplied for:

- A list of vertices — these are the nodes which form the boundary outlines for each character. They take the form of (x, y) pairs.
- A list of segments, i.e. the connection information needed to construct the boundary polygon. These are a list of integer pairs corresponding to

```
# Vertices, dimension, attributes, boundary markers
#
286 2 0 0
#
# Vertex no., x, y
#
0 0.906 0.09
1 0.734 -0.021
      .
      .
284 0.528219 0.394703
285 0.527145 0.369736
#
# Segments, boundary markers
#
286 0 0
#
0 0 1
1 1 2
2 2 3
  .
  .
192 192 193
193 193 0
194 194 195
  .
  .
243 243 244
244 244 194
245 245 246
  .
  .
284 284 285
285 285 245
#
# Holes
#
2
#
0 0.640961 0.323633
1 0.6685625 0.6155
```

Figure 6: A Triangle '`.poly`' file showing how vertices, segments and holes are set up. Note the termination segments which close each polyline and the two hole coordinates.

the vertices mentioned previously. Since all the vertices are correctly ordered, this list can be generated easily; and since all polygons are of the simple closed form, the last entry for a given polyline will be of the form $(n+m-1, n)$ where n is the starting vertex and m is the number of points in a given closed polyline.

- A list of 'holes', if any. These are points which lie within regions inside certain polylines which are not to be triangulated. In the case of the Fraktur $\mathfrak{B}$, it is clear that there are two interior polygons which enclose regions which are not to be triangulated. By specifying a hole point anywhere in a given region, Triangle is instructed not to triangulate that region.

Triangle produces a set of triangulated elements connecting polyline vertices from this input and stores the elements in a `.ele` file.

Vertices and segments are essentially provided by pstoedit but holes need to be calculated explicitly. As mentioned previously, the sense of a polygon (clockwise or anti-clockwise) determines if it should be filled or not. If it is not to be filled, then a hole coordinate must be placed somewhere within the polygon.

In the Type 1 PostScript format an anti-clockwise polygon is one forming an inner boundary and therefore containing a hole; for TrueType it's the other way round. A simple algorithm exists [5] for determining if a polygon is clockwise. For a closed polygon with n points $(x_0, y_0), \ldots, (x_{n-1}, y_{n-1})$, calculate the quantity:

$$\mathcal{A} = \frac{1}{2}\sum_{i=0}^{n-1}(x_i y_{i+1} - x_{i+1} y_i) \text{ with } (x_n, y_n) \equiv (x_0, y_0)$$

If $\mathcal{A} > 0$ then the polygon is anti-clockwise, otherwise it is clockwise. Once a hole polygon has been identified, any point within it serves as a hole point for Triangle.

The following algorithm is used [3] to construct an interior polygon point:

1. Identify a convex vertex v.
2. For each other vertex q do:
   (a) If q is inside avb, where a and b are the adjacent vertices to q, compute distance to v (orthogonal to ab).
   (b) Save point q if distance d is a new minimum.
3. If no point is inside, return midpoint of ab, or centroid of avb.
4. Else if some point inside, qv is internal: return its midpoint.

Application of this algorithm usually results in a hole point being set very close to a boundary segment — so close, that to the naked eye the point often seems to lie *on* the segment.

4 Putting it all together

Two Perl scripts — g2poly and font2dx — have been written to automate the above procedure. g2poly converts a gnuplot input file (generated by pstoedit) into a `.poly` file suitable for input into Triangle. Everything else is handled within the font2dx script, which in fact calls g2poly. font2dx can optionally reencode a font according to several common encoding schemes. At present font2dx outputs fonts only in

text (ASCII) format, rather than the more compact binary format.

The general form of a font2dx command is:

```
font2dx [OPTION]... FILENAME
```

where `FILENAME` is a PostScript Type 1, OpenType or TrueType font file. At present, the following options are available:

`--noclean` Don't clean up intermediate files (usually there are *hundreds* of these!) — by default these files are deleted, leaving just the generated and original fonts.

`--scale=<integer>` Attempt rescaling of the font. This can be used to correctly scale a font in cases where the default scale factor fails.

`--negate` Reverse the normal convention for inner and outer closed polygons.

`--flat=<number>` Set the pstoedit 'flat' parameter. This defaults to 1.0 and the acceptable range of values is [0.2–100.0]. This parameter controls how accurately curves in fonts are approximated by polylines — higher numbers give rougher approximations.

`--enc=<encoding>` Change font encoding. There is a choice of many pre-defined schemes and if the encoding is not one of these, then an encoding file is looked for, with the assumed name `<encoding>.enc`. Hence, many of the standard encoding schemes in TEX can also be used.

`--help` Print usage information and help.

font2dx will process only the first 256 character glyphs in a font. Modern fonts often have many more than this. If there is a 'hidden' glyph not in one of the first 256 slots, then you could try manually re-encoding the font with a tool such as fontforge prior to running font2dx.

A further issue is that OpenDX font characters have a width attribute, but no explicitly defined height. However, TEX is perfectly happy using characters which have *zero* width. In such cases, it is impossible to correctly scale such characters unless there is a corresponding height (so that scaling ratios can be calculated). font2dx therefore adds a `char height` attribute, equivalent to $\langle height \rangle + \langle depth \rangle$ enabling proper scaling even for zero width characters.

4.1 Quality Issues

It can be worth experimenting with the `--flat` option to optimize font quality. The default value for this parameter is 1 which generally produces very good quality fonts, i.e. unless the fonts are greatly enlarged, it is almost impossible to detect the polygonal character of the outlines. In fact, a value of 10 produces pretty decent results for most text fonts we have tested. Figure 7 illustrates the effect of the `--flat` parameter for the URW Times-Roman font. For exceptionally fine and detailed fonts a flat parameter of less than 1 may prove necessary.

Figure 7: OpenDX rendering of URW Times-Roman for different flat parameters: flat = 100 (top); flat = 10 (middle); flat = 1 (bottom). Even a flat value of 100 produces recognizable text albeit in effectively a different font!

There is a trade-off between font size and quality with smaller flat parameters leading to larger file sizes, as might be expected. However it is generally true that as flat parameters get very large, the space saved is not worth the enormous loss in quality. This is illustrated in Table 1 where it is clear that there is not much space to be gained in going from a flat parameter of 10 to a flat parameter of 100 in the case of URW Times-Roman — but there is an enormous loss of quality. In the rest of this paper, we use fonts generated with a flat parameter of 1.

'flat' parameter	100	10	1
URW Times	181717	254580	543254
WebOMints-GD	417797	709443	2036248

Table 1: Font file sizes (in bytes) for different flat parameters in the case of URW Times and the ornament font WebOMints-GD.

5 Annotation in OpenDX

This ability to create native OpenDX fonts from industry-standard outlines, as described above, has the potential to greatly improve annotation quality within OpenDX. It has been common for users to post-process their OpenDX-generated images with graphical editing tools such as Gimp or Photoshop in order to add text elements. Either that, or the *Pitman* font was grossly overused (because it was the only good quality font available) making many annotated images produced by OpenDX immediately

identifiable.[1] One remaining issue is the typographical quality of OpenDX annotation, particularly with regard to mathematics. This is one area where TEX can certainly help!

5.1 OpenDX *Text* and *Caption* Modules

Text within OpenDX is treated just like any other OpenDX object. It can be scaled, rotated, coloured, and manipulated in many different ways. There are two modules within the core OpenDX system which facilitate text entry and annotation.

The *Text* module allows text to be positioned anywhere in 3D space with any rotation, size and orientation. The position and height are given in world (user) coordinates.

The *Caption* module displays a caption on the screen independently of any other OpenDX objects representing the user's data. This produces text which remains in the same position relative to the screen. The position of the text is given in screen (viewport) coordinates, i.e. a position of $[0.9, 0.5]$ means 9/10 of the way along the horizontal axis and half way up the vertical axis. The height is given in pixels.

Text and *Caption* have very rudimentary typesetting capabilities. Escape sequences (using 'backslash' as the escape character) can be used to obtain characters not available on some keyboards (e.g. diacriticals) and spacing is achieved via the 'space' character (ASCII 32) of the particular font in use. Now this latter point raises a problem if one wishes to use, say, Computer Modern Roman because this font doesn't have a space character! Position 32 is taken up by the *suppress* character — -. Of course, this isn't a problem in TEX since all spacing is calculated internally — removing the need for an explicit 'space' character.

5.2 The *LaTeXText* and *LaTeXCaption* Macros

Two new OpenDX macros were developed as alternatives to the *Text* and *Caption* modules: *LaTeXText* and *LaTeXCaption*. In OpenDX a macro is a combination of modules (and other macros) and can be created through the OpenDX visual programming editor.

The above macros take LATEX commands as their main argument. The user may enter, optionally, a set of LATEX *preamble* commands if certain packages are required. Hence,

```
\\usepackage{cmbright}
```

might be entered as a preamble option if the CM Bright fonts were required. The double backslash is not an error — it's an unfortunate consequence of the previously mentioned fact that backslash itself is treated as an escape character in text arguments of the *Text* and *Caption* modules.

If there is a lot of text to be typeset, it is more convenient to supply a file containing the text rather than type the text in as an argument to a macro. For this reason, two modified versions of *LaTeXText* and *LaTeXCaption* are available, which accept a file of LATEX commands rather than a string of commands. These macros are *LaTeXFileText* and *LaTeXFileCaption* respectively. Another advantage of using these modules, in addition to their primary purpose, is that backslash characters do not need to be doubled-up. Within the VPE, the macros appear like this:

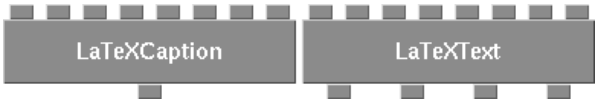

Figure 8: *LaTeXText* and *LaTeXCaption* macros as they appear in the OpenDX VPE.

LaTeXText takes the following inputs:

`latex_string` A string of LATEX commands.

`height` Height of text in user (world) coordinates.

`position` Position vector of reference point (see below) in user coordinates.

`baseline` Direction of baseline expressed as a vector.

`angle` Euler-type angle specifying rotation about the baseline axis.

`preamble` A string of LATEX preamble commands — for example, to load font definitions or special packages.

`extrusion` A scalar defining the extrusion in user coordinates. A number ≤ 0 produces no extrusion.

`reference` An integer (1–9) specifying the reference point on the formatted text object to be used for positioning. (1) refers to bottom left (the default); (2) is bottom centre; (3) is bottom right, etc., up to (9) which refers to top right.

There are 4 outputs:

`text` Complete object including extrusions and surfaces.

`nosurface` Just the extrusion (no upper or lower surfaces).

[1] Not unlike the situation some years ago where almost any document produced using TEX used the Computer Modern fonts — not because TEX was incapable of using other fonts but because at that time it was not straightforward to do so.

`top_surface` The top surface.

`bottom_surface` The bottom surface.

The four outputs allow the upper/lower surfaces and extrusion to be handled differently. For example, the upper and lower surfaces can be given different colours.

LaTeXCaption has just a single output and the following inputs:

`latex_string` A string of LaTeX commands.

`coords` An integer specifying the type of coordinates used: (1) viewport, (2) pixel, (3) world or (4) stationary. Using stationary coordinates, the text string will be attached to a particular point in world coordinates but will retain the same orientation with respect to the viewing camera.

`direction` Direction of baseline expressed as a vector.

`priority` An integer specifying how the text is layered relative to the other OpenDX objects: (−1) behind, (0) equal or (1) in front.

`position` The screen position. How this vector is interpreted depends on the value of the `coords` parameter.

`height` Height, in pixels unless stationary position, in which case world coordinates are used.

`preamble` A string of LaTeX preamble commands.

`reference` An integer (1–9) specifying the reference point on the formatted text object to be used for positioning. See description above for *LaTeXText*

The conversion of LaTeX commands to OpenDX objects is handled by two Perl scripts — dvidx and latex2dx:

dvidx is a TeX `dvi` driver program similar to dvips et al. It takes a `dvi` file as input and generates an OpenDX object. This object contains the correctly scaled and positioned characters from the OpenDX fonts converted from outline originals. It understands dvips colour specials and can output in two different OpenDX formats: a *compact* form which consists of external references to OpenDX fonts; and an *inclusive* format in which all the relevant data from the external font files is included in the output. dvidx can be used standalone to produce OpenDX output if desired. Multiple pages are handled by collecting individual pages in an OpenDX *Group* object.

latex2dx is essentially a wrapper Perl script around dvidx. It takes raw LaTeX input, produces a temporary `dvi` file and then calls dvidx to generate OpenDX output.

It is latex2dx that is actually called by the *LaTeXText* and *LaTeXCaption* macros but it is dvidx which does all the hard work.

6 The dvidx Perl Script

The writing of dvidx was made considerably easier by the use of two clever Perl packages written by Jan Pazdziora — `Font::TFM` and `TeX::DVI::Parse` [7]. The working of dvidx is roughly as follows:

1. First, run dvicopy[2] on the original `dvi` input to translate all the virtual font references to base fonts.
2. Parse the dvicopy output using the Perl package `TeX::DVI::Parse`.
3. For each font encountered, obtain the appropriate metrics from the TeX font metric (`tfm`) file using `Font::TFM`.
4. Map the base font to a raw OpenDX font and extract the appropriate characters.
5. Position and scale the character via an OpenDX rotation/translation operation (in OpenDX jargon, this is an *XForm* transformation object).

dvidx cannot read the packed font (PK) files traditionally used by `dvi` drivers and usually created (indirectly) from METAFONT source files. There is no reason in principle why it could not be made to use such files but the approach described here produces higher quality and is much easier to implement. However, it does mean that METAFONT sources which have not been converted to outline form cannot currently be rendered using dvidx.

6.1 The dvidx Map File

The mapping of raw TeX font names to OpenDX font files is done via a map file similar to (but much less versatile than) the map file used by dvips. The dvidx map file contains two columns, the first column giving the name of the raw TeX font and the second column giving the name of the corresponding OpenDX font file. It is possible that a single OpenDX font file may map to more than one raw TeX font but not vice-versa. If a raw TeX font maps to more than one OpenDX font file then the *last* entry in the map file is the one that is used.

One of the features of the dvips map file is that one can re-encode a PostScript font on the fly via a re-encoding directive within the map file itself.

[2] dvicopy is a program which is routinely available as part of all modern TeX implementations. Its primary purpose is to expand virtual font definitions. This is useful in cases where a `dvi` driver doesn't understand virtual fonts.

For example, a re-encoding to TeXBase1 is achieved within a dvips map file with the following directive:

```
" TeXBase1Encoding ReEncodeFont " <8r.enc
```

where 8r.enc is a file containing the appropriate PostScript encoding commands. This type of functionality could be added to the dvidx map file, but in many cases would be redundant. This is because OpenDX fonts always contain *exactly* 256 characters whereas Type 1 PostScript fonts generally contain 'hidden' glyphs that are not contained within the visible 256 character slots. It is often the case that the purpose of re-encoding is actually to place many of these hidden glyphs in visible slots.

Our solution to this problem is somewhat brute-force but effective. A program such as fontforge can be used to re-encode the original outline font using the required encoding file (such as 8r.enc, for example). The re-encoded PostScript font is then converted to an OpenDX font using font2dx but given a different name to the original. The convention we use, is to append the string -<enc> to the OpenDX file name. This produces map file entries like:

```
ptmri8r          Times-Italic-8r.dx
tii              Times-Italic-8y.dx
```

whereas in the dvips map file we would have:

```
ptmri8r Times-Italic
 " TeXBase1Encoding ReEncodeFont " <8r.enc
tii Times-Italic
 " TeXnANSIEncoding ReEncodeFont " <texnansi.enc
```

A similar brute-force approach can be used to deal with 'slanted' fonts created on the fly via a dvips map file entry such as:

```
ptmro8r Times-Roman " .167 SlantFont ...
```

7 Examples of Text Annotation in OpenDX

Suppose we wish to add a title to the image shown in figure 1. The normal way to do this in OpenDX would be via the *Caption* module. The visual program would look like that shown in figure 9. We have created a caption object and added it to the original object (using the *Collect* module). The result is shown in figure 10.

Now the function we are plotting is quite a complicated one:

$$z = \frac{\sqrt{\sin(\omega x)\cos(2\omega y) + 1}}{1 + \sqrt{x^2 + y^2}}$$

and we have tried to indicate the form of the function in the caption. The core facilities of OpenDX limit what we can do here and the result is both difficult to read and cumbersome to position because it consists of one relatively long line of monospaced text.

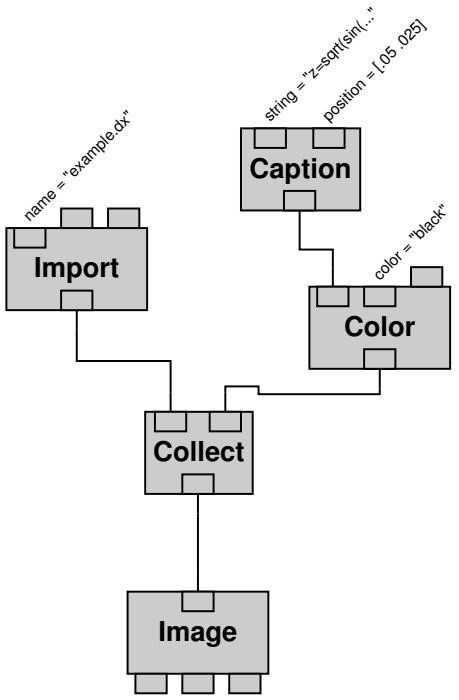

Figure 9: Modified visual program using *Caption*.

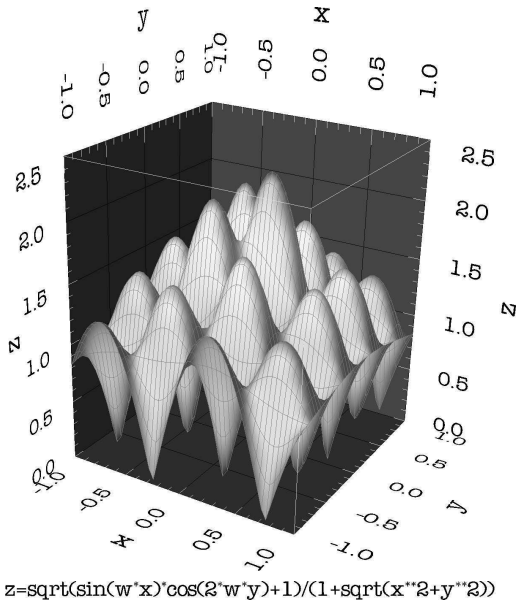

Figure 10: OpenDX-annotated figure.

So, we instead use the *LaTeXCaption* macro. In addition, to make the mathematics easier to read on-screen, we anti-alias the output using the *Text-Alias* macro. The resulting visual program is shown in figure 11. The main argument to *LaTeXCaption* is the following piece of LaTeX code:

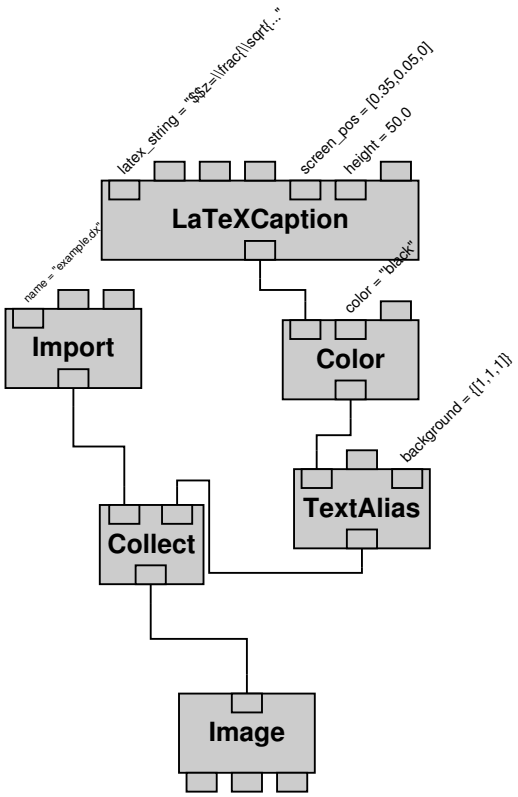

Figure 11: Modified visual program using *LaTeXCaption*.

```
$$z=\\frac{\\sqrt{\\sin(\\omega x)
   \\cos(2\\omega y)+1}}{1+\\sqrt{x^2+y^2}}$$
```

where, as mentioned earlier, the backslashes have been doubled because backslash is an escape character in OpenDX. *LaTeXCaption*'s other arguments include an orientation vector, a position vector and optional LaTeX preamble text. Standard OpenDX modules can be used to make other modifications, such as colour changes.

8 Special Effects with *LaTeXText*

LaTeXCaption provides flat 2D screen annotation within a 3D OpenDX space. *LaTeXText* has similar functionality except that it operates in 3D and the text can be oriented and positioned arbitrarily in 3D space. In this section we show how OpenDX can be used as a tool to produce 3D special effects. All the examples which follow were typeset with LaTeX via the *LaTeXText* macro (or its equivalent *LaTeXFileText*) but the effects described can all be applied to standard OpenDX text objects no matter how they were created. These examples really just scratch the surface of what can be done with special effects in OpenDX — the possibilities are almost limitless.

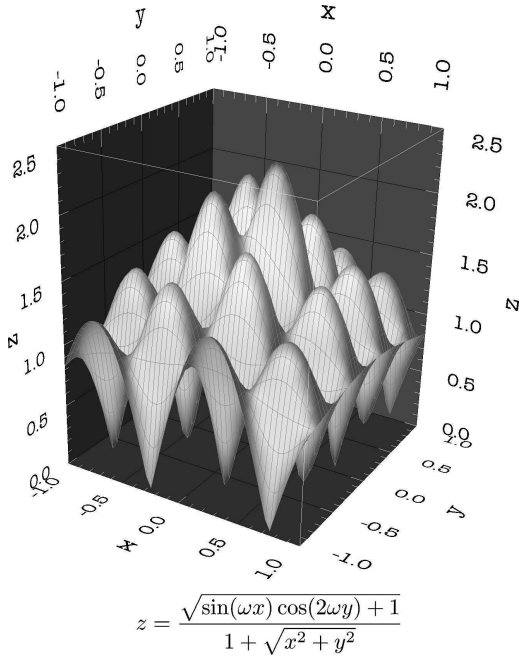

Figure 12: LaTeX-annotated figure.

8.1 Warped Text

Many of the operations that can be done on OpenDX objects can also be done on text. So, for example, we can *warp* a piece of text by transforming its coordinates as shown in figure 13. Note that the warped text has a distinct 'sheen' due to reflected light. The properties of the lighting can be precisely controlled within OpenDX, as can the reflective properties of the surfaces of objects.

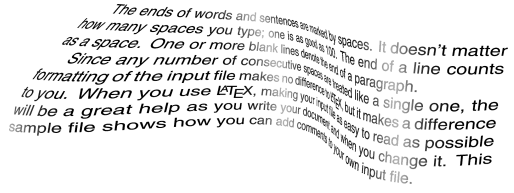

Figure 13: Warping text.

8.2 Texture Mapping

Texture mapping, i.e. the overlaying of a 2D image onto a 2D surface, is a common technique in computer graphics. It is most useful when the surface itself has a low resolution. Overlaying a high resolution image then produces an impressive visual effect but with an underlying simplicity allowing fast geometric transformations. In figure 14 we overlay an image (texture) onto the font characters. Note that as well as being texture mapped, the text has

also been *extruded* to give it a thickness. We show another example of extrusion later. At present, texture mapping in OpenDX relies on OpenGL hardware rendering and there are some technical limitations on the quality of hard-copy output one can obtain.

Figure 14: Texture mapping.

8.3 Text Boundaries

It is easy within OpenDX to embellish the character boundaries of text in many different ways. In figure 15, spherical glyphs are used to define boundaries but almost anything is possible — and often easy to set up. The density and size of the glyphs can be adjusted within OpenDX by first extracting the outline information for each character and populating the outlines with graphical glyphs.

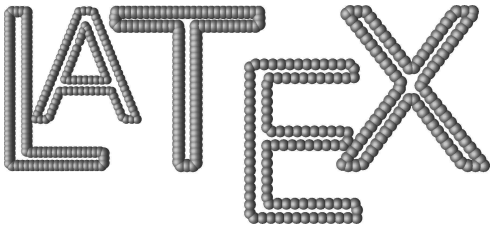

Figure 15: Glyph boundaries: spheres in this case.

8.4 Exploiting Transparency

The opacity of the 3D text can be changed to make it semi-transparent. For example, a stained glass effect can be achieved by wrapping a tube around the boundary of each character (to simulate the lead) and reducing the opacity of the text to some suitable value (< 1). This is illustrated in figure 16.

Figure 16: Stained glass effects — note the transparency of the 'glass'.

8.5 Extrusion

In figure 17, we illustrate *extruded* 3D text — i.e. the text has a thickness as well as width, height and depth!

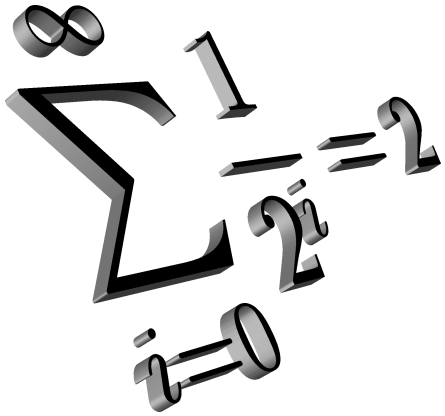

Figure 17: Extruded 3D text.

It should be emphasized at this point that as far as OpenDX is concerned, this is just an ordinary 3D object. In the image display window, you can rotate, zoom and even walk through the zero in the subscript on the $\sum_{i=0}^{\infty}$ sum! Such manipulations are possible with all the objects we have described. Note that in figure 17, the text has been rotated and the image generated with perspective. These properties can be controlled interactively from the OpenDX image window.

8.6 Foreign Languages

LaTeX has excellent support for many of the world's languages, including an increasing number of non-Latin languages such as Arabic, Japanese, Chinese and Hebrew, through packages such as ArabTeX, CJK and babel. Many of these packages have been

successfully applied to OpenDX captioning using *DXfontutils*.

9 Getting *DXfontutils*

The complete *DXfontutils* system consists of:

- three Perl scripts, font2dx, dvidx and latex2dx;
- five OpenDX macros: *TextAlias*, *LaTeXText*, *LaTeXCaption*, *LaTeXFileCaption* and *LaTeXFileText*;
- a set of TEX fonts in OpenDX format: Computer Modern, AMS Euler and a selection of others converted from outlines in the TEX Live 2004 distribution;
- several example OpenDX networks showing how to achieve the various effects described in this document and a sample dvidx map file.

The complete system can be downloaded from:

`http://www.njph.f2s.com/dxfontutils`

10 Summary

We have shown one way that LATEX can be used as a back-end typesetting engine: to enhance the limited annotation facilities provided in the core OpenDX system. To facilitate this, a font conversion program was written, allowing native OpenDX fonts to be utilized throughout for maximum efficiency. Additional requirements were an OpenDX `dvi` driver and a set of macros to be used in the visual programming editor of OpenDX.

We have also demonstrated how OpenDX can be used as a powerful tool for producing special effects on LATEX generated typeset material. LATEX and OpenDX are therefore complementary, each able to enhance the output from the other.

The main problem to date has been the speed with which the various Perl scripts, external programs, and OpenDX macros link together. For example, the standard OpenDX *Text* macro is much, much faster than the *LaTeXText* macro. The bottleneck is primarily the dvidx script.

OpenDX has a caching mechanism which means that for a given piece of LATEX formatted text, the dvidx and latex2dx scripts are run just once. Subsequent operations such as rotation or shading are done on the cached object. Of course, if the LATEX commands are changed, then the scripts are automatically re-executed. Generally, re-execution happens only if any of the inputs to *LaTeXText* or *LaTeXCaption* are changed.

Finally, there are several other improvements which could be made, such as:

- modifying font2dx to produce OpenDX fonts in the more compact binary format;
- modifying dvidx to read binary format OpenDX fonts (at present it works only with text format fonts);
- adding "SlantFont" transformation directives to the dvidx map file following a similar scheme to that used by dvips;
- adding `\special` support within dvidx for the inclusion of native OpenDX objects;

Currently, *DXfontutils* is more a proof-of-concept system than a well-tuned production software product. However, it is a proof-of-concept with considerable functionality. We have illustrated its use with LATEX, but there is no reason why plain TEX or other formats could not be used instead — all that would be required are minor edits to the latex2dx script.

References

[1] *IBM Visualization Data Explorer User's Reference*. `http://opendx.npaci.edu/docs/html/refguide.htm`, 1999–2004.

[2] *OpenDX*. `http://www.opendx.org`, 1999–2004.

[3] *comp.graphics.algorithms FAQ, §2.06*. `http://www.faqs.org/faqs/graphics/algorithms-faq`, 2004.

[4] *Gnuplot*. `http://www.gnuplot.info`, 2004.

[5] P. Bourke. *Determining whether or not a polygon (2D) has its vertices ordered clockwise or counter-clockwise*. `http://astronomy.swin.edu.au/~pbourke/geometry/clockwise`, 2004.

[6] W. Glunz. *pstoedit*. `http://www.pstoedit.net`, 2004.

[7] J. Pazdziora. `TEX::DVI::PARSE`, `Font::TFM`. `http://www.cpan.org`, 2004.

[8] J. Schewchuk. *Triangle*. `http://www.cs.cmu.edu/~quake/triangle.html`, 2004.

[9] D.L. Thompson, J.A. Braun, and R. Ford. *OpenDX: Paths to Visualization*. Visualization and Imagery Solutions Inc., 2001.

[10] G. Williams. *Fontforge*. `http://fontforge.sourceforge.net`, 2004.

◇ J. P. Hagon
Physics Centre
School of Natural Sciences
University of Newcastle upon Tyne
NE1 7RU
United Kingdom
`jerry.hagon@newcastle.ac.uk`