Improvements to the `graphics2d` package of the FreeHEP*Java library

Mark Dönszelmann†, Simon Fischer‡and Sami Kama§

September 13, 2001

Abstract

This report summarises the improvements made to the `graphics2d` package of the FreeHEP open source Java library within the scope of the Summer Student Programme 2001 of the European Organisation for Nuclear Research (CERN). A new PDF driver was written and classes for embedding Type 1 and Type 3 fonts were introduced.

Contents

*`http://www.freehep.org`

†CERN, Geneva, Switzerland, `Mark.Donszelmann@cern.ch`

‡University of Dortmund, Germany, `simon@united-wizards.de`

§Middle East Technical University, Ankara, Turkey, `sami_kama@yahoo.com`

1 Introduction

The FreeHEP `graphics2d` package provides a set of output drivers for vector oriented document formats among which are the portable document format (PDF), Encapsulated PostScript (EPS) and scalable vector graphics (SVG). The latter is mainly used for the world wide web. A special `PixelGraphics` implementation can be used to render to the screen. Compared to screen dumps, vector oriented formats have the advantage of retaining their high precision at any scale. This is especially valuable when rendering to a high resolution device, for instance a printer.

The `graphics2d` package is designed to be easily pluggable into existing applications. It is therefore resembles the Java graphics classes.

2 Features

2.1 Class hierarchy

The class hierarchy (see figure 1) of the FreeHEP graphics2d package adopts the Java graphics context class hierarchy. Parallel to the abstract graphics context classes `Graphics` and `Graphics2D` that come with Java 1.1 and 1.2, there are two interfaces `VectorGraphics1` and `VectorGraphics2` which declare methods that are guaranteed to work with the respective Java versions. Among these are all methods known to Java plus methods to implement some features that will be focussed on in the next section. Finally there are the abstract classes `VectorGraphics` (not shown) and `VectorGraphics2D`, each of which implements one of the interfaces and extends one of Java's graphics contexts. They contain default implementations for some methods, where this is possible, plus the wrappers from integer to double methods. All actual graphics drivers for the various output formats inherit from one of these classes. The drivers for PostScript and PDF are described in 2.3 and 2.4.

2.2 Overview of the `VectorGraphics2D` Features

Being a subclass of `Graphics2D`, `VectorGraphics2D` inherits all methods for drawing of lines, shapes, and text, setting of strokes, paints, and fonts, and transformations. Additionally all drawing methods are available in double precision. These methods require no further discussion. See [1] and [2] for details.

2.2.1 Additional features

In addition to the standard Java features there are some handy methods especially interesting for applications in high energy physics.

Displaying a physics event might easily require some thousand tiny boxes, triangles or circles. The method drawSymbol() renders these symbols at high performance and lower filesize.

In order to format text on a diagram, one does not need a complex text layout system. FreeHEP chooses the approach of limited HTML strings which can be used to format the text while drawing. Tags are available for italic, bold, typewriter, superscript, subscript, underline, and overline text. Furthermore strings can be underlaid by banners and surrounded by boxes.

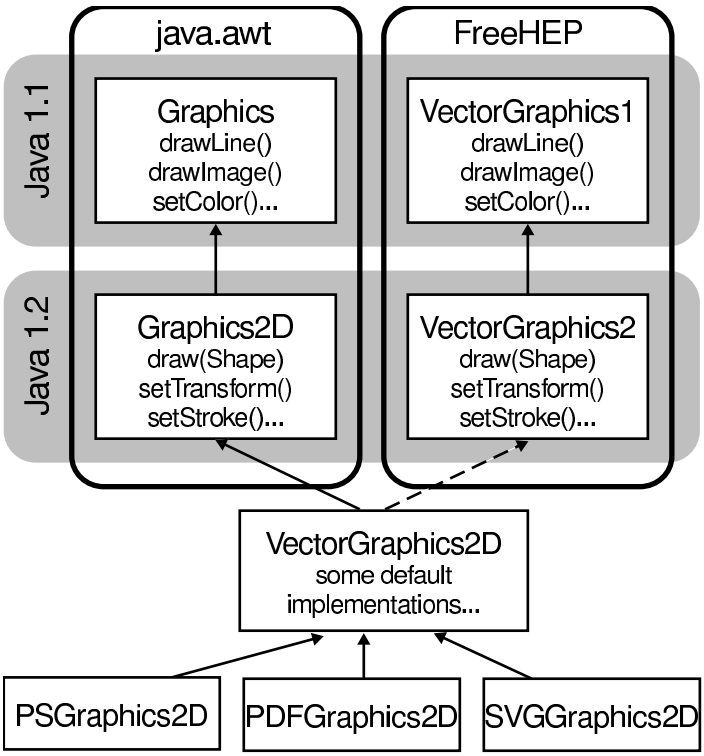

Figure 1: The `graphics2d` inheritance tree

2.2.2 Multi page documents

Some of the output drivers additionally implement the `MultipageDocument` interface which, as one can easily guess, facilitates output documents containing a set of graphics each of which is drawn on a separate page. The only thing to do is to frame the drawing of the page by invoking openPage() and closePage().

2.2.3 Drawing to a `VectorGraphics2D` context

PostScript and PDF files have a special stack-based file organisation. The paint, stroke, transformation, clipping area, and font altogether form the *graphics state*. Some of the output formats might allow only the modification of these parameters but not their direct setting. Such a modification might be the intersection of the clipping area or a concatenation of the current transformation matrix. Due to this, a special approach for drawing on `VectorGraphics` contexts is highly recommended. First one creates a new graphics context as a copy of the existing graphics state by calling create(). Then one draws on it narrowing down the clipping area or making arbitrary transformations if desired, thereby changing the current graphics state. Finally one disposes of the newly created graphics context by calling dispose() and continues drawing on the original context using the previous graphics state.

2.3 The new PDFGraphics2D driver

2.3.1 Features

As PDF is a quite powerful format the PDF driver supports almost all features. It facilitates multi page output and embedded thumbnails as well as bookmarks for easy navigation within the document. Fonts can be embedded either as Type1 or Type3.

2.3.2 Implementation

The PDF driver was written from scratch along the lines of the existing PostScript driver. PDF files [3] are organised in a tree-like structure consisting mainly of dictionaries and streams. A page dictionary might reference another dictionary describing the required resources and a stream containing the contents of the page. This stream might again reference an image, stored as a so called *XObject* in a separate stream. The actual page contents consist of a set of commands for constructing and filling paths, showing text, and many more.

In order to write syntactically correct PDF output, utilities from the `org.freehep.util.pdf` package were used. The main class `PDFWriter` has a set of methods to open and close dictionaries, streams, and objects. It counts byte offsets and lengths of these objects and keeps track of references. These values are needed for the reference table at the end of a PDFfile.

Additional utility classes for delaying the writing of objects like images and patterns to the end of the page stream were introduced and added to the package.

2.4 Newly implemented features of PSGraphics2D

Some improvements have been made to the PostScript [4] driver. Apart from some minor changes like adding support for shading patterns, there are two major changes. `PSGraphics2D` now implements the `MultipageDocument` interface and is capable of embedding Type1 and Type3 fonts.

3 Using the library

Generally the usage of the library does not require any changes to existing code. As the `VectorGraphics2D` subclasses extend `Graphics` as well, they can be used as an argument to the paint() method of any component.

3.1 Creation of and drawing on graphics contexts

In order to make use of the additional features, one has to use the `VectorGraphics` interface. This section gives an example how to do this.

It is possible to create a document independent of any panel or frame. It can simply be created by a constructor or factory method and prepared for drawing onto it by making the desired settings with the respective methds. After drawing onto the context, the file can finally be closed. See the javadoc of `VectorGraphics2D` and its subclasses for details [5].

More probably you will want to use `VectorGraphics2` in the context of a `JPanel` or a similar component. Therefore you should override the component's paint() or print() method and create a `VectorGraphics2` instance as shown in the following example.

```
public void paintComponent(Graphics g) {
  if (g != null) {
    // create a VectorGraphics2 instance
    VectorGrahpics2 vg =
      VectorGraphicsUtilities2.makeVectorGraphics2(g);
    // paint your graphics
    vg.drawSymbol(90, 105, 10,
                  VectorGraphicsConstants.SYMBOL_UP_TRIANGLE);
    vg.drawString(new TagString("Hello <b>World</b>!"),
                  100, 100);
  }
}
```

When you draw to the graphics context keep in mind to use create() and dispose() as described in 2.2.3. Notice that the factory method used will produce a `PixelGraphics2D` to display graphics on the screen in case `g` is a standard Java graphics context. If `g` already is a `VectorGraphics2`, for instance `PDFGraphics2D` it will simply return `g` itself.

3.2 Exporting components to a file

An easy way of exporting components is provided by the `ExportDialog`. It brings up a file chooser and lets the user select a format to save a given component or array of components in.

```
public class ExportFrame extends JFrame {

    private ExportDialog dialog;
    private JComponent contents;

    public ExportFrame() {
        super("Export Frame");

        dialog = new ExportDialog();
        dialog.addExportFileType(new PDF2DExportFileType());
        // ... add ps, eps, svg, gif,...

        // [...] initialize the component "contents", which
        // is to be exported, here and add it to the frame
        getContentPane().add(contents);
    }

    // call this method when an appropriate event is fired
    public void export() {
      dialog.showExportDialog(this, contents);
    }
}
```

The showExportDialog() method will bring up a dialog which lets the user choose a format and a file. Then it will create an appropriate graphics driver and use it as the argument to the paint() method of `content` to produce the output.

4 Font embedding

A font is a program that defines a particular shape (glyph) for every character in a character set. In order for a program to be able to display a given text properly it is necessary to access these fonts. Usually there are several fonts installed in a system. But sometimes, especially when transferring the documents between different computers, a processing program might not be able to find the necessary fonts. In order to prevent this, one must be sure about the existence of the necessary fonts. One way to do this is embedding the font into the document itself. Embedding a font into a document needs special encoding depending on the type of the document, font type and system type. In the FreeHEP graphics2D package this is done by several classes. `FontIncluder` is abstract superclass of all font embedding classes. It is extended by the `FontEmbedder` class, which is the superclass of several type font embedding classes. See figure 2.

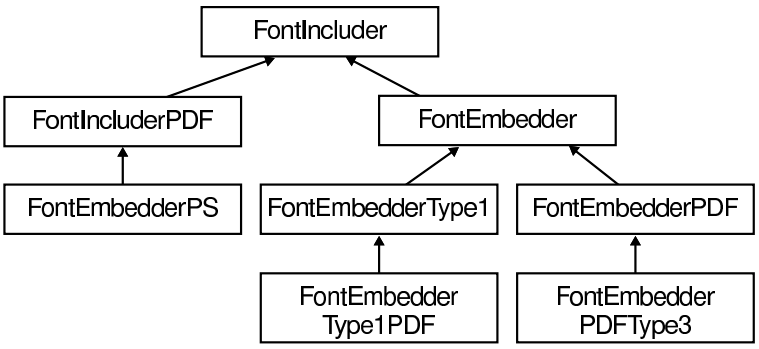

Figure 2: The font embedder inheritance tree

These classes are used for extracting the data from the Java and putting them into font formats supported by PostScript and PDF. Usually fonts are divided into two main section an Encoding array, which is a mapping between character values and shape definitions, and a glyph dictionary that contains necessary information for drawing the shapes. To show a character, a processing program follows the procedure shown in figure 3. The argument to the text showing operator is a string. Actually this string consists of a set of bytes, specifying the character codes. Looking up their unique unicode names in the encoding table, the application can use these as keys for the glyph table and finally retrieve the characters shapes.

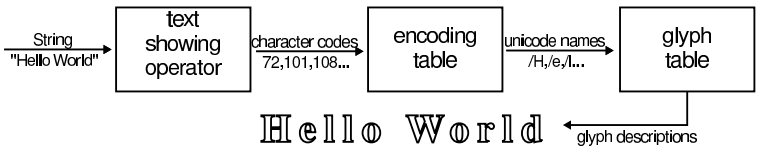

Figure 3: Encoding scheme

Since we are getting the information from Java and since it uses unicode, having different Encoding array, we must generate the Encoding array ourselves.

4.1 Encoding

To get the proper encoding arrays for a font, the `Lookup` class is used. `Lookup` class holds several `CharTable` classes , which are generated by the `CharTableGenerator` class from a set of definition files4. One can get the necessary `CharTable` class for a specific font from the `Lookup` class. The `CharTable` instances contain character names, the encoding array and unicode numbers of the characters of that font. Once one has the unicode number, name and the encoding of a character one can get the shape of that character from Java. Having all necessary information the font can embedded into the document. Currently Type 1 and Type 3 fonts are supported. The Encoding array of a font is constructed by looping over all possible character codes and calling the toName() method of the proper `CharTable` instance. After the Encoding array is created, it can be used to generate the simpler Type 3 fonts or the more complicated Type 1 fonts.

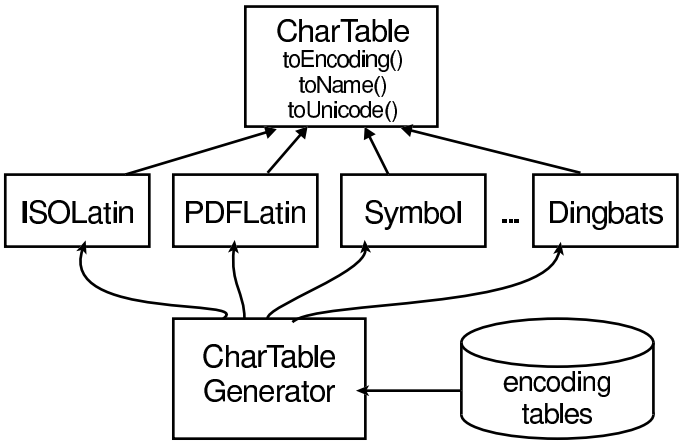

Figure 4: Generation of CharTables

4.2 Type 3 fonts

Type 3 fonts are PostScript programs that are written in clear text. In Type 3 fonts shapes are constructed with standard PostScript operators such as curveto, lineto, fill, moveto, etc. This property makes them easy to construct and understand. However, since they are not encoded in any way, they consume more space than the Type 1 fonts. Type 3 font embedding is handled by `FontEmbedderPDFType3` for PDF files and `FontEmbedderPS` for PostScript files. Both classes start processing with a chartable which they got from the `Lookup` class, generate the Encoding array and the CharProcs dictionary, which contains glyph definitions as PostScript procedures. `FontEmbedderPS` also generates a Metric array to set the font advances widths properly. Finally both `FontEmbedderPS` and `FontEmbedderPDFType3` put the created font into the document in proper formatting for the document type.

4.3 Type 1 fonts

Type 1 [6] fonts are a special case of a PostScript program. It contains both a clear text part and an encoded-encrypted part. It also contains *hints* to preserve character shapes to some extend in extreme cases, e.g. at small sizes. Type 1 font embedding is done by the `FontEmbedderType1` class for both PDF and PostScript files, since both formats can handle Type 1 fonts. `FontEmbedderType1` uses several utility classes to generate the encoded part of the font file. These are the `CharStringEncoder` and the `EEXECEncryption` utility classes. First the `FontEmbedderType1` class gets the proper `CharTable` instance from `Lookup` with a `getTable()` call and constructs the Encoding array. Then, from the names in the encoding array it constructs the `CharProcs` dictionary. Subsequently it sends the `CharProcs` dictionary through `CharStringEncoder` and `EEXECEncryption` classes to encode and encrypt the dictionary. Finally it sends all data to the proper output stream. However these classes do not support hints yet, because Java does not contain hint information. One could get that information from a TrueTypeFont file.

4.4 TrueType Fonts

In order to obtain the necessary font description we can use two sources of information. Using the standard Java `Font` and `GlyphVector` classes one can easily get the glyphs' shapes. Getting the hints for the glyphs is more difficult. TrueType font files [7], which are organised in tables, contain this information. The tables can be retrieved via Java's `OpenType` interface by their names as uninterpreted byte arrays. Unfortunately no font implements this interface yet. Therefore the `TTFFile` class can read a TrueType font from a file and should (which we cannot test yet) be able to read them from the `OpenType` data as well. Subclasses of `TTFTable` contain all the interpreted data available in the TrueType font.

Since JDK 1.3 Java is shipped with some Lucida fonts which are in TrueType format. As of JDK 1.4 these fonts also include hints.

5 Conclusion

The `VectorGraphics` interfaces proved to be very suitable to build further output drivers upon. It was possible to implement all features for the PDF driver within a short time. Redundancy between the old and new drivers was taken as an occasion to do some refactoring. Methods were moved up in the class hierarchy or to utility classes. The generated output including Type1 and Type3 fonts was tested with several versions of Adobe Acrobat Reader, Ghostview and Ghostscript and displays properly. The work was completed within eleven weeks.

The `graphics2d` package, as well as the entire FreeHEP library, can be used in a wide field of applications. The improvements done are tested with the WIRED[1] event display plug-in for JAS[2] and work fine. It should be no problem to include the library in existing software projects.

There are still some things that need to be done.

[1] http://wired.cern.ch/
[2] http://jas.freehep.org

- Check the next Java version for `OpenType` implementations and check whether or not the `TTFTable` implementations read them correctly
- Add TrueType font embedding
- Complete some minor features in the graphics drivers like cyclic gradients fill for PDF and image tiling for PostScript

6 Acknowledgements

We like to thank Charles Loomis, who built the fundaments of the `graphics2d` package, designed its interfaces, and class hierarchy and implemented the PostScript drivers.

References

[1] Satyaraj Pantham: *JFC 2D Graphics and Imaging*, SAMS, 2000

[2] *Java 2 Platform, Standard Edition, v 1.3.1 API Specification* `http://java.sun.com/j2se/docs/api/index.html`

[3] Jim Meehan, Ed Taft, Steve Chernicoff, Caroline Rose: *PDF reference, version 1.3*, 2nd ed., Addison Wesley (2000)

[4] Ed Taft, Steve Chernicoff, Caroline Rose: *PostScript language reference*, 3rd ed., Addison Wesley (1999)

[5] *FreeHEP Overview* `http://java.freehep.org/lib/freehep/api/index.html`

[6] *Adobe Type 1 Font Format*, Addison Wesley (1993)

[7] *TrueType 1.0 Font Files*, Microsoft Technical Specification (1995)