CREATING REUSABLE WELL-STRUCTURED PDF AS A SEQUENCE OF COMPONENT OBJECT GRAPHIC (COG) ELEMENTS.

Steven R. Bagley, David F. Brailsford and Matthew R. B. Hardy
Electronic Publishing Research Group
School of Computer Science & IT
University of Nottingham
Nottingham NG8 1BB, UK
{srb, dfb, mrh}@cs.nott.ac.uk

ABSTRACT
Portable Document Format (PDF) is a page-oriented, graphically rich format based on PostScript semantics and it is also the format interpreted by the Adobe Acrobat viewers. Although each of the pages in a PDF document is an independent graphic object this property does not necessarily extend to the components (headings, diagrams, paragraphs etc.) within a page. This, in turn, makes the manipulation and extraction of graphic objects on a PDF page into a very difficult and uncertain process.

The work described here investigates the advantages of a model wherein PDF pages are created from assemblies of COGs (Component Object Graphics) each with a clearly defined graphic state. The relative positioning of COGs on a PDF page is determined by appropriate 'spacer' objects and a traversal of the tree of COGs and spacers determines the rendering order. The enhanced revisability of PDF documents within the COG model is discussed, together with the application of the model in those contexts which require easy revisability coupled with the ability to maintain and amend PDF document structure.

Categories and Subject Descriptors
I.7.2 [Document and Text Processing]: Document Preparation — *Markup languages*; I.7.4 [Document and Text Processing]: Electronic Publishing.

General Terms
Algorithms, Documentation, Experimentation.

Keywords
PDF, graphic objects, Form Xobjects, Tagged PDF.

FINAL DRAFT of paper accepted for:
DocEng'03, November 20–22, 2003, Grenoble, France.

1. INTRODUCTION
Since its introduction in 1993 Adobe's Acrobat viewer software and the underlying PDF (Portable Document Format) have established themselves as important *de facto* standards for the faithful representation of page based, graphically rich, electronic documents. At the time of writing release 6.0 of Acrobat has just become available and the PDF specification has also been updated to a revision level of 1.5.

Although it is possible to generate PDF directly from text preparation software, a large number of PDF documents are still created by passing the PostScript output from the given front-end application into the Adobe Distiller software, which converts PostScript into PDF. Distiller can optimise the PDF it generates by removing PostScript procedures, for loops and so on and replacing them by 'in-line' code but it has to be much more careful in performing any optimisations that involve the graphic state of objects on the page. In particular it is not generally safe to alter the rendering order of text and graphic objects that was laid down in the original PostScript. For this reason the quality of the PDF, in terms of its flexibility for re-purposing the page content, is very much determined by the original PostScript. There is absolutely nothing, for example, that forces PostScript to render its pages in 'reading order'. An article such as this one, laid out in two-column format, could very well be rendered in *baseline sort* ordering wherein each sentence fragment terminates at the inter-column gutter and the renderer then 'hops the gutter' to typeset an unrelated sentence fragment to the right of the gutter. Indeed even if the columns *have* been rendered in reading order, rather than baseline sort order, the Acrobat text selection tool, relying as it does on x and y positioning of individual words, will often continue to jump the gutter, resulting in a selection across the full width of the page when, perhaps, only a subset of the left-hand column was needed.

1.1. Selecting text and graphic objects
The problems caused by arbitrary rendering order on a PDF page are not too serious provided that the PDF file is regarded as being a strictly 'read only' format. However, starting with Acrobat 3.0 (and PDF release 1.2) there have been some simple facilities for performing 'touch up' on PDF pages in order, perhaps, to correct page proofs immediately prior to printing. More recently this Touch Up facility has been enhanced to allow graphic objects, as well as text objects, to be selected and moved around the page with a view to adjusting the final detailed layout of a page without having to revert to the software that initially created and/or integrated the PDF.

Unfortunately use of the graphic Touch Up tool can reveal all the previously discussed problems encountered with text: clicking on the graphic object may result in selecting only a subset of it; an attempt to drag a marquee around the entire object may result in selecting objects outside the marquee simply because these were originally part of the same PostScript path. In an ideal world it would be possible to select a 'clean' subset of any text or graphic object but very often it is necessary to import the entire PDF page into an application such as Adobe Illustrator in order to edit out extraneous material and to leave only the desired objects.

We decided to tackle the problems outlined so far by designing a new model for creating and encapsulating PDF text and graphic objects such that each object has a well-defined graphic state. These objects we call *COG*s (**C**omponent **O**bject **G**raphics)[1] and we now look at their properties and investigate ways in which pages of COGs can be created.

2. THE COG MODEL

The COG model for documents attempts to alter the way a page is described in a digital document format such as PDF. Many current digital document formats can describe a page in a programmatic fashion by using a Page Description Language (PDL) to specify how the page raster is marked to create the final appearance. Depending on the complexity of the PDL, these marks can range from simple drawing primitives, offering support for drawing lines, through to facilities for handling text, transparency and raster images; some formats, notably PostScript, are full programming languages in their own right. The document renderer executes the PDL operators, in sequence, to produce the final output. Generally speaking the overall *graphic state* of the page is cumulative and builds up as a result of the particular operators that are executed.

The COG model however, works from a different viewpoint. It encapsulates the low-level primitives within distinct graphical objects, which are executed in sequence to generate the page in a logical and 'object oriented' manner.

2.1. COGs

Our encapsulated graphical objects are what we have termed COGs, where a COG is a representation of a logical block of content on the page. This current paragraph would be an example of a COG, as would the various headings on this page. The description is deliberately left somewhat vague; the concept of what constitutes a COG, and its optimum granularity, depends on the page in question but even so a few basic ground rules can be laid down.

Ideally, the COG should represent a logical block of content that one might want to use again in some other context. If the granularity of the COGs becomes too small, then reusing them gives few benefits. Conversely, if the granularity becomes too large then it becomes very difficult to reuse any part of interest because the COG will then contain much extraneous material that is of no use. The most appealing compromise seems to be to keep the size of a textual COG at roughly the level of a paragraph and for graphical COGs at a sensible level of encapsulation depending on the context of their first use. A graphic of three cars in a sales brochure for a motor company would probably be represented by three COGs (one for each car), though in another context the meaning of the graphic might suggest that they are better represented as one COG.

2.2. Describing COGs

The COG represents a graphical block on the page, and requires some 'code' to draw it. This code is nearly identical to that which would draw the same material in a normal PDF document, but a few adjustments are needed because of the nature of COG encapsulation.

Traditionally, a PostScript document can be one long piece of code with no page independence. That is to say the settings for various properties (font, point size, text colour, stroke width etc) are set up at an early stage and are changed only where necessary. Indeed, finding the fragments of code that determine the graphics state for some part of a given page can be a complex programmatic task in its own right.

It is common to encounter graphics that have been drawn in an order that defies all logic. A line diagram of the authors' campus, converted to PostScript from DXF, is famous for breaking off half way through drawing the Computer Science building and then, in the same 'path', proceeding to draw just a small portion of an ornamental lake some 20 metres away.

A further problem with PDF produced from PostScript is that there is nothing to prevent the components of a graphic being placed using absolute page coordinates, rather than relative ones. If this happens, any attempt to move the graphic on the page requires the application to know where the graphic was originally intended to be drawn.

The design of the COG model prevents both of the above problems at the outset. The COGs do not know in which order they will be drawn and must make no presumptions about inherited graphics state. Secondly, COGs are intended to represent distinct and logically separate objects. Thirdly, a COG has no idea whereabouts on the page it will be drawn.

To realise the above properties a COG must be completely encapsulated; it can make no assumption about inherited graphics state and so must set up within itself all the properties it needs. Finally, a COG must be drawn using a system of relative coordinates so that it can be freely positioned anywhere on the page.

2.3. Drawing COGs on the Page

The method of placing COGs on the page to create a final layout is notionally similar to the way newspapers used to be laid out in the days of metal type i.e. as blocks representing different articles, photos, headlines and so on.

In the COG model the page is represented as an ordered list of 'spacer' objects, but at the very end of each spacer, once the position coordinates are established, there appears a pointer to the COG object itself containing the code to draw the object. A traversal of this ordered list will draw each COG in the intended reading order of a document. For example, a two-column document like this one would lay out all the COGs in the left hand column, followed by all those in the right hand column.

COGs can be moved around the page by altering the spacer code, or the position of a particular spacer in the list. Any COG can be

[1] The acronym reflects the properties of COGs that draw upon the **C**omponent Programming and **O**bject-Oriented paradigms, as applied to self-contained **G**raphic objects.

replaced with a different one, very simply, by changing the COG that the spacer object points to.

3. IMPLEMENTATION

Traditionally, a PDF page is described as a monolithic stream of drawing operators that are executed in sequence to create the markings that constitute the final output. Each PDF operator is executed in the context of the graphical state left by the previously executed operator, although there are commands that will save and restore the current graphics state as we shall shortly discover.

When implementing COGs within PDF, we need to modify this approach in two distinct ways: firstly, we need to ensure a clean graphics state for each distinct block – in other words the graphics operators executed in COG X should not have an effect on the graphics operators executed in COG Y (and vice versa). Secondly, we must be able to split an amorphous stream of PDF content into distinct blocks of content (the COGs themselves).

3.1. Ensuring a clean graphic state

PDF provides us with mechanisms for manipulating the graphics state as a whole and these can be used to allow each COG to inherit a well-defined graphic state. Some details now follow.

3.1.1. External State

The PDF `gs` operator loads the graphics state from a dictionary (known as an **ExtGState** dictionary) that defines the default values for parameters such as line width, line style, etc. along with various controls used in the prepress and professional printing environments. At first sight this seems useful: we could load a default graphic state at the beginning of each COG and proceed from there. Unfortunately the graphic state loaded by the `gs` operator is cumulative; each property of that state is added to the current state. If a property is not explicitly defined in the newly loaded state then the setting in the current state is retained. This alone rules out the `gs` method for our purposes because properties that are not explicitly defined in the new state will be inherited from the previous COG object to be executed. In other words we cannot specify a known graphics state using the `gs` operator alone.

A second problem is that the **ExtGState** dictionary allows the specification of many graphic state properties but there are some strange omissions, such as the default stroke and fill colours. This compounds the difficulties of using this method to fully specify a graphic state for a COG.

3.1.2. Save and restore

PDF's other methods of manipulating the graphics state, `q` and `Q` (analogous to the PostScript operators `gsave` and `grestore`) allow the current graphics state to be saved at any point, and then restored later in the stream. Multiple save/restore operations are allowed and these are implemented in a standard stack-like LIFO fashion.

The effect of this is that the graphics state after a restore operation is identical to what it was when the save operator was executed (though this does not affect the marks already placed on the page).

This save and restore method provides a method that preserves the graphics state in its entirety. By saving the graphics state before we execute a COG, and restoring it afterwards, we know that the next COG will execute in exactly the same graphics state as the first. This, then, provides an easy mechanism for defining the state in which a COG will be executed and is our method of choice for implementing COGs.

3.2. Structuring PDF Content

Each page in a PDF is represented by a COS[2] dictionary[1] similar to the following example:

```
<<
  /Type       /Page
  /Parent     2 0 R
  /MediaBox   [0 0 595 842]
  /Contents   4 0 R
  /Resources
  <<
       /ProcSet [/PDF /Text  ]
       /Font  <<
                    /F1 5 0 R
              >>
  >>
>>
```

The dictionary contains key-value pairs that describe various properties of page, such as its position within the pages tree structure (`Parent` key), size (`MediaBox`), and what resources – fonts, class of drawing operators, etc – the page uses (`Resources`). Finally there is the `Contents` key. This is usually a pointer to a stream that contains the actual sequence of imaging operators that are used to draw the page.

A PDF file is a collection of many different objects (streams, dictionaries, arrays etc) that are interlinked into a tree-like structure[3]. To bring this about, objects within PDF can be assigned a numerical index that allows them to be referred to from elsewhere within the PDF. An example of object indexing can be seen in the above example in the **Parent** and **Contents** keys, which use syntax such as `2 0 R` to reference the correct object. The first numerical value refers to the object index; the second is a generation number which allows objects to be easily updated with a later version without having to rewrite the whole file. The various generations of objects are held as addenda to the original cross-reference table. The final keyword, `R,` denotes a reference to an indirect object which exists somewhere else in the file. The PDF Reference Manual [1] gives further details of all the above properties.

While the above description describes the structure of the majority of PDF files that are produced, the format itself provides two mechanisms for splitting the actual content stream into logical blocks. The first is by splitting the operators into multiple streams and then having the content key of the dictionary point to an array which, in turn, points to these streams. The second method is by using a structure known as a *Form XObject*.

3.2.1. Content Arrays

The option of PDF content being described as an array of streams, instead of a single stream, developed out of the need to be able generate a PDF in a single-pass. Some imaging operators (typically those involving bitmap images) require the size of the data to be presented in the stream before the data itself is presented. By splitting the original stream for the whole page into

[2] COS – COS Object System, a recursively defined name for the objects that represent the internals of a PDF.

[3] This is the internal structure of the PDF data format and bears no resemblance to the logical structure contained within a structured or Tagged PDF.

multiple streams, data for a bitmap image can be written to stream *n* (say) and its length can then be calculated and appended to the end of stream *n-1*. The remainder of the page can be written out to streams beginning at array position *n+1*. When a PDF interpreter comes to render the page, it effectively concatenates the streams together and reads them as one long stream.

Fortunately there are no restrictions on how arrays of streams can be used in PDF. It is perfectly feasible to use them, as we do, for segmenting a content stream into many smaller streams based on an 'object' model in which each stream is responsible for drawing one block of logical graphical content.

3.2.2. Form XObjects

The PDF *Form XObject* (pronounced as if it were one word, *FormXObject*) is a remnant of PDF's PostScript heritage and is the equivalent of the PostScript *Form*. The name change in PDF was made in order to avoid confusion with the representation of conventional forms (e.g. for tax returns, expenses claims, job applications etc.) The PostScript Form [2, page 206] was a specialised PostScript procedure which when executed made no alterations to the programming environment outside of itself, the end result being that the output of the form could be cached by the PostScript interpreter, so that its rendering would be much faster.

In PDF, the Form XObject [1] follows the same idea – it is an externalised set of drawing operators that can be called at any point within a page's drawing stream. To identify a Form XObject it is given a name in the page's dictionary, which is linked to the stream containing the drawing operators. To render a Form XObject, one executes it using the `Do` operator:

```
/MyFormXObject Do
```

The Current Transformation Matrix (CTM) at the time of callout sets the size and positioning of the Form XObject. A bounding box, into which its output will be clipped, is included within the Form Xobject's definition.

A Form XObject is defined in the PDF as a so-called COS Stream [1]. The header dictionary for this stream contains extra information as shown in the example below (Note that the standard COS Stream keys have been removed for clarity):

```
<<
      /Type         /XObject
      /Subtype      /Form
      /FormType     1
      /BBox         [0 0 1000 1000]
      /Matrix       [1 0 0 1 0 0]
      /Resources    <<
                    /ProcSet [/PDF]
                    >>
            …
>>
```

Most of the keys are obvious: the `Type` and `Subtype` keys define it as a Form XObject, the `BBox` sets the bounding box and the `Matrix` key sets a transformation matrix between the Form Xobject's graphics space and that of the page.

Form XObjects have the useful property that when they are executed they make no changes to the graphics state; their execution is implicitly wrapped up between `q` and `Q` (graphics save and restore) operations.

3.3. Implementing COGs within PDF

Initial COG-PDF tests began by using the method of segregating the page content COS stream into an array of multiple streams. This was a development of similar work carried out by Smith and Brailsford [3] in which the possibility of using Form XObjects, rather than stream arrays, had been envisaged but had had to be rejected because of their limited implementation in early releases of Acrobat.

A close reading of later revisions of the PDF specification, coupled with advice from Adobe's Acrobat Engineering group, led us to the conclusion that the problems Smith and Brailsford faced with PDF Form XObjects no longer existed. Form Xobjects were now the clear choice for implementing COGs.

Although the implementation is now predominantly based around Form XObjects, it still uses the original method of arrays of streams. The contents of these arrays are the spacer objects, which in turn contain pointers to COGs.

3.3.1. COG PDF Spacers

The job of the spacer is to image a specific COG at a specific point on the page. The COG itself is drawn by executing the Form XObject in the standard fashion.

To position the COG at the correct place on the page, we alter the CTM of the page to move the origin to the bottom-left hand position of the COG. Since the COG's content stream is designed to be drawn with respect to an origin of (0,0), the spacer does not need to be aware of the COG's latent positioning operators in order to translate it to the correct position on the page. This makes programmatic manipulation of the COGs on a page very simple—the spacer can be manipulated to alter the COG's position without knowledge of what is inside the COG itself.

Even though the Form XObject is responsible for cleaning up its own alterations to the graphics state, the spacer itself still needs to save and restore the graphics state. This is because the translate operator's effects are cumulative. A translation of (100,100) followed by a translation of (50,50) is identical to a single translation of (150, 150). By saving and restoring the graphics state at the beginning and end of the spacer we are able to nullify these cumulative effects.

The final COS stream for a spacer is of the form:

```
q 1 0 0 1 0 0 300 300 cm /CogName Do Q
```

The `q` and `Q` make sure that the spacer doesn't affect the current graphics state. The `cm` operator is the standard CTM manipulation operator in PDF. Its integer arguments precede it, in postfix notation, and the last two of these denote the translation that is to be performed before rendering the COG..

3.3.2. The inner structure of a COG

COGs are just standard Form XObjects, with a few extra entries in the dictionary that enable the COG system to identify them. A COG's dictionary will look similar to what follows (note that the standard Form XObject information has been greyed out for clarity – details of these standard dictionary entries can be found in PDF Reference Manual)

```
<<
      /Type         /XObject
      /Subtype      /Form
```

```
/Cogged     true
/Name       /Cog00000000
/Width      640
/Height     200
/FormType   1
/BBox       [0 0 595 100]
/Length     423
/Resources
<<
      /Font
      <<
      /PB 3 0 R
      >>
/ProcSet    [/PDF /Text  ]
>>
>>
```

The first extra key is the `Cogged` key; this tells the consumer that this is a COG and not just an ordinary Form XObject. The `Name` key gives the COG a unique name by which it can be referenced. Eventually this will be a variation on the standard UUID [5], though initial tests have used a simpler naming scheme.

The `Width` and `Height` key are self-explanatory, and are provided so that applications can determine whether it is possible for a particular COG to fit in a particular space in the page.

The Content stream of the COG Form XObject is identical to a normal Form XObject. However, they must be drawn with their bottom-left corner at (0,0) or the spacer will not position them correctly. Also, they must not make any assumptions about default graphical state; if they want 10pt Times Roman, they must ask for 10pt Times Roman explicitly.

3.3.3. Extra Details

In addition to the definitions of the COGs and spacers themselves, a few other rules are needed in order to produce a correct COG-PDF.

Firstly, the COGs must be imaged on a page in reading order. That is, if COG *B* contains a paragraph that follows on from COG *A* then the spacer in the page Contents for COG *B* must be after that for COG *A*.

Secondly, and importantly, the only objects allowed in the pages Content Stream array are spacer objects with embedded pointers to COGs. No other Content streams are allowed which might draw extra items on the page, for this would break the concept of COGs – where everything is drawn inside a COG. Any extraneous drawing operators would firstly, not be COGs and could therefore not be manipulated by programs expecting COGs. Secondly, and more importantly, they could alter the graphics state from the norm, which would alter the way subsequent COGs would be imaged.

3.4. Generating COG PDF

None of the current PDF code generators (MacOS X, Adobe Distiller, etc) provides any kind of hooks to coerce them into generating COG-PDF files and so it has been necessary to develop our own tools. But once a COG-PDF file has been created it can be viewed with the help of demonstration plugin that we have developed for Adobe Acrobat. This enables the bounding boxes of the COGs to be seen and each COG can be dragged and dropped to a new position on the page. This enables users to see the possibilities of the COG approach as opposed to using Adobe Touch-Up on monolithic, non-COG, PDF files.

Two approaches have been developed for producing COG-PDF; the first is a simple extension of the idea behind Juggler [3] and the second is a back-end processor for the *ditroff* [4] typesetting system.

3.4.1. COG script

COG script, like Juggler before it, was a proof-of-concept exercise. A COG script functions by taking many PDFs and extracting the Content stream of the first page from each of them and converting these pages to COGs. The software, implemented as a plugin for Adobe Acrobat, is driven by a simple script, which lists the PDF files from which to import COGs and specifies where to position the eventually created COGs on the page. At the present stage the script is just a list of file names together with the desired (*x,y*) positions for the COGs on the composite page.

Experiments with this plugin, were helpful in allowing ideas to be quickly tested and developed. These experiments resulted in standardization of the COG-PDF implementation to the one described above.

3.4.2. pdfdit

The COG script method suffers from the constraint that it can only make a COG out of the smallest portion of PDF material that can be guaranteed to have a clean graphics state. Given that most PDF files are still produced by Distiller this means that the smallest COG has to be a whole page.

It was decided that some system was needed that could generate COG-PDF directly from a suitable originating application. The application chosen was the *ditroff* typesetting system developed by Brian Kernighan in the late 1970s [4].

Ditroff is a device-independent version of the original *troff* program. It takes input in the form of a marked-up ASCII text file and outputs commands to draw the page in what we shall term Ditroff Intermediate Code (DIC). This format predated PostScript by several years and aimed to be a simple yet comprehensive code that could be translated into any imaging device's own native language. Indeed, DIC is sufficiently simple and expressive that it helped tip the balance in favour of choosing *ditroff,* though another major factor was the authors' familiarity with the system and its inner workings. Alternative formats were looked at, but they either had excessively complex file formats (Microsoft Word) or would require substantial time investment to write an equivalent backend driver (*dvi*/LATEX).

DIC output consists of a stream of operators (usually expressed by single letter codes) that inform the typesetter how to image the page content. With a few exceptions, the output consists of the operator followed by any parameters (typically either integers or a single character). White space is used to disambiguate parameters or operators. Table 1 presents a brief list of commonly used operators and their purpose; a full list can be found in [4].

s*N*	set point size to *N*
f*N*	select font mounted at N
c*X*	Image character X at current point
H*N*	Move to absolute horizontal position N

	(N > 0)
v*N*	As above, but vertical (down is positive)
h*N*	Move relative N units (to the right; N > 0)
v*N*	As above, but vertical (down; N > 0)
NNc	Move right *NN*, then image character *c* (*NN* is exactly 2 digits!)
n*b a*	end of line (for information; *b* = space before line, *a* = space after)
w	paddable word space (for information)
p*N*	new page N begins – resets cursor to top of page

Table 1. The ditroff standard operations

To understand how the output of *ditroff* is converted to a COG-PDF, it is necessary to be familiar with the general structure of *ditroff*'s output. Presented below is the DIC output that prints 'Hello World' on the page. The code has been split into blocks to help the subsequent discussion of its operation.

```
x T psc
x res 576 1 1
x init

V0

p1

x font 1 R
x font 2 I

…

s10
f1

H576

s18

V1920
cH
h104ce
64l40l40o

w

h108cW
h136co
72r48l40d

n192 0

H576
V2112

…

x trailer
V6336
x stop
```

The first block of code uses the x operator, which is used to embed device operations into the output. This is initialisation code which tells the output postprocessor about the output device that *ditroff* was expecting and its resolution (specified in dots per inch). PDF and PostScript's shared graphics model enables us to use a pseudo-resolution of 576 dpi; this resolution was chosen because it is close to that of many output devices and it has a large number of factors (including 72 – the number of PostScript points per inch).

The second block moves the cursor to vertical position zero, whilst the third block begins page 1 (also moving the cursor to vertical position zero). The fourth block maps the fonts that ditroff believes are mounted in each font position. In this case we are using only Times-Roman (R) and Times-Italic (I).

As the next few blocks show, the output from *ditroff* is not optimised – largely due to its single-pass nature. The default point size (10pt) and face (Times-Roman) are set up in block five, even though block seven alters the point size (to the required 18pt). Block six moves the cursor to horizontal position 576, which equates to a one inch indent.

Block six starts to render text. The vertical cursor is moved to the text's baseline (1920 units, 3.33inches from the top), and an 'H' is imaged. Next, the cursor is advanced 104 units to the right of its current position. This stops the next glyph being imaged over the top of the previous (unlike other systems, imaging a character in DIC does not alter the cursor position). Next an 'e' is imaged. Again the cursor is advanced, this time using the optimised form where a move and an image operation can be encoded as precisely two digits followed by the character to image (this limits the move to a maximum of 99 device units, hence the previous moves are performed with the h operator). The rest of the word 'Hello' is imaged in the same way.

The next operator in the stream (ninth block) informs the post-processor of a word break. While no action is necessary, it is useful in ensuring that the output is Tagged PDF compliant by explicitly marking all spaces in the PDF content stream.

Block ten images 'World' onto the page in a similar fashion to block eight. The w operator in block nine does not advance the cursor, so the first part of block ten has to move the cursor over the last character and the space gap.

Block eleven signals the end of a line, again this is for information only. However, it does provide a copy of the current vertical spacing in *troff* (as the space before parameter), which is useful for inferring where COGs begin and end. The next block resets the cursor to the left margin and moves it to the baseline of the next text line.

The final block signifies to the post-processor that the stream has finished and it can send a stop signal to the output device.

A back-end program for *ditroff*, called *pdfdit*, was written to convert the DIC output code into COG-PDF. The expressiveness of DIC output (not the least its clear signalling of the end of each output line—a facility lamentably missing from LATEX's *dvi*) is good enough for *pdfdit* to infer what might constitute a COG when using a standard layout of the kind imposed by the use of the popular *troff ms,* or *mm,* macros. The standard layout imposed by these macro sets separates all paragraphs/headings and other logical graphical blocks by a few points over and above the standard leading. Given that DIC makes the current line height available at all times, for every line, it is easy to see if the drop is

greater than the line height and if so the program closes the current COG and starts work on the next.

In addition to inferring implicit COG breaks, extra macros were added to the *ms* set to allow the explicit definition of a COG block. These new macros place extra, device-specific, commands (using the x operator) into the DIC output which are then recognised by *pdfdit* to encapsulate the graphical output into a COG.

Although its purpose differs from the traditional usage, *pdfdit* acts like any other program language compiler. As in traditional compilers, *pdfdit* can be split into separate phases; a syntax analyser, semantic analyser, a code optimiser, and a code generator. An extra phase inserted into *pdfdit* is the COGifier that groups graphics operations into COGs. However in the interests of program efficiency and due to the relative lack of structure in the *ditroff* output, the phases are not kept rigidly distinct within *pdfdit*.

Syntax analysis of the input stream is performed, and for each operator a handler function is called. The function performs a semantic analysis of each operator and also assembles all the character glyphs into word blocks (though, depending on kerning, they may end up being sub-word blocks, especially if accents are involved). The width and height of the text runs, and their position on the page, are calculated using the font metrics, and this information is then stored into a queue. Additional information is queued relating to line breaks, COG break markers and page breaks. This approach has the added benefit of optimising out the additional superfluous moves that can be seen in the DIC input.

After the input has been parsed, the queue is processed by the 'COGifier'. This works by segmenting the queue of word blocks into separate COGs. Explicitly marked COGs are simple to handle. Inferred COGs are created by breaking down the queue into lines of text, and then seeing if the distance between the average baseline of two adjacent lines is different to the vertical spacing given in the DIC code, if it is then a new COG is said to have been started. In both cases, the bounding box of the COG is calculated by taking the union of the area of all the (sub)word blocks within it.

This leaves with an array of COGs for each page. The final phase is to convert these COGs to use relative coordinates (at this point they are still using the *ditroff* device units) and to correct the axis (*ditroff* use the top left corner as the origin whilst PDF uses the bottom-left). The resultant COGs still use *ditroff* device units, and so these are scaled to PDF space (making sure that no rounding errors occur which might cause characters to shift slightly). The converted COGs are then compiled down to PDF code and linked to form the final document.

Although *pdfdit* produces a COG-PDF file which is completely compatible with any properly engineered PDF viewing application, it offers no advantage over a conventional PDF file unless the viewer application is equipped with a plugin to exploit the advantages of the COG representation. It should also be noted that if a viewing application such as Acrobat attempts to optimise or rewrite a COG-PDF file it would destroy the COG nature of the PDF.

3.4.3. COG shuffler

We have already described our Acrobat plugin, which allows COGs to be highlighted and moved around the page. Figures 1a–1c show some of the features in action.

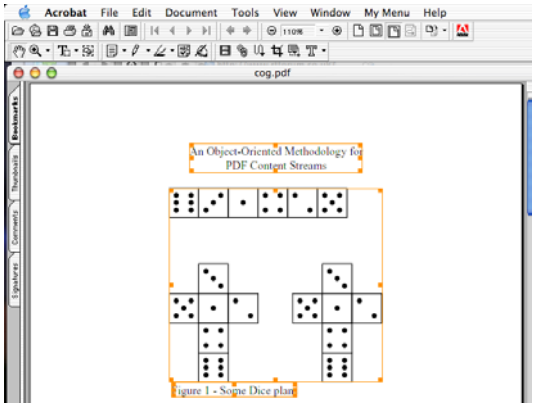

Figure 1a – Acrobat displaying the bounding box of the COGs on the page.

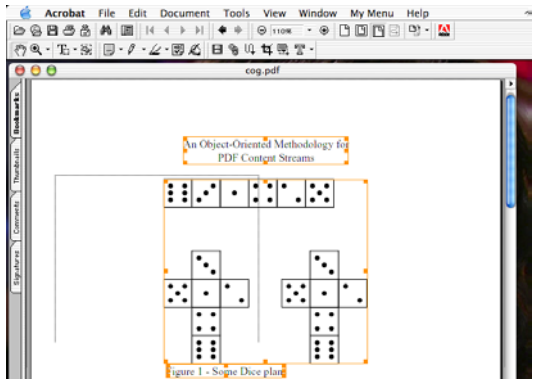

Figure 1b – A COG being dragged to a new location

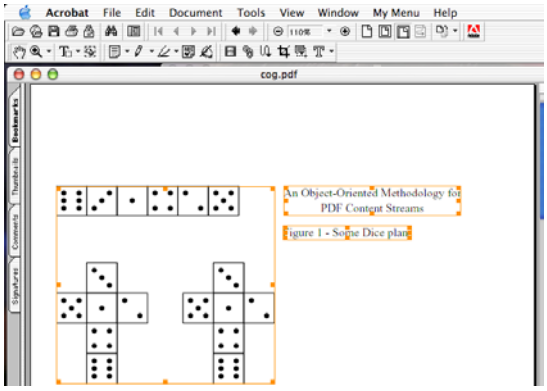

Figure 1c – The COGs in their new location

To check that the appearance of the COGs was truly invariant with respect to their positioning on the page, a 'shuffle' button was added to the user interface of the plugin. This button causes a random permutation of the COGs on the page.

4. BENEFITS OF COGS

It might seem eccentric to undertake this COGs research if the displayed result appears, to the user, very much like a conventional PDF file. While this reaction is understandable, it fails to take into account the possibilities opened up when a PDF is generated using COGs.

4.1. Extracting PDF Content

It can sometimes be useful to be able to extract content from a PDF, for reuse elsewhere, whether interactively within an Acrobat-like environment, or programmatically (possibly within some sort of server-based application). Recent versions of Adobe Acrobat have shipped with a tool called the Touch-Up tool. This tool allows the user to make minor alterations to the PDF within the PDF environment.

Another facility offered by Touch Up is the ability to extract content from a PDF file. This is done by using the 'Touch-Up Object Tool', which allows the user to select graphical objects. However, Touch-Up's opinion as to what constitutes a graphical object will not always coincide with that of the user. As far as Touch-Up is concerned a graphical object is the smallest possibly extraction of graphics commands from the content stream material that corresponds to where the user has clicked. This is demonstrated in figure 3 below:

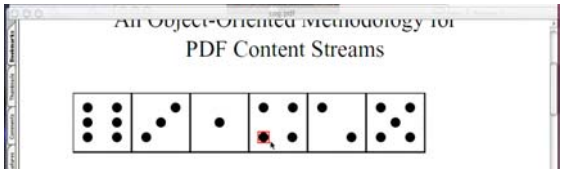

Figure 3 – A user attempting to select the dice with the Touch-Up tool.

Here the user wishes to select the dice plan displayed in Acrobat, so she clicks over part of the dice, only to find that Touch-Up selects just a single spot underneath the mouse pointer. The user is then forced to either select each part of the diagram by hand, or to attempt to select all of it by dragging a marquee around the desired material.

Although this is not a problem for a simple diagram, it becomes more and more difficult as the image complexity increases. Problems become acute if the graphic we are interested in has become occluded behind another object. Indeed, it is possible for an object to become occluded even when there is no apparent graphic object nearby to overlapping it. At this stage, in a conventional PDF file it becomes necessary to import the page of the PDF into a package such as Adobe Illustrator to perform a thorough clean-up prior to editing out unwanted objects.

4.1.1. Programmatic extraction

Extracting a graphic or an image programmatically can be quite a challenge in a conventional PDF file because there is not necessarily any clearly defined order in which the graphic is drawn on the page. To faithfully extract a graphic, the extraction program would need to parse the content stream and build up a table describing each mark laid down on the page, its properties (stroke width, fill colour etc) and its location. It would then have to walk over this list and work out which operators contribute to the graphic of interest. It is likely some operators (for example, a border drawn around the whole page) may appear to overlap the area of interest, when in fact they are not part of the graphic itself. Finally there is the problem of signalling to the extraction program which graphic you are interested in. All of these issues, and more, are what the Adobe Touch-Up plugin has to cope with when being used interactively.

4.1.2. Programmatic extraction using COGs

With a COG PDF, the process of extraction becomes much simpler than in conventional PDF. Extracting a particular graphic is now just a matter of selecting the correct COG on the page, and copying its definition out of the PDF. Identification of the COG can either be done interactively or by using the COG's unique ID. The user does not need to worry about selecting all parts of the graphic, because the COG explicitly groups them, but the extraction tool does need to be COG-PDF aware.

The biggest advantage of all in the COG approach is that an extraction tool does not need to worry about how a COG is drawn, or how its drawing operators are distributed within the content stream or even what the graphics state is for each of the executed PDF operators. These problems are shielded from the extraction program by the fact the COGs are encapsulated.

4.2. Updating content

Sometimes it is necessary to update content on a PDF page with a more recent version at the last minute before a page is sent to an image setting device. Last-minute replacement of raster images with higher-resolution versions is a well-established technique using methods such as Open Prepress Interface (OPI), but replacing vector images is much harder. The example outlined below was brought to our attention by a digital prepress manager as this paper was being authored.

A company logo featured on every page of a large print job. Unfortunately the graphic itself did not rasterise properly and so needed to be replaced on every page. The customer, in response to this problem, had created a new, clean, version of the logo that rasterised cleanly, and he wanted to update every page with the new version.

Using Touch-Up each part of the object had to be selected using a marquee but, unfortunately, this also had the unstoppable side-effect of selecting the page number next to the diagram.

The page number would then have to be deselected manually on each page and, to compound the problem, it was not possible on some pages to select the logo at all because it was occluded by other objects. The digital prepress manager ended up having to load each page into Adobe Illustrator, and found he was spending about 10 minutes per page correcting the problem. In the end, he gave up and solved the problem manually by masking out the damaged logo on the film prior to preparing the printing plates.

4.2.1. Shareability of COGs

If this problem PDF had been a COG-PDF, our digital prepress manager's job would have been considerably simpler. The COG format would have trivialised the selection of the logo itself and would have made the insertion of a new version very simple.

At first glance it might seem necessary to perform this task on every page of the document but the encapsulation property of COGs enables them to be shared and reused. There is nothing to prevent any number of spacer objects from indirectly referencing the same COG and in this way a COG-PDF file could point to the same company logo COG from each page. A simple replacement of this one COG with a later version creates a revised version of the document in one operation.

4.3. Further Benefits

Although, only two benefits have been outlined in detail above, they are by no means the only ones. Our demonstration plugin shows how COGs enable the user to manipulate a page layout far more easily than with a normal PDF.

As the next section shows, there are more benefits to be gained as we extend the COG format still further.

5. FUTURE WORK

The basis of the COG model for documents, and its PDF implementation, have now been set out. Some of the more interesting prospects for further development are outlined below.

5.1. COG Linking

As it stands, a COG-PDF, however it is produced, is a final-form document and there is no way to add external COGs that are not part of the original. One way to achieve extra flexibility is to go beyond the one-COG-per-page of Juggler and to adopt the idea of a *linker* from programming language compiler technology.

In this case a COG-producer such as *pdfdit* would no longer produce a stand-alone COG-PDF file; rather it would produce many small PDF files containing just one COG each or perhaps a library of COGs, in a single file, with clear demarcations between them. To accompany these COGs, there would be a script that states on what page, and at which location, each COG would be positioned. A second program, the COG linker, would take this script file and produce the final COG-PDF. This would allow *pdfdit* to include references to 'foreign' COGs, including graphical elements above and beyond *troff*'s capabilities, which could then be included at link time to produce the final document.

5.1.1. Late Binding

Following on from the idea of the linker, the next step would be to remove the need for static linking and to extend the Acrobat environment (by the use of plugins), so that it can load in a COG script directly, rather like a dynamic linker. This would mean that a COG-PDF file, when viewed, would always contain the latest version of the COGs (although there may be times when this would be undesirable and so facilities must be added to freeze a document to a particular version of a COG).

5.2. Metadata

As it stands, a document is a collection of COGs laid out on a page in a particular order. However, no checks are made to see if the COGs make sense when placed in that order. For example, if a journal paper contains several COGs in 12pt Times Roman, and suddenly one of them is set in 36pt Helvetica before continuing in 12pt Times, there is a fair chance that the COG might be out of place. On the other hand, if the document in question were a newspaper, this might make perfect sense.

What is needed is to add some more information to each COG describing both the physical appearance of the COG and what it supposedly represents. It would then be possible for a program to use this information to check the document for consistency.

5.3. Automatic layout

The discussion of COG scripts so far envisages the COGs being laid out at specific positions on a page. There is no reason why this should always be the case. An alternative would be to supply the COG linker with general rules about how the document should be laid out: e.g. there should be 6pts of space between each paragraph, but only 3pts separating a heading and a paragraph; captions should always be centred under the object they are describing, and so on. The COG linker would then use this information along with the metadata to lay out the COGs on the page, in a similar fashion to the *pm* program developed by Kernighan and Van Wyk [6].

This approach would help greatly if a COG needed to be updated; any alterations in its dimensions would no longer run the risk of it overflowing other COGs further down the page.

6. COGS AND LOGICAL STRUCTURE

Appearance-based mark-up (like that in PDF and PostScript) describes the appearance of a page very precisely so that it can be displayed identically anywhere. To this end, details of the fonts used, their size and their positioning, are attached to the text strings making up the document. We have now seen how these properties (e.g. 9 pt. body text; 12 pt. headings) can be attached as metadata to the COG objects. However, a COG for the present paragraph would have no innate knowledge that it actually represented a 'paragraph'.

A particularly interesting possibility is to create COGs of a granularity that aligns, in some way, with the logical structure of a document. Logical mark-up (as typified by XML applications and to some extent by formatters such as LaTeX), tags the components of a document by what they actually are. For example, this particular paragraph would be tagged as being a *paragraph*; the heading as being a *heading* (probably at a specific level). These logical tags, or *elements*, can then be nested together to form a hierarchical Document Structure Tree.

6.1. PDF and Structure

In PDF version 1.3, and Acrobat 4, Adobe added support to the PDF standard for carrying a Document Structure Tree. A brief outline of this implementation is now given, though readers are encouraged to consult reference [1] for a fuller description.

Unlike XML, where the tags that build the tree hierarchy are interleaved with textual content, PDF models the structure tree separately and its elements point to the required content data. Within the PDF format the structure tree is built up from dictionaries that represent the different elements, similar to the way an XML tree is presented to a programmer when loaded via the Document Object Model (DOM).

To enable nodes of the PDF structure tree to point to the content they enclose, markers are placed within the content stream itself to demarcate the blocks of content. These markers are then given a unique number called an MCID (Marked Content Identifier) that allows them to be referenced from the element dictionary. Sometimes the content stream may not allow a single MCID to wrap the content of some logical element without including other disjoint content. In this case the element dictionary can point to multiple MCIDs. The content for the element is taken to be the union of these MCID blocks. It is also possible to specify children in the PDF structure tree as a mixture of element dictionaries and MCIDs. This is closely analogous to the idea of mixed content in XML.

With the advent of PDF 1.4, the structure implementation was refined further and given the name *Tagged PDF* to differentiate it from PDF 1.3's *Structured PDF*. This was basically a refinement

of the rules for structure, which included a mandatory end marker to terminate each word (a space character or equivalent must be provided, in addition to the horizontal movement needed to justify the text). Furthermore, the page content has to be set out in reading order. The rules of *Tagged PDF* allow PDF files to be read out aloud by voice synthesiser software, or reflowed for display on devices such as PDAs and mobile phones.

6.2. Editing PDF Structure

It is possible, in Adobe Acrobat, to edit the structure tree of a PDF interactively and a similar editing can be achieved programmatically via an Acrobat plugin. New elements can be added, old elements deleted and the whole structure rearranged as necessary.

However, contrary to users' expectations, manipulation of the structural ordering will generally have no effect on the page appearance. For example, if one changes the order of two paragraphs in the structure tree, this will not be reflected within the document as seen on screen (though it will lead to a different reading order if it is read by screen-reading software).

This effect arises because the structure tree is external to the Page Content. Unlike an XML document, where the structure tree defines the route through the document, and the content is interleaved within it, the structure tree in a PDF is an external construct that has often been added in after the document content has been established.

6.3. Structured COGs

At present a COG-PDF file is unstructured. This does not have to be the case and by amending the COG-PDF specification it is possible to incorporate logical structure into COG-PDF. An outline of these amendments now follows.

In structured PDF, the content of an element is wrapped inside an MCID that is then pointed to by the structure tree. This approach causes problems if we try to marry structure with COG-PDF because MCIDs have a back pointer into the structure tree. This, in turn, means that if a Form XObject, contains an MCID it points back to a specific place in the structure tree thereby rendering the COG 'impure' and capable of being used just once in a document. Given that the whole idea of a COG is that of a shareable object, which can be used multiple timesent, this limitation poses quite a problem. Fortunately, if the MCID is placed just after the spacer code, and immediately prior to the call of a COG FormXObject, then the COG remains 'pure' and can be reused, in a Tagged PDF, without any limitations.

We propose, therefore, to limit the structure tree in a COG PDF file so that it points to block-level elements whose minimum granularity is a single COG. We can then place the MCID operators into the spacers and link the spacer objects into the structure tree as normal.

One advantage of structuring PDFs around a COG model is that it becomes easy for an application to update the appearance of the PDF, if desired, whenever the structure tree is edited. It becomes simply a matter of manipulating the spacers to lay the document out in a different way. Consistency checks can be carried out if the ideas outlined in section 5 are followed. The addition of metadata to COGs, to describe their properties can check whether it makes sense to move some particular content to a new place. The structure tree, in essence, becomes a script for driving the order of the layout.

7. CONCLUSIONS

We feel enthusiastic and optimistic about the future of the COG model of PDF files Its application in creating material such as catalogues is plain to see: descriptions of items for sale could be authored in a variety of applications and placed onto pages, either via a linker script or via interactive placement, as currently implemented in our plugin [7]. The beauty of this approach is that last-minute alterations to a page, prior to going to press, are easily accommodated and the very nature of COGs allows one instance of an object to be shared among many pages rather than needing to be replicated many times.

Allied to all of the above advantages is the fact that COGs, of the granularity we propose, are natural candidates for participating in PDF structure trees, especially if the Adobe Standard Structure Tags, as set out in reference [1], are used. The structure tree can act as a template for reordering the COGs into any desired rendering order on the page.

Once some more experience has been gained in generating and manipulating COG-PDF files the time will come to tackle the much harder task of attempting to rewrite badly composed 'legacy' PDF pages into sequences of COGs.

8. ACKNOWLEDGEMENTS

We thank the Acrobat Engineering team at Adobe Systems Inc. for many useful comments and suggestions,

9. REFERENCES

1. Adobe Systems Incorporated, *PDF Reference (Third Edition) version 1.4,* ISBN 0-201-75839-3, Addison-Wesley, December 2001.

2. Adobe Systems Incorporated, *PostScript Language Reference Manual (Third Edition),* ISBN 0-201-37922-9, Addison-Wesley, February 1999.

3. Philip N. Smith and David F. Brailsford, "Towards Structured Block-based PDF," *Electronic Publishing—Origination, Dissemination and Design*, vol. 8, nos. 2 and 3, pp. 153–165, June/September 1995. Available on-line at `http://cajun.cs.nott.ac.uk/compsci/epo/papers/epoddtoc.html`

4. B. W. Kernighan, "A Typesetter Independent TROFF," Computing Science Technical Report No. 97, Bell Laboratories, Murray Hill, New Jersey 07974, March 1982.

5. *Universally Unique Identifiers (UUID).* `http://www.globecom.net/ietf/draft/draft-leach-uuids-guids-01.html`

6. Brian W. Kernighan and Christopher J. Van Wyk, "Page Makeup by Postprocessing Text Formatter Output," *Computing Systems*, vol. 2, no. 1, pp. 103–131, 1989.

7. The COG-PDF home page `http://www.eprg.org/cogs`