Some PostScript Transparency Experiments Using Acrobat 5.0

Don Lancaster
Synergetics, Box 809, Thatcher, AZ 85552

http://www.tinaja.com
don@tinaja.com
(928) 428-4073

What follows is a work in process. I had hoped to be able to show some simple **PostScript** utilities that let you explore the transparency features of **Acrobat 5**. My present solution works, but is horrendously awkward and incomplete. So, I am presenting it here in hopes you can add to it and help solve some key puzzles.

What we really want is one of these…

… gotten by sending ordinary **PostScript** code to **Acrobat Distiller**. Because PostScript uses an opaque imaging model, only an indirect method will normally work to produce transparent .PDF files. And such transparent .PDF files can only be transformed **back** into PostScript by an ungainly and cumbersome process. Thus .PDF will remain the format of choice for transparency distribution.

The following documents and tools are essential to understanding transparency…

PDF Reference Manual
PDFMark Reference Manual
PostScript Reference Manual
Transparency in PDF
Making a Transparent Color Rectangle
Acrobat SDK Development Kit

Some of these are included in the Acrobat 5 CD disk, while others can be gotten by using the above links. It is especially important to master the **COS Objects** section of the **PDFmark Reference Manual**.

I've placed a simple PostScript "transparent square on square" sourcecode as my **g9demox.psl**; its interim .PDF file as **g9demox.pdf**; and its final hand altered and fully transparent .PDF file result as **g9demo.pdf** Having these three files on hand as printouts will greatly ease understanding what follows.

It also pays to create a **BARE** Distiller Job Option ahead of time. A "minimal" one that does a 1.4 distill but does **not** do any compression. Or anything else complex such as embedding fonts. Later on and if needed, you can use the **FLATVUE.PSL** utility of **GuruGram #8** to convert any Flate streams to plaintext.

PDF Graphic State Objects

The .PDF format maintains a **graphics state** that is both open-ended and more complex than PostScript. Besides PostScript's usual **setgray** or its **setlinewidth** and similar operators, new graphics operators can be dictionary defined and passed to Acrobat objects.

Of crucial interest here is the **/CA** operator used to set the stroke transparency, the **/ca** operator used for the fill transparency, and the **/BM** operator used to pick all of the dozens of super fancy optional transparency mode names.

Those **COS Objects** are easily used by **pdfmark** to let Distiller create a new graphics state object to be installed in the final transparent .PDF file…

```
% make a new graphics state
[ /_objdef {newgsstate} /type /dict /OBJ pdfmark

% fill the new overlay graphics state
[  {newgsstate}
      <<
       /Type /ExtGState
       /ca 0.5                    % fill transparency is 0.5
       /CA 0.5                    % stroke transparency is 0.5
       /BM /Normal
       /AIS false
       /OP false
       /OPM 1
       /op true
      >> /PUT

      pdfmark
```

One new graphics state object will be needed for the transparency overlay. Since new graphics operators are cumulative and "remembered", use of bracketing saves and restores as **q** and **Q** will be needed to localize the tranparency to the desired graphics. As was the case on our previous page.

Note that this only makes a .PDF graphics state object. It does **not** link or install it for use by other objects.

Xobject Graphic Stream Objects

Adobe would like you to use a special type of subroutine called an **XObject** for your transparency graphics. They call these **/Group** objects having new subtype **/Transparency**. But apparently, any old graphics state in the page stream can be forced to take place at any time. Letting you (as we have on page one) go from opaque to transparent and back again in the same page stream.

I have yet to find any obvious way to create a Group object "subroutine" with PostScript. The closest I've found is the **BP**, short for Begin Picture generator…

```
% start a Begin Picture graphics overlay subroutine...
[ /BBox [0 0 1000 1000 ] /_objdef {overlaypix} /BP pdfmark

% fill the Begin Picture graphics overlay Xobject subroutine...
0 0.5 1 setrgbcolor     % on the aqua side of blue
4 4 6 6 rectfill

% complete the Begin Picture graphics overlay Xobject...
[ /EP pdfmark
```

Such a Xobject "subroutine" can be shown where desired by…

```
[ {overlaypix} /SP pdfmark
```

We now have the ability to create opaque objects, transparent objects, and the graphics states that let you be one or the other. Sadly, I have not yet found out how to make PostScript cause the crucial "**Please change to a new graphics state named zorch**" that forces distiller to enter a **/GS4 gs** or similar command into the present page stream. While simultaneously listing **/GS4 19 0 R** or wherever as an available resource.

This may be because I missed the obvious, or that **Adobe** omitted or made this capability difficult. Since **GhostScript** easily forces graphics states, leaving such a fundamental capability off of **PostScript** is utterly unthinkable.

My present sneaky and ungainly workaround instead involves…

Force Feeding a Python

OK, we can now use PostScript to create a .PDF file with all the graphics resources needed for transparency. But apparently do not yet know how to activate the links from the PostScript end. So, (he-he-he), we'll simply **post-edit the Acrobat file** to force the needed linkings.

To do this, bring the .PDF file up in Wordpad and save to a new filename. Then snoop around till you find the object number(s) for your new graphic state(s). Then find the object number of the page stream that is to include your transparency change. Then find the object number of the Resources for that page. Write these numbers down.

Next, decide how many new graphic state objects you have, and shop around for the lowest **unused** /GS variable names. In our square-on-square demo, **/GS2** is the first available. It is super important to not use the same variable name for two different graphics states!

To correct the graphics state linkings…

(1) Insert an appropriate new "change transparency" command such as **/GS2 gs** into the correct place in the page stream.

(2) Insert an appropriate "link graphics state" command such as **/GS2 19 0 R** into the correct place in the resources dictionary. The second number is the desired graphics state object.

(3) Save the new file and bring it up in Acrobat to auto-rebuild the cross reference and byte count. Then do a **save as** to preserve the repairs as your final file.

Should you need more than one linking, repeat the above process as needed. In the **g9demox.psl** example, a **/GS2 gs** will go into stream Object #2 , and a **/GS2 1 0 R** goes into Resource Object #8. The new graphics state is Object #1.

Failed attempts at linking...

There is an obvious **/PUT** command available to the **pdfmark** operator, but I seem unable to get it to do anything useful in the way of linking resources..

Ferinstance, the command string…

[{ThisPage} <</Resources <</ExtGState <<
/GS4 {newgsstate}>> >> >> /PUT pdfmark

… seems to create a **new** and **duplicate(!)** Resources entry that is ignored by Acrobat. Same goes for **{NextPage}**. And, while **{Catalog}** seems to enter just fine, this appears to be out of the resource inheritibility parentage range.

< insert many frustrating lost hours here >

I did manage to find an alternate method that is pretty nigh but not plumb. While somewhat more complex, it lets you actually create any XObject you like, but still demands a simple (often changing only one byte) post patch. I call it…

The Bait and Switch Method

A PDF stream object apparently consists of a data fork (the actual stream) and a resource fork (a companion dictionary). You can use pdfmark's **/PUT** command to

enter either stream or dictinary items. So long as you do only one or the other with any single **/PUT** command. Let's try it.

You have to work from the inside out. First, create a transparent graphics state pretty much like we did before…

```
[ /_objdef {XGS1} /type /dict /OBJ pdfmark

[  {XGS1}
      <<
        /Type /ExtGState
        /ca 0.5              % fill transparency is 0.5
        /CA 0.5              % stroke transparency is 0.5
        /BM /Normal
        /Name /XGS1
        /AIS false
        /OP false
        /OPM 1
        /op true
       >> /PUT pdfmark
```

Next, create a resource dictionary object for your Xobject…

```
[ /_objdef {XR1} /type /dict /OBJ pdfmark

[ {XR1} <<
    /ProcSet [/PDF ]
    /ExtGState << /XGS1 {XGS1} >>
      >> /PUT  pdfmark
```

And then create your actual Xobject, starting with its dictionary…

```
[ /_objdef {XOBJ1} /type /stream /OBJ pdfmark

[ {XOBJ1} <<
    /Type /XObject
    /Subtype /Form
    /FormType 1
    /Name /XOBJ1
    /BBox [0 0 1000 1000]
    /Resources {XR1}
    /Matrix [1 0 0 1 0 0]
        >> /PUT pdfmark
```

Note that Distiller has used an internal name of **{XGS1}** to link the Resource Dictionary to the graphics state. And **{XR1}** to link the Xobject to its Resource Dictionary. Note further that these variables are internal and local to Distiller only.

We then add the stream to our new Xobject…

```
[ {XOBJ1}
(0 0.5 1 rg
/XGS1 gs
4 4 6 6 re
f) /PUT pdfmark
```

Sadly, there are apparently still two big gotchas here. The first is that **you have to enter your stream data as PDF commands rather than PostScript**. Ferinstance, a **4 4 6 6 rectfill** becomes **4 4 6 6 re**.

If your PostScript is too complex to sight convert, try doing another PDF whose sole goal is to convert the code for you. The big advantage of this method is that you can in fact create and link graphic states to both your XObject and its stream any way you like. And transparency groups become possible.

The second infuriating gotcha is that I found no way to do a…

```
[ {XOBJ1} /SP pdfmark
```

…using **/SP** to enter your new XObject. Despite the Xobject and its Resource file done via **/BP** and **/EP** begin and end picture looking nearly identical to me. I am thoroughly puzzled over this apparent restriction. Chances are there is an internal picture name directory in Distiller I have not yet found.

So, to continue, you create a regular and temporary external form object the old **/BP** and **/EP** way. Just like we did earlier…

```
[ /BBox [0 0 1000 1000 ] /_objdef {tempoverlay} /BP pdfmark

0 0.5 1 setrgbcolor          % on the aqua side of blue
4 4 6 6 rectfill             % new square temporarily opaque

[ /EP pdfmark
```

When **/SP** gets called and distilled, a form object usually named **/FM1** gets created. The only tiny remaining problem is that the form resources are pointing to the wrong and opaque Xobject. We can finally do our "bait and switch" by post patching.

Post Patching is done simply by inspecting the file and finding the names of the wanted and unwanted transparency Xobjects. The patch itself is usually as easy as changing **/FM1 4 0 obj** to **/FM1 3 0 obj**. Normally **no xref rebuild should be needed**, so long as the old and new object numbers have the same number of digits. Be sure to watch this detail.

Thus, our post patch still remains but is much simpler and safer than before. New example code for you to inspect and use appears as **g9demoy.psl**; its intermediate .PDF file as **g9demoyx.pdf**; and its final hand altered and fully transparent .PDF file result as **g9demoy.pdf**.

A Few More Gotchas

Using Wordpad for post editing can be dangerous as it may trash an occasional control command character in a font or other compressed stream. Word itself is often acceptably better, but your best bet is to do a true PDF document search and replace that has no excluded characters and makes no changes in line ending terminations. A custom **PostScript-as-language** routine can easily handle this. As could a **custom plugin**.

While you can apparently add any dictionary item or append any stream content to your external Xobject at any time, apparently the Xobjects created internally by the **/BP** and **/EP** process will **not** let you append stream content. But they will let you add less useful dictionary items. Note once again that **/PUT** will create a duplicate and useless dictionary entry instead of overwriting an existing one.

PDFmark defined streams are apparently forced into Flate Decoding whether you like it or not. If needed, you can uncompress these using **FLATEVUE.PDF** or with the **SDK Uncompress plugin**. Note that the later infuriatingly "erases" anything not yet correctly linked.

Adobe does **not** want you to make graphics state changes inside an Xobject. Since our example internally forces an internal Xobject transparency state change, certain post PDF plugins might not work as they may get overridden. Specifically, the **UncompressPDF** in the **Acrobat SDK** apparently will not let you change the transparency of the blue box on page one.

How Many Mathematicians Does it Take...

… to change a light bulb? Only one, who hands the bulb to four Californians. Thus reducing the problem to a previously solved riddle.

We thus now have a method to explore and actually use Acrobat transparency by using distilled PostScript. It works just fine but needs an annoying hand patch at end process. To make the solution general and convenient, your help is needed in solving the (apparently) simpler problem of using PostScript to force a graphics change and autolink in Distiller.

As of this latest **GuruGram** update, we have full PostScript transparency via Distiller, but still require a one byte, non index trashing post patch. The remaining problem boils down to "How do I feed the **/SP** pdfmark command in Distiller my own named Xobject that was not created by **/BP** and **/EP**?"

Let's hear from you.

Consulting services available per **http://www.tinaja.com/info01.asp**.