A Gonzo PostScript PowerPoint Emulation

Don Lancaster
Synergetics, Box 809, Thatcher, AZ 85552
copyright c2003 as GuruGram #20
http://www.tinaja.com
don@tinaja.com
(928) 428-4073

I was rather disappointed when I first tried to web distribute a slide show that I initially created in PowerPoint. Files were fragmented and huge, appearance was awful, display was glacial, and linking was cumbersome. Worst of all, it was not letting me do my things my way.

So, I instead decided to use my **Gonzo Utilities** written in **raw PostScript** to emulate much if not all of PowerPoint. The new emulation can be used from the ground up to create new slide shows, or can be used to convert an existing PowerPoint presentation to one that web distributes significantly better. It should be most useful on Linux and similar open source platforms as well. The results are totally GIF free. Large bitmaps are also easily avoided.

A demo of the new emulator **appears here** along with its **generating sourcecode**. The actual gonzo utilities are **found here**. As always, **Gonzo** gives you incredible control, total device independence and complete flexibility. But is proudly not WYSIWYG. Gonzo also has a moderate to steep learning curve.

What follows will make the most sense if you have an open **sourcecode** window and a **demo slideshow** nearby. Let's first pick up a few of tools we will need...

Backgrounds and Patterns

PowerPoint slide and splash backgrounds are normally .GIF files. Besides licensing restrictions, these have problems with file size and PDF conversion. The **PostScript pattern** capability found in the **PostScript Language Reference Manual** would seem to be a good workaround to huge .GIF bitmaps.

A pattern is simply a "rubber stamp" that gets repeated over an area. These can have quite small file sizes, yet can quickly replicate themselves. Easily covering what would normally take a huge bitmap. Any repeatable image or artwork is a good pattern candidate.

Here's how to take a small image and convert it into a replicating pattern...

You first create a pattern dictionary…

```
/bodypatdict
  << /PatternType 1                % Tiling pattern
    /PaintType 1                   % Colored
    /TilingType 1
    /BBox [0 0 50 50]
    /XStep 5                       % one-tenth for gonzo grid
    /YStep 5
    /PaintProc {begin bodyrandimage end}
  >>  store
```

And an image (or other proc) that you want to replicate…

```
/bodyrandimage { gsave  5 dup scale
 <</ImageType 1                    % always one
   /Width hpixels                  % width in pixels
   /Height vpixels                 % height in pixels
   /ImageMatrix [hpixels 0 0
      vpixels neg 0 vpixels ]
   /DataSource bodypatstring       % proc to get string data
   /BitsPerComponent 8             % color resolution
   /Decode [0 1 0 1 0 1]           % per red book 4.10
        >>
   image                           % call the image
   grestore} def
```

In this case **bodypatstring** is a string source of image data which we will look at shortly. This could also be a small bitmap read as a file or inline as an **currentfile** embedding.

And here is how the pattern actually gets used…

```
save /snap1 exch store             % show the body background
   bodypatdict
   matrix                          % Identity matrix
   makepattern                     % Instantiate the pattern
   /bodypat exch def
   0 0                              % for the actual slide size
   slidewide slidehigh
   /Pattern setcolorspace
   bodypat setcolor rectfill       % Fill rectangle with pattern
snap1 restore
```

A **save** and **restore** should also be used around each individual slide. Otherwise a "consumed" string pointer on the previous slide may generate errors.

Some Random Patterns

One problem I found with patterns is that they can be excruciatingly slow if the underlying generating proc is highly complex. A solution to this is to initially convert any really slow code into an **image**, and then replicate the image.

We've already seen some interesting random images used for **photo backgrounds** and for our **Dodge & Burn** utilities. Here's a new random pattern generator with some unique features.

You start with an array of **[r g b texture]**. Where **r**, **g**, and **b** are the "nominal" color desired. And **texture** is the "spread" of the three pixel colors used to make up the random pattern. A "spread" of **40** might be appropriate for adding modest background interest, while **120** will give more dramatic splash color variations.

Three rgb pixel values are then generated whose values deviate from "nominal" by a zero-centered random number whose limit is set by **texture**. Like so…

```
/buildpixels {aload pop /texture exch store /b0 exch store
/g0 exch store /r0 exch store 123 srand

/r1 r0 texture random texture 2 div sub add cvi
   dup 255 ge {pop 255}if  dup 0 lt {pop 0} if store
/g1 g0 texture random texture 2 div sub add cvi
   dup 255 ge {pop 255}if  dup 0 lt {pop 0} if store
/b1 b0 texture random texture 2 div sub add cvi
   dup 255 ge {pop 255}if  dup 0 lt {pop 0} if store

/r2 r0 texture random texture 2 div sub add cvi
   dup 255 ge {pop 255}if  dup 0 lt {pop 0} if store
/g2 g0 texture random texture 2 div sub add cvi
   dup 255 ge {pop 255}if  dup 0 lt {pop 0} if store
/b2 b0 texture random texture 2 div sub add cvi
   dup 255 ge {pop 255}if  dup 0 lt {pop 0} if store

/r3 r0 texture random texture 2 div sub add cvi
   dup 255 ge {pop 255}if  dup 0 lt {pop 0} if store
/g3 g0 texture random texture 2 div sub add cvi
   dup 255 ge {pop 255}if  dup 0 lt {pop 0} if store
/b3 b0 texture random texture 2 div sub add cvi
   dup 255 ge {pop 255}if  dup 0 lt {pop 0} if store

    } store
```

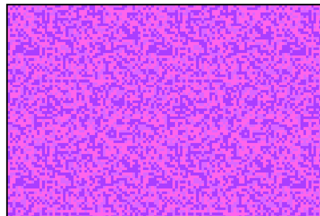
We then take these pixels and generate a 50 x 50 random array of them…

```
/makeimagestring {  12345 srand          % make repeatable
  /iarray mark
  issize { r1 g1 b1} repeat ] store      % fill all w/color 1

  issize 2 div cvi { iarray  issize      % fill half w/color 2
  random 3 mul cvi  [ r2 g2 b2 ]
  putinterval} repeat

  issize 2 div cvi { iarray  issize      % fill third w/color 3
  random 3 mul cvi  [ r3 g3 b3 ]
  putinterval} repeat

  iarray makestring
              } store
```

And then convert the array to an image string using this disgustingly elegant off-the-wall routine…

```
/makestring {dup length string dup /NullEncode filter
3 -1 roll {1 index exch write} forall pop} def
```

Here is what our initial body and splash random patterns look like…



A 50x50 image adds 3x2500 = 7500 bytes to the length of your PDF file. Being random, compression is not going to help at all. A body and splash pattern pair will thus need 15K. You can reduce these values by going to smaller replicating images. Smaller images may add unattractive repeating patterns, though.

Speaking of which, note the **PostScript srand** random seed operator. This guarantees you will get the **same** random pattern each time. Should a chosen pattern have glitches or repeats in it, simply try a different seed.

Handling URL's and Links

There are two elements to an Acrobat link: The url itself and an **action box** in which that url will be activated on clicking. Manually reentering and repositioning on-page action boxes with **Acrobat** can be an enormously painful process.

Instead, **Gonzo** provides for **fully automatic action box tracking** by placing markers inside its text. Thus, the action boxes will automatically move around as text gets longer or shorter. As well as automatically scaling the action box size to approximate the font height and message length. Final page placement requires no further effort at all.

Full details appear in our **AUTOURL.PDF** tutorial found in our **GuruGram** library.

Emulator Organization

The key to the Gonzo emulation is to have a **slideshowdict** dictionary that names and positions everything that is likely to be needed. In general, you'll want to use **indirect** references when and where possible, so you can add as many styles and options as you care to. Eventually building up an entire library of useful patterns and layouts.

One very important **slideshowdict** dictionary entry is the **sequence** array. Which sets the order that your slides will appear in…

```
/sequence [
  /an_introduction_to_magic_sinewaves
  /magic_sinewaves_are
  /magic_sinewave_features
    . . . . . . . .
  /for_additional_help
  /this_has_been
              ] store
```

Slides are easily added, removed, duplicated, or rearranged by changing this array. Slides can be hidden by commenting them with a leading "**%**". Note that the slides can be defined in any order in your code. Only their position in this **sequence** array determines their output order.

The actual slideshow emulator routine is amazingly simple…

```
/makeslideshow {setupshow
  sequence {
      save /snapxx exch store
      cvx exec showpage
      snapxx restore
         } forall
              } store
```

This first initializes your pattern strings and does a few other housekeeping items. It then grabs the slides one by one and executes them in order. A slide description will typically be a title and a titleproc, a body and a bodyproc, and perhaps some additional stroke graphics.

A single **makeslideshow** command generates the entire presentation.

A Guided Tour

The emulated **slide show** is around 50K long, or less than **one-fifth** the original. As noted, the code can be further shortened if desired.

Each and every slide images nearly instantly without lengthy bitmap delays. All slides have outstanding typography and fully magnifiable superb artwork. All links work as expected and need no further prep.

The final show is totally device independent and requires only a .PDF reader for the platform of interest. No GIF's are used at all, avoiding all licensing problems. Nor is any part of the PowerPoint code. Single-file sourcecode is a mere 35K long.

Here's some slide-by-slide comments on features and details of note…

an_introduction_to_magic_sinewaves — This title slide uses an ivory background and a magenta splash, helped along with a few magenta lines. The URL link is auto generated by the sourcecode. My preference is to never underline a URL.

magic_sinewaves_are — A typical body slide. I chose to use gray as a title color to reduce the harshness of black. Same with the bulleted entries. Bullets are a Zapf Dingbat and once set, stay the desired size. Bullets could be tinted if desired.

and_limitations — colored text is easily highlighted, but use red sparingly and only for key points.

magic_sinewave_appearance — This uses a **real** magic sinewave to **exactly** generate the needed artwork. Only a few dozen bytes of code are needed. Display time is ridiculously faster than a converted .GIF bitmap. The degree symbols are faked using a superscript font and a plain old "**o**".

typical_unfiltered_spectrum — Again, this uses a few bytes of Gonzo stroke graphics to create a graph. Horizontal and vertical lettering can be mixed and matched at will. PowerPoint .GIF graphs do not translate lettering very well.

typical_unfiltered_spectrum_FLASH — This adds a "key point" message on top of an existing slide. It also shows us how we can do progressive builds. Flashing has been added to this slide by using **JavaScript** as detailed in the **PDFFLASH.PDF** of **GuruGram #48**.

magic_sinewave_generation — A typical electronic schematic. This takes heavier lettering than the normal **Gonzo** electronic symbols. As before, the viewing time is much faster than an converted .GIF bitmap.

two_important_magsine_types — Two colored text entries were added to prevent the "everything bold" from overwhelming. Note the fine kerning of the math expressions. An extra point of spread goes between the parenthesis or the slashes.

key_magsine_secret_I — A mix of a simple stroke graphic, text, and a red highlight.

key_magsine_secret_II — A slightly smaller font is used for the numerals. Gonzo places no sane limits on font sizes or variations. Again, there is slight kerning on the parenthesis.

fourier_pulse_properties — The big math expressions form a graphic in themselves. Note that a "*" is normally too small and too high to look good. So a larger and subscripted font was used for the asterisks. As usual, some subtle kerning makes for top appearance.

the_magic_equations — This is **waaay** too much math detail to show in a slide. But it is the key point of the presentation and is easily magnified. Most viewers eyes will blur over anytime they see either math or poetry. While still delivering the message "this is a key bunch of very messy math". I didn't fix the asterisks here, but easily could have done so.

equation_simplification — Always work in a terrible and unexpected pun.

quantization — A typical slide with an auto-tracking, auto-boxing link.

how_big_should_n_be — Use exclamation points very sparingly.

for_additional_help — Combining several URL's with text.

this_has_been — Keep "penalty of death" notices small. But still obvious.

Getting Fancy

Note that **Acrobat** PDF has a **full screen** mode that lets you do fancy transitions, automated sequencing, and such. Thus, most PowerPoint features are easily emulated. Flashing is easily done using the JaveScript concepts of **PDFFLASH.PDF** found in **GuruGram #48**.

For some reason, **Adobe** steadfastly refuses to let you run full screen .PDF from within a browser. Thus, you'll have to actually download the file before running a full screen display.

For More Help

Additional background along with related utilities and tutorials appears on our **GuruGram**, **PostScript**, **Acrobat**, and **Fonts & Bitmaps** library pages.

Consulting assistance on any and all of these and related topics can be found at **http://www.tinaja.com/info01.asp**. As can our presentation development services.

Additional **GuruGrams** await your ongoing support as a **Synergetics Partner**.