A Digital Airbrushing Algorithm

Don Lancaster
Synergetics, Box 809, Thatcher, AZ 85552

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

Classic airbrushing was a technique of using elaborate masks and a miniature paint sprayer to cover up and improve portions of a photo. So they looked better or eliminated perceived defects. We can instead define digital airbrushing as…

> **DIGITAL AIRBRUSHING —**
>
> **Given four points on a bitmap image p0, p1, p2, and p3, match the color at those points and linearly gradient blend all intermediate points inside that defined area.**

You might want to do this because there is a bad lighting burn. Or dirt to get removed. Or are doing historical retouching. Or want to improve the top cover on a test equipment image. Or are preparing **eBay** resale images .

We looked at some fundamental new image manipulation tools in **BMP2PSA.PDF** as **GuruGram #84**. As our first application example, we will expand these tools into a set of **digital airbrushing utilities**. As additional features, we will allow the airbrushing to be **any** quadrilateral shape and not a simple aligned rectangle.

We will further allow adding randomized **texture** of arbitrary depth to improve the "interest" and "match" of the corrected area.

While the algorithms we will look at can be adapted to any language that can access host disk files, easily manipulate data arrays, handle the **HSB** and **RGB** conversions, and cleanly detect path insidedness, we'll focus on using **PostScript arrays of strings** and its superb **infill** operator as a preferred solution.

A preliminary algorithm study appears **here** with its results **here**.

The working utilities are found as our file **AIRBRUT1.PSL**. An example demo appears **here**. A bad lighting burn has been partially eliminated lower right and some speckle upper left. Mild texture has been added to match the image areas.

The Algorithm

There is some subtlety and sneakiness that makes digital airbrushing nontrivial. Here is a useful approach…

A DIGITAL AIRBRUSHING ALGORITHM—

(1) Establish four points x0,y0 x1,y1, x2,y2, and x3,y3 CLOCKWISE FROM THE LOWER LEFT in the area to be airbrushed.

(2) Calculate a bounding box that will contain all of the control points.

(3) Find the HSB values at the corners of the bounding box that would be needed to create a linear two dimensional gradient that matches the four control HSB values. This will involve solving four linear equations in four unknowns.

(4) Establish the quadrilateral path and make corrections to only those inside pixels.

(5) Normalize the bounding box to a unit square. Then use bilineal interpolation to determine all needed internal pixel values.

(6) Optionally randomize each pixel RGB value to add a desired depth of textural interest.

It is very important to note that **you normally will want to work in HSB (hue-saturation-brightness) space rather than RGB (red-green-blue**). The reason is that a linear RGB transition from red to green goes through gray, while a similar HSB transition goes from the more expected red to orange to yellow to green.

PostScript is especially adept at converting between these two color standards…

```
HSB to RGB:
   hue sat bright setHSBcolor currentRGBcolor
RGB to HSB
   red blue green setRGBcolor currentHSBcolor
```

Several minor points before continuing: Our **PostScript** examples optionally use our **Gonzo Utilities**. Our provided .BMP image airbrushing **utility** textfile is

first modified and then gets run by **Using Acrobat Distiller as a General Purpose PostScript Computer**. Which will pick up a selected **.BMP** image, modify it, and save it to a new filename. No knowledge of PostScript is needed for this utility.

Distiller versions newer than 8.1 prevent diskfile access as a default. To use these, enter Distiller from your command line using "**acrodist -F**".

This application expands upon our **Bitmap to PS Array Conversions tutorial** and **utility** from our **GuruGram** library. Other enhancements and expansions are planned in a continuing series.

The **.BMP Data Format** may demand one, two, or three **padding bytes** to make each new row start on a 32 bit boundary. While the math to do this is somewhat obtuse, the needed count apparently simplifies to **xpixels 4 mod**. This detail may not yet be picked up in some of the earlier material.

A Preliminary Study

A simplified preliminary study was first done to explore the airbrushing algorithm. It is found **here** with its results **here**. Our usual approach to bitmap adjustments is to convert the bitmap to a **PostScript array of strings**. Those strings are then modified and then either saved as a new bitmap or as a PostScript image.

Only a single array of strings is used in the study. This could represent one plane of HSB or RGB color space. The data values were first set to **46** which will appear as an ASCII period. Quadrilateral control points of **97**, **104**, **112**, and **122** were then selected clockwise from lower left. In the output log reports of the demo, these will appear as lower case **a**, **h**, **p**, and **z**.

A rectangular **bounding box** is next found that will exactly contain all of the quadrilateral control points. A lower boundary can be found by comparing two points and discarding the larger one; comparing another two points and again discarding the larger one; and then comparing and discarding the larger of the two remaining points…

```
/bbxmin llx ulx 2 copy gt {exch} if  pop
        urx urx 2 copy gt {exch} if  pop
                2 copy gt {exch} if  pop store
```

The upper boundary is found by discarding the smaller of each pair, and the **y** max and min values are handled similarly. These four corners are placed in the demo log as lower case ASCII "**o**" characters of **111**.

A **bilineal interpolation** is a suitable tool to generate gradients in most images. Assume you have a unit square and a new sought after point fractional distances **x** and **y** from the corners. The general bilineal interpolation will be…

```
newvalue = (1-x)(1-y) * ( llvalue ) +
           (1-x)(y) * ( ulvalue ) +
             (x)(y) * ( urvalue ) +
           (x)(1-y) * ( lrvalue )
```

We already know the four input quadrilateral values and their fractional **x** and **y** distances. But we do not know the four bounding box values. We can write four linear equations in four unknowns to solve for our needed corner values…

```
A0 = W0*w  + X0*x + Y0*y + Z0*z
A1 = W1*w  + X1*x + Y1*y + Z1*z
A2 = W2*w  + X2*x + Y2*y + Z2*z
A3 = W3*w  + X3*x + Y3*y + Z3*z
```

Here **A0** through **A3** are the input values. And **W0** through **Z3** are the known fractional distance products inside the unit square to be interpolated. And **w** through **z** are our sought after corner bounding box values.

Note that this is "backwards" from normal and here we are really seeking out an **inverse transformation** solution. Such equations are easily solved with **Gaussian Elimination** using this **specific 4x4 Utility**.

We could now scan all pixels inside our bounding box and find their interpolated gradient values. But we will only want to **change** the pixels that are **inside of** the original quadrilateral.

To decide which pixels to change and which to leave alone, we can first create a new **PostScript** path. This path will be an analog of the **mask** used in traditional airbrushing. It represents a **1:1 map** of the pixels to be modified…

```
newpath
llx lly moveto
ulx uly lineto
urx ury lineto
lrx lry lineto
closepath
```

The little known and wonderfully enigmatic PostScript **infill** operator can then be used inside a double loop to decide which pixels to update. **infill** tests an input point to decide whether it is outside or inside of the path. And returns a **true** or a **false** result…

```
bbymin 1 bbymax {/cury exch store
bbxmin 1 bbxmax {/curx exch store

  curx cury infill {changepixel} if

        } for
      } for
```

Doing this on the study produces a "gradient alphabet". On our real program, the process gets repeated once for each **HSB** plane per pixel. Speaking of which…

The Actual Airbrushing Utility

Inputs needed to use **AIRBRUT1.PSL** are the input and output filenames and paths. Plus an **airbrushboundaries** array that holds eight **x** and **y** data values representing four quadrilateral points **clockwise from lower left**.

The utility then pretty much follows an expanded version of the study. Input data points are grabbed and **HSB** converted. As a reminder, the **HSB** color space gives more realistic blends than **RGB**. The input file is then read and converted to a group of three **PostScript** arrays of strings, one each for hue, saturation, and brightness.

A rectangular bounding box is next found. Its **HSB** corner values are calculated by repeatedly solving the above four equations in four unknowns. From there, a new **PostSript** masking path is created, and the **infill** operator used to selectively alter the chosen pixels. Finally, the three arrays of strings are written to a new output **.BMP** file of your choosing.

Details on all of these operations can be reviewed by studying the comments in the **AIRBRUT1.PSL** utility code. Extensive printout options can be commented in or out to study each portion of the utility in more detail.

Adding Texture

Most bitmap images will involve some grain or variance. If you try overwriting them with a "pure color" gradient, the results may look too "computer like". Our airbrushing utility has a provision to **slightly randomize** the actual RGB pixels being placed. This is done by picking a small random number range, such as **-4** to **+4** and adding the value to each pixel plane. A limit check is performed after addition to make sure results stay in an integer **0-255** range.

Minor randomizing in **RGB** space ends up about the same as **HSB** would end up. A **textdepth** lets you adjust your texture to match that of the original image. Because of its speed penalty, a **textureflag** flag allows you to selectively turn the texturizing on and off.

Like so…

```
/texture {   textureflag {textdepth random
             dup 2 div sub  add cvi } if

             dup 255 gt {pop 255} if
             dup 0 lt {pop 0 } if
          }  store
```

The **random** operator is extracted from our **Gonzo Utilities**. A **5 random** would return a random digit from **0** to **4**.

Some Extensions

Speed has not yet been optimized. Present operation is acceptably fast for smaller airbrushing areas. Rates are in the **30,000** pixels per second range. An obvious speedup on multiple airbrushing tasks can be done by converting bitmaps to and from **PostScript** arrays of strings only once per work session.

Another obvious improvement would be to allow mouse point entry instead of typed data coordinates. This possibly could be provided through extensions to **GhostScript**, Acrobat SDK tools, or a supervisory language linking.

At present, **blends through red are not permitted**. A workaround can be based on a pseudo color phase shift. **There always will be one hue that you must not cross during blending**. A fix has been added below.

A detail: **.BMP** and **PostScript** images build from the bottom up. The **y** coordinate in **Paint** builds from the top down. To convert **Paint** data, **subtract the needed y coordinate from your total Paint height**.

And Some Revisions

Two improvements were made to **AIRBRUT1.PSL** in April of 2008. These attempt to deal with two quirks of HSB space airbrushing. The first being that **a blend is not allowed through any hue with an 0/1 ambiguity**. This is red by default. With a symptom of the blends going "the wrong way" around the hue circle. And can be dealt with by temporarily **phase shifting** all of the hues by a new constant.

The second quirk being that **hue becomes arbitrary and ambiguous very near a true gray**. This can be dealt with by using a RGB blend instead of a HSB blend. Or by making sure that "hints" of the blending colors remain in the gray itself.

There is now a new **usehsb** Boolean. When true, the blends will be in hsb space. Blends will be bright and closer to saturation, but rarely may go "the wrong way around" for one particular color. Which defaults to red. Blends may also have slight hue fringes near any true grays.

When false, a conventional RGB blend is done. This will be less saturated, "darker", and more into the gray. But should eliminate any gray hue shifting.

To shift the "problem" hue to a new value, there is now a new **hueshift** variable whose intended range is **-0.5 to +0.5**. The zero default value blends through aqua and has the **1/0** hue discontinuity at red. A value of **+0.16** blends through blue with an orange problem. A value of **-0.16** blends through green with a purple problem. And a value of **-0.5** blends through red with an aqua problem.

Hue shifting adds or removes a chosen amount of color phase through the 0 to 1 hue space. After interpolation, the opposite phase shift is then cranked in, resetting the colors to their normal values. As done by two new **hue+** and **hue-** routines.

A second route to dealing with tinting near true grays is to **customize the gray that is being used in the target image**. For instance, if you are blending blues into a gray, a RGB gray of **50 50 51** is much less likely to create problems since a "hint" of blue remains and no blending is called for.

These new techniques should rarely be needed. They can be activated when and as any specific problem comes up on any particular image. Defaults should be a **usehsb** of true and **hueshift** of zero. At present, a modest slowdown is caused by the new code.

For More Help

Once again, our preliminary airbrushing study appears as **AIRSTUDY.PSL** and its viewable results file as **AIRSTUDY.LOG**. The working utilities are **AIRBRUT1.PSL** A demo is found at **AIRDEMO1.JPG**. This **GuruGram** file is named **AIRBRUSH.PDF** and its source code is **AIRBRUSH.PSL**.

The underlying **.BMP** to and from PostScript strings utility is **AOSUTIL1.PSL** and its companion tutorial is **BMP2PSA.PDF**. A **.BMP** format tutorial called **EXPBMP.PDF** is also in our **GuruGram** library. Our Gonzo utilities are found as **GONZOUTILS.PSL** and its tutorial as **GONZOTUT.PDF**.

Similar tutorials and additional support materials are found on our **PostScript** and our **GurGram** library pages. As always, **Custom Consulting** is available on a cash and carry or contract basis. As are seminars. For details, email **don@tinaja.com**.

Or call **(928) 428-4073**.