https://doi.org/10.2352/issn.2169-2629.2019.27.12

Color processing and management in Ghostscript

Michael J. Vrhel, Artifex Software, 1305 Grant Avenue, Suite 200, Novato, CA 94945

Abstract

Ghostscript has a long history in the open source community and was developed at the same time that page description languages were evolving to the complex specification of PDF today. Color is a key component in this specification and its description and proper implementation is as complex as any other part of the specification. In this document, the color processing and management that takes place in Ghostscript is reviewed with a focus on how its design achieves computational efficiency while providing flexibility for the developer and user.

Introduction

Page description languages (PDLs) such as PDF offer a wide variety of definitions of color for images, graphics and text. In addition, PDF includes specifications for over-printing and transparency blending that can become complex. For these reasons, the design of software to render the PDF content to images or convert it to other PDL formats requires careful thought. In addition, this rendering software is often embedded within a printer, and so it must be able to operate efficiently with limited resources.

In this paper, the color processing and management methods in the open source project Ghostscript are reviewed. Various steps that must take place to ensure proper color rendering of PDF files are discussed. The efficiency of the code and its use of multiple cores as well as SIMD (Single Instruction Multiple Data) operators are discussed and performance results are given.

PDLs

The first page description languages were PostScript, developed by Adobe Systems in 1984 [1], and PCL introduced by Hewlett-Packard in 1984 [2]. PostScript is a stack-based programming language and as such has great flexibility. Over the years, several versions of each of these specifications were developed, culminating with PostScript Language Level 3 (PS-LL3) in 1997 and PCL6 in 1995. The color model in PCL is gray or RGB based with the assumption that the data is contained in sGray[1] or sRGB [3] color space. PS-LL3, on the other hand, has a rich definition of color with the use of multi-dimensional lookup tables (MLUTs) for source color spaces as well as output color spaces[2]. CIE-based color support was introduced in PS-LL2 in 1990 with the ability to specify matrix-based mappings for source colors and MLUTs for destination devices. Spot color support was introduced in PS-LL2 with the separation color space. This addition to the specification allowed the drawing of an object using a single colorant that need not be RGB, gray or CMYK based. To ensure that devices that were unable to use a spot colorant could create a reasonable reproduction of the document, an alternate tint mapping to a common color space such as RGB, gray, CMYK or a CIE-based color space is required. PS-LL3 introduced a DeviceN color space, which allowed the use of arbitrary spot colorants possibly mixed with standard CMYK colorants. As with the separation color space, the DeviceN color space description is required to include a mapping to a more common color space.

The PDF 1.0 specification, which was introduced in 1993, had support for device-dependent gray, RGB and CMYK colors. Initially, it was not possible to make any device-independent color specification. In 1999, PDF 1.3 was introduced with support for ICC color spaces. In 2001, PDF 1.4 introduced transparency into the PDF specification with the ability to specify particular color spaces in which to perform transparency blending. The PDF 1.7 specification was turned into an ISO standard (ISO 32000-1:2008) in 2008. In 2017, version 2.0 was introduced (ISO 32000-2), which supports the use of ICC profile version ICC.1:2010 (ISO 15076-1:2010) [4].

Spot color support in PDF was introduced with the separation color space in version 1.2 in 1996 and the DeviceN color space in version 1.3 in 1999. These color spaces were specified in a manner similar to the PostScript separation and DeviceN color spaces.

The PDF specification includes several specialized versions including PDF/A for archiving, PDF/E for engineering and technical documents, PDF/UA for universal access, PDF/VT for variable and transactional printing and PDF/X for graphics exchange, which is geared toward the print industry. PDF/X allows the specification of a particular output intent for which the document is targeted. This output intent can be specified either by a specific ICC profile embedded in the document or a key word (e.g. CGATS TR 001 SWOP). Version PDF/X-5n includes the ability to specify an xCLR ICC profile for the output intent.

Ghostscript

First released in 1988, Ghostscript is an open source project originally developed by L. Peter Deutsch as an implementation of a PostScript interpreter [5]. Since that time, Ghostscript has been expanded to support PostScript LL3, all versions of PDF through the current 2.0 version, Microsoft XPS and ECMA 388 Open-XPS format [6] as well as all the various versions of PCL. Ghostscript is currently distributed under the GNU AGPLv3 license as well as a commercial version. It is a key component of Linux CUPS (Common UNIX Printing System) and, as such, has millions of installs and users.

Ghostscript can be roughly divided into three sections: the interpreters, the graphics library and the output devices, as shown in Figure 1. The interpreter has the task of converting the commands of the input language to common forms that the graphics library and/or the device understand. For example, the languages

[1] A colorimetrically neutral color space with a gamma and white point that is the same as sRGB.

[2] These source MLUTs are called color space arrays, and the destination MLUTs are called color rendering dictionaries (CRDs). The PS-LL2 specification includes CRDs but no color space arrays.

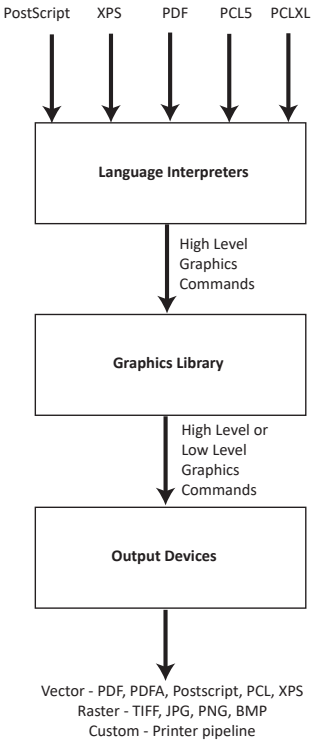

Figure 1. *High-level view of Ghostscript's architecture.*

all have the concept of drawing a path and filling it with a particular shading or placing an image at a specific location on the page. Each language has different syntax for describing these operations, and it is the job of the interpreter to map these commands to a common command such as fill_path or put_image.

The output device may directly understand these higher level vector-based commands. For example, if the output device is a PDF output device, the device will take the fill_path command and construct the PDF syntax needed in the output PDF file to specify the path and the color to fill the path. In this way, Ghostscript can map from one high-level language (e.g. PDF) to another high-level language (e.g. XPS) maintaining scalability. If the output device is a raster format (e.g. an image or printer engine), then the device will not know directly what to do with a command like fill_path. In this case, the graphics library must process the command into the highest level command that the device can handle. For example, the device may have hardware capability to do shading fills of trapezoids. As such, as the graphics library breaks down a fill_path command to more fundamental fills, shading fills of trapezoids when encountered in that process would be handled directly by the device.

Ghostscript Color Management

Prior to 2010, Ghostscript color management was primarily based on PostScript color methods with embedded ICC profiles handled with the open source Argyll color management system [7]. With the release of version 9.0 in 2010, Ghostscript switched to a pure ICC-based color processing approach using the open source Little CMS color management engine as its CMM [8]. At

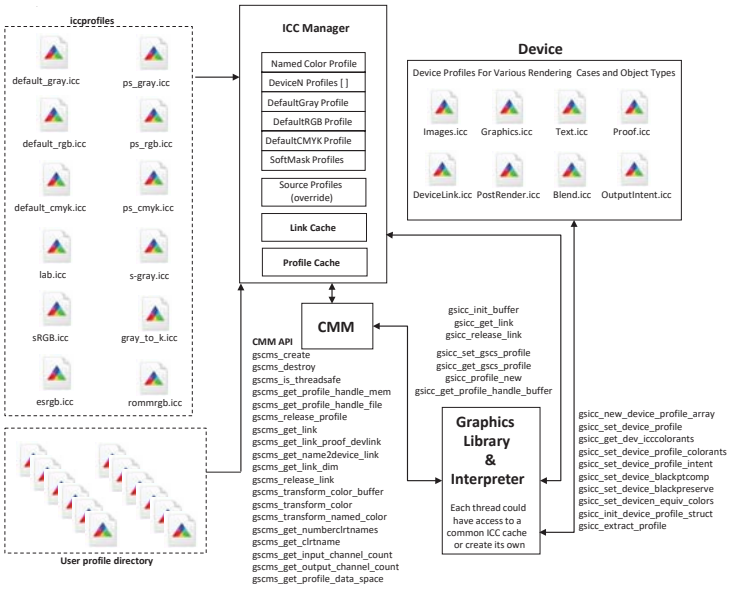

Figure 2. *Color architecture and API for CMM in Ghostscript.*

this time, an API[3] was specified that enabled the straight-forward use of any CMM with Ghostscript. Today, Ghostscript uses a thread-safe fork of little CMS [9, 10]. A diagram of the system as it exists today (with the current release of version 9.27) is shown in Figure 2.

Non-ICC CIE-based color spaces

Although Ghostscript has moved to a pure ICC work-flow, non-ICC color spaces that are CIE-based still exist in documents and, as such, Ghostscript must handle those in a manner that enables them to be processed by the CMM. The CMM is unlikely to understand how to handle PostScript color space arrays or PDF's non-ICC CIE-based color spaces of CalRGB and CalGray. To handle these color spaces, Ghostscript converts them to equivalent ICC profiles that can be understood by the CMM. To avoid repeated conversions of the same color spaces, Ghostscript will compute a hash of the contents in the case of PostScript or make use of the color space object resource number in PDF and store this hash number with the equivalent ICC profile in a most recently used (MRU) cache as shown in Figure 3.

Device-dependent color spaces

PDF allows the use of device-dependent color spaces specified by the keywords DeviceGray, DeviceRGB and DeviceCMYK. In some cases, a PDF file will specify in a Resource Dictionary values for DefaultGray, DefaultRGB and/or DefaultCMYK color spaces in terms of ICC profiles. In this case, those ICC profiles are used wherever the related device-dependent color space is used. If the PDF file is of the PDF/X form with an output intent specified (for example as a CMYK ICC profile), then all DeviceCMYK colors in the document are assumed to be specified by that ICC profile.

[3] Application Programming Interface

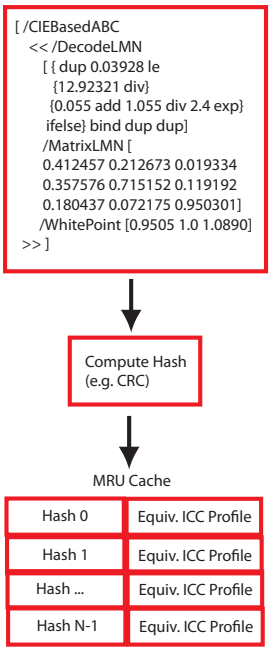

Figure 3. *Caching of equivavlent ICC profiles based upon hash of the color space contents. Note PS description for sRGB color space.*

If none of the above definitions for these device-dependent color spaces are contained in the document, Ghostscript will treat DeviceGray as being defined by the sGray ICC profile, DeviceRGB as being defined by the sRGB ICC profile and DeviceCMYK as being defined by a profile based upon CGATS/SWOP TR003 2007 CMYK. Ghostscript allows the user to define other ICC profiles for the device-dependent color spaces with its command line options, where, for example, including

```
-sDefaultCMYKProfile=''MyCMYKProfile.icc''
```

means that the code will use that profile to define DeviceCMYK color spaces in the document.

Spot Colors

The handing of spot colors by Ghostscript can occur in a number of different ways depending upon the capabilities of the device and the needs of the user. For example, if a document contains a spot color (e.g. Pantone 631) and the output device only understands CMYK, then that spot colorant is converted to CMYK using the alternate tint transform specified in the document. This is Ghostscript's default behavior when processing spot colorants for devices that do not support spot colorants. In this case, a user is at the mercy of the alternate tint transform for his/her simulated view of the spot colorant.

Instead of relying upon the alternate tint transform, using an option in Ghostscript, the user can specify an xCLR ICC profile for the spot colorant (or combination of spot colorants if it is a DeviceN color space) and have that ICC profile associated with and used when managing those spot colors. The xCLR ICC profiles must contain the colorantTableTag so that the colorant names can be compared to the names encountered in the document. The

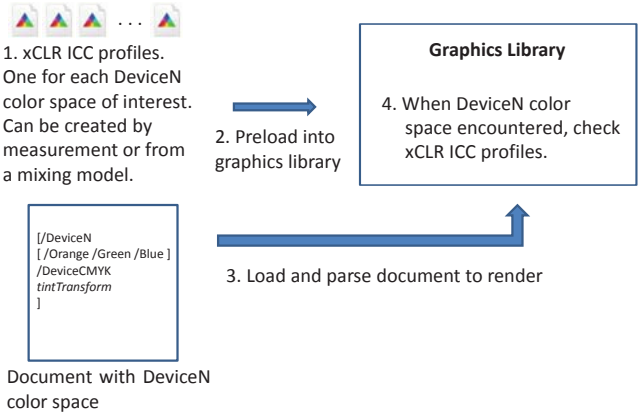

Figure 4. *xCLR ICC based handing of source DeviceN colors.*

colorant orders specified by the names in the document may be different than they exist in the xCLR ICC profile, necessitating the use of a permutation of the tint values prior to color management [11]. The xCLR ICC profile is provided to Ghostscript on the command line using an option like

```
-dDeviceNProfile=''OrangeViolet.icc''
```

See Figure 4 for an overview of the process.

Instead of the alternate tint transform or the xCLR approach, users can use a custom based method in Ghostscript for the handling of spot colorants. If the user has her own proprietary manner in which she wants to process DeviceN colors, she can load any data she needs using the command line option

```
-sNamedProfile=''MyNamedColorStructure.icc
    ''
```

She will need to implement the `gsicc_transform_named_color` method in Ghostscript. For this method, there is currently an example implementation that uses a look-up-table with colorant names and associated CIELAB values along with a mixing model to create a simulated color.

Ghostscript has devices included with it that will handle all spot colorants (up to 60 + CMYK per page). One device is the psdcmyk device, which creates Photoshop output image files (with a Photoshop specification max of 56 colorants). The other device is the tiffsep device, which creates a composite TIFF image of all the colorants as well as individual separation TIFF images for each colorant on the document page.

Device Profiles

As shown in Figure 2, there are several ICC profiles associated with the output device. This design allows for object-dependent color management, since Ghostscript knows the object type that it is processing. For example, different destination ICC profiles (as well as rendering intents) can be used for graphics, images and text. These destination ICC profiles are specified on the command line using options such as

```
-sGraphicICCProfile=''MyGraphicsProfile.
    icc''
```

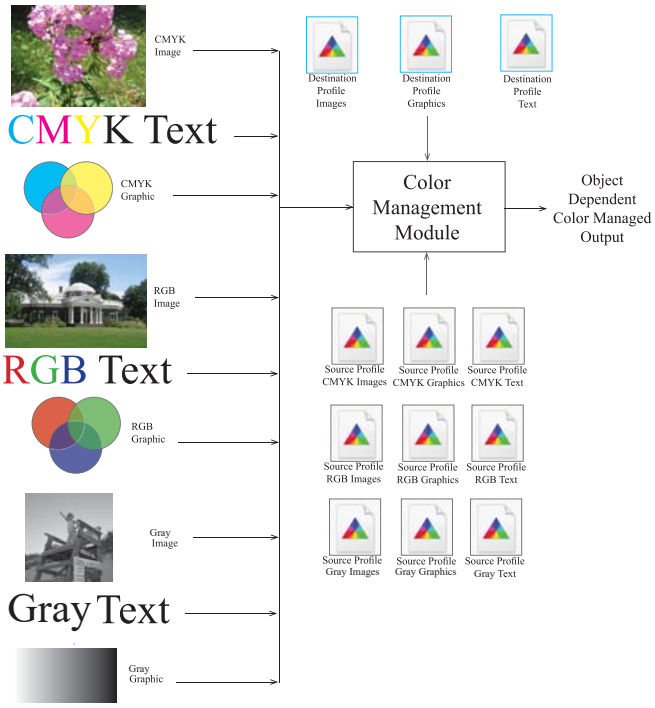

Figure 5. *Object-dependent color management. Depending upon the source type and color, the CMM is provided different source and destination ICC profiles.*

to specify the device destination profile for graphics objects. Similar options exist for text and images.

In addition to specifying the destination profile, Ghostscript offers the flexibility for the user to specify the source profile that she wants to use for graphics, images and text objects that exist in the document in the DeviceGray, DeviceRGB and DeviceCMYK color spaces. Since this requires the specifying of as many as nine additional ICC profiles, the profile information for any of these that are needed to be set are contained in a single file that is provided to Ghostscript through the command line option

```
-sSourceObjectICC=''MySourceInformation.
    txt''
```

Detail on the exact format of this file is contained in the Ghostscript documentation [12]. Figure 5 provides an overview of the various options that can be specified.

By default, for the tiffsep device, the CMYK colorants are color managed using the same work flow used for a standard CMYK device, while the spot colorants are rendered directly to separations without any color management. If desired, the user has the option to specify an output ICC profile for the tiffsep device that has more than four colorants. For example, if the output device used the colorants CMYK+Orange+Violet an ICC profile for these six colorants can be specified for the device on the command line using the options

```
-sOutputICCProfile=''MyDeviceNProfile.icc'
    '
-sICCOutputColors=''Cyan, Magenta, Yellow,
    Black, Orange, Violet''
```

The list of colorant names must be in the same order that they occur in the ICC profile. If a matching colorant name is found in the document (e.g. Orange), then that colorant will be associated with the related separation.

Transparency

As mentioned earlier, PDF 1.4 and above allow the use of transparency in the imaging model. Transparency blending in PDF is quite complex. It includes a variety of blending operations as well as the capability for the blending color space to be specified in the document. To add to the complexity, transparency groups can be embedded within groups each with their own blending color space along with additional properties of knockout and isolation. Ghostscript will honor the color space in its blending operations per the specification. Occasionally PDF files are encountered for which the page transparency group, which is to say the top most or first transparency group, does not specify a color space. In this case, the specification states that the transparency group should inherit the color space of the target device. This dependency of the blending color space on the target device can lead to rendering that looks quite different on an LCD screen compared to rendering on a printed page. The difference is caused by blending in RGB color space for the LCD display vs. blending in CMYK color space for the printer. To work around this issue, Ghostscript enables the user to specify a blending color space to use to ensure consistent rendering across devices. This is done with the command line option

```
-sBlendColorProfile=''MyBlendingColorSpace
    .icc''
```

With transparency blending occurring between different object types, performing object-dependent color managment becomes problematic. For example, if a text object is 50 percent blended with an image object, should those pixels be treated as image or text during color conversion? This decision is left to the designer of the output device in Ghostscript. To provide the flexibility for this color managment to be handled in the output device, Ghostscript can use an additional image plane during transpareny blending to maintain information about the type of objects that were used when a particular pixel was blended. In this way, when the drawing of the transparency buffer is completed, the device can determine based upon this additional image plane how it wants to perform color processing of the buffer.

The blending operations of transparency can take a signification amount of computational time. For example, when blending one transparency group with its parent (the backdrop), for each pixel in the group, Ghostscript has to compute

$$C_r = \left(1 - \frac{\alpha_s}{\alpha_r}\right) \times C_b + \frac{\alpha_s}{\alpha_r} \times \left[(1-\alpha_b) \times C_s + \alpha_b \times B(C_b, C_s)\right] \quad (1)$$

$$\alpha_r = 1 - \left[(1-\alpha_b) \times (1-\alpha s)\right] \quad (2)$$

where C_r is the resultant color, α_r is the resultant alpha, C_b is the backdrop color, α_b is the backdrop alpha, C_s is the source (or current group) color, α_s is the source alpha and $B(x,y)$ is a blending function for which PDF defines twelve different options[4].

[4]These equations in the PDF specification are based on the work of Porter and Duff. [13]

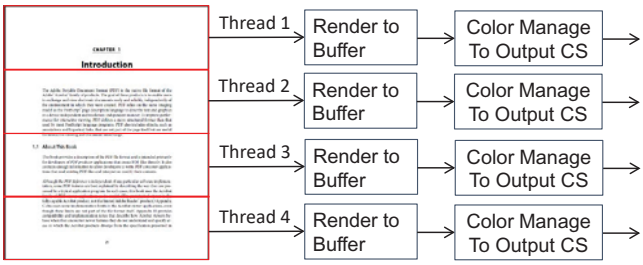

Figure 6. *Multi-threaded rendering.*

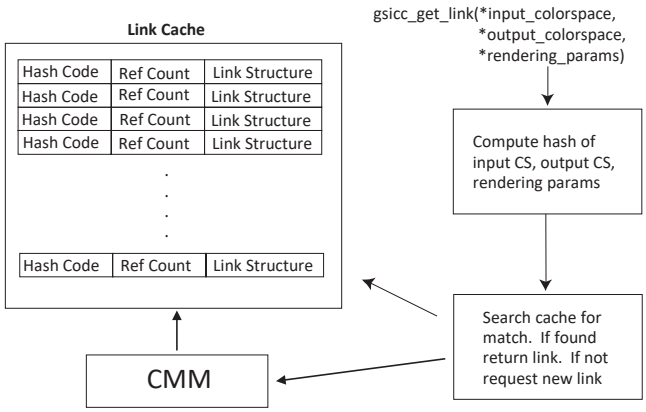

Figure 7. *Link cache*

Where the architecture allows for it, Ghostscript can make use of single instruction multiple data (SIMD) operations to provide acceleration on this process. Performance results for this will be discussed in the following Efficiency section.

Output Intent and Threaded Rendering

It was previously discussed that a PDF/X file can include an output intent ICC profile, which specifies the intended color space to which the document was to be rendered. Often, this color space is significantly different than the actual target device color space. For example, the output intent could be Fogra 39 CMYK, but the document is being displayed on an organic LED RGB display with a P3 gamut. In this case, the proper rendering process would be to render the document as a CMYK Fogra 39 image and then transform that data to the RGB color space defined by the P3 display device.

Implementing the above transformation in a design takes some thought. The amount of data that needs to be processed has increased significantly since the entire page is being color managed twice. A developer might be tempted to pack the color transforms into one operation eliminating the rendering to the intermediate Fogra 39 color space. Unfortunately, in general, that approach is not possible due to the fact that PDF includes a drawing state called overprint in which subsequent drawing of colorants might not erase colorants that have already been drawn in that location. This drawing state means that the previous colors in the Fogra 39 color space must be maintained through the entire drawing of the page to ensure the intended rendering is performed.

Most processors today have multiple cores; so to provide some efficiency in this rendering process, Ghostscript can make use of multiple threads as it renders the page. This process is shown in Figure 6, where individual threads are used to render different bands of the page to buffers in the output intent color space, which is then color managed by the thread to the target color space. Other operations such as halftoning, scaling or trapping can be performed by these threads. In low memory situations, the band size can be made quite small (down to a single row, for example), and as a thread completes its processing, it will begin processing the next band. In this way, the number of processing threads can be set to the number of cores in the CPU enabling significant speed-up in the processing.

Efficiency

Commercial users of Ghostscript have aggressive performance requirements since they are often trying to reach a particular page/minute goal. In addition, printer manufacturers often need to achieve certain certifications, such as Apple AirPrint, for which performance is a key factor. To reach performance goals, the first place to look for improvements is at portions of the code that are the most computationally expensive. Profiling of Ghostscript points to color transformations, transparency blending and creation of ICC color transformations as some of the more expensive oprations.

In the case of Little CMS, the creation of color transforms or links involves the creation of a relatively large multi-dimensional look-up table. Ideally these links should only be created once and then shared between the rendering threads. To enable link sharing among threads, Ghostscript uses a thread-safe version of Little CMS. [9] In addition, Ghostscript uses a cache to store opaque pointers to links supplied by the CMM. A hash of the source and destination ICC profiles as well as the rendering conditions and input/output data organization and bit depth is included in the hash. Figure 7 shows a graphical illustration of this caching process.

In addition to methods of multi-threaded rendering and caching, performance improvements can be achieved through the use of SIMD operations or (potentially) GPGPU resources if they are available. Ghostscript can use SSE4.2 SIMD intrinsics [14] to achieve parallelism in expensive operations such as transparency blending as well as tetrahedral interpolation of MLUT data used in color conversion [15]. SSE4.2 has 128-bit sized registers making it possible to operate on four 32-bit floats, eight 16-bit integers or sixteen bytes in parallel. As such, depending upon the operation that is being implemented, significant speed-ups are possible. The SSE tetrahedral interpolation is implemented as a plug-in to the thread-safe lcms2-MT fork of Little CMS [9, 10].

To demonstrate the speed-up achieved with the use of SSE operations for tetrahedral interpolation, Figure 8 displays the contents of a PDF page consisting of six images in sRGB color space, each of size 4608x3456 pixels with a PDF page size of 23.9 x 26.7 cm. The file was rendered to a CMYK color space defined by the eciCMYK ICC profile (FOGRA53)[16] with a bit depth of 32bits/pixel at 600dpi resolution. This output format amounts to approximately 142 MB data for this file. To avoid timing overhead due to file output, the actual output data was written to the

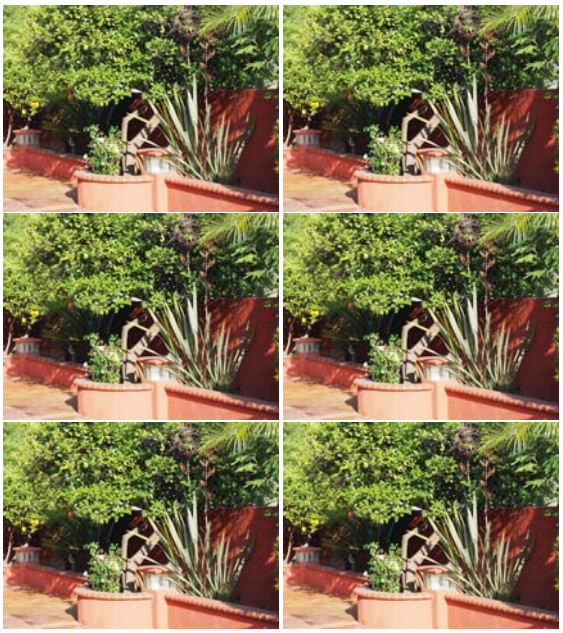

Figure 8. *PDF Images document with six images in sRGB color space.*

null device. The exact command line used was

```
./gs -sDEVICE=bitcmyk
-r600  -o /dev/null
-sOutputICCProfile=eciCMYK.icc
-dGrayValues=256 -f input_file.pdf
```

The code was run on single thread on an Intel i7-6700HQ at 2.6GHz. Profiling was performed using the Visual Studio 2017 profiler enabling investigation into the total amount of time spent on tetrahedral interpolations going from the sRGB color space to the FOGRA53 color space.

In addition to the above tetrahedral interpolation testing, SSE 4.2 implementations were made to the transparency compositing operations given in Equations 1-2. To test the performance improvements, a file from the Apple AirPrint Conformance Test Suite that contains transparency content was rendered to a continuous tone CMYK output device (32-bits/pixel) at 600dpi where again the actual output was sent to the null device to avoid timing overhead for file transfer time. The command line used was

```
./gs -sDEVICE=bitcmyk
-r600  -o /dev/null
-dGrayValues=256 -f Airprint.pdf
```

Figure 9 displays the speed-up achieved using SSE 4.2 for the tetrahedral interpolation and transparency blending compared to an efficient C-code implementation. This figure shows the percent speed-up for processing the entire file (in blue) as well as the speed-up within the function doing the operation as well (in orange). Note that the file speed-ups were similar for the two files at about 10 percent, but the function speed-ups were much different with the SSE 4.2 transparency blending operation being 73 percent faster than the standard C-code implementation. This difference in function vs. file speed-up is caused by differences in the time spent within those functions when rendering those files.

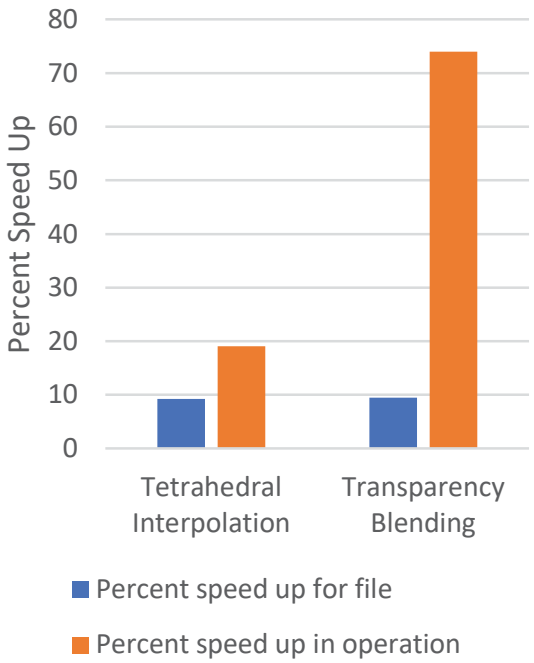

Figure 9. *Percent speed-up for using SSE 4.2 methods on tetrahedral interpolation and transparency blending.*

For the images file shown in Figure 8, the tetrahedral interpolation operation consumes almost 50 percent of the total page processing time, in which case every 1 second speed-up in that operation will result in a 0.5 second speed-up on the entire file. In the case of the Apple AirPrint file with transparency, only about 12 percent of the processing time is spent doing transparency blending. Achieving a 10 percent speed-up in the file requires a much greater speed-up in the transparency blending function.

References

[1] PostScript® Language Reference, Third Edition, Adobe Systems, Addison-Wesley, Reading Massachusetts (1999)

[2] PCL5 Printer Language Technical Reference Manual Part 1, Hewlett Packard (1992)

[3] IEC 61966-2-1, Part 2-1: Colour management Default RGB colour space sRGB, IEC (1999)

[4] ISO 32000-1:2008. Document management – Portable document format – Part 1: PDF 1.7 (2008)

[5] Interview with L. Peter Deutsch, http://web.archive.org/web/20041013082602/http://devlinux.org/deutsch-interview.html (retrieved 2019)

[6] ECMA 388 Open XML Paper Specification. Available at http://www.ecma-international.org/publications/standards/Ecma-388.htm (2009)

[7] https://www.argyllcms.com/ (2019)

[8] Source code at https://github.com/mm2/Little-CMS (2019)

[9] M. J. Vrhel, R. Watts and R. Johnston, LittleCMS-MT: A thread-safe open source color management library, CIC 26, Vancouver BC (2018)

[10] Source code at http://git.ghostscript.com/?p=thirdparty-LittleCMS2.git;a=shortlog;h=refs/heads/LittleCMS2-art (2019)

[11] M. J. Vrhel, System and method for improving color management of color spaces in electronic documents, US Patent US8488195B2 (2013)
[12] Source code at http://git.ghostscript.com/ghostpdl.git/. Documentation at https://www.ghostscript.com/Documentation.html (2019)
[13] T. Porter and T. Duff, Compositing digital images, Computer Graphics, vol. 18., no. 3, July (1984).
[14] https://software.intel.com/sites/landingpage/IntrinsicsGuide/ (2019)
[15] H. R. Kang, Color Technology for Electronic Imaging Devices, SPIE Press (1997)
[16] European Color Initiative, profile and details available at http://www.eci.org/en/colourstandards/workingcolorspaces August (2017)

Author Biography

Michael Vrhel was awarded his PhD from North Carolina State University in 1993; during his PhD, he was an Eastman Kodak Fellow. He has many years' experience working in digital imaging, including biomedical imaging and signal processing at NIH; color instrument and color software design at Color Savvy Systems Ltd, and positions at Conexant Systems, TAK Imaging and Artifex Software. A senior member of the IEEE, he has a number of current and pending patents and is the author of numerous papers in the areas of image and signal processing including a book, The Fundamanentals of Digital Imaging. His current interests include efficient computational color rendering methods as well as deep learning applications.