Accelerating Vector Graphics Rendering using the Graphics Hardware Pipeline

Vineet Batra[1], Mark J. Kilgard[2], Harish Kumar[1], and Tristan Lorach[2]

[1]Adobe Systems [2]NVIDIA

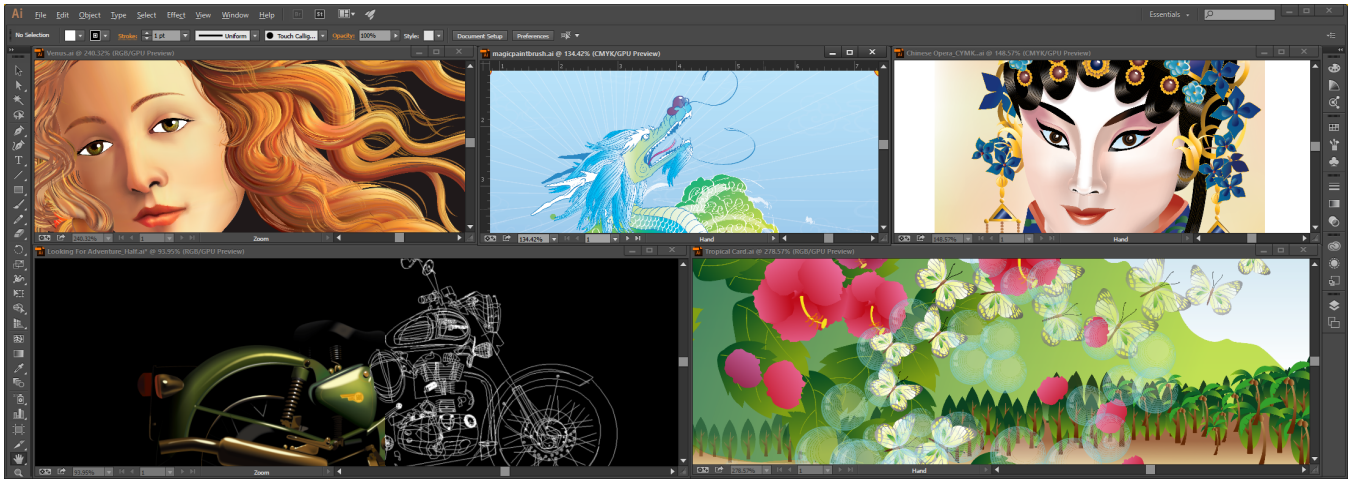


Figure 1: *Five complex RGB and CMYK documents GPU-rendered by Illustrator CC; all rendered with "GPU Preview" enabled.*

Abstract

We describe our successful initiative to accelerate Adobe Illustrator with the graphics hardware pipeline of modern GPUs. Relying on OpenGL 4.4 plus recent OpenGL extensions for advanced blend modes and first-class GPU-accelerated path rendering, we accelerate the Adobe Graphics Model (AGM) layer responsible for rendering sophisticated Illustrator scenes. Illustrator documents render in either an RGB or CMYK color mode. While GPUs are designed and optimized for RGB rendering, we orchestrate OpenGL rendering of vector content in the proper CMYK color space and accommodate the 5+ color components required. We support both non-isolated and isolated transparency groups, knockout, patterns, and arbitrary path clipping. We harness GPU tessellation to shade paths smoothly with gradient meshes. We do all this and render complex Illustrator scenes 2 to 6x faster than CPU rendering at Full HD resolutions; and 5 to 16x faster at Ultra HD resolutions.

CR Categories: I.3.4 [Computer Graphics]: Graphics Utilities—Graphics Editors;

Keywords: Illustrator, path rendering, vector graphics, OpenGL

1 Introduction

Designers and artists worldwide rely on Adobe Illustrator to design and edit resolution-independent 2D artwork and typographic content. Illustrator was Adobe's very first application when released over 27 years ago.

Prior to our work, no version utilized graphics hardware to accelerate Illustrator's rendering. All rendering was performed entirely by the CPU. This situation is in stark contrast to the now ubiquitous GPU-acceleration of 3D graphics rendering in Computer-Aided Design, Animation, and Modeling applications. So while other graphical content creation applications readily benefit from the past 15 years of improvements in GPU functionality and performance, Illustrator could neither benefit from nor scale with the tremendous strides in GPU functionality and performance. Our work remedies this situation as Figure 1 shows.

The starting point for our work is OpenGL 4.4 [Khronos Group 2014] and the GPU-accelerated "stencil, then cover" path rendering functionality described in [Kilgard and Bolz 2012]. While the `NV_path_rendering` OpenGL extension [Kilgard 2012] provides very fast and resolution-independent rendering of first-class path objects just as we need, Illustrator's rendering model requires much more than merely rendering paths. We had to develop our own strategies to handle features of Illustrator that, while dependent on path rendering, require considerably more sophisticated orchestration of the GPU. We focus on the GPU-based techniques we developed and productized to accomplish full GPU-acceleration of Illustrator.

Adobe originally developed Illustrator as a vector graphics editor for PostScript [Adobe Systems 1985] which implements the imaging model described by [Warnock and Wyatt 1982]. Illustrator today depends on the Portable Document Format (PDF) standard for its underlying rendering capabilities as described by the ISO 32000 standard [Adobe Systems 2008]. The evolution of PostScript to PDF introduced a number of sophisticated graphics capabilities for dealing with printer color spaces, compositing, and photorealistic artistic shading. These capabilities developed in parallel with but

ACM Reference Format
Batra, V., Kilgard, M., Kumar, H., Lorach, T. 2015. Accelerating Vector Graphics Rendering using the Graphics Hardware Pipeline. ACM Trans. Graph. 34, 4, Article 146 (August 2015), 15 pages.
DOI = 10.1145/2766968 http://doi.acm.org/10.1145/2766968.

SIGGRAPH '15 Technical Paper, August 09 – 13, 2015, Los Angeles, CA.

DOI: http://doi.acm.org/10.1145/2766968

isolated from contemporaneous hardware-oriented improvements in interactive 3D graphics. PDF and Illustrator incorporated features and solutions relevant to the print industry such as subtractive color spaces, typography, and rasterization of 2D spline-based content targeting print engines with extremely high pixel density. However GPU hardware designers essentially ignored such concerns—instead focusing on interactive 3D graphics.

For example, GPUs specialize at rendering RGB colors—typically with alpha, so RGBA—for display on emissive RGB monitors. Indeed the framebuffer, texture, and memory subsystems of GPUs are tailored specifically for handling 4-component RGBA colors. However Illustrator users target printed color output and expect accurate color reproduction so the default document color mode in Illustrator is CMYK, a color space intended for printed content. Illustrator also does support RGB documents but CMYK matters more to professional users.

Rather than just 4 color components as GPUs are designed to handle well, CMYK involves at least 5 components (the process ink colors *cyan*, *magenta*, *yellow*, and *black* + alpha) and then additional *spot* colors assigned to specific custom inks. Simply converting CMYK colors to RGB and rendering in an RGB color space is not the same as rendering into a proper CMYK color space as Figure 5 shows. Among the differences, CMYK is a subtractive color space while RGB is an additive color space so color arithmetic such as blending must account for this difference. When a GPU's data paths are so specialized for processing RGBA values with exactly 4 components, handling 5 or more components in a manner consistent with PDF requirements and at reasonable efficiency requires novel treatment.

1.1 Motivation

Our work to GPU-accelerate Illustrator has the obvious goal of improving the rendering speed and interactivity to improve user productivity. Illustrator documents can become extremely complex and so poor rendering performance can frustrate a designer's creativity.

We also considered display technology trends, particularly increasing screen resolutions and pixel densities. So-called *4K* resolutions such as Ultra HD at 3840x2160 are particularly interesting to us. CPU performance scaling for rendering in Illustrator has clearly not been adequate to keep up with the increasing number of pixels needing to be rendered. Recent 5K Ultra HD (5120x2880) displays make this more pressing.

While our focus has been fully accelerating *existing* features of the PDF rendering model, we anticipate our transition of Illustrator's rendering to the GPU allows us to accelerate graphics operations, known as *effects* in Illustrator, that are handled "above" the PDF rendering model currently. Gaussian blurs and image warps are obvious examples of effects the GPU could significantly accelerate.

1.2 Contributions and Outline

Our key contributions are:

- Novel repurposing of the GPU's multiple RGBA render buffers for *NChannel* color space rendering of CMYK process colors and additional spot colors with full blending support.
- GPU algorithms to composite both isolated and non-isolated transparency groups properly.
- Tessellating gradient meshes to shade paths via GPU hardware tessellation.
- Harnessing "stencil, then cover" path rendering to support arbitrary path clipping, pattern shading, and knockout groups.
- Rendering complex vector scenes with the GPU many times faster than comparable multi-core CPU rendering.

These contributions are broadly applicable beyond Illustrator and benefit any GPU-accelerated software system that utilizes PDF, SVG, or similar printing, web, or vector graphics standards. We anticipate web browsers, other 2D digital content creation applications, document previewers, and even printers will use our contributions to accelerate existing standards for resolution-independent 2D vector graphics.

Furthermore we expect GPU-acceleration of Illustrator to motivate graphics hardware architects to focus on specific optimizations for this important rendering model. Likewise research in GPU-based techniques for vector graphics becomes substantially easier to productize.

Section 2 outlines relevant prior work. Section 3 provides useful background for Illustrator's software architecture relevant to rendering. Section 4 discusses GPU support for blend modes, a prerequisite for our contributions. Sections 5 to 7 describe the primary techniques we developed to GPU-accelerate Illustrator fully. Section 8 compares and contrasts our GPU-acceleration to Illustrator's existing CPU renderer. Section 9 presents our performance improvements. Section 10 concludes with a call for standardization and discussion of future plans.

1.3 Status

The June 2014 release of Adobe Illustrator CC first incorporated our GPU-acceleration efforts. Adobe reports over 3.4 million Creative Cloud subscribers as of December 2014. A substantial fraction of these subscribers use Illustrator. The complete contributions we discuss are refinements beyond the initial release. In particular, the 2015 version introduces CMYK (Section 5) and tessellation-based gradient meshes (Section 7.2).

2 Related Work

The PDF specification [Adobe Systems 2008] describes in detail the rendering model Illustrator implements. Kilgard and Bolz [2012] provides a good review of GPU-acceleration approaches for path rendering and is the crucial underlying functionality upon which our work relies.

Our interest in rendering complex vector graphics scenes is similar to the random-access vector texture approach of [Nehab and Hoppe 2008] and its refinement by [Leben 2010]. However Illustrator's raison d'être is arbitrary editing of vector graphics scenes in a very general setting (permitting CMYK color spaces, blend modes, transparency groups, etc.) so the need to re-encode the scene's vector texture whenever the scene is manipulated is unappealing.

Vector textures are also unappealing because we must support all of Illustrator's rich feature set but a vector texture scheme effectively centralizes support for the scene's entire rendering requirements. This means the fragment shader used to decode the vector texture approach must be prepared to incorporate every possible vector graphics feature in the scene. The advantage of random-access decoding when rendering from a vector texture, while interesting, is not particularly relevant to Illustrator.

Recent work by [Ganacim et al. 2014] has significantly advanced the vector texture approach by using CUDA to implement

massively-parallel vector graphics on the GPU rather than the conventional graphics pipeline. Section 9.3 compares their reported performance to ours so we defer more discussion until then.

2.1 Alternative Vector Graphics Editors

While there are many alternative path-based vector graphics editors, such as Inkscape [Kirsanov 2009] and CorelDRAW [Bouton 2012], no existing vector graphics editor significantly exploits graphics hardware acceleration to the best of our knowledge.

While not a conventional path-based editor, the Mischief application [61 Solutions Inc. 2013] for scalable drawing, sketching, and painting is GPU-accelerated. Mischief takes an Adaptive Distance Field (ADF) approach [Frisken and Perry 2006]. ADFs represent shape implicitly rather than the explicit path-based approach used by Illustrator and PDF to represent shape. A hybridization of Mischief's implicit approach and our explicit approach may be possible given the commonality of GPU-acceleration.

2.2 Smooth Shading

The PDF standard provides several shading operators for continuous color. Linear and radial gradients are very familiar to 2D digital artists and are a standard way for integrating continuous color into vector graphics content. For example, the SVG standard [SVG Working Group 2011a] supports both linear and radial gradients. Our GPU-acceleration efforts accelerate both linear and radial gradients using straightforward cover shaders as described in [Kilgard and Bolz 2012].

Illustrator's *gradient mesh* tool provides a way to shade paths using a mesh of Coons patches to assign and edit continuous color within a path [Adobe Systems 2008; Coons 1967]. Skilled artists use gradient meshes to create editable photorealistic resolution-independent artwork, often sampling colors for the mesh control points from photographs.

A different approach to assigning smooth color to vector graphics artwork called *diffusion curves* [Orzan et al. 2013; Sun et al. 2012; Ilbery et al. 2013] relies on partitioning 2D space with vector-based primitives and letting color naturally "diffuse" through the scene to a steady-state that respects the partitions. The color diffusion process is computationally intensive and well-suited for the massively parallel nature of the GPU. Our immediate interest is accelerating Illustrator's existing gradient mesh functionality though we anticipate our transition to the GPU facilitates the adoption of more intuitive gradient methods otherwise too expensive for the CPU.

3 Illustrator Rendering Architecture

The software architecture of Illustrator comprises several interoperable modules shown in Figure 2 that have evolved over decades.

Our paper focuses on the Rendering Subsystem for rasterizing vector content at arbitrary resolutions so any further detail about other modules is beyond our scope. Despite the complexity of Illustrator overall, our GPU-acceleration effort required modifying only the Rendering Subsystem and its subcomponents.

An Illustrator *artwork* (analogous to a 3D scene graph but for vector graphics) comprises multiple art objects (Bézier-based path, text, image, mesh, etc.), each attributed with a collection of *appearances* (fill and stroke with solid color, pattern, gradient, etc.) and *effects* on these appearances (blur, feather, shadow, etc.). When stacked in Z-order, these art objects interact with each other to produce rich content. This interaction (or composition) is based in the PDF specification. Illustrator also provides several high level primitives

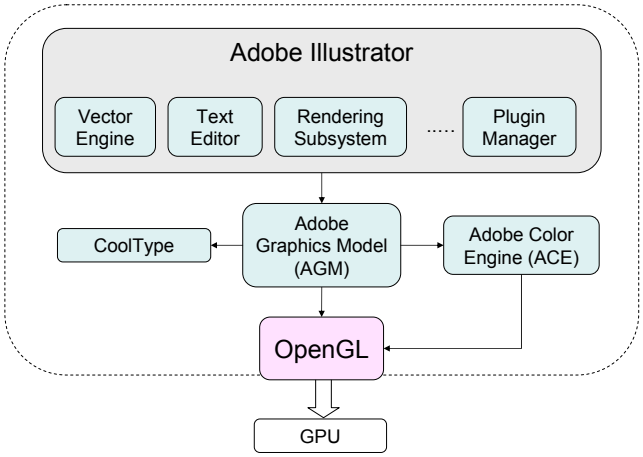


Figure 2: *Organization of major rendering-related modules within Illustrator.*

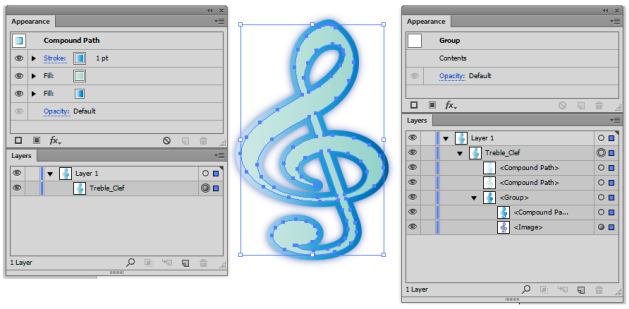


Figure 3: *How high-level artwork with effects applied (left) for a treble clef scene (center) reduces to a tree of PDF constructs (right).*

that are not supported directly in the PDF specification but are reducible to PDF constructs. An example is an Illustrator path with both fill and stroke applied, which is reduced to two paths—a path with stroke placed on top of a path with fill. Figure 3 shows another example where an art object with *OffsetPath* and *OuterGlow* effects is reduced to a group comprising three compound paths and an image.

In service to the Rendering Subsystem, the *Adobe Graphics Model* (AGM) layer provides a pipeline for rendering PDF compliant artwork using the CPU and now—with our work—the GPU. For managing the color output by a target device, AGM uses Adobe Color Engine (ACE) which implements color conversions between different profiles on the GPU using GLSL shaders. ACE profile management handles many different target devices (monitor, mobile devices, printer, etc.) to ensure accurate color reproduction. Font support is implemented via CoolType that provides programming interfaces for retrieving font and glyph information (including glyph outlines) from system fonts. Figure 2 shows the relationship of AGM, ACE, CoolType and OpenGL. Our acceleration effort focused essentially entirely on accelerating AGM so we rely on higher level graphics primitives to be reduced to the PDF rendering model.

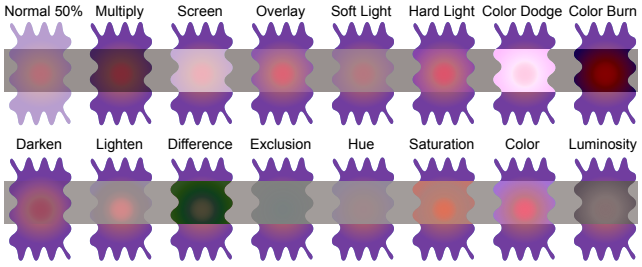


Figure 4: *Example of all sixteen blend modes showing the interaction between a blob with an opacity gradient interacting with an overlapping rectangle with 90% opacity.*

4 Blend Modes

Illustrator version 9 introduced a palette of sixteen blend modes. These blend modes were subsequently incorporated into the PDF standard's transparency model [Adobe Systems 2000].

4.1 GPU Blend Mode Support

Of these modes only two (*Normal* and *Screen*) are sufficiently simple that they can be implemented with conventional OpenGL fixed-function blending. Several of the advanced PDF blend modes require intermediate numeric range exceeding the clamped [0,1] range used in fixed-function GPU blending. Some of the modes require simple "if/then/else" conditions, division, and square root operations. To a hardware designer or anyone else first encountering these modes, they may seem *ad hoc* and arbitrary, but *HardLight*, *ColorDodge*, and the rest are firmly established in the vocabulary and training of digital artists [Valentine 2012]. Figure 4 demonstrates their various effects.

In anticipation of Adobe's requirements for blend mode support, NVIDIA developed the OpenGL `NV_blend_equation_advanced` extension [Brown 2013] for advanced blending. It provides all the blend modes needed by Illustrator, PDF, and SVG [SVG Working Group 2011b]. The first-class *coherent* form of the extension is implemented via hardware support in Tegra K1 and Maxwell GPUs. For older GPUs without hardware support for the coherent form, the advanced blending functionality is exposed through an *incoherent* form of the extension. For the incoherent form, drivers are expected to implement the advanced blending as a driver-generated epilogue to the application's fragment shader. The incoherent form requires the application to use blend barrier commands to ensure proper blend ordering to guard against read-modify-write hazards. A special exception is made for `NV_path_rendering` operations, since "stencil, then cover" path rendering naturally lends itself to proper blending.

Khronos subsequently standardized OpenGL advanced blending functionality with the `KHR_blend_equation_advanced` [Brown 2014] extension, also with coherent and incoherent forms. AGM uses either OpenGL extension as available.

4.2 Premultiplied Alpha

One point of interest for GPU blending is how colors are represented. Colors stored in GPU framebuffers and textures must to be stored in pre-multiplied alpha form [Smith 1995] for correct blending (including blend modes) and texture filtering. In contrast, the CPU-based AGM renderer stores color values with non-premultiplied alpha consistent with the PDF specification.

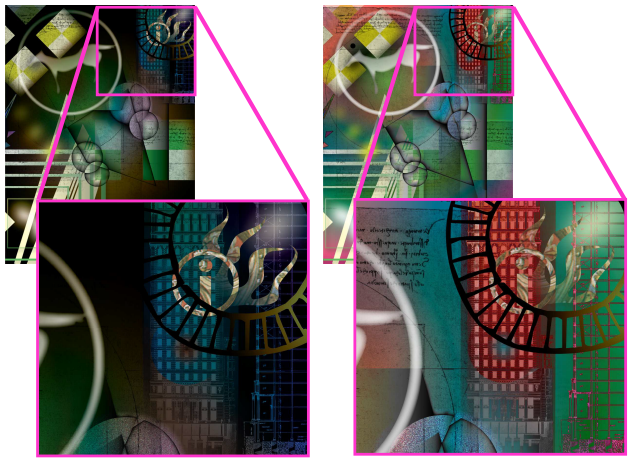


Figure 5: *An artistic CMYK Illustrator document (left, correct) properly rendered in its intended CMYK color space; naïve RGB rendering (right-side, wrong) of same scene by converting all inputs to RGB . Magnified portion show obvious gross color shifts.*

5 CMYK Support

Illustrator artwork is typically authored in the CMYK color space—optionally with spot color components—to best facilitate high-quality color printing. Illustrator uses AGM to render CMYK documents to a true CMYK framebuffer. This means the color components in the rendered framebuffer correspond to actual CMYK process colors plus any spot colors so blending and other color math operates on and maintains the components independently.

What the artists sees "on screen" when editing a CMYK document is a color conversion from CMYK plus any spot colors to RGB. Importantly this conversion happens on the *final* CMYK rendering result; the conversion may even emulate the specific color profile of a particular print device using ACE. Converting CMYK rendering results to a print device's CMYK color profile results in better color fidelity and gives an artist better access to the printer's full color gamut including spot color inks. Importantly spot color components stay segregated from process colors.

While it is possible to force the conversion of all CMYK color inputs to RGB and render in the GPU's conventional RGB mode, Figure 5 shows the inadequacy of rendering CMYK content with this approach; notice the obvious color shifts.

Now we explain our approach to orchestrate CMYK rendering with multiple RGBA color buffers. The technique we present works not just for Illustrator but any GPU application that requires CMYK rendering semantics.

There are two problems we must address:

1. CMYK is a subtractive color space so conventional GPU blending modes do not blend appropriately for CMYK.
2. At a minimum, CMYK rendering needs 5 framebuffer components—plus additional components for any spot colors.

5.1 CMYK Blend Modes and Color Math

Adobe's technical note introducing transparency to PDF [Adobe Systems 2000] explains how blending in a subtractive color space requires special handling:

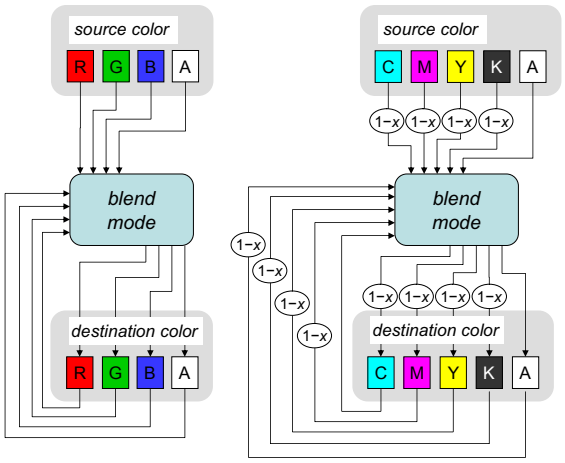


Figure 6: *RGBA blending works normally (left); CMYK's subtractive color space must complement colors components on input and output of blending (right).*

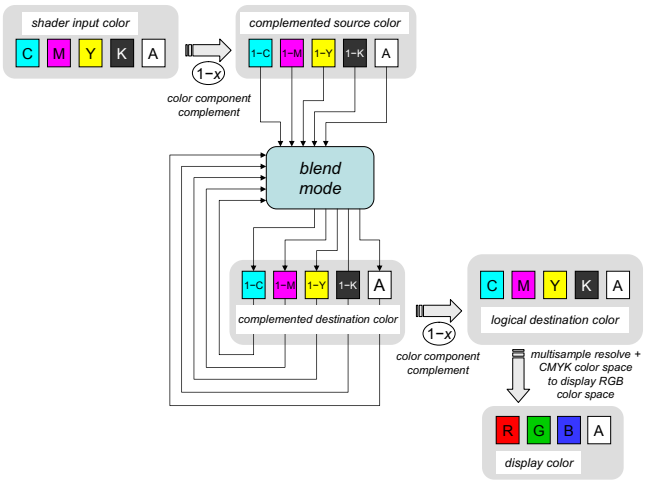


Figure 7: *Conventional GPU blending works for CMYK when color components (but not alpha) are stored as complemented values.*

> When performing blending operations in subtractive color spaces, we assume that the color component values are complemented before the blend mode function is applied and that the results of the function are then complemented before being used. By complemented we mean that a color component value c is replaced with $1 - c$.

An example helps appreciate why: Consider a black ink at 90% of its full application (so a dark black). Now consider how to get 40% of the apparent brightness of that ink. Intuitively 40% brightness should be an even darker black. Naïvely multiplying 0.9 by 0.4, as appropriate for additive color components, is 36% but less black ink is clearly incorrect. Adjusting the blending math for the subtractive nature of black ink, $1 - ((1 - 0.9) \times (1 - 0.4)) = 94\%$ results in more black ink and the darker black we intuitively expect.

Figure 6 illustrates how RGB and CMYK color values must be treated for a blend mode to operate correctly. Conventional fixed-function GPU blending lacks the ability to complement inputs and outputs to fixed-function blending.

The blend mode extensions described in Section 4 assume an additive color space. Naïvely adapting these blend modes to operate correctly for a subtractive color space such as CMYK would mean adding a complement to each input color component and, likewise, a complement to each output color component. We avoid the naïve approach because

1. Additive blend modes avoid input & output color component complements so involve fewer operation and are therefore more efficient.

   This is particularly important for legacy hardware that predates hardware blend mode support. For such hardware, performing additional input & output complements for CMYK rendering would force expensive shader emulation even for the default *Normal* blend mode that otherwise can be performed with fast standard hardware blending.

2. Not just blending math requires this adjustment—*all* color math such as in a shader or texture filtering must follow the rule.

Our solution stores CMYK color components always in complemented form on the GPU. Figure 7 illustrates this approach. Incoming CMYK colors, no matter what the source, must be complemented. Alpha values are *not* complemented and simply stored normally as alpha is always additive.

The implications of this are far reaching and demand rigorous consistency. Any CMYK or RGB color inputs must be converted to complemented CMYK. For example, when color channels are logically "cleared to zero," that really means clearing to one (but alpha components still clear to zero). By storing complemented CMYK colors in the framebuffer *and* in textures, existing hardware texture accesses to such resources including texture filtering operate properly. Likewise programmable shaders are simpler and faster by skipping the requirement to complement input and output colors by performing color math on complemented subtractive color values.

Importantly our solution means the blend modes provided by the blend equation advanced extensions operate in subtractive CMYK color mode with exactly the same blending math as additive RGB color mode.

Only when rendered results are ready to be displayed or read back to system memory should the complemented color components be reversed to uncomplemented CMYK. Often color profiles are applied when displaying or reading back rendering results so the necessary complement can be folded into a more complex conversion thereby avoiding an explicit complement. The bottom right two steps in Figure 7 show this.

5.2 Representing an NChannel Framebuffer on a GPU

When we speak of Illustrator's CMYK color mode, we mean a subtractive color spaces supporting the 4 standard print process colors (CMYK), alpha, and possibly some fixed number of spot colors. PDF supports a variety of color spaces with different properties and degrees of sophistication. The most general of these is the *NChannel* color space so we use the term NChannel to refer to a general multi-component color space.

5.2.1 Multiple Color Buffers

GPUs lack native support for color buffer configurations with more than 4 components. However modern GPUs also support multiple (typically eight) color buffers to which a single fragment shader can output a distinct RGBA value to each color buffer. Each color buffer is independently blended with its respective RGBA output value.

We orchestrate multiple color buffers to "construct" an NChannel

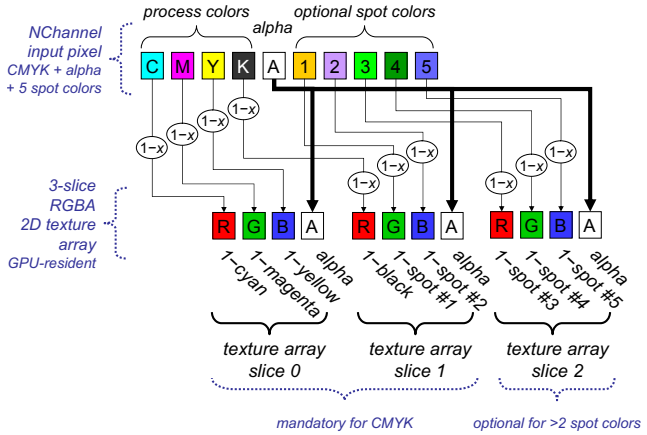


Figure 8: *How NChannel color components in a CMYK color space with spot colors are converted to a GPU-resident version.*

framebuffer with CMYKA components plus any additional spot color components. Figure 8 illustrates how we take a CMYKA input color with five spot color components and convert this 10-component vector into a GPU-amenable representation by spanning three RGBA color buffers. We can span additional color buffers to support more spot colors but we always require at least two RGBA color buffers for CMYKA.

Figure 8 shows we *replicate* the alpha (A) component in the alpha of each RGBA color buffer. This is done because blend mode equations for color values require access to the alpha component. As GPU blending performs each color buffer blending operation separately, replication of alpha ensures every three color components of the NChannel color spanning multiple RGBA buffers has an accessible alpha value. We deliberately replicate alpha this way and ensure all alpha blending is identically configured so the alpha values of each RGBA color buffer for any specific color sample maintain the *same* alpha value. This invariant must be maintained for correct operation.

5.2.2 Memory Requirements

Our approach wastes some storage. Every additional RGBA buffer adds an additional replicated alpha component. We also waste storage if the actual CMYK color mode configuration has fewer spot colors than our multiple RGBA color buffers provide. For example, CMYKA with zero spot colors means the G and B components of the second color buffer are wasted—assuming R is used to store K. The unfortunate implication is that a document in CMYK color mode without spot colors requires double the framebuffer memory as an RGBA document.

This waste is substantial when coupled with Illustrator's reliance on 8x multisampling for antialiasing. When also accounting for the stencil buffer requirements, representing CMYK at 8x on today's GPUs requires 96 bytes of storage per pixel.[1] Each gigabyte of memory roughly corresponds to representing 5.5 million CMYKA pixels this way. The waste is ameliorated by the enormous memory capacity and bandwidth of modern GPUs. Graphics boards with 12 gigabytes of memory are available today and capacities are sure to increase.

We anticipate GPU hardware innovations will provide less wasteful memory organizations in future GPUs. In the interim when the memory burden is just too taxing, settling for 4x antialiasing quality easily halves the memory requirements at the cost of diminished rendering quality.

[1] 96 bytes = 8× (4 bytes for depth-stencil + 8 bytes for CMYKA)

5.2.3 Fragment Shader and Blending Implementation Details

Once we have orchestrated multiple color buffers to store NChannel color values, our fragment shaders must adapt to outputting color values appropriately. This means making sure alpha is replicated in each output color buffer alpha component and each process and spot color is routed to its appropriate component in the correct color buffer.

GPU color blending should be configured the same across all the framebuffer components.

One irksome detail: The blend equation advanced extensions are restricted to operate only where outputting to the first color buffer and requiring all other color buffers disabled (otherwise an OpenGL error is generated; this reflects a hardware limitation). We workaround this limitation with multiple "cover" steps. We "stencil" the path once and then perform the path rendering "cover" operation repeatedly, once for each color buffer (binding each color buffer in turn as the first and only color buffer) and reset the stencil values only on the last color buffer. Our fragment shader must be aware in this case which logical color buffer it is outputting in each repeated "cover" step. While expensive, we only require this workaround for blend modes (typically the less common ones) that do not correspond to fixed-function GPU blending—so importantly not the *Normal* mode.

5.2.4 Reading an NChannel Framebuffer from a Shader

Illustrator occasionally needs to read a framebuffer from a shader. (see Section 6.1.2 for an example). To enable efficient texture lookup, we organize the multiple color buffers used to represent an NChannel buffer as layers of a multisample 2D *texture array* (`sampler2DMSArray` in GLSL shaders). A 2D texture array is a single OpenGL texture object with a fixed format, width, height, and number of layers. When multisampled, the number of samples is also fixed. The `texelFetch` command in a shader fetches an RGBA floating-point vector sample from a given 2D location, layer index, and sample index (effectively a 4-dimensional array access). Multiple `texelFetch` fetches, one to each layer, are necessary from the fragment shader to read all the components of a pixel in an NChannel framebuffer.

The `ARB_texture_multisample` extension [Bolz 2009] introduced support for multisampled 2D texture arrays with OpenGL 3.2 mandating the functionality. Among the advantages of a multisampled 2D texture array is all the layers belong to a single texture memory allocation making the layers faster to bind as a unit and managed as a single large memory allocation. The OpenGL command `glFramebufferTextureLayer` allows each layer of the texture array to be attached to different color buffer of single framebuffer object.

6 Transparency Groups

In Section 4's discussion of blend modes, we assumed objects are simply blended in object stacking order with blend modes directing the blending. The PDF 1.4 transparency model becomes more intricate when graphics objects are grouped. Transparency groups, or simply *groups*, allow a sequence of consecutive objects to be collected together and composited to produce a single color and opacity at each color sample. Groups facilitate independent subscenes to be composited together. Artists also use groups to combine objects for artistic effects such as darkening or lightening re-

gions. Groups can be nested within other groups to form a tree of groups. When a group is reduced to a single color and opacity, the group itself has a blend mode and a per-group opacity used to composite the group with its backdrop. Transparency groups are a distinct concept from other mechanisms to group objects such as groups formed to manage hierarchical transforms or inherited properties.

6.1 Isolated versus Non-isolated Groups

Groups can be either *non-isolated* (Illustrator's default for a new group) or *isolated*.[2] This distinction is which backdrop is used when compositing objects in the group. With a non-isolated group, the backdrop is "inherited" from whatever has already been rendered prior in the object stacking order. This allows a group to interact with the prior objects "beneath" the group. With an isolated group, the backdrop is fully transparent so it has neither color, shape, nor opacity. This is sometimes called rendering "on glass" because there is really nothing for the group to interact with when the group itself is rendered. The group's rendering is—as the name implies—isolated.

Both modes are useful in their proper context. Non-isolated groups make sense when rendering a group expects to interact with the artwork beneath it. For example, a non-isolated group makes sense when an artist wants to use a blend mode such as *ColorDodge* or *ColorBurn* where painting with black or white respectively preserves the backdrop color. Compositing a source object using these blend modes with an isolated group would not make much sense because the initial backdrop is fully transparent so there is no color to preserve.

Isolated groups are more appropriate when the group is considered a fully resolved piece of artwork you simply want to composite into the scene. For example, a piece of vector clip art consisting of objects rendered with the *Normal* blend mode and that otherwise has no blending relationship with what's been rendered so far.

6.1.1 Framebuffer Management for Groups

Of the two types of groups, non-isolated is the more expensive to implement. Both types of groups conceptually create a transient framebuffer necessary to resolve the color, shape, and opacity of the group. We call this a *framebuffer instance* and we implement the group's framebuffer instance with an OpenGL framebuffer object (FBO) distinct from the current framebuffer instance. Allocating and discarding transient FBOs during rendering is inefficient so we manage a set of framebuffer resources sufficient to handle the scene's maximum non-trivial group nesting. Each framebuffer instance needs independent color buffer storage—and CMYK needs multiple color buffers as Section 5.2 discusses. However all the framebuffer instances can share a single stencil buffer. This stencil sharing is useful for maintaining the clip path nesting and conserving GPU memory usage. Because each FBO used to manage transient layers is preallocated and may be used to render a group positioned arbitrarily within the scene's viewport, each FBO is maintained at the same dimensions as the base framebuffer instance (typically sized to match the maximum window-space view size) and expects to share a single stencil buffer.

We speak of *non-trivial* groups because in common cases where every blend mode within a group is *Normal* and the group opacity is fully opaque (and other uncommon group features such as knockout are inactive), rendering a group to its own framebuffer instance is functionally identical to simply rendering the group's objects in sequence into the current framebuffer instance. Recognizing trivial groups and not instantiating a framebuffer instance for them is an important performance optimization.

But in cases when a non-trivial group is present, we carefully orchestrate rendering to a framebuffer instance and when the group is resolved, compositing the resolved framebuffer layer for the group back into the previously current framebuffer layer. Because groups can be nested to form a tree, this process is conceptually recursive but practically limited by the scene's maximum non-trivial group nesting.

From this point on, our discussion deals with non-trivial groups.

[2]The SVG Compositing specification [SVG Working Group 2011b] has the same concept but calls the property *enable-background* where the value *accumulate* matches non-isolated and *new* matches isolated.

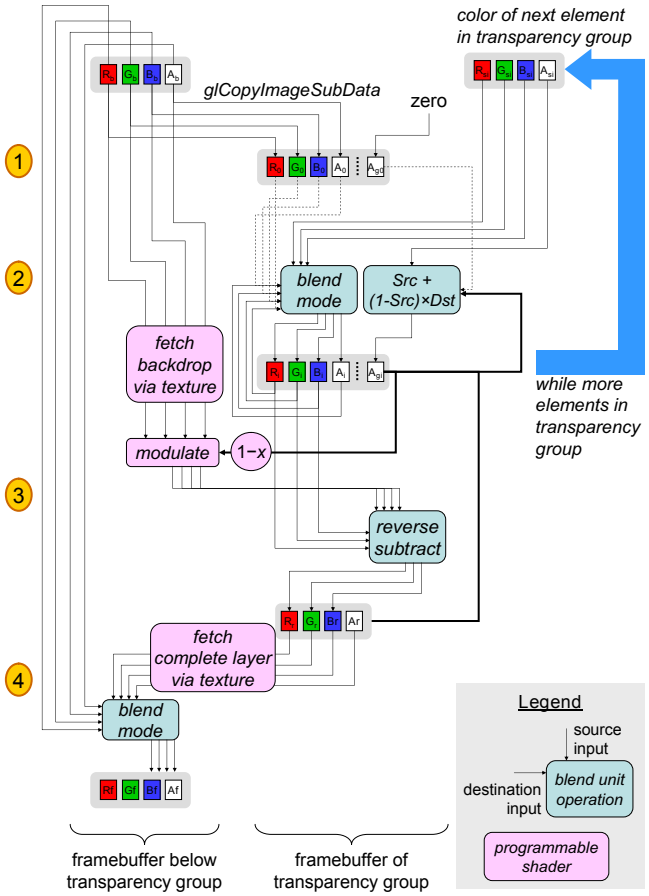


Figure 9: *Four steps in rendering a non-isolated group with the GPU.*

6.1.2 Implementing Non-isolated Groups

Figure 9 illustrates the steps to process and resolve a non-isolated group assuming an RGB color space. Numbered circles down the figure's left side indicate the steps 1, 2, 3, and 4 to be discussed in turn.

Step 1: Establishing a non-isolated group requires copying the backdrop from the current framebuffer instance to the group's transient FBO's color buffer(s). OpenGL's `glCopyImageSubData` command [Gold and Sellers 2012] copies a rectangular region of texel color values from one color buffer's underlying texture object to another. When the color buffers are multisampled, the command copies each pixel's individual color samples. In the worst

case, we may have to copy the entire color buffer contents but often we can bound the window-space bounding box for the objects within the group (including any nested groups). CMYK has multiple color buffers configured as layers of a texture array but a single `glCopyImageSubData` command can copy all the texture array layers.

An additional single-component (red) color buffer is also required for the FBO to maintain the non-isolated group's *group alpha* and labeled A_{g0} and A_{gi} in the Figure 9. A scissored `glClear` command must clear the *group alpha* color buffer to zero The motivation for *group alpha* will be more clear in Step 3.

Step 2: Each element of the group must be rendered in object stacking order. Any object that is a non-trivial group requires that nested group to be rendered and resolved. In such cases, the resolved color and opacity of the nested group is composited using the group element's blend mode and group opacity into this framebuffer instance. Elements of trivial groups can simply be rendered in sequence.

During step 2, in addition to compositing group elements into the RGB color buffer (or buffers for CMYK), the *group opacity* buffer is configured for PDF's *Normal* blending (implemented in OpenGL with the `GL_ONE,GL_ONE_MINUS_SRC_ALPHA` blend function). The fragment shader is responsible to output the alpha of each group element to this color buffer. The *group alpha* buffer is used to keep a distinct running accumulation of alpha but starting from zero from the alpha component(s) in the other (4-component) color buffers.

Step 3: Before the resolved color in a non-isolated group can be composited back to the prior framebuffer instance from before processing the group, we must "subtract out" the backdrop color and alpha used to initialize the group's framebuffer instance by `glCopyImageSubData`. Otherwise when the resolved color of the group is composited back into the prior framebuffer instance (Step 4), the prior framebuffer instance's color and alpha would be accounted for twice.

The PDF specification computes the resolved result of a non-isolated transparency group with the equations:

$$C = C_n + (C_n - C_0) \times \left(\frac{\alpha_0}{\alpha_{gn}} - \alpha_0 \right) \quad (1)$$

$$\alpha = \alpha_{gn} \quad (2)$$

where C is the resolved group color, C_n is the final color in the framebuffer instance after all n group elements are composited, C_0 and α_0 are the backdrop color and alpha respectively from the prior framebuffer instance, and α_{gn} is the final *group alpha* from the single-channel color buffer after all n group elements are composited.

This equation is written with non-premultiplied alpha but the GPU represents colors in pre-multiplied alpha form. Combining the Equations 1 and 2 to find αC simplifies to:

$$\alpha C = \alpha_n C_n \left(\frac{\alpha_{gn} + (1 - \alpha_{gn})\alpha_0}{\alpha_n} \right) - (1 - \alpha_{gn})\alpha_0 C_0 \quad (3)$$

We recognize the fractional expression in Equation 3 reduces to unity because $\alpha_{gn} + (1 - \alpha_{gn})\alpha_0$ is α_n because the alpha composition of the final *group alpha* with the backdrop alpha α_0 is simply α_n as alpha compositing is associative.

So Equation 3 simplifies to:

$$\alpha C = \alpha_n C_n - (1 - \alpha_{gn})\alpha_0 C_0 \quad (4)$$

Equation 4 can be realized in OpenGL by rendering a conservative window-space rectangle matching the rectangle used for the earlier `glCopyImageSubData` command with this subtractive blend

```
glBlendEquation(GL_FUNC_REVERSE_SUBTRACT);
glBlendFunc(GL_ONE, GL_ONE);
```

and a per-color sample fragment shader that outputs the product of fetching color values from the prior framebuffer instance to get C_0 and one minus the texel α_{gn} fetched from the single-component *group alpha* color buffer.

Step 4: Lastly composite the resolved group color αC and opacity α back to the prior framebuffer instance by rendering with per-sample shading another conservative window-space rectangle, here applying the blend mode for the group. The logical Steps 3 and 4 can be advantageously combined into a single rendering pass.

6.1.3 Implementing Isolated Groups

Isolated groups are easier. The isolated group rendering process follows the same general structure as shown in Figure 9 except Step 1 simply clears the group's framebuffer instance color buffer(s) to fully transparent. For RGB, this is clearing the color buffer to zero. For CMYK, this is clearing the RGB components to one and alpha component to zero. No `glCopyImageSubData` command is necessary.

Since an isolated group does not copy the backdrop from the prior framebuffer instance, there is also no need to "subtract out" that backdrop in Step 3 so this step can be skipped.

Steps 2 and 4 operate in the same manner as for non-isolated groups.

6.2 Knockout

By default, groups composite the elements of the group in their stacking order and use the prior element's rendering result as their backdrop. A group marked for knockout, known as a *knockout group*, always uses the group's initial backdrop. One common use of knock-out is rendering semi-opaque annotations where the annotations may overlap but double-blending of annotations is undesirable so only the last rendered annotation at any given pixel should blend with the prior framebuffer instance (the backdrop).

"Stencil, then cover" path rendering provides an efficient way to implement knockout by rendering the elements of the group in reverse stacking order. Blend normally during the "cover" step but mark every updated stencil sample by setting an upper bit in the stencil buffer for each updated color sample. Then have further rendering by other elements in the knockout group fail the stencil test if the stencil sample value's upper bit is set. This ensures color samples are only updated and blended by the last element in the group's stacking order to cover the color sample. When all the group's elements have been rendered, draw a conservative covering rectangle to unmark the stencil values so normal "stencil, then cover" rendering can proceed.

Note that this approach will not work if any of the immediate elements of the knockout group are non-isolated-groups. This uncommon case requires the non-isolated group element to use the backdrop of prior group elements in the stacking order which the reverse order will not have rendered so an explicit and involved knockout approach is required.

7 Shading

Illustrator supports constant color, linear gradients, radial gradients, and raster shading. These straightforward shading modes are performed by "cover" fragment shaders much as described by [Kilgard and Bolz 2012] with short shaders. The shaders must be tweaked to support outputting to NChannel framebuffers but are otherwise not particularly noteworthy. PDF's support for patterns and gradient meshes however present more interesting shading challenges.

7.1 Patterns

A *pattern* consists of a small graphical figure called a pattern cell, which is replicated at fixed horizontal and vertical intervals to fill an area. This process is called *tiling* the area. The pattern cell comprises graphical elements (such as paths, text, and images), may be non-rectangular in shape, and the spacing of tiles can differ from the dimensions of the cell itself. To draw a pattern, the pattern cell is drawn as many times as necessary to fill the given area. We accomplish this in one of two methods:

1. If a pattern object contains a non-isolated group with a blend mode, the contents of a pattern cell are drawn at each step, clipped to the tile area.
2. Otherwise, the contents of the pattern cell are drawn once into a separate texture (which is of the same size as pattern cell), and this texture is copied at each tile location.

The actual process of clipping to a tile is the same method described by [Kilgard and Bolz 2012] to clip to an arbitrary path.

7.2 Gradient Meshes

The PDF specification provide several mesh-based shading techniques for vector objects:

- Free-form and lattice-form smooth shaded triangle meshes,
- Coons patch [Coons 1967] meshes, and
- Tensor-product patch meshes.

Triangle meshes are fairly straightforward as the GPU is excellent at rendering smooth-shaded color triangles. The only caveats are we stencil test these triangles against the shaded object's stenciled region and use a final "cover" step but with color writes disabled to make sure the stenciled region is reset.

Patch meshes are more challenging than triangle meshes as edges of the patches are bicubic Bézier segments and the patch may fold over itself. The naïve approach would expand the patch into a triangle mesh and render in the same manner as the shaded triangle meshes. This has the disadvantage that a sufficiently tessellated triangle mesh to approximate each patch in a patch mesh (which might be hundreds of patches) is expensive for the CPU to generate and store. Having to CPU-tessellate all the patches undermines the compactness and editability advantages of Coons patches. Moreover the tessellated triangle meshes would be resolution-dependent so would not support fast zooming of the scene.

Fortunately modern GPUs support hardware tessellation units [Schäfer et al. 2014], but the application of this hardware is primarily directed at depth-tested 3D models formed from tessellated patches.

We harness this same tessellation hardware to render PDF's Coons and tensor-product patch meshes, but we identify some limitations of existing GPU hardware applied to our 2D tessellation task. Hardware tessellation splits the process of rasterizing patches into three programmable domains:

Vertex shading facilitating the transformation of control points from object space to other spaces.

Tessellation Control shading accepting an array of control points (transformed by vertex shading) and outputting a fixed (possibly different) number of control points, uniform patch values, and level-of-detail parameters to define a patch to evaluate.

Tessellation Evaluation shading evaluating the patch output from the tessellation control shader at a given (u,v) location within the patch as part of a mesh topology generated by fixed-function hardware.

At first glance, this hardware is readily amenable to tessellation of our 2D gradient mesh patches. The vertex shader can transform vertices from object space into window-space. The Tessellation Control Shader (TCS) subsequently performs a basis change from a Coons patch to a bicubic Bézier basis for ease of evaluation by the Tessellation Evaluation Shader (TES); the tensor-product patch is already a Bézier bicubic. The TCS uses the window-space control point positions to compute appropriate level-of-detail parameters to ensure every triangle in the tessellated topology is on the scale of about 1 to 2 pixels to minimize under or over tessellation. The TES should evaluate the 2D position at its (u,v) and interpolate a color based on color values assigned to the corner control points. Still there are three notable issues to address.

Resolving Mesh Overlap Render Order First GPU hardware tessellation does not guarantee the precise triangle rasterization order for a patch. This is justified because 1) 3D models are expected to be depth-tested to resolve hidden surface occlusion so there is no mandatory intra-patch triangle ordering (though the order is reasonably expected to be deterministic); and 2) the hardware is more efficient if it can group vertices into triangles to maximize vertex reuse. However PDF mandates a particular order:

> Patches can sometimes appear to fold over on themselves—for example, if a boundary curve intersects itself. As the value of parameter u or v increases in parameter space, the location of the corresponding pixels in device space may change direction so that new pixels are mapped onto previous pixels already mapped. If more than one point (u, v) in parameter space is mapped to the same point in device space, the point selected shall be the one with the largest value of v. If multiple points have the same v, the one with the largest value of u shall be selected. If one patch overlaps another, the patch that appears later in the data stream shall paint over the earlier one. [Adobe Systems 2008] §8.7.4.5.7

This provides a tidy, well-defined order but does not match the actual hardware rendering order. To resolve this, a combination of (u,v) and the current patch number (indicated by `gl_InvocationID`) can be combined into a [0,1] value to use as a depth value. For example:

```
gl_Position.z = float(gl_InvocationID*65536
  + int(v*255)*256 + int(u*255))/ 16777215.0;
```

Larger magnitude depth values are later in the rendering order. When the gradient mesh is opaque such that double blending is not a concern, depth buffering with a `GL_GREATER` depth buffer is sufficient to ensure the gradient patch mesh ordering. If that is not the case, a depth-only rendering pass to resolve the last update to every color sample followed by a second `GL_EQUAL` depth function pass to assign interpolated color and blend is necessary.

While we understand it is only the ordering of rasterized triangles within a patch that is implementation-dependent, one or more

patches may overlap a color sample so depth samples from different patches always compare with the latter patch in render order "winning".

Prior to rendering any set of patches, a depth clear to zero is necessary to reset the depth buffer. This could be done with a "cover" operation that simply zeros the depth buffer (without modifying other buffers) or with a scissored depth buffer clear.

Once the render order issues are resolved, color shading is a matter of bicubic interpolation [Sun et al. 2007] in the TES.

This is a lot of complexity to match the PDF specification's patch rendering order. Certainly if the hardware's tessellation generator simply guaranteed an order consistent with the PDF specification, even at the cost of some less optimal hardware efficiency, rendering PDF gradient meshes would be much more straightforward.

Another option is detecting via CPU preprocessing of the patch mesh whether or not actual mesh overlaps are present [Randrianarivony and Brunnett 2004]. When not present, gradient mesh rendering could be much more straightforward and efficient. In practice, we know overlaps are rare in real gradient mesh content.

Coarse Level-of-detail Control Graphics hardware tessellation has a limited maximum level-of-detail for tessellation. When the level-of-detail is clamped to a hardware limit for tessellation, tessellation artifacts may arise. We monitor the relative size of tessellated patches such that their maximum level-of-detail does not grossly exceed the scale of two or three pixels in window space. If this happens, patches need to be subdivided manually to ensure the patch mesh avoids objectionable tessellation artifacts. Care is necessary to maintain a water-tight subdivided patch mesh. This is done by ensuring exactly matching level-of-detail computations on mutual edges of adjacent patches.

8 Comparing GPU versus CPU Rendering

Our contributions for GPU-acceleration are best understood in contrast with Illustrator's pre-existing CPU rendering approach. All but a cursory description of Illustrator's CPU rendering approach is beyond the scope of this paper. Illustrator's CPU rendering closely follows the PDF standard [Adobe Systems 2008]. AGM's CPU renderer relies on a robust, expertly-tuned, but reasonably conventional active edge list algorithm [Foley et al. 1990] for rasterizing arbitrary paths including Bézier segments [Turner 2007]. Table 1 lists the differences between the CPU and GPU approaches in organizing the framebuffer storage for rendering. Table 2 lists the ways rendering is different between the CPU and GPU approaches.

9 Performance

We benchmarked our GPU-accelerated rendering mode against AGM's CPU-based renderer on six Illustrator documents pictured in Figure 10. We selected these scenes for their availability, artistic content, and complexity. Table 3 quantitatively summarizes each scene's complexity. We consider these scenes representative of the kind of complex artwork we wish to encourage by making its authoring more interactive.

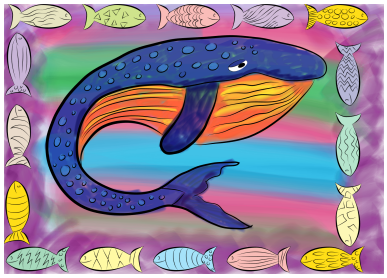
9.1 Benchmarking RGB Artwork

Table 4 presents our benchmarking results for RGB color model rendering. Our benchmarking method executes a script that zooms and pans over the content to mimic the kind of fast view changes an

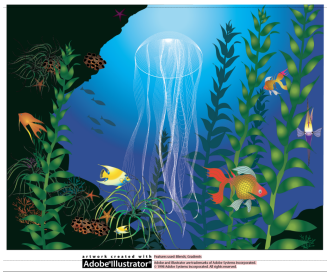
(a) WF_BambooScene.ai

(b) archerfish.ai

(c) Blue Mirror.ai



(d) whale2.ai



(e) Tropical Reef.ai

(f) bigBlend2.ai

Figure 10: *Challenging Illustrator artwork for benchmarking.*

Capability	CPU	GPU
Per-component storage	8-bit fixed-point components	8-bit fixed-point components (same)
Color opacity representation	Non-premultiplied (*straight*) alpha	Premultiplied §4.2
Per-pixel storage	One color value per pixel	Multisampling: 8 distinct (hardware compressed) colors per pixel + 8 stencil samples §5.2.2
CMYK representation	CMYK color components stored "as is"	Components stored complemented §5.1
CMYK organization	N channels + alpha channel allocated as linear image in system memory	Multiple RGBA buffers, rendered as multiple render targets §5.2.1
CMYK excess framebuffer storage	None	Color components allocated in multiples of 3, with alpha duplicated for every 3 color components §5.2.1,5.2.2

Table 1: *Comparison of framebuffer storage and representation in CPU and GPU rendering modes for Illustrator.*

Capability	CPU	GPU
Rendering approach	Scan-line rasterization, professionally optimized	GPU "stencil, then cover" rendering via `NV_path_rendering`
Rendering granularity	Cache of sub-image tiles for image reuse	Direct rendering to GPU framebuffer
Resolve to displayed color	Downsample of high resolution tile samples + apply color profile via CPU	Multisample downsample, then apply color profile via GLSL shader
Rendering implementation	C++ code directly manipulates pixels with CPU	CPU orchestrating OpenGL commands using GLSL shaders; pixel touching by GPU
Parallelism	Multi-core CPU rendering, divvying screen-space tiles among threads	GPU pipeline parallelism; NVIDIA dual-core OpenGL driver mode
Path control point transformation	CPU-based math	GPU-based vertex shading internal to `NV_path_rendering`
Stroking	Conversion of stroked paths to fills	`NV_path_rendering` stencils stroked paths directly—approximating cubic Bézier segments as a sequence of quadratic strokes
Antialiasing mechanism	Fractional coverage during scan-line rasterization	8x multisampling §5.2.2
Blend modes	Optimized C++ with MMX and SSE intrinsics	Conventional GPU blending for *Normal* and *Screen*; `KHR_blend_equation_advanced` for advanced PDF blend modes §4
Non-isolated transparency group	Initialize (system memory) group framebuffer from lower layer contents; CPU render group with both non-isolated *and* isolated alpha; once group is rendered, composite from group framebuffer to lower layer after first subtracting out lower layer contents from group framebuffer	Allocate renderable GPU color texture(s) for layer *and* 1-component "isolated alpha" buffer cleared to zero; blit lower layer contents to new framebuffer object color contents; GPU render group with extra 1-component buffer accumulating isolated alpha; reverse subtract lower layer color from group color texture(s) based on isolated alpha; composite with blend mode color texture(s) for layer to lower layer framebuffer object §6.1.2
Isolated transparency group	Initialize group framebuffer (system memory) to clear; GPU render group with single alpha; once group is rendered, composite with blend mode from group framebuffer to lower layer	Allocate renderable GPU color texture(s) for layer; GPU render group; composite with blend mode color texture(s) for layer to lower layer framebuffer object §6.1.3
Knockout	Tagging pixels as done immediately once written in a knockout group	Reverse rendering of objects with the layer; unless combined with non-isolated group when depth buffering is used §6.2
Path clipping	Concurrent scan-line rasterization of clip path	Stencil clip path with `NV_path_rendering` into upper bits of the stencil buffer; then stencil draw path against stencil buffer upper bits §7.1
Gradients	CPU shading code in C++	GLSL fragment shaders with 1D texture for filtering and attribute generation by glPathTexGenNV §7
Gradient mesh implementation	CPU-based recursive mesh subdivision	Direct evaluation of 2D mesh vertex position & color via GPU programmable tessellator, then hardware rasterization §7.2
Implementation of Illustrator Effects (FX)	CPU-based effect plugin code processes layer image to make new image	Layer image must be read to CPU; CPU-based effect plugin code processes layer image to make new image; new image loaded as a texture and applied as image gradient §3

Table 2: *Comparison of rendering approaches in CPU and GPU rendering modes for Illustrator.*

Scene	Paths	2D Control Points	Clipping Masks	Transparent Groups	Gradients	Embedded Images	Native Color Space
WF_BambooScene	84,995	618,926	32,367	25,614	126	0	CMYK
whale2	14,403	481,928	14,333	14,313	0	0	RGB
ArcherFish	11,214	203,771	6,777	2,105	1,973	524	RGB
Tropical Reef	1,041	6,698	0	0	291	0	CMYK
BlueMirror	132,138	1,685,979	66,803	66,792	0	0	RGB
bigblend2	9,428	92,645	2,531	1,457	1,096	1	CMYK

Table 3: *Scene complexity metrics.*

RGB mode Scene	Monitor HD Resolution	CPU ms	GPU ms	Gain
WF_Bamboo Scene	Full	320	174	3.53x
	Ultra	1071	219	5.23x
whale2	Full	336	37	8.41x
	Ultra	1015	129	8.02x
ArcherFish	Full	259	28	9.04x
	Ultra	979	100	9.59x
Tropical Reef	Full	64	20	3.36x
	Ultra	179	30	6.28x
BlueMirror	Full	1209	100	11.39x
	Ultra	2971	279	10.98x
bigblend2	Full	828	44	14.38x
	Ultra	3211	142	17.54x

Table 4: *Average frame time in milliseconds rendering complex art scenes in RGB document mode (CMYK artwork is forced to RGB) at varying zooms and panning, comparing the existing CPU rendering mode to our new GPU-accelerated mode.* Gain *is the geometric mean of the speedup of GPU over the CPU mode for corresponding benchmark frames.*

CMYK mode Scene	Monitor HD Resolution	CPU ms	GPU ms	Gain
WF_Bamboo Scene	Full	392	405	1.34x
	Ultra	1520	630	2.37x
Tropical Reef	Full	99	22	4.80
	Ultra	311	38	9.28x
bigblend2	Full	861	111	6.73x
	Ultra	3542	381	10.55x

Table 5: *Average frame time in milliseconds rendering complex CMYK art scenes at varying zooms and panning, comparing the existing CPU rendering mode to our new GPU-accelerated mode.* Gain *is the geometric mean of the speedup of GPU over the CPU mode for corresponding benchmark frames.*

artist would use during interactive inspection of the artwork. During the benchmark, the Illustrator application and document view are both *maximized* for the most visible pixels.

Our system configuration is a Windows 7 PC with a Xeon E3-1240 V2 CPU @ 3.40GHz (4 cores), 8 GB RAM, and NVIDIA GeForce GTX 780 Ti GPU. AGM's CPU-based renderer automatically takes advantages of the CPU's multiple cores for parallel rendering. NVIDIA's OpenGL driver is automatically configured for dual-core operation so the application thread communicates OpenGL commands to a driver thread so application and driver processing operate concurrently. We benchmarked the 64-bit version of the latest Illustrator. Illustrator always renders with 8x multisampling.

We are particularly interested in how GPU-accelerated Illustrator can improve the user experience in expectation of the mass-market adoption of *4K resolution* displays so we report frame times using Full HD resolution (1920x1080) and Ultra HD (3840x2160) monitors. Increasing the display resolution from Full to Ultra HD increases the geometric mean of the relative increase in CPU render time by 190%; but only 22% for the GPU-accelerated transition. We note that the CPU rendered Ultra HD frame render times are on the order of seconds; the GPU-accelerated frame rates are 5+ frames per second (7.9 average) so still within what artists tolerate as interactive.

9.2 Benchmarking CMYK Artwork

Table 5 presents CMYK benchmarking results for the three "native CMYK" scenes listed in Table 3. Whereas the benchmarking of these scenes in Table 4 forced the scenes be converted to RGB, Table 5 benchmarks these scenes in their native CMYK color space by rendering with a CMYK framebuffer.

The WF_BambooScene scene is included specifically because it demonstrates rather poor GPU rendering performance, particularly when rendered in the scene's native CMYK color space. The WF_BambooScene scene is an example of a scene constructed in ways that stress GPU rendering with poor results—though the scene is quite challenging for CPU rendering too! First the scene itself has a large number of paths, a variety of blend modes, and uses knockout. The scene already has a large number of transparency groups, but many become non-trivial groups when used with non-*Normal* blend modes and knock-out. Recall `NV_blend_equation_advanced` limitations that make blend modes overly expensive in CMYK rendering (Section 5.2.3). Additionally when zoomed out of the scene, there is a large number of intricate paths placed invisibly outside the scene's framed view (this is not uncommon for artists to do as a way of stashing fragments of artwork). The effect is acceptable GPU-acceleration when zoomed into the CMYK scene but worse-than-CPU-rendering performance when zoomed out. Performance is good zoomed in because the pathological features of the scene get view culled away.

We now look at a more typical scene in detail. Figure 11 graphs the render time for at a variety of zoom levels for the (native CMYK) Tropical Reef scene. Illustrator documents are in "real world" dimensions so a 100% zoom corresponds to its printed dimensions. We graph the zoom factor squared (so 1 is a 100% zoom and 4 is a 200% zoom) because this normalizes the screen-space area of a given scene region. The graph shows the scene rendered in all eight combinations of CPU/GPU rendering, Full/Ultra HD resolution, and CMYK/RGB color models over a range of zoom levels. While the original scene is authored in CMYK, by forcing a conversion to the RGB color mode, we can compare the relative expense of CMYK relative to RGB rendering.

While the window size in pixels is constant at either Full or Ultra HD resolution, the render time is not stable at increasing zoom lev-

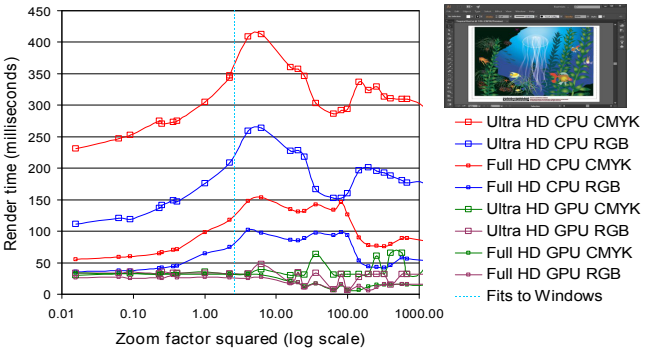


Figure 11: *Rendering time for Tropical Reef.ai scene, comparing GPU vs. CPU, Full vs. Ultra HD, and RGB vs. CMYK at over a large range of zoom factors squared. Dashed vertical lines indicate the zoom factor when the scene "Fits to Window" for Full and Ultra HD respectively.*

els because many objects can be culled from the scene as the zoom increases while also small objects become large and hence more expensive to draw. This factor affects both the CPU and GPU render times but in different ways we now explore.

The CPU renderer is sensitive to having a large number of objects, and hence active edges, to process. Also complex shading and blending is relatively more expensive for the CPU while shading and blending are quite efficient for the GPU. In contrast while the CPU's scan-line rasterizer is quite work-efficient and "cache friendly" because it operates on just a scan-line at a time, the GPU renderer is challenged when paths are large in screen space so the overdraw from the "stencil" step becomes rasterization bound. Lots of stencil counting that ultimately cancels to zero or generates large winding number magnitudes create costly rasterization burdens for the GPU. Likewise expensive quadratic discard shaders for curved stroked segments become expensive when the stroke width is more than a fix pixels wide in screen space.

Even so, GPU performance is consistently faster than the CPU performance but subject to more significant variations at different zoom levels. To help quantify the relative cost of CMYK rendering via the GPU we can compare native CMYK rendering on the GPU to an RGB-converted version of the Tropical Reef content in Figure 11. RGB-converted rendering averages 36% (Full HD) to 43% (Ultra HD) faster than CMYK rendering with the CPU. The GPU averages 6% (Full HD) to 9% (Ultra HD) faster when the CMYK artwork is converted to RGB but these averages mask significant variability. So while the framebuffer memory consumption for CMYK is at least double the storage for RGB color mode rendering, the observed performance cost is not nearly so bad and is still much faster than CMYK rendering on the CPU.

9.3 Comparison with Recent Work

[Ganacim et al. 2014] provides performance results for a number of SVG scenes using a massively-parallel vector graphics (MPVG) system based on CUDA. Table 6 presents a subset of their scenes most relevant to an Illustrator artist with results from our comparable PC configuration. Our rendering performance relies on `NV_path_rendering` but performs markedly better than the `NV_path_rendering` performance reported in their paper. We attribute this to forcing use of NVIDIA's dual-core driver and Illustrator's tuned OpenGL usage and scene traversal. For all but two of scenes in our table, GPU-accelerated Illustrator is faster than their published results—the exceptions are a detailed but inefficiently authored (so CPU bound) Paris map and their—arguably pathological—Contour scene.

Input	Resolution	MPVG 8x	MPVG 32x	Illustrator GPU 8x
Car	1024x682	12.86	14.73	2.94
	2048x1364			3.09
Drops	1024x1143	14.28	18.59	2.29
	2048x2286			6.21
Embrace	1024x1096	15.50	19.38	1.69
	2048x2192			3.63
Reschart	1024x625	8.51	11.14	1.92
	2048x1250			2.60
Tiger	1024x1055	12.89	17.24	1.48
	2048x2110			5.34
Boston	1024x917	37.22	41.81	2.66
	2048x1834			5.02
Hawaii	1024x844	26.16	29.48	3.83
	2048x1688			8.75
Contour	1024x1024	30.07	30.36	90.53
	2048x2048			328.57
Paris 50K	1024x1024	26.82	25.22	65.53
	2048x2048			64.23

Table 6: *Comparison of SVG content render times (in milliseconds) reported by [Ganacim et al. 2014] to our work with Illustrator.*

MPVG's strengths are antialiasing quality and an ability to handle what would typically be considered inefficiently structured scenes—strengths we readily acknowledge. Since Illustrator supports just 8x multisampling and their reported image resolutions are rather low, we provide "double resolution" rendering results as an alternative for their 32x rendering quality.

While MPVG's rendering quality and novelty of approach is impressive, Illustrator must support the entire PDF rendering model including CMYK, blend modes, transparency groups, etc. all while making everything editable.

10 Conclusions

We have succeeded in GPU-accelerating Illustrator despite decades of being unadvantaged by graphics hardware. Our benchmarking shows significant speed-ups, particularly at 4K resolution. Our contributions introduce novel techniques for supporting CMYK color space rendering on existing GPUs, proper transparency group support including non-isolated groups and knock-out, and mapping PDF's gradient mesh shading to GPU tessellation.

10.1 Broader Hardware Support

OpenGL 4.4 is the multi-vendor standard baseline for our GPU-acceleration effort but Illustrator also requires the `NV_path_rendering` and `KHR_blend_equation_advanced` extensions. As a practical matter, today just NVIDIA GPUs (Fermi generation [Wittenbrink et al. 2011] and beyond) on Windows support the prerequisite OpenGL functionality. Illustrator supports a wide range of system configurations so we naturally want these GPU-acceleration benefits supported more broadly. We are exploring ways to support a broader range of GPUs. Standardization of `NV_path_rendering` would make this much easier.

10.2 Future Work

Much of the user interface of Illustrator today assumes re-rendering the scene is slow and expensive. For example, the user interface encourages users to *isolate* a portion of the scene to avoid rendering the complete scene. Similarly editing often happens by dragging overlaid "blue lines" rather than a more *what-you-see-is-what-you-get* interaction model. We hope the GPU-acceleration we have introduced into Illustrator will facilitate more powerful, intuitive, and fluid user interfaces for vector graphics editing.

We note a number of inspired research efforts relevant to vector graphics editing that are either conceived with GPU-acceleration in mind—such as diffusion curves [Andronikos 2013]—or introduce vector graphics complexity that overwhelms conventional CPU-based vector graphics rendering—such as digital micrography [Maharik et al. 2011]. We hope by bringing Illustrator to the GPU, these techniques will become tractable to support within Illustrator and thereby be adopted by digital artists.

Our framebuffer memory usage is substantial. We want to incorporate NVIDIA's `NV_framebuffer_mixed_samples` OpenGL extension [Bolz 2014] that allows an OpenGL framebuffer object to have fewer color samples than stencil samples (and no depth samples) to reduce memory usage without reducing the rasterization quality of path rendering.

We believe graphics hardware architects can improve the support for print-oriented features—in particular the CMYK color space. Doing so would greatly reduce the memory bandwidth, memory footprint, and greatly simplify the complex orchestration of multiple RGBA color buffers required to accomplish CMYK rendering.

Acknowledgments

We thank: David Aronson, Rui Bastros, Jeff Bolz, Rajesh Budhiraja, Nathan Carr, Qingqing Deng, Vineet Punjabi, E Ramalingam, Anubhav Rohatgi, Lekhraj Sharma, Gopinath Srinivasan, Tarun Beri, and our anonymous reviewers.

References

61 SOLUTIONS INC., 2013. Mischief | Sketching & Drawing & Painting Software. http://madewithmischief.com/ .

ADOBE SYSTEMS. 1985. *PostScript Language Reference Manual*, 1st ed. Addison-Wesley Longman Publishing Co., Inc.

ADOBE SYSTEMS, 2000. Transparency in PDF, Technical Note #5407, May.

ADOBE SYSTEMS. 2008. *Document management–Portable document format–Part 1: PDF 1.7*. Also published as ISO 32000.

ANDRONIKOS, N., 2013. What's so cool about diffusion curves.

BOLZ, J., 2009. `ARB_texture_multisample` extension.

BOLZ, J., 2014. `NV_framebuffer_mixed_samples` extension.

BOUTON, G. D. 2012. *CorelDRAW X6 The Official Guide*. McGraw-Hill Osborne Media.

BROWN, P., 2013. `NV_blend_equation_advanced` extension.

BROWN, P., 2014. `KHR_blend_equation_advanced` extension.

COONS, S. A. 1967. Surfaces for Computer-aided Design of Space Forms. Tech. Rep. MIT/LCS/TR-41, MIT, May.

FOLEY, J. D., VAN DAM, A., FEINER, S. K., AND HUGHES, J. F. 1990. *Computer Graphics: Principles and Practice (2nd Ed.)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

FRISKEN, S. F., AND PERRY, R. N. 2006. Designing with distance fields. In *ACM SIGGRAPH 2006 Courses*, ACM, New York, NY, USA, SIGGRAPH '06, 60–66.

GANACIM, F., LIMA, R. S., DE FIGUEIREDO, L. H., AND NEHAB, D. 2014. Massively-parallel vector graphics. *ACM Transactions on Graphics (Proceedings of the ACM SIGGRAPH Asia 2014) 36*, 6, 229.

GOLD, M., AND SELLERS, G., 2012. `ARB_copy_image` extension.

ILBERY, P., KENDALL, L., CONCOLATO, C., AND MCCOSKER, M. 2013. Biharmonic diffusion curve images from boundary elements. *ACM Trans. Graph. 32*, 6 (Nov.), 219:1–219:12.

KHRONOS GROUP. 2014. *The OpenGL Graphics System: A Specification*, Version 4.4 (Compatibility Profile) ed.

KILGARD, M. J., AND BOLZ, J. 2012. GPU-accelerated path rendering. *ACM Trans. Graph. 31*, 6 (Nov.), 172:1–172:10.

KILGARD, M., 2012. `NV_path_rendering` extension.

KIRSANOV, D. 2009. *The Book of Inkscape: The Definitive Guide to the Free Graphics Editor*. No Starch Press.

LEBEN, I. 2010. *Random Access Rendering of Animated Vector Graphics Using GPU*. Master's thesis, RMIT University, Melbourne, Australia.

MAHARIK, R., BESSMELTSEV, M., SHEFFER, A., SHAMIR, A., AND CARR, N. 2011. Digital micrography. *ACM Trans. Graph. 30*, 4 (July), 100:1–100:12.

NEHAB, D., AND HOPPE, H. 2008. Random-access rendering of general vector graphics. In *ACM SIGGRAPH Asia 2008 papers*, SIGGRAPH Asia '08, 135:1–135:10.

ORZAN, A., BOUSSEAU, A., BARLA, P., WINNEMÖLLER, H., THOLLOT, J., AND SALESIN, D. 2013. Diffusion curves: A vector representation for smooth-shaded images. *Commun. ACM 56*, 7 (July), 101–108.

RANDRIANARIVONY, M., AND BRUNNETT, G., 2004. Necessary and sufficient conditions for the regularity of a planar coons map.

SCHÄFER, H., NIESSNER, M., KEINERT, B., STAMMINGER, M., AND LOOP, C. 2014. State of the art report on real-time rendering with hardware tessellation.

SMITH, A. R. 1995. Image Compositing Fundamentals. Tech. Rep. Technical Memo 4, Microsoft, Aug.

SUN, J., LIANG, L., WEN, F., AND SHUM, H.-Y. 2007. Image vectorization using optimized gradient meshes. *ACM Trans. Graph. 26*, 3 (July).

SUN, X., XIE, G., DONG, Y., LIN, S., XU, W., WANG, W., TONG, X., AND GUO, B. 2012. Diffusion curve textures for resolution independent texture mapping. *ACM Trans. Graph. 31*, 4 (July), 74:1–74:9.

SVG WORKING GROUP, 2011. Scalable Vector Graphics (SVG) 1.1 (2nd edition).

SVG WORKING GROUP, 2011. SVG compositing specification. W3C working draft March 15, 2011.

TURNER, D., 2007. How freetype's rasterizer work [sic], Feb. http://git.savannah.gnu.org/cgit/freetype/freetype2.git/tree/docs/raster.txt .

VALENTINE, S. 2012. *The Hidden Power of Blend Modes in Adobe Photoshop*. Adobe Press.

WARNOCK, J., AND WYATT, D. K. 1982. A device independent graphics imaging model for use with raster devices. In *Proceedings of the 9th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '82, 313–319.

WITTENBRINK, C., KILGARIFF, E., AND PRABHU, A. 2011. Fermi GF100 GPU architecture. *Micro, IEEE 31*, 2, 50–59.