A Realistic 2D Drawing System

Category: System

Abstract

Two dimensional graphics systems are often characterized as just 3D graphics with a fixed Z value. This paper describes a graphics system built to support high quality 2D rendering by applications while taking advantage of hardware acceleration designed for 3D graphics.

The system is divided into three stages: tessellation, rendering and compositing. The tessellation stage contains a novel algorithm for handling splines stroked with an elliptical pen. The rendering stage provides precise pixelization semantics and access to hardware acceleration. Image compositing is enhanced with new operators that extend the capabilities of traditional Porter/Duff compositing to provide a formal model that supports common incremental drawing methods.

CR Categories: I.3.2 [Computer Graphics]: Graphics Systems—Distributed/network graphics; I.3.3 [Computer Graphics]: Picture/Image Generation—Antialiasing, Bitmap and framebuffer operations; I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling—Boundary representations;

Keywords: Bézier Splines,Document Manipulation,Image Composition,Antialiasing

1 Introduction

In this paper, we describe some features of a new realistic 2D graphics system designed to support everyday 2D applications. We use the term "realistic" to describe both high quality results and also to distinguish our system from the bulk of existing 2D graphics systems. These existing 2D graphics systems may suffice for drawing dialog boxes but often fail to provide sufficient power or control to support sophisticated applications. Without such support applications are often forced to render an image using custom code. The native graphics system serves only to transport the completed image to the screen.

Our system provides interfaces at a variety of levels enabling applications to take advantage of whatever pieces of the system are appropriate. The result is a system which provides access to useful hardware acceleration and shared drawing code without constraining the application architecture. An overall view of the system architecture is presented in Figure 2. Applications present objects in the form of splines, trapezoids or images. Splines are tessellated into trapezoids, trapezoids are rendered to images and the images are composited to the graphics surface. Figures 1, 3 and 11 show some images generated with this system.

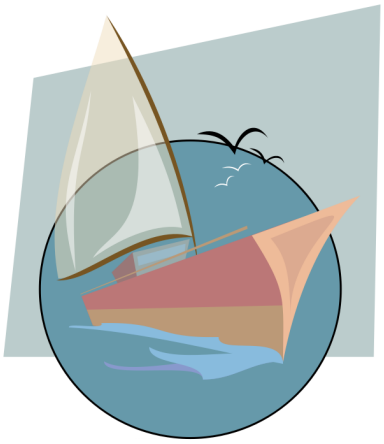


Figure 1: Sample rendering of SVG (artwork courtesy of Petr Vlk)

Unlike other 2D systems, the tessellation stage is done by an application library which can be used to drive several low level graphics devices in addition to the graphics display. The trapezoid rendering and image compositing stages are precisely defined so they can be duplicated for each graphics device, providing exact image matching across all supported devices.

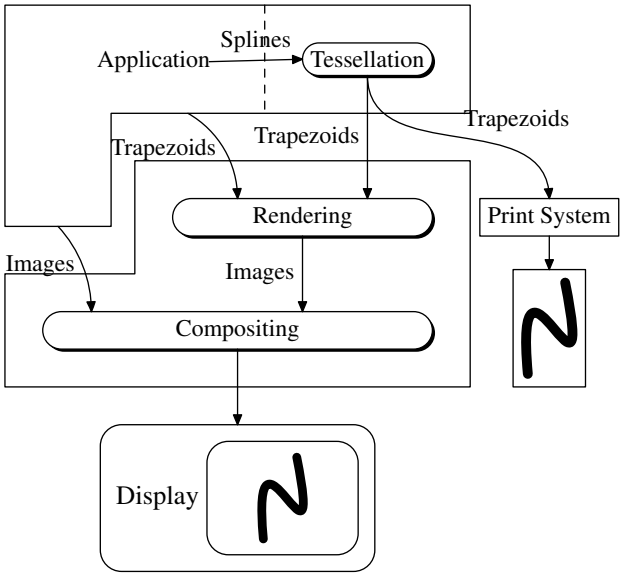


Figure 2: System Overview

Our system is designed to take advantage of hardware built to accelerate OpenGL[Segal et al. 1999] and DirectX[Bargen and Donnelly 1998] while providing semantics suitable for 2D applications.

In addition to these architectural benefits, this work includes novel extensions to the Porter/Duff compositing model and a new algorithm for efficient drawing stroked splines based in polygon convolution theory[Guibas et al. 1983].

Figure 3: Sample rendering of SVG (artwork courtesy of Petr Vlk)

2 Motivation

This new graphics architecture is motivated by the needs of applications designed to produce documents using two dimensional graphics operations. Such applications are not well served by existing graphics systems and often resort to custom code. Where implemented on top of an existing window system, this custom code cannot readily take advantage of hardware acceleration. Similarly, printing support becomes difficult and often available only for a limited class of output devices.

Exploring the requirements for current page description mechanisms, we found that many have the same basic structure in common; a "painter's" model that closely follows Porter/Duff[Porter and Duff 1984] image compositing and splines that can either be stroked or filled for geometric objects. We provide cubic Bézier splines as the only spline representation as they are common in practice and they are readily targeted by other spline forms.

2.1 Precisely Matched Rendering

With the input of the graphics system specified, an analysis of the required output reveals a relatively short list. Applications either generate graphics for display in the window system, construct local images in memory or produce page description languages for printing. One common requirement among applications is that all forms of output produce precisely the same results; variation from display to image or printer is not well tolerated by users.

Application requirements for precise matching of rendering between display, memory and printing necessitate a greater level of control over output than existing graphics APIs provide. Instead of exposing only the highest level graphics primitive with no guarantees of pixelization, this new system provides a range of interfaces at many levels so that applications can manipulate objects where necessary to produce the desired results.

Our system splits geometric rendering into three distinct steps and exposes well defined interfaces between them, allowing applications control over how each step is performed.

Tessellation is performed within the application through a library interface. The resulting trapezoids may be rendered locally in memory or sent to a printer or the window system. Because the tessellation is always performed by the same library for all output media, the results can match precisely. This particular design is similar to OpenGL where applications tessellate geometry and present triangles to the graphics system. Our system is unique among 2D rendering systems all of which expose a broad set of high level operations but no lower level primitives.

Rendering and image compositing may be performed locally within the library to manipulate images in memory, or within the window system for display. The pixelization semantics of the system are defined so that results computed in either place will match exactly. The X Window System [Scheifler and Gettys 1992] provides precise pixelization semantics for its operations, but polygons are specified with integer coordinates making them unsuitable for tessellation. OpenGL and the other 2D graphics systems do not provide any precise specification for pixelization making it impossible to duplicate the rendering mechanism for in-memory images.

2.2 Hardware Acceleration

Modern graphics hardware is designed to efficiently accelerate the OpenGL and DirectX APIs. These both provide image compositing using the Porter/Duff operators; our system is designed to parallel these 3D libraries at the lowest levels so that accelerated implementations can be developed. Image compositing is done in color value space, as that is most commonly supported by hardware. Future versions of our system will incorporate support for gamma-corrected compositing when the hardware capabilities are better understood by the authors.

Existing graphics chipsets have some variability in polygon rendering. As precise pixelization is not required by all applications, our system permits the selection of an imprecise rendering mode. An implementation of imprecise rendering is constrained to preserve the semantics necessary for reasonable output.

The precise rendering specification was chosen to permit a reasonably efficient software implementation so that applications wouldn't be forced to use imprecise polygons for acceptable performance. Several different algorithms were analyzed and the current one chosen as representing the best balance between speed and accuracy.

3 Tessellation

The highest-level geometric input accepted by our graphics system is a path consisting of cubic Bézier splines and straight line segments. A path can be filled or stroked by a circular/elliptical brush. In either case, the rendering of a path consists of tessellating the desired form into a series of trapezoids which are passed to the next stage of the rendering system as shown in Figure 2. The tessellation of a filled path is straightforward using known techniques, while the rendering of a stroked path deserves further discussion.

3.1 Stroking Splines via Convolution

An interesting problem for 2D drawing systems is the rendering of a curved path as swept out by a curved, convex brush. The result of sweeping a brush along a path is described by the notion of a *Minkowski Sum*. Given two regions A and B in the plane, their Minkowski sum is the result of all pairwise vector sums for all points in A and B, that is:

$$A + B \equiv \{\mathbf{a} + \mathbf{b} \,|\, \mathbf{a} \in A \text{ and } \mathbf{b} \in B\}$$

This operation is depicted in Figure 4.

Our approach to computing the Minkowski sum is based on the theory of tracings and convolutions as set forth by Guibas, Ramshaw, and Stolfi. A tracing is defined around the contour of each region, and the convolution of the two tracings computes a boundary whose interior, (as determined using the winding rule), is

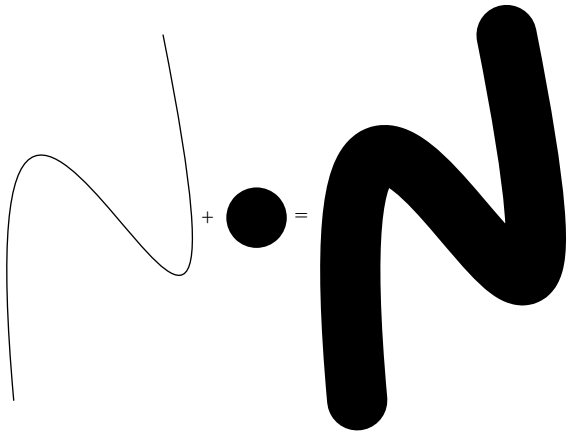


Figure 4: Minkowski sum of shape and pen

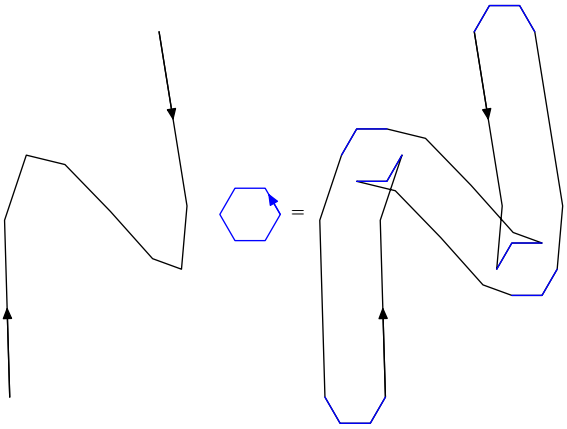


Figure 5: Convolution of tracings for shape and pen

the Minkowski sum of the two regions. Figure 5 depicts the convolution operation given polygonal tracings approximating the boundaries of the regions shown above.

The computational benefit of this technique stems from the fact that the polygonal boundary formed by the convolution of two polygonal tracings consists of nothing more than a series of polygonal segments from each tracing as translated by appropriate vertices from the other tracing. This is significantly simpler than the true outline of a spline as stroked by a circular pen, yet we are still able to approximate the exact result within a user-specified error bound.

3.2 Convolution Algorithm

This section presents an algorithm to render a Bézier spline as stroked by a circular brush of diameter *width*. The computed boundary is within an error bound of *flatness* from the exact result.

We first decompose the spline into a series of straight line segments using the algorithm developed by deCastlejau and described by Farin[Farin 1990]. The recursive decomposition is terminated when the straight line segments deviate from the exact curve by less than *flatness*. It is worth noting at this point that the accuracy required for the approximation of the spline is a function only of *flatness* and is independent of the width of the pen.

We also construct a polygonal pen as a regular polygon approximating the brush within *flatness*. The minimum number of pen vertices needed is computed as:

$$\left\lceil \frac{\pi}{\arccos\left(1 - \frac{2 \cdot flatness}{width}\right)} \right\rceil$$

The number of vertices is then increased as needed to result in an even integer ≥ 4.

For the purpose of accurately computing caps for the spline, it is convenient if the polygonal pen breaks apart into two halves, with one half acting as the endcap. We arrange this by inserting new vertices into the pen as necessary.

Next, we establish a *tracing* from our polygonal pen by traversing it in counterclockwise order. Imagine an observer walking straight along each segment around the polygon, turning in place at each vertex. The pair of entry and exit directions are computed and stored for each vertex. This pair of directions define the *active tangent range* for each vertex. If we imagine the polygonal pen being dragged along the path, the vertices which contain the current path tangent within their active range are precisely those vertices that contribute to the convolution boundary.

With those preliminaries out of the way, the body of the algorithm is quite simple:

```
CONVOLVE: shape, pen → convolution
    convolution := empty
    v := initial vertex of shape
    a := find active vertex of pen given tangent of shape at v
    while v is not final vertex of shape
            add v + a to convolution
            slope := tangent of shape at v
            if slope is counterclockwise of active range of a
                    a := next vertex of pen
            else if slope is clockwise of active range a
                    a := previous vertex of pen
            else
                    v := next vertex of shape
```

We begin by finding the active pen vertex a at the beginning of the shape. Then, we simply iterate over each vertex v in the shape. At each iteration, we add to the convolution the current shape vertex translated by the current active pen vertex. Then, if the tangent of the shape lies within the active range of a we continue on to the next vertex v in the shape. Otherwise, we step a around the pen until its active range contains the current tangent. As we step around the pen, we add the traversed pen segments to the convolution.

For sake of demonstration, we have shown an algorithm that computes only half of the convolution. The second half could be generated by a second traversal of the path in the opposite direction as suggested by Guibas et al. We improve on this by always using a pen with 180° radial symmetry. In this way, the active pen vertex for the "backward" half of the convolution is always directly opposite the "forward" active vertex. Our implementation is able to generate the full convolution in a single pass.

Figure 6 depicts 4 stages of the algorithm from the convolution shown earlier. In 6(a) the pen has just advanced to v. At this point, the shape turns sharply in place and the exit tangent of the shape does not lie within the active range of a. Therefore, in 6.(b), the active vertex a has been stepped twice clockwise around the pen so that its active range now contains the tangent. In 6(c), the shape is again turning in place, but this time the new tangent remains within the active range. So we simply advance the pen again resulting in the final convolution of 6(d).

Generating the convolution takes linear time with respect to the number of vertices in the approximation of the shape. The tessellation of the convolution into trapezoids is performed incrementally

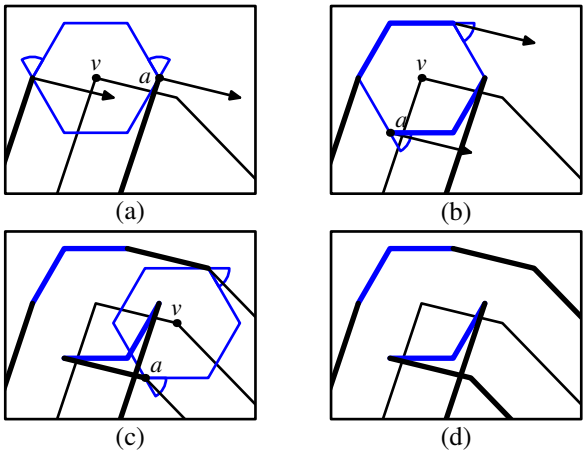


Figure 6: Detail of convolution operation

along with the generation of the convolution. We have discussed the algorithm in terms of a Bézier spline and a circular brush, but since the convolution algorithm operates solely on polygonal tracings, the algorithm is general enough to accurately render any shape stroked by any convex brush as long as both can be approximated as polygons within an arbitrary error bound.

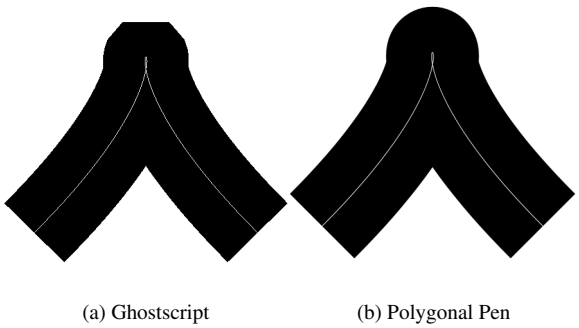
3.3 Related Work

Many 2D graphics systems employ a simplistic technique for stroking curves with a brush: decomposing the curve into straight line segments, then stroking those segments. For example, the PostScript `flattenpath` operator declares this as the method used in PostScript. There are a number of difficulties in accurately rendering a stroked curve with this approach. First, it is readily apparent that computational requirements are increased since many more line segments must be used to approximate the spline in order to guarantee smooth edges on the stroke outline. Second, it is difficult to determine how accurately the spline must be approximated since this is a function of both the width of the pen and the curvature at each point along the spline. Third, for very wide pens and very sharp turns, there may not be sufficient numerical precision available to represent the very short line segments necessary.

Figure 7(a) demonstrates clear visual artifacts in the Ghostscript rendering of a spline. The image of Figure 7(b) shows the result from our system using the convolution algorithm described above. Each image is generated on a 450x450 pixel array with the *flatness* parameter set to 0.2 pixels. We have demonstrated similar artifacts using a PostScript interpreter available in a current model laser printer.

The convolution of polygonal tracings has been applied to the problem of drawing stroked shapes before. Indeed, this application was the original motivation for the mathematical framework invented by Guibas et al. Hobby[Hobby 1989] and Knuth[Knuth 1986] both use this convolution when stroking splines. Hobby takes particular care in the design of *pen polygons*—polygonal pens which achieve uniform stroke width when rendering to a bi-level output device.

Our system does not need Hobby's carefully constructed pen polygons since antialiasing removes the artifacts which pen polygons are designed to eliminate. However, there is still a significant advantage to using a polygonal pen to stroke the path rather than an exact circle. After performing the convolution of a polygonal pen with a path, the result contains portions of the path translated by pen vertices interspersed with portions of the pen itself. In Knuth's



(a) Ghostscript (b) Polygonal Pen

Figure 7: Comparison of spline rendering

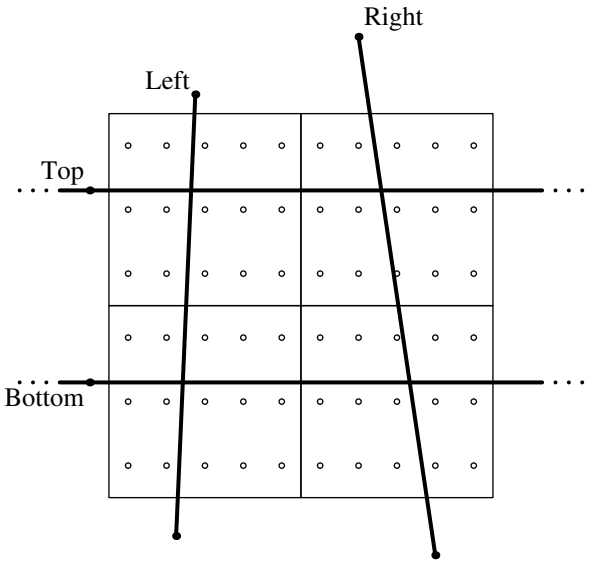


Figure 8: Trapezoid and sample grid for four pixels

work, the convolution consists of both curves, (from the path), and linear segments, (from the pen). Our technique yields a simpler representation for the convolution, (piece-wise linear), while still approximating the exact convolution with user-defined precision.

4 Rendering Polygons

Several existing 2D graphics applications and libraries (Adobe Postscript[Adobe Systems Inc. 1985], Adobe Acrobat, Ghostscript[ghostscript 2003], Gnome libart[Pennington 1999]) tessellate geometric figures into horizontal bands split into trapezoidal regions. These regions are delineated by arbitrary lines; the use of an arbitrary line provides increased accuracy when a single straight edge spans multiple bands.

Our system represents these as separate trapezoids with two horizontal lines and two arbitrary non-horizontal lines as seen in Figure 8. As the trapezoid representation is rather large, applications are given the choice to use triangles specified as three vertices in any of several representations (disjoint, strips and fans). Trapezoids are used in preference to triangles in 2D graphics because of the number of rectilinear elements in a typical 2D user interface. Rendering triangles would introduce seams across every rectangular object.

The following discussion will use the term polygon, but that should be understood to mean only these limited trapezoids and triangles.

4.1 Polygons Rendering Requirements

To permit hardware acceleration, our system provides for both imprecise and precise rendering of polygons, selectable by the application. Imprecise polygons are constrained to ensure reasonable results with tessellated objects:

- Abutting edges must match precisely. When specifying two polygons abutting along a common edge, if that edge is specified with the same coordinates in each polygon then the sum of alpha values for pixels inside the union of the two polygons must be precisely one.
- Translationally invariant. The pixelization of the polygon must be the same when the polygon is translated by any whole number of pixels in any direction.
- Sharp edges are honored. When the polygon is rasterized with Sharp edges, the implicit alpha mask will contain only 1 or 0 for each pixel.
- Order independent. Two identical polygons specified with vertices in different orders must generate identical results.

Simple box-filtered (super) sampling hardware should match these constraints. The specification for Precise polygons also matches these constraints.

4.2 Precise Polygons

Precise polygons are specified so that different implementations can produce identical results. While matching at this level is rarely necessary in interactive environments, system which blend images from several different rasterizers do need this level of conformance. However, the actual pixelization is often not important, only that every implementation agree. With that in mind, a relatively simple super-sampling specification was developed that could be easily implemented in software.

When computing coverage incrementally polygons, the sum-to-one requirement means that each step in the alpha value must be covered by precisely one polygon. As the number of polygons increases, the area of each decreases and eventually represents a single point in the pixel. Hence, any sum-to-one incrementally computed rendering algorithm is equivalent to some kind of point sampling.

To provide maximum resolution, we set the number of sample points equal to the maximum alpha value. For depth $2n$, we create a rectangular grid of points 2^n+1 by 2^n-1, which is square enough to avoid significant variation in appearance on rotation. Figure 8 shows the resulting grid at depth 4. For depth $2n+1$, we simply line the points up along the midline of the pixel. This would generate significant errors were it ever used in practice for depths > 1, but alpha is generally expressed in 1, 4, 8 or 16 bits.

The current algorithm computes Bresenham coefficients for each non-horizontal edge and then walks them simultaneously stepping down to each sample row and counting the number of samples present in each pixel. This algorithm scales as the number of rows instead of the total number of samples which is somewhat more efficient than a straight forward super sampling technique.

We experimented with one point sampling technique which would automatically adjust the point positions to reduce the error between the point coverage and the area coverage but discovered that the local nature of incremental computation couldn't compensate for the global nature of this automatic placement, so many errors resulted. Of course, the maximum error in any point sampling technique is unity as the area of the points is zero, so additional complexity to reduce errors in some cases wasn't really beneficial.

4.3 Related Work

Edwin Catmull[Catmull 1978] coined the term "area sampling" to describe the box filter used in our system and many others. His area sampling was analytic instead of super sampled. As discussed above, true area sampling is not consistent with our incremental rendering architecture.

Several works use precomputed filter values with lookup tables indexed by the location of the line intersection along the pixel edge [Abram and Westover 1985; Carpenter 1984; Fiume et al. 1983; Fiume 1991; Greene 1996]. Such lookup tables permit the use of arbitrary filter constructions, but that is incompatible with our system's requirement for precise sum-to-one semantics. Because these systems must subtract alpha values, they need to be carefully constructed to avoid negative results from that subtraction and to ensure precise sum-to-one semantics. Our system provides higher alpha resolution with some performance impact, a reasonable tradeoff given the image precision goals of many 2D graphics applications.

Other systems perform antialiasing by computing the distance from the pixel to the spline. This approach also allows the use of arbitrary filters. Some of these[Klassen 1991; Lien et al. 1987] are limited to single-pixel wide curves. Others[Fabris and Forrest 1997] are much more expensive because many pixels must be measured against many spline sample points, instead of directly computing the affected pixels and their coverage values.

5 Compositing

Our system uses Porter/Duff image compositing as the fundamental pixel manipulation primitives in our system, just as in OpenGL, Quartz, PDF 1.4 and other current graphics systems. This provides a complete set of operations as well as access to hardware acceleration on current graphics chipsets designed for these other systems.

We start with basic Porter/Duff compositing and extend it in a novel way to support the common rendering operations of tessellation and repeated pen application.

5.1 General Compositing Function

The Plan 9 window system[Pike 2000] uses a single image compositing function as the foundation for the whole graphics system:

$$dest = (source\ \mathbf{IN}\ mask)\ \mathbf{OVER}\ dest$$

Providing different values for *source* and *mask* permits the complete expression of most graphics operations. As Jim Blinn says[Blinn 1998], the **OVER** operator is very nearly the only one ever needed. An exception to this is where the source must replace the destination irrespective of alpha. Plan 9 has a special case for this used, for example, when scrolling windows.

Extending this compositing function to permit any of the compositing operators to be used in place of **OVER** completes the expression of these remaining operations, eliminating the special case:

$$dest = (source\ \mathbf{IN}\ mask)\ \mathbf{OP}\ dest$$

In our system, the *source* and *mask* operands may be tiled or projectively transformed, or be synthesized from a gradient specification. The *mask* operand may also be synthesized from the union of a collection of polygons; the various masks are added together and applied in a single operation.

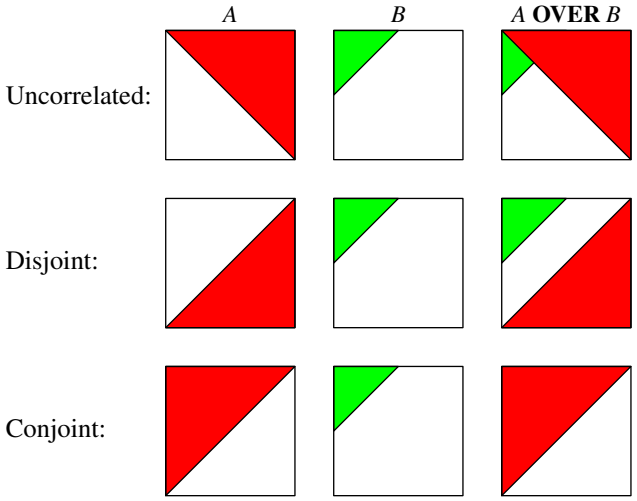


Figure 9: Behavior of **OVER** operators

The result is a complete graphics system which can always be reduced to it's component parts; there are no opaque high level operations. An application interested in different polygon rendering techniques can compute the resulting alpha mask and use that with the general compositing formula; the alpha computation may not be accelerated with hardware, but the compositing stage can be. This is relatively unique among other 2D window systems graphics models; in most other systems, applications often resort to low level pixel arithmetic as a part of implementing unsupported higher level operations.

5.2 Sub-pixel Geometry in Image Compositing

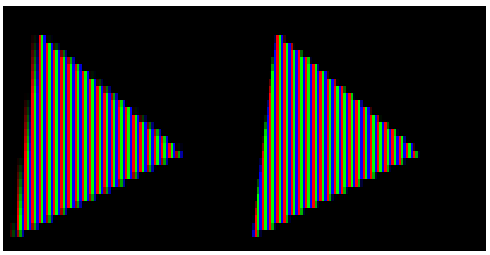
Porter and Duff built an algebraic structure to codify image compositing techniques of their day. Most modern color graphics systems inherit partial pixel coverage and translucency computations from that definition. The coverage formulae are defined assuming that the geometry of the various objects are unrelated. Porter and Duff state if the object geometries were known composite image colors could be more accurately computed.

Using the actual object geometries to compute each pixel is unwieldy in practice, but in at least two common cases the general character of the geometry is inherent in the algorithms used. The first is when tessellating a complex object into simple shapes; in this case, the simple shapes are non-overlapping and we call the subpixel geometries *Disjoint*. The second is when repeatedly applying a pen or brush, in which case the shapes are overlapping; these subpixel geometries are called *Conjoint*. When the subpixel geometry is unknown, we call it *Uncorrelated*. This taxonomy could be extended to other sub-pixel geometries.

In the Disjoint case, as long as the pixel is not completely covered, each operation treats the pixel as transparent as the assumption is new coverage occurs only in areas not yet covered; only excess coverage blends with the existing pixel. In the Conjoint case, as long as new coverage is less than existing coverage, each pixel is treated as opaque as the assumption is that new coverage always overlays existing coverage and only excess coverage is applied to translucent areas. Figure 9 shows these relationships.

Limiting the analysis to just the **OVER** operator for brevity, we can now develop functions that express this relationship. Given two images, A and B, the uncorrelated operator is:

$$A \textbf{ OVER } B = A + (1 - \alpha_A)B$$



(a) Per Pixel (b) Per Component

Figure 10: Comparing pixel-level and component-level compositing

The disjoint operator depends on whether B covers more of the pixel than is transparent in A. Over coverage reduces the contribution of B by the ratio of the remaining area to the coverage of B:

$$A \textbf{ OVER}_{disjoint} B = A + \begin{cases} B & \text{if } \alpha_B \leq 1 - \alpha_A \\ \frac{1-\alpha_A}{\alpha_B} B & \text{otherwise} \end{cases}$$

This operator is equivalent to the OpenGL compositing operation using FUNC_ADD, SRC_ALPHA_SATURATE, ONE which is commonly used for the same purpose.

The conjoint operator depends on whether B covers more of the pixel than A. Where it does, the area covered by B and not by A is the only contributing part.

$$A \textbf{ OVER}_{conjoint} B = A + \begin{cases} 0 & \text{if } \alpha_B \leq \alpha_A \\ \left(1 - \frac{\alpha_A}{\alpha_B}\right) B & \text{otherwise} \end{cases}$$

Similar equations can be constructed for the other compositing operators.

5.3 Per-Component Compositing

The prevalence of LCD displays in 2D environments suggests that the graphics system should permit optimizations specific to such devices. LCD screens differ from CRTs in many ways. One important difference is that each color component (red, green or blue) on the screen is visibly discrete and in a known geometric relation to other components.

Our system exposes this underlying hardware capability by permitting the *mask* operand to provide separate alpha values for each component. As deployed today, this is done for text to good effect. Extensions to the tessellation operation to exploit this capability are expected to be provided in a future version of the system. Figure 10(a) shows a magnified view of an LCD screen with a triangle rendered with per-pixel compositing while figure 10(b) shows a triangle rendered with per-component compositing.

Note that on an actual screen, interactions of the object geometry with the component position requires the use of some filtering to reduce visible color fringing around the objects. For the current text implementation, that filtering is done as the glyphs are rasterized. For the future geometric rendering mechanism, a post-rendering filter step will be needed for good looking results.

J. Platt et al [Betrisey et al. 2000] describe an algorithm for computing component intensities for a particular image with respect to the human visual system. Their work doesn't describe how the resulting intensities should be used within a rendering system. Current released systems using this particular technique are limited to pre-computing colored versions of glyphs as they have no support for per-component compositing as described here.

6 Related Systems

This new work builds a complete 2D graphics architecture, comparable systems are largely those present in existing 2D window systems and other 2D graphics applications. Our system is unique in providing a carefully layered architecture which permits the integration of application-rendered and system-rendered images for the support of applications with demanding rendering requirements while offering hardware accelerated performance. The system includes a novel algorithm for efficient tesselation of stroked splines, provides disjoint and conjoint extensions to the compositing operators and the exposes per-component compositing capabilities for improving image presentation on LCD monitors.

The Quartz[Apple Computer Corp. 2002] 2D graphics system is the most comparable fielded system to this work. Differences are both architectural and functional. Both Quartz and this work expose Bézier splines as the fundamental geometric primitive and use Porter/Duff image compositing to manipulate pixels. Our system splits the graphics layer underneath the tessellation function for rendering to various output devices and also provides applications the ability to supplant the supplied tessellation functionality with purpose-built code while still permitting a reasonable degree of hardware acceleration. The Quartz API can be used to generate PDF output, but all of the tessellation and rendering is performed by the external PDF engine and so it cannot provide assurances of identical results. Quartz also lacks the new compositing operators for disjoint and conjoint sub-pixel geometry as well as per-component compositing operations.

The Microsoft Windows GDI[Yuan 2001] provides output device independence, but the bitblt rendering model is essentially identical to that found in the Xerox Alto[Thacker et al. 1982] which has been supplanted for color graphics by image compositing. GDI also provides a wealth of simple geometric figures including lines and arcs, but not including Bézier splines or paths. Like Quartz, GDI does not provide any access to the underlying mechanism for drawing tessellated figures, so alternative geometric objects cannot be accelerated.

The X Window System is largely equivalent to GDI, except that it provides only display services which eliminates the ability to preserve or present information in other contexts. The Xprint extension provides limited support for printing, but there is still no way to generate images within the application.

The OpenGL Graphics System provides similar capabilities for presenting geometric objects on the screen, but it's focus on interactive 3D applications produces visible artifacts in static images; tessellation is limited to triangles with floating point coordinates and so longer edges produce visible knots. OpenGL also has no support for alternative rendering targets. Given the prevalence of OpenGL in applications and as a target for hardware vendors, it did serve as a model for hardware interaction so that hardware designed for OpenGL could also be used to accelerate our system.

Page description languages like PostScript[Adobe Systems Inc. 1985], PDF 1.4[Adobe Systems Inc. 2001] and SVG are all consumers of lower level geometric rendering mechanisms and are related to our system, although not comparable. The PostScript specification requires that strokes containing curves are implicitly converted to sequences of lines before being rendered. This produces visible artifacts in the resulting image as demonstrated in figure 7.

7 Implementation

This system has been implemented within the X Window System. Tessellation is performed by an application library while rendering and compositing are performed by an X extension. Hardware acceleration of the image compositing operators is implemented. No hardware acceleration has yet been implemented for the trapezoid rendering step. The application library will be extended to include support for PostScript or PDF output for printers as well as to perform the rendering and compositing steps to local memory.

While the implementation is not yet mature, it has already proven valuable in several applications and to be a significant advance in 2D drawing.

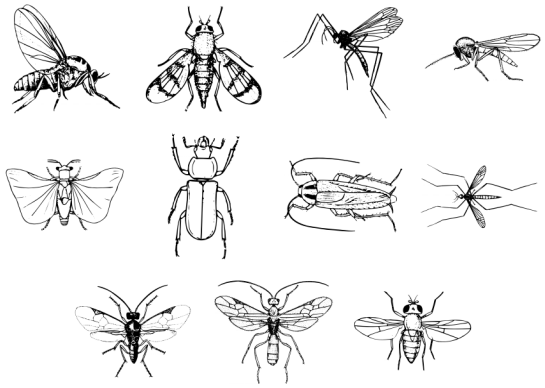


Figure 11: Sample rendering of SVG (artwork courtesy of Lauri Järvlepp)

References

ABRAM, G., AND WESTOVER, L. 1985. Efficient alias-free rendering using bit-masks and look-up tables. In *Proceedings of the 12th annual conference on Computer graphics and interactive techniques*, ACM Press, 53–59.

ADOBE SYSTEMS INC. 1985. *PostScript Language Reference Manual*. Addison Wesley.

ADOBE SYSTEMS INC. 2001. *PDF Reference: Version 1.4*, 3rd ed. Addison-Wesley.

APPLE COMPUTER CORP., 2002. Inside mac os x: Drawing with quartz 2d. `http://developer.apple.com/techpubs/macosx/CoreTechnologies/graphics/Quartz2D/drawingwithquartz2d/drawingwithquartz.pdf`.

BARGEN, B., AND DONNELLY, T. P. 1998. *Inside DirectX*. Microsoft Press.

BETRISEY, C., BLINN, J. F., DRESEVIC, B., HILL, B., HITCHCOCK, G., KEELY, B., MITCHELL, D. P., PLATT, J. C., AND WHITTED, T. 2000. Displaced filtering for patterned displays. In *Society for Information Display Symposium*, Society for Information Display, 296–299.

BLINN, J. 1998. *Jim Blinn's Corner: Dirty Pixels*. Morgan Kaufmann.

CARPENTER, L. 1984. The a-buffer, an antialiased hidden surface method. In *Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, 103–108.

CATMULL, E. 1978. A hidden-surface algorithm with anti-aliasing. In *Proceedings of the 5th annual conference on Computer graphics and interactive techniques*, ACM Press, 6–11.

FABRIS, A. E., AND FORREST, A. R. 1997. Antialiasing of curves by discrete pre-filtering. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, ACM Press/Addison-Wesley Publishing Co., 317–326.

FARIN, G. 1990. *Curves and Surfaces for Computer Aided Geometric Design: A Practical Guide*, second ed. Academic Press.

FIUME, E., FOURNIER, A., AND RUDOLPH, L. 1983. A parallel scan conversion algorithm with anti-aliasing for a general-purpose ultracomputer. In *Proceedings of the 10th annual conference on Computer graphics and interactive techniques*, 141–150.

FIUME, E. 1991. Coverage masks and convolution tables for fast area sampling. *CVGIP: Graphical Models and Image Processing 53*, 1, 25–30.

GHOSTSCRIPT, 2003. `http://www.ghostscript.com`, acessed: Jan 21 9:31 UTC 2003.

GREENE, N. 1996. Hierarchical polygon tiling with coverage masks. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, ACM Press, 65–74.

GUIBAS, L., RAMSHAW, L., AND STOLFI, J. 1983. A kinetic framework for computational geometry. In *Proceedings of the IEEE 1983 24th Annual Symposium on the Foundations of Computer Science*, IEEE Computer Society Press, 100–111.

HOBBY, J. D. 1989. Rasterizing curves of constant width. *Journal of the ACM (JACM) 36*, 2, 209–229.

KLASSEN, R. V. 1991. Drawing antialiased cubic spline curves. *ACM Transactions on Graphics (TOG) 10*, 1, 92–108.

KNUTH, D. E. 1986. *METAFONT: The Program*, vol. D of *Computers & Typesetting*. Addison Wesley.

LIEN, S.-L., SHANTZ, M., AND PRATT, V. 1987. Adaptive forward differencing for rendering curves and surfaces. In *Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, ACM Press, 111–118.

NELSON, G. 1991. *Trestle Window System Tutorial*. Prentice Hall, ch. 7.

PENNINGTON, H. 1999. *GTK+/Gnome Application Development*. New Riders Publishing.

PIKE, R. 2000. *draw - screen graphics*. Bell Laboratories. Plan 9 Manual Page Entry.

PORTER, T., AND DUFF, T. 1984. Compositing Digital Images. *Computer Graphics 18*, 3 (July), 253–259.

SCHEIFLER, R. W., AND GETTYS, J. 1992. *X Window System*, third ed. Digital Press.

SEGAL, M., AKELEY, K., AND (ED), J. L. 1999. *The OpenGL Graphics System: A Specification*. SGI.

SYMBOLICS. 1988. Programming the user interface — dictionary, revised for genera 7.2. Technical Report Book 7B, Pub. No. 99 90 58, Symbolics, Inc., Cambridge, MA, Feb.

THACKER, C. P., ET AL. 1982. Alto: A personal computer. In *Computer Structures: Principles and Examples*, S. et al., Ed. McGraw-Hill, ch. 33. Also CSL-79-11, Xerox Palo Alto Research Center (1979).

YUAN, F. 2001. *Windows Graphics Programming: Win32 GDI and DirectDraw*. Prentice Hall.