# Non-Invasive, Interactive, Stylized Rendering

Alex Mohr *     Michael Gleicher *

University of Wisconsin, Madison

## Abstract

In this paper, we show how many interactive 3D applications' visual styles can be changed to new, different, and interesting visual styles non-invasively. Our method lets a single stylized renderer be used with many applications. We implement this by intercepting the OpenGL graphics library and changing the drawing calls. Even though OpenGL only receives low-level information from an application, computation on this data and assumptions about the application can give us enough information to develop stylized renderers.

**Keywords:** Non-photorealistic rendering, Real-time, Stylized, Interactive, 3D

## 1 Introduction

Many different visual styles are now possible for interactive 3D applications. To date, most applications' visual styles have been tightly coupled to the applications themselves. This makes prototyping new visual styles and experimenting with different visual styles difficult. In this paper, we show that we can change many interactive 3D applications' visual styles non-invasively. This effectively divorces an application's visual style from the application itself. Using this method, we develop interesting stylized renderers.

Visual style is a key element of a graphical application and often dictates much of the design and development of an application. For example, computer video games often seek to immerse their players in rich, believable environments. Accomplishing this requires the visual style of the game to reflect the environment. Significant amounts of design and artwork are often necessary and in many cases the game's code and data are specifically designed to achieve a certain style. In id Software's Quake III Arena, for example, many texture maps and programmable shaders create a richly detailed, futuristic, techno-gothic world. While typically less extreme, other kinds of applications also exhibit carefully designed styles.

Because visual style is such an intrinsic component of an application, it must typically be designed in from the core. Extensive changes are required to explore alternate styles, making exploration of new application/style pairings extremely difficult.

Our goal is to explore varying visual styles with existing applications. This requires us to alter the visual style of applications non-invasively, as we rarely have access to the source code of existing software. To achieve these alterations, we are limited to intercepting the output of applications at a common level: calls to the graphics library. We must "hijack" a program's drawing calls, and reinterpret them to create alternate styles.

This paper describes our system and methods for non-invasive visual style changes. Standard operating system mechanisms allow us to intercept calls to a graphics library. The challenge, however, is that the only information we receive from the application is low-level drawing commands and primitives. This lack of high-level information precludes many current stylized rendering techniques. The contribution of this paper is to show that despite only getting low-level information, interesting stylized renderers are possible, enabling non-invasive style alterations.

We first present a brief overview of the mechanics involved in intercepting the OpenGL graphics library and providing for pluggable renderers. We then show some alterations that illustrate how simple transformation techniques lead to relatively uninteresting visual styles. To create more interesting styles, we devise methods to gather more information. We demonstrate a set of renderers of increasing quality, all operating non-invasively. The most sophisticated are capable of providing plausible artistic styles. While these methods may not produce imagery to rival state of the art non-photorealistic rendering systems, we can dynamically apply them to applications without access to the source code.

## 2 Related Work

All programs have a visual style, to some degree. In the early history of computer graphics, visual style was dominated by either performance limitations or attempts to achieve photorealism. A movement within the community to display images in non-photorealistic, artistic styles began to gain momentum with Haeberli's impressionist painting programs [6] and Saito and Takahashi's Comprehensible Renderering paper [16]. These were soon followed by commercial tools such as Fractal Painter [3].

Since then, there has been an explosion of new styles presented at SIGGRAPH and other venues. Just a few examples include impressionist painting [12], technical illustration [5], pen and ink [17], engraving [14], watercolor, and even the style of Dr. Seuss [9].

As stylized renderers improved in performance due to hardware advances and clever algorithms, interactive stylized rendering became practical. An early example was demonstrated in the Sketch system [18] that employed a sketchy look to augment their sketching interaction technique. Many other techniques have been employed to acheive interactive rates. Hertzmann and Zorin demonstrate a real-time method to find all silhouettes in a model [7]. Lake, Marshall et al. [10] also present techniques for interactive 3D stylized rendering. The work of Markosian, Kowalski, et al. [13] emphasized speed for interactive non-photorealistic rendering. Several other systems also demonstrate interactive non-photorealistic rendering techniques. The work of Kaplan et al. [8] describes a method

* {amohr,gleicher}@cs.wisc.edu, http://www.cs.wisc.edu/graphics

for interactively rendering non-photorealistic subdivision surfaces.

In all of these cases, a new rendering style was explored in the context of a single, specific application. At least one commercial application, ViewPoint corporation's LiveArt 98 [4], has allowed users to switch among multiple drawing styles, and even add new styles to the application. NPRQuake [1] also demnostrates pluggable renderers for stylized rendering. However, these applications were explicitly designed for internal switchable renderers—they cannot be applied to other applications.

The key challenge of non-invasive stylization—creating styles with limited information—was also addressed by Litwinowicz [12]. His work creates impressionist styles from unstructured video but relies on computer vision techniques to gain information required to drive image-based effects. Unfortunately, such techniques are too computationally expensive for interactive applications.

# 3 Intercepting OpenGL

To change applications' visual styles non-invasively, we intercept the OpenGL graphics library. We chose OpenGL because we want our technique to apply to a broad class of applications and OpenGL is used by a large number of interactive 3D applications. To intercept OpenGL, we rely on standard operating system services.

Most modern operating systems, including Windows and today's Unix systems, rely on dynamic library loading to allow system libraries to be updated independently of applications. This provides our mechanism to intercept applications' calls to OpenGL: we replace the OpenGL system library with one of our own. While some newer systems, such as Windows 2000 attempt to avoid such security loopholes, under many systems (such as Windows NT) this replacement simply requires naming our library the same as the expected system library, and insuring that our library precedes the real library in the search path.

There are two engineering considerations in performing library interception. First, our library must fully implement the same interface as the system library. Second, since our library needs to call the real system library, we need to load the real library and provide a name mapping mechanism.

We have created our own Windows OpenGL shared library (`opengl32.dll`) that can dynamically load rendering algorithms to specify the behavior of drawing commands. Because we have faithfully recreated the documented OpenGL interface, applications are unaware that such alterations occur. We note that the functionality we replace the drawing calls with is not limited to drawing. For example, we have created a plug-in renderer that logs graphics calls to a file, allowing us to capture 3D geometry.

Our pluggable rendering architecture handles many of the engineering concerns and low-level data processing for plug-in renderers. It handles intercepting OpenGL commands, distilling the myriad of library calls to a smaller, more manageable set, managing OpenGL state, providing user interfaces for dynamically switching and configuring renderers, providing plug-in renderers access to the *real* OpenGL, and perfoming common functions such as the data buffering and coordinate system conversions discussed in the following sections.

# 4 Plug-in Renderers

A plug-in renderer has access to all of the infromation that a program sends to the graphics library, but it can *only* receive this information. We get no information about the meaning or intent of drawing operations, only a stream of commands.

The simplest transformations just change individual calls or insert additional commands into the drawing stream. Color masking or disabling texture mapping are examples. While such alterations clearly change the look of an application, they are far from producing compelling visual styles.

More complex effects require the renderer to examine more than one library call. Buffering a number of calls allows for both transformations on groups of commands as well as analysis of the OpenGL command stream to get more information.

## 4.1 Simple Techniques

The simplest buffering scheme, implemented in our first prototype, buffers a single geometric primitive at a time. In OpenGL, this requires recording each vertex in a polygon as it is specified along with its associated data. This scheme permits each geometric primitive to be altered but does not allow for effects that require information about groups of primitives.

This section presents three of our initial pluggable renderers. They produce some interesting results but serve to show that we are limited without high-level information.

### 4.1.1 Depth-cued Wireframe

The depth-cued wireframe renderer, shown in Figure 4 on a student's introductory graphics course assignment, illustrates a simple visual style transformation that can be achieved with very little information from an application.

Mimicing an old calligraphic display is certainly not a traditional artistic style. However, a wireframe renderer has utility: it is very good for seeing a scene's underlying geometry. For example, the tessellation of the trees in Figure 4 is apparent with the wireframe renderer. Also, a drawing error in this student's project is visible—the black line intended to outline the train cars is too high. This is normally invisible, since the background is also black. Our wireframe renderer exposes this flaw.

The only information this renderer requires is the location of each line and triangle's vertices. Only lines and triangles are necessary because our system can convert quadrilaterals and arbitrary polygons to triangles. The operation of this pluggable renderer is simple. When it receives a primitive, if it is a line, it is drawn. If it is a triangle, its edges are drawn.

### 4.1.2 Simple Sketch

Our jittered line renderer tries to mimic a gestural pencil sketch style. In this style, artists typically trace the same contour several times with fast strokes. The fast strokes lead to inaccuracies in line placement, which gives some of the appeal to gestural drawings. In Figure 3, we see the jittered line renderer running on id Software's popular game, Quake III Arena. This renderer requires only a little more information than the wireframe renderer, but provides a crude approximation to the artistic style.

Our jittered lines renderer behaves as the wireframe renderer, except that it draws underlying triangles, jitters the locations of the geometry lines, and draws each line several times. In order for this renderer to work, the jittering must be done in screen space. If lines were jittered by constant amounts in eye or world space, lines close to the near clip plane would have wildly changing locations while objects far from the near clip plane would experience almost no jitter.

To create screen space jitter, a system must record more information. The camera transformation must be known such that the screen space position of each point can be determined. Our rendering framework was augmented to record the transformation matrices and compute the screen space positions of all vertices in a manner identical to OpenGL's pipeline. First an incoming vertex in object coordinates is transformed by the current modelview matrix, then transformed by the projection matrix, and finally normalized by its homogeneous coordinate. It is important to clip geometry to the view volume or this computation can give incorrect results.

The jittered-line renderer creates a more interesting visual effect by buffering and computing more information than the simpler wireframe renderer. While its results would not be mistaken for a hand-drawn sketch, or even a common non-photorealistic renderer, it did work with a wide variety of applications and provide inspiration for the techniques that follow.

#### 4.1.3 Simple Colored Pencil

A simple extension to the jittered line renderer attempted to simulate a sketchy colored pencil style. By making some assumptions about the scene geometry and computing more information, this more interesting renderer is possible. An illustration of this renderer is shown in Figure 5.

The simple colored pencil renderer assumes that it is drawing a 3D scene that makes sense to be lit from above. The renderer computes a false face normal for each triangle by calculating the cross product of two vectors in the directions of two different sides of the triangle. If the normal is pointing away from the viewer, the normal is flipped to point toward the viewer. This must be done because there is no reliable way to distinguish front faces from back faces. Once a face normal $\vec{n}$ is computed, the renderer simply scales each vertex's color values by $(\vec{n}_y + 1)/2$. This simulates a directional light from the top of the viewport. The resulting colors are quantized to simulate a restricted color palette. Finally, lines are drawn from near each vertex to random points jittered along the opposite side of the triangle. Texture is used to simulate paper grain.

This renderer is significantly more interesting than the jittered lines renderer, but it requries no new information. This demonstrates that making assumptions about applications can give enough information to implement interesting rendering styles. This renderer is indeed the most interesting so far, but again, falls far short of both real artistic styles and existing non-photorealistic art renderers.

### 4.2 Limitations of These Techniques

The problem with these simple renderers most obvious in the static frames is that they expose the underlying scene geometry. For example, the lines drawn by the jittered line renderer in Figure 3 are not chosen because they most effectively convey overall shape. Instead, they are the boundaries of the geometric primitives—artifacts due to the low-level nature of the information we get.

Some of the problem stems from insufficient buffering: because the renderer only treats individual primitives, it must draw each one independently. It cannot choose to discard an edge adjacent to another primitive because it does not know the other primitives exist. This leads to the excess lines that show the underlying mesh structure (clearly visible in Figures 3 and 5). It also leaves us without a mechanism to avoid drawing discontinuities across primitive groups. In the next section, we will demonstrate how additional buffering can alleviate these issues.

A deeper issue is that OpenGL only gets low-level information about primitives in a scene and we are therefore fundamentally limited in our ability to correctly stylize applications. The key issue is *intent*. With no notion of the intent of drawing commands, we are forced to make assumptions about the underlying application. For example, there is no way to know whether the triangles in a scene comprise a large mesh, twenty small meshes, or faces of buttons in a user interface. The next section will explain how we have dealt with this limitation, and present two renderers of much higher quality than those already shown.

## 5 Getting More Information

Because we had access to little direct information, we redesigned our system with a new execution model. The second version of our system intercepts all OpenGL calls and buffers and performs computations on the incoming data for an entire frame at a time. The plug-in renderer is only executed when a frame is completed. A plug-in renderer requests the data it requires for the whole scene at once, processes it, and emits OpenGL calls to draw a frame. Buffering all the data for whole frames allows our system to do computations on entire scenes and gather more meaningful information than just the data associated with each primitive.

Upon intercepting an OpenGL command, our system stores the data in a hash table. Hashing is done so we can gather higher level data about frames being generated. To use any techniques that give us meaningful global information, we need to detect repeated data passed to OpenGL. For example, to detect silhouette edges or find distinct objects, we must know mesh connectivity. Often, OpenGL programs draw connected meshes as *disjoint* primitives. For instance, cubes are frequently drawn as six distinct quadrilaterals. Thus, if we take each location to be distinct, we will not accurately reconstruct the connected cube.

Our system buffers data until a command that requires a complete image arrives. The most common command in this class is `SwapBuffers()` for a double buffered application, but other examples include commands that read the framebuffer into a texture, or copy parts of the framebuffer onto itself. Since an image is required at this point, our system calls the plug-in renderer to generate the image. Plug-in renderers are free to examine all scene data and render a frame. After this, the data buffers are flushed the process begins again.

To reduce drawing discontinuities in our stylized renderers (see the tiled plane in Figure 5), we compute and record per-vertex normals in eye space. We assume that vertex normals will vary consistently across meshes, so if we write our renderer to draw triangles that vary smoothly with vertex normals, we will reduce drawing discontinuity across objects.

To address the visible geometry problem, we implemented a naïve silhouette edge detector. We simply keep a hash table of winged-edges that the application has drawn. As primitives are drawn, we continually update a list of silhouette edges in this winged-edge list. When a frame is complete, we have a complete list of all the silhouette edges. We implemented this brute-force method because it was easy to do in our system. A technique like those presented in [7] or [2] could be more efficient.

### 5.1 Colored Pencil and Pencil Sketch

Our second colored pencil renderer, shown in Figure 2 running on a research animation system, takes advantage of the this information. A thick dark line is drawn for every silhouette edge in the scene, which suggests a hand-drawn look and gives a good indication of shape and contour. This helps make form apparent without showing objects' internal structures.

The triangle rendering routine has been revised to improve triangle-to-triangle continuity. This is done by taking the triangle's surface normal as the average of every vertex normal in eye space, and crossing it with a direction vector that represents an incoming directional light. The result is used as a drawing direction for the triangle. The triangle is then shaded with lines that are parallel to this drawing direction, but slightly jittered. Assuming the geometry has vertex normals, this drawing direction will change slowly and smoothly across a smooth surface. Even though the ends of the shading lines do not match up from triangle to triangle, the shading trend is consistent across large areas. Compare the checkered plane in Figure 5 to that in Figure 1. While the individual triangles are still visible upon close inspection, the quality is much improved.

The pencil sketch renderer shown in Figure 1 uses techniques similar to the colored pencil renderer. However, the pencil sketch renderer takes advantage of all the vertex normals in eye space. The more a vertex normal becomes more aligned with the light direction, shading lines not only become lighter, but also become more sparse.

Although the images from these renderers in Figures 1 and 2 do not match the quality of many application-specific renderers due to lack of high-level information, we consider them of sufficient quality to be termed "artistic" styles. Some of the internal structure in Figure 2 is still visible, but is much improved from Figure 5. Similarly, while the renderer shown in Figure 1 is not perfect, it can produce plausible pencil-sketch images.

## 6 Conclusion

We have shown that applications' visual styles can be changed to different and interesting styles without modifying the applications themselves. Instead, we intercept graphics library calls and reinterpret the stream of commands to create images in the desired style.

A unique challenge of this approach is that the graphics library gets no high-level information from applications. Renderers must rely on buffering, analysis, and assumption to create the required information. This last category leads to a number of failure modes. If an application uses OpenGL to draw user interface elements like buttons and scrollbars or to display text to a user, stylizing these elements may render parts of the application unusable. Another example is evident in the work of [15]. Their multipass shading technique treats OpenGL as a general SIMD computer. There are also applications that use OpenGL for things other than graphic display of images such as robot path planning [11]. Our technique could cause such an application to fail.

At present, the image quality of our results falls short of state of the art non-photorealistic renderers. This is not suprising: we have more limited information, are operating with more strict performance goals, and have not adequately explored different techniques. In time, we hope to develop improved analysis techniques and rendering algorithms that will allow us to approach current non-photorealistic renderers, but to provide these results across a much broader class of applications.

### Acknowledgments

Andrew Gardner, Erik Bakke, Steve Dutcher, and Christopher Herrman helped build NPRQuake, which was much of the impetus for this work.

Dr. John Hughes provided invaluable suggestions and advice for revision. Rob Iverson and Min Zhong assisted with paper production. The baby model depicted in Figures 1, 5, and 6 was provided by Hou Soo Ming. Figure 3 contains a scene from the freely available Quake III Arena Demo, Copyright id Software, 2000. The general model shown in Figure 2 is freely downloadable courtesy ViewPoint Corp.

This research is supported by an NSF Career Award "Motion Transformations for Computer Animation" CCR-9984506, support from Microsoft, equipment donations from IBM and Intel, and software donations from Microsoft, Intel, Alias/Wavefront, SoftImage, Autodesk and Pixar.

## References

[1] Erik Bakke, Steven Dutcher, Christopher Herrman, Andrew Gardner, and Alex Mohr. NPRQuake. Computer Science Course Project, 2000.

[2] John W. Buchanan and Mario C. Sousa. The edge buffer: A data structure for easy silhouette rendering. *NPAR 2000 : First International Symposium on Non Photorealistic Animation and Rendering*.

[3] Fractal Designs Corp. Fractal painter. Computer Software, 1991-1997.

[4] ViewPoint Corp. Liveart 98. Computer Software, 1998.

[5] Bruce Gooch, Peter-Pike J. Sloan, Amy Gooch, Peter Shirley, and Rich Riesenfeld. Interactive technical illustration. *1999 ACM Symposium on Interactive 3D Graphics*.

[6] Paul E. Haeberli. Paint by numbers: Abstract image representations. *Computer Graphics (Proceedings of SIGGRAPH 90)*, 24(4).

[7] Aaron Hertzmann and Denis Zorin. Illustrating smooth surfaces. *Proceedings of SIGGRAPH 2000*.

[8] Matthew Kaplan, Bruce Gooch, and Elaine Cohen. Interactive artistic rendering. *NPAR 2000 : First International Symposium on Non Photorealistic Animation and Rendering*.

[9] Michael A. Kowalski, Lee Markosian, J. D. Northrup, Lubomir Bourdev, Ronen Barzel, Loring S. Holden, and John Hughes. Art-based rendering of fur, grass, and trees. *Proceedings of SIGGRAPH 99*.

[10] Adam Lake, Carl Marshall, Mark Harris, and Marc Blackstein. Stylized rendering techniques for scalable real-time 3d animation. *NPAR 2000 : First International Symposium on Non Photorealistic Animation and Rendering*.

[11] Jed Lengyel, Mark Reichert, Bruce R. Donald, and Donald P. Greenberg. Real-time robot motion planning using rasterizing computer graphics hardware. *Computer Graphics (Proceedings of SIGGRAPH 90)*, 24(4).

[12] Peter Litwinowicz. Processing images and video for an impressionist effect. *Proceedings of SIGGRAPH 97*.

[13] Lee Markosian, Michael A. Kowalski, Samuel J. Trychin, Lubomir D. Bourdev, Daniel Goldstein, and John F. Hughes. Real-time nonphotorealistic rendering. *Proceedings of SIGGRAPH 97*.

[14] Victor Ostromoukhov. Digital facial engraving. *Proceedings of SIGGRAPH 99*.

[15] Mark S. Peercy, Marc Olano, John Airey, and P. Jeffrey Ungar. Interactive multi-pass programmable shading. *Proceedings of SIGGRAPH 2000*.

[16] Takafumi Saito and Tokiichiro Takahashi. Comprehensible rendering of 3-d shapes. *Computer Graphics (Proceedings of SIGGRAPH 90)*, 24(4).

[17] Georges Winkenbach and David H. Salesin. Rendering parametric surfaces in pen and ink. *Proceedings of SIGGRAPH 96*.

[18] Robert C. Zeleznik, Kenneth P. Herndon, and John F. Hughes. Sketch: An interface for sketching 3d scenes. *Proceedings of SIGGRAPH 96*.