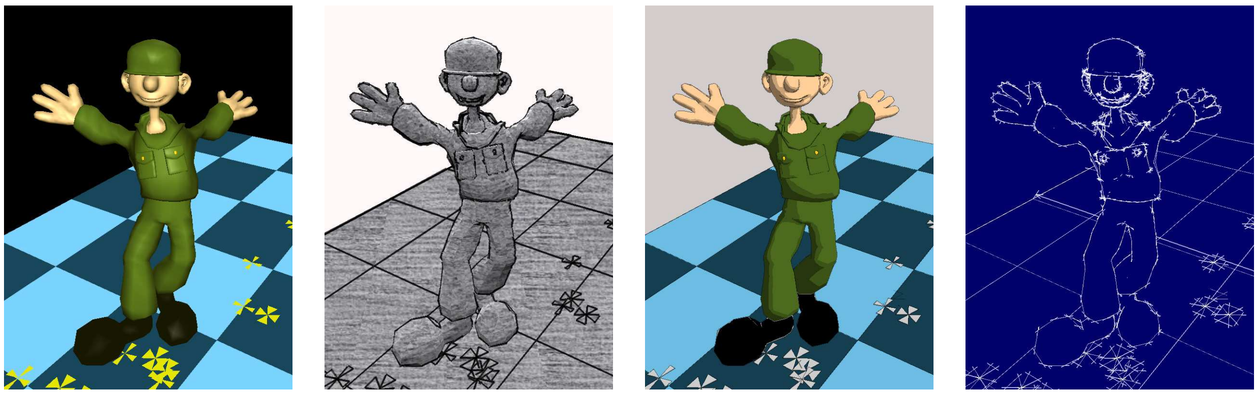
# HijackGL: Reconstructing from Streams for Stylized Rendering

Alex Mohr [*] Michael Gleicher [*]

University of Wisconsin, Madison



## Abstract

This work shows that intercepting a low-level graphics library command stream and reconstructing a declarative representation is practical and useful, especially for exploring new rendering styles. We show not only how the basic mechanics of intercepting an OpenGL command stream lead to a non-invasive extension mechanism for graphics applications, but also how simply manipulating the stream severely limits the kinds of styles we can consider. We describe how our system efficiently reconstructs a declarative representation of the geometry implicit in the graphics library command stream. We present a set of application extensions built with this framework including several stylized renderers. Extensions built using our system are capable of changing the rendering style of applications on the fly at interactive rates.

**Keywords:** non-invasive, non-photorealistic, real-time, stylized, interactive, 3D

## 1 Introduction

Over the past several years, computer graphics techniques for producing intentionally stylized images have emerged. This non-photorealistic rendering (NPR) has proven interesting not only for purely aesthetic reasons, but also to enhance comprehension of automatically generated images. Because of this, NPR techniques are useful across a wide variety of applications. Unfortunately, most NPR methods create images in unique ways—very different from traditional rendering methods, and even from other NPR techniques. Typically, applications must be retrofit or even re-architected to support new visual styles. This has precluded extensive experimentation with matching new styles to existing applications.

Non-invasive methods for altering the behavior of graphics applications can be used to extend graphics applications without modification. These methods work by intercepting calls to the underlying graphics library and have allowed for a variety of useful changes including changing the visual style of interactive graphics programs. The benefit of these techniques is that they allow new functionality to be added to existing applications without modifying the applications. The problem with these techniques is that they obtain only limited information about what the application is doing—merely the information inherent in the stream of library calls from the application.

In this paper, we explore the potential for creating new and interesting visual styles for interactive 3D applications using only this limited information in the graphics library command stream. First, we examine the information obtained by intercepting the stream of calls to the low level graphics library OpenGL. Directly manipulating this stream can lead to some simple stylistic changes. Unfortunately, we find this direct transformation too limited to support many desirable stylized rendering techniques.

More sophisticated NPR techniques require a declarative representation of the geometry in the scene as opposed to the inherently imperative representation contained in the OpenGL command stream. We will show how such a declarative representation can be constructed from the stream of graphics library calls, and we provide several examples of NPR styles implemented using this representation. The information provided by such a reconstruction is still limited, however, so we examine these limitations and their impact on the types of renderers possible using our techniques.

A benefit of our interception-based stylization is that it allows stylized rendering algorithms to be used with varied programming languages and programming styles. Even in cases where we do have the ability to modify the source code of an application program,

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

it is sometimes simpler to use our interception and reconstruction techniques to provide stylized rendering. This frees the application programmer from having to worry about providing data in a form compatible with renderers, as is common in a scene graph library. There is a trade off, however. Interception has a performance cost, and can only provide limited information to a renderer. While our interception system may not be able to compete with a carefully hand-crafted application, it can provide interactive performance on a wide variety of renderers across a wide variety of applications.

In a previous paper [2001], we introduced the concept of non-invasive stylization of interactive applications. In this paper, we make a further contribution to the topic by introducing the more generally useful technique of reconstructing a declarative representation of the geometry from the OpenGL command stream and provide a number of new renderers that can work with this limited data. We present a prototype system called *HijackGL* that is more general and efficient than our previous system, and a set of new renderers that are more efficient and attractive than those presented previously.

The rest of this paper is organized as follows: We begin by examining the mechanism of intercepting applications' calls to the graphics library, including reviewing the previous work that uses this technique. We briefly review the stylization capabilities of systems that are able to manipulate the OpenGL command stream and show how they are unable to directly support an important class of NPR techniques. We then consider the problem of reconstructing a declarative representation of the geometry from the command stream, and the issues in computing the information required for NPR algorithms. We describe the reconstruction mechanisms used by HijackGL and the software architecture this reconstruction enables. Given this representation, we then consider how several NPR algorithms can be realized within this framework. We conclude by assessing the apparent limitations of the non-invasive approach.

## 2 Intercepting OpenGL

Modern operating systems provide access to common functionality to applications through shared libraries. Shared libraries are similar to traditional libraries, except they are dynamically loaded and linked at run time rather than statically linked at compile time. Shared libraries have the advantage that a library may be easily updated or replaced, by just replacing the library file. Since linking happens dynamically, when a shared library is replaced the next invocation of a program that uses the library will operate with the updated version.

Shared libraries provide a mechanism for modifying the behavior of applications without altering them. Changing the shared library can change the behavior of an application that uses it. Providing that the new library provides exactly the same interface as the original, applications should be unaware of the change. This provides a mechanism for temporarily modifying the behavior of an application. By causing the linker to dynamically link the application to an alternate library, different behavior can be effected. Under Windows and UNIX, this interception can be easily accomplished by placing the alternate library before the system library in the system's library search path (for example, in the same directory as the application on Windows).

The intercepting library can make use of the original library by loading it explicitly. This allows intercepting libraries to act as filters, optionally "passing through" any calls to the original library, or using the original library to implement their own algorithms.

Creating such an interception library requires engineering the new library to faithfully reproduce the interface of the original. It also requires the library to be created in a way that reproduces enough of the functionality of the original library so that applications that use it can function. While this is may require a large amount of engineering, it is straightforward if the library is well documented.

The intercepting library has no access to the internals of an application that calls it. The only information of the application that the library has access to is the sequence of calls the application makes to the library, and any global variables shared between the application and the library. We refer to the sequence of calls made by the application to the library, including the parameters of those calls, the *command stream*.

For extending existing graphics applications, a low-level graphics API seems to be the most logical choice. We could choose to intercept a higher level scene graph API, such as OpenInventor; however, this would severely limit the number of applications our program would be useful for. We could choose to intercept at a much lower level, such as the pixels going to the frame buffer, but this provides data in too unstructured a form, requiring expensive computer vision analysis to create any structure required for processing. Examples of this approach are the painterly rendering systems of [Litwinowicz 1997] and [Hertzmann and Perlin 2000].

Most low-level graphics APIs, including OpenGL, do not explicitly represent the geometry for full scenes at a time. The command stream represents a programmatic or *imperative* representation of the geometry: it provides a sequence of commands that, when executed in order, draw a picture of the geometry. In contrast, most descriptions of geometry for geometric processing algorithms, and high-level APIs (such as Renderman [Upstill 1989] or OpenInventor [Strauss and Carey 1992]) represent geometry *declaratively* as triangle meshes with connectivity or subdivision surfaces, for instance.

Our system for graphics application extension, HijackGL, operates by intercepting calls to the OpenGL graphics library under the Windows operating system. The system could have been equivalently implemented using another low-level graphics API such as DirectX, or under a different operating system such as UNIX. OpenGL has the advantage of being a very common, stable, popular, and well documented API that we are familiar with.

### 2.1 Related Work: Intercepting OpenGL

Many existing systems intercept calls to the OpenGL graphics library for various purposes. The most common category of tools assist programmers in performance analysis and debugging. SGI's glTrace [SGI and Miles 1997] records the sequence of library calls as they are made to assist in debugging and to allow performance monitoring and hardware timing simulation. IBM's ZAPdb OpenGL debugger [IBM 1998] uses this interception technique to aid in debugging OpenGL programs. Intel's Graphics Performance Toolkit [Intel 1997–2000] uses a similar method to instrument graphics application performance. None of these tools provide any facility for changing the behavior of the intercepted application.

In 1996, SGI developed a stream codec for OpenGL [SGI and Dunwoody 1996]. This feature allowed OpenGL command streams and associated data to be captured and recorded to disk or elsewhere. This stream capture ability makes it easy to examine and process streams offline; however, it does not allow for adding new functionality to interactive applications.

WireGL [Humphreys et al. 2001] and its successor Chromium [Humphreys 2001] also intercept OpenGL to generate different output like distributed displays. Chromium provides a mechanism for implementing plug-in modules that alter the stream of GL commands, allowing the simple transformations we have presented [2001] or Section 2.2 to be implemented.

All of these previous systems either view the OpenGL command stream executed by the application, record it, or transmit it. While Chromium allows for some simple stream transformations, it still considers the "data" of the application to be the stream itself. All processing and manipulation of the data is done directly on the stream representation.

Our own work on non-invasive, interactive, stylized rendering [2001] also intercepts the OpenGL stream. This permitted making localized changes to the stream, as described in Section 2.2. To combat the limitations of these localized changes, some of the renderers buffered an entire frame of data and processed it to render the image. In a sense, these renderers were performing a special-case of the scene reconstruction described in Section 3. In this paper, we formalize, generalize and encapsulate this reconstruction, and provide new and improved rendering algorithms that use it.

## 2.2 Stream Transformations

Once the OpenGL interception mechanism is in place, some very simple but useful stream transformations can be made. The simplest transformations change the operation of a single library call. For example, one simple method is to cause every triangle to be drawn in wireframe. This may be done by simply drawing a triangle's outline every time a triangle should be drawn. This is useful for observing how the geometry changes in programs that use level-of-detail methods. Other similar changes include perturbing normals and colors, quantizing color values, or making objects translucent.

Changing the stream directly like this can support interesting extensions, but this method is limited. These modifications are "local" in the sense that they cannot consider information contained in other parts of the stream when making changes. This means that information like connectivity cannot be obtained. Also, since the stream is processed serially, there is no easy way to reorder operations; for example, to draw all the polygons in a scene in a sorted order. All the performance evaluation and debugging tools cited earlier can be implemented by processing the stream of graphics commands in this manner.

In a previous paper [2001], we explored some of the potential for stylistic alterations possible with these local stream transformations. Some of these included a simple depth-cued wireframe renderer, a very simple pencil-sketch renderer that drew the outlines of every triangle in the scene, and a colored-pencil style that drew sketchy approximations of each triangle with quantized colors.

There is a great potential for even more effects to be implemented by local transformations. For example, we might consider performing non-linear color shifts (to better meet the usability concerns of partially color blind users), non-linear spatial distortions (for example, to zoom in on regions of interest yet retain context of the whole scene), or even selective omission of various geometric elements. However, any such local transformation of the stream is fundamentally limited.

## 2.3 Stylized Rendering Techniques

Stylized rendering is a rapidly growing area of computer graphics. See [Reynolds 1999–2002] for a an excellent annotated bibliography. Techniques for stylized[1] rendering fall into three broad categories: 3D techniques that render images based on geometric descriptions, image post-processing methods that alter existing images, and interactive methods that enhance users' inputs. For stylization of existing applications, the third category is inappropriate.

It is possible to apply image-processing stylization algorithms to the output of 3D graphics applications. The processing would occur in the frame buffer after rasterization. We have not considered such an approach because the methods are usually more costly (since they involve per-pixel operations, and reading images from the frame buffer which is expensive on modern graphics hardware), and often needs to reconstruct information that existed in the geometric representations available before rasterization.

Most stylized rendering algorithms operate on a declarative description of the geometry. Because these algorithms rely on non-local properties of the geometry, or simply wish to avoid the artifacts of the underlying representation, they must consider the geometric models as a whole. For example, many existing stylized rendering algorithms require silhouette edges [Lake et al. 2000; Hertzmann and Zorin 2000], front-to-back ordering of polygons, surface parameterizations [Praun et al. 2001; Hertzmann and Zorin 2000], or implicit shape representations [Bremer and Hughes 1998]. Others operate by processing subdivision surface meshes [Hertzmann and Zorin 2000] or multi-resolution meshes [Lake et al. 2000].

To achieve stylized rendering that might truly be called "artistic," even more information than just the geometry is required. To better convey information, a renderer needs to understand the communicative goals of the image [Seligmann and Feiner 1991], or at least to know what and why a scene is being drawn. Such information is most certainly not in an OpenGL command stream, and is unlikely to be contained in generic geometric representations.

# 3 Reconstructing Geometry

To be non-invasive, we must limit our system to obtaining only the OpenGL command stream from applications. Given that many existing stylized rendering algorithms require a declarative representation of the geometry, we are faced with two options. We could devise new algorithms that function on the command streams, as we did previoiusly in [2001], or we can devise a scheme to reconstruct the necessary declarative representations from the OpenGL stream. Because we hope to draw on the wide ranging literature of existing algorithms, we choose the latter approach. An additional argument in favor of reconstruction is that it is unclear that stream transforms can support rich enough visual styles. Rather than debate which representation is better, we chose to construct a system that can produce both.

The types of declarative representations used for stylized renderers are different from the internal representations used by many interactive graphics programs. Many OpenGL programs have no need to sort objects or primitives because of z-buffering hardware and have no use for silhouette edges or connectivity information. Therefore, they often neither compute nor contain explicit representations of this information.

Our goals in building HijackGL were to provide a system that allowed the greatest possible range of renderers to be applied to the

[1]We prefer to use the term *stylized* rendering rather than the more common *non-photorealistic* rendering because little in computer graphics is truly photorealistic. Intentionally non-photorealistic is too cumbersome a term, and we are not arrogant enough to call the output of our algorithms artistic; however, this term is often applied to this class of imagery.

greatest possible range of applications, and to maintain interactive performance. These concerns drive the system design described in this section. In Section 5 we assess how successful we are at achieving these goals.

### 3.1 Declarative and Imperative Representations

Parsing the OpenGL command stream on the fly and constructing a useful higher-level representation poses many challenges. Because OpenGL is an extremely flexible API and it reflects an imperative model that maps well to graphics hardware, the command stream does not directly correspond to a declarative representation of the geometry. In contrast, recent work shows the OpenGL command stream can be viewed as an assembly language program with each command corresponding to a single instruction [Peercy et al. 2000].

The hardware-centric drawing model does provide a lowest common denominator for managing the complexity of the API. Ultimately, OpenGL turns streams of commands into hardware specific rasterization commands. Most OpenGL commands either update the library's internal state (which determines where and how the primitives will ultimately be rasterized), or draw primitives that are ultimately broken down into lines and triangles.

Constructing a useful declarative representation of the geometry requires more than simply buffering all of the stream data for each frame. While buffering all of the information is an important step, we also must construct a meaningful representation of that information sufficient for the algorithms we wish to employ.

To demonstrate some of the more basic issues in reconstructing geometry from an OpenGL stream, consider interpreting a program that draws a cube with its faces colored differently. Ultimately, we would like to tell the renderer "there is a cube" or even, there are 8 vertices connected into 6 faces, by 12 edges, with a certain set of details. While OpenGL gives programmers a wide variety of ways to specify the cube, these two geometric descriptions are not among them. Instead, an OpenGL program instructs the graphics hardware to draw the cube one primitive at a time. For each primitive, the geometric information for each vertex must be provided. In the case of the cube, each vertex must be specified multiple times, once for each primitive [2]. OpenGL provides a number of methods for specifying the primitives: it provides for all of the numeric data types in the language (integral, floating point, ...) and a variety of equivalent mechanisms for generating primitives (individual triangles, quadrilaterals, triangle strips, ...).

### 3.2 Processing the Command Stream

The first step in processing the OpenGL command stream is to homogenize the data. HijackGL homogenizes data in different ways. First, simple data type homogenization is performed. All incoming data values are translated to a common format. For example, vertex location data and texture coordinates are converted to full double precision[3] four-vectors. Next, higher-level primitives like quadrilaterals, triangle strips, and triangle fans are converted to collections of more fundamental primitives. That is, quads are represented by pairs of triangles, and triangle strips and fans by collections of triangles. This is done so that all the surface geometry in the scene may be examined by walking a list of triangles without having to write code to support each possible surface primitive type.

The next step in processing is to build data structures that represent geometry as entities, rather than drawing instructions. In OpenGL, we must distinguish between a *vertex*, which is what OpenGL uses to describe the primitives that are drawn, and a *location* which is a position in space. In OpenGL, vertices have several attributes including location, color, normal, and texture coordinate. A vertex is instantiated with the OpenGL command glVertex. When this command is issued, all of a vertex's attributes are fixed. The location of the vertex is set by parameters to the glVertex command, and all the other attributes are set by the current OpenGL state. OpenGL does not keep track of vertices any longer than is necessary to rasterize the primitives associated with it. In contrast, HijackGL buffers vertices for entire frames at a time.

Because vertices have many attributes, *different vertices* may share the *same location*. More generally, any two vertices differ if and only if one or more of their attributes differ. For example, in the cube described above, there are eight points but twenty-four *different* vertices because for the different faces, the vertices have different normals and colors.

HijackGL stores a hash table for each vertex attribute. Hashing is used to recognize and remove duplicate values. Removing duplicate values for locations is necessary to compute connectivity information described later. Hashing is also used with other attributes to reduce the amount of memory required to run HijackGL. Because we must cache associated data for every value, the memory footprint of each vertex attribute can be large enough that sharing is necessary to avoid excessive memory usage. The size of data for a single frame can be very large, depending on the complexity of the scene. Storing vertex attributes with every vertex directly can consume a lot of space. Using hash tables to only store the unique values reduces the space requirements. For example, scenes that contain many vertices that vary only in their locations only store a single instance of the shared color, texture coordinate, and normal that vertices refer to. This is potentially a very large savings over storing every vertex attribute explicitly. For scenes that do contain very many unique values, the hashing does not save much space. In practice, many applications duplicate many vertex attributes.

To determine whether vertices share the same location or normal, we must place them into a common coordinate system so that their positions can be compared. It is important to note that there is no notion of a "world" coordinate system in OpenGL. Because OpenGL gives the programmer flexibility in how coordinate systems are defined, we choose to convert all positions and normals to eye coordinates (i.e. using OpenGL's MODELVIEW matrix, or in the case of normals, the inverse transpose MODELVIEW matrix). Once we have transformed locations to eye space, the values are truncated slightly because the least significant bits may differ slightly due to numerical precision limits. Once in this format, a simple hash value is computed using the binary representation of the floating point values.

Using the hashing idea further to remove duplicates, HijackGL stores many objects including all OpenGL primitives in hash tables. This is illustrated in Figure 1. Vertices reference Locations, Colors, Texture Coordinates, and Normals. In turn, Points reference single Vertices, Line Segments reference pairs of Vertices, Triangles reference triples of Vertices, and Quadrilaterals reference pairs of Triangles. In addition to these OpenGL primitives, HijackGL stores some structures it computes itself in the same manner. For example, Edges reference pairs of Locations.

[2]Because each face has a different normal, the OpenGL mechanism for sharing vertices does not apply.

[3]In hindsight, this was a poor decision. Single precision would have provided better performance and there are few applications that require double precision and even fewer graphics devices that support it.

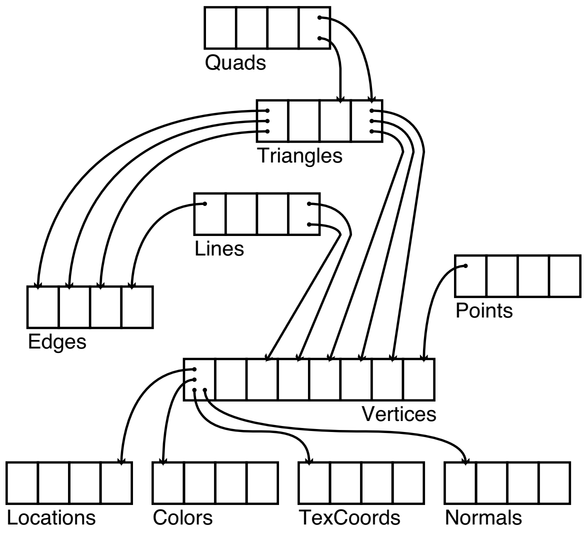


**Figure 1:** The data structures of HijackGL's declarative representation of geometry. This figure does not show the complete set of information tracked by HijackGL. Rather, it is meant to indicate the overall structure.

## 3.3 Computing Additional Information

While processing the stream of commands and building our data structures, HijackGL computes connectivity information. Edges are recorded as line and triangle primitives are added to the data structure. Figure 1 shows a hash table of edges kept by HijackGL. Edges reference two *locations* (as opposed to vertices). In addition, lines reference a single edge, and triangles reference three edges. As edges are added to the hash table, if they come from triangles, two triangles are remembered from each edge as "neighbors". This resulting winged-edge-like data structure encodes the connectivity of the geometry. Also, silhouette edges (defined as those edges whose neighbors overlap when projected to the screen) are flagged, and the sharpness of edges (defined by the angle between an edge's neighbors) is recorded.

While a more clever algorithm for computing connectivity information and silhouette edges might be faster, our simple technique fits well inside HijackGL, and it is fast enough to maintain interactivity with many applications.

Several applications, including popular games, do their own lighting. In this case, the application never sends normals to OpenGL. To handle this, HijackGL computes the face normals of triangles as an approximation.

Many NPR styles benefit from knowing geometry information in screen-space. Therefore, HijackGL computes the screen-space locations of vertices when it stores them.

## 3.4 Assumptions and Inferences

The reconstruction mechanism just described computes useful information given the command stream. However, there is much more information that we might like for constructing non-photorealistic renderers and other extensions. For example, we might like to know what geometry comprises logically distinct objects. This would allow us to treat different objects separately. We might also like to know what objects should be considered foreground objects and what might be considered background objects. More practically, we might like to know which objects should be changed and which should not. For example, stylizing user-interface elements can make a program difficult or impossible to use.

Unfortunately, this kind of information is not explicitly represented in the stream. In order to reconstruct any of this information, we must either make guesses and assumptions about how the program we are intercepting operates or we must have some prior knowledge of the program's operation. We choose not to examine the latter case, because this solution is not general. However, we note that if this kind of information is desired, it can be obtained by using HijackGL as a tool to monitor the commands the application in question is executing.

There are some simple assumptions and guesses that operate reasonably well in a number of cases. For example, many programs use OpenGL's transformation matrix stack to position different objects. We can make the assumption that whenever the transformation matrix changes, a logically distinct object is being drawn. While this certainly does not work all the time, it works well for some applications.

Ultimately, the lack of high-level information of an application's intent in drawing places a fundamental limitation on the capabilities of the interception approach. We will explore this limitation in Section 5 .

## 3.5 Software Architecture

The buffering and reconstruction performed by HijackGL to encapsulate the geometry information creates a good abstraction for renderers to process. HijackGL then provides the infrastructure for renderers and extension modules to use our reconstructed geometry.

Renderers for HijackGL are dynamically loaded plug-in modules. One of these modules is loaded for use when an application starts. HijackGL then intercepts all the OpenGL calls and builds a declarative representation of the geometry data. When an image is required (for example, at the end of a frame, or when a read-framebuffer command is encountered), HijackGL calls the plug-in module. The plug-in module may then process the reconstructed data and call into the system's OpenGL implementation to draw images on the screen. HijackGL also supports a mode where plug-in modules may execute once in response to a keypress. This is useful for creating extensions like screen-capture modules, as described in Section 4.5.

Because of our declarative data representation, it is very easy to implement simple renderers in HijackGL. For example, a simple wireframe renderer would essentially consist of a for-loop to walk over every edge in the scene and draw it. Adding a single condition to check the silhouette flag on each edge would make a silhouette edge renderer. To draw solid geometry, another for-loop that walks all the triangles and renders them is all that must be added. Our most complex renderer to date, the pencil sketch renderer of Section 4.2, is about 500 lines of code.

## 4 Example Renderers

In this section, we examine several plug-in renderers that we have constructed for HijackGL. Ideally, these renderers would be straightforward implementations of standard stylized rendering techniques. However, because of the fundamental limitations of the type of information interception can provide (Section 3.4 and Section 5), the pragmatic limitations of our current implementation, and our efficiency concerns, we have had to develop modified versions of existing algorithms to meet our needs.

The renderers chosen here were selected because they demonstrate the utility of geometric reconstruction and the potential and problems of interception-based approaches. We do not claim that these renderers provide the visual quality that state of the art, hand-coded stylized rendering applications provide. However, they are of sufficient quality to be interesting, and further, we are able to provide our renderers across a wide variety of existing applications. Much of the visual appeal that results from our demonstrations comes from choosing well-designed applications.

## 4.1 Wireframe Rendering

In Figure 2 we show a wireframe renderer running on HijackGL. A wireframe renderer is easy to produce with our system. As stated in section 3.5, this renderer is essentially a single for-loop that walks the list of edges in the scene and draws them.

Wireframe renderers are not often considered very artistic styles, but they are useful. For example, to see how a level-of-detail algorithm operates, it is often ineffective to examine the running application itself. In fact, many level-of-detail algorithms are designed to minimize the overall visual changes as the underlying geometry changes. With a simple plug-in wireframe renderer, it is easy to see what is happening with the underlying geometry as levels-of-detail change.

While this renderer does not explicitly need a declarative representation to operate, its implementation becomes very simple and straightforward with this representation.

An improved wireframe renderer may accent the silhouette edges and sharp edges. This gives enough information to make the shape apparent but does not clutter the view with all the internal structure. This simple style is difficult or impossible without reconstructing the geometry so that silhouette edges may be computed.

## 4.2 Pencil-Sketch Rendering

In previous work [2001] we presented a very simple pencil sketch renderer that simply drew triangles in white, and then outlined them with jittered, sketchy lines. This renderer operated by making local stream transformations. It was not convincing because it was effectively a wireframe renderer, showing the structure of the triangle mesh. A more sophisticated renderer in that paper draws pencil strokes that cover multiple triangles, but this renderer was too inefficient for interactive use.

Figure 4 shows an improved, interactive pencil-sketch style renderer running on a demo of the game Quake III Arena by id Software. This renderer is implemented as a plug-in module for HijackGL. The module takes advantage of connectivity information to render silhouette and sharp edges with thick dark lines. These edges help convey the shape of geometry. Pencil-sketch textures are applied to each triangle in the scene with an orientation that varies smoothly with surfaces' orientations. This is accomplished by choosing a "stroke direction" that is perpendicular to the projection of the triangle's normal onto the view plane. A pencil-sketch texture with strokes all going in one direction is applied to the triangle by choosing texture coordinates such that the pencil stroke direction is perpendicular to the projected normal. This causes smooth surfaces to have consistent shading. Since we have computed the positions of vertices after perspective projection, we can draw the triangles and apply the pencil-sketch texture in screen space and thus the pencil strokes remain at constant density across images.

This pencil sketch renderer is inspired by the work of [Lake et al. 2000]. However, their method always forces sketch textures to be applied in the same orientation relative to the screen while our pencil sketch renderer allows the sketching direction to vary with the surface orientation. Specifically, we choose the sketching direction to be perpendicular to the projection of the surface normal biased upwards in screen space. This procedure guarantees that the sketching direction will vary smoothly for smooth surfaces, and will change sharply for discontinuous surfaces.

## 4.3 Blueprint Rendering

We created a simple blueprint rendering style using HijackGL shown in Figure 5. This renderer draws translucent white lines on a blue background to give the impression of a blueprint drawing. The lines are only drawn on silhouette and sharp edges, and the lines are extended beyond their end points to further suggest a blueprint or draft-drawing style. Dimension lines are drawn for selected edges greater than a certain length to further suggest a blueprint style. We would like the edges that have dimension lines drawn for them to change with low frequency because randomly switching every frame can be distracting. However, HijackGL does not get any inter-frame coherency information, so we cannot do this reliably. However, since many applications draw their scenes the same way from frame to frame, we make the assumption that edges that appear in the same place in the edge list are in fact the same from frame to frame. This works extremely well for some applications and the dimension lines change with low frequency. However, this fails with other applications, causing distracting flicker. g

## 4.4 Cartoon Rendering

Figure 6 shows a cartoon-style renderer running on a research animation system. This module is a direct implementation of the algorithm presented in [Lake et al. 2000]. We implemented this algorithm to demonstrate that implementing an existing popular style in HijackGL is possible.

The fundamental algorithm is straightforward. Very simply, traditional diffuse lighting is computed per-vertex by computing the dot product of the unit surface normal with a unit vector in the direction of the light. Then the material color of the surface is scaled by this resulting value to obtain the final color. Cartoon rendering is characterized by a harsh quantization of lighting values. Often, objects are rendered in a two-tone fashion: the parts of objects illuminated directly appear bright, while those that are not illuminated directly appear as darker versions of the material color.

To simulate this, we compute the dot product of the surface normal and the direction to the light like normal, but instead of scaling material color values directly, we instead quantize the dot product value, using it to index into a one-dimensional texture map. This one dimensional texture map is made up of two or three constant color segments, ranging from near black to near white. By using the dot product as a texture coordinate in this one dimensional texture, we effectively quantize the dot product value. We apply this texture map to the objects in the scene and configure OpenGL to multiply the objects' material colors by the texture map, thereby properly two-tone shading the objects. The result is rather convincing, as seen in Figure 6.

While this renderer is a common style, by implementing it in HijackGL, we can automatically apply it across many applications—even to those whose source code we do not have.

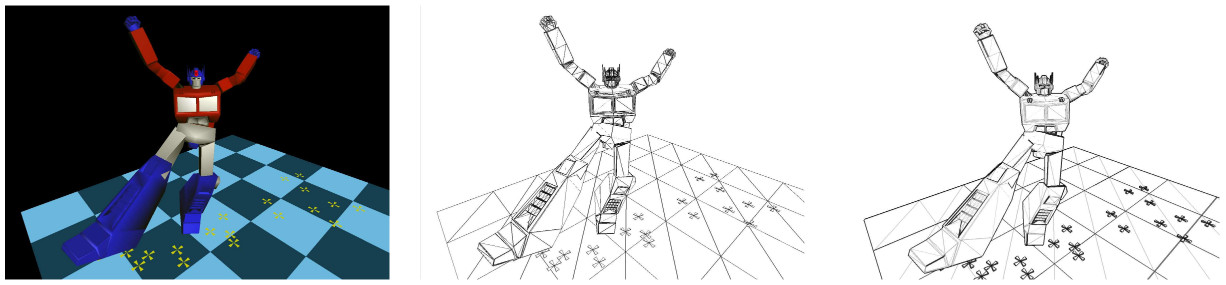


**Figure 2:** Wireframe renderers. Left: Original application. Center: Wireframe renderer that displays all edges. Right: Wireframe renderer accenting silhouette edges.

### 4.5 Other Applications

While our focus in this paper is the use of HijackGL for stylized rendering, we briefly consider some other relevant uses here. In each case, a plug-in renderer makes use of the reconstructed geometry to extend applications non-invasively.

Generating different output from graphics programs is useful for many tasks. Often, generating high-quality output for print media from a graphics program requires programming investment. Extra code must be added to render to a high-resolution off screen buffer, and then write that buffer to disk. This is reasonably straightforward; however, if scalable vector art is desired or required for print media, complex code to do projections and generate properly sorted primitives must often be added. With our techniques, an extension similar to the gl2ps library [Geuzaine 2001] can be implemented and then used with any OpenGL application. We have built a module for HijackGL that generates Adobe Illustrator vector art files from 3D applications. This module takes advantage of the reconstructed scene information to sort triangles from back to front. We note that this module could be chained with a visual style change module to produce vector output of one of our stylized renderers.

Chromium and its predecessor, WireGL [Humphreys et al. 2001] exist primarily to generate different output. Their system, like ours, uses library interception techniques and sends OpenGL commands to a cluster of machines that render to a tiled display for large format output. However, because our reconstruction step removes many of the redundant values in an OpenGL stream, it may be possible to send less information across the wire to the cluster of renderers.

Figure 3 demonstrates the results of our geometry capture module. This extension works by processing the reconstructed scene data and writing a 3D file format of the scene data. Without the reconstructed scene data, the models we generate would effectively be bags of disjoint triangles. Instead, because we compute connectivity, we may convert our data to any standard mesh boundary representation. After we have taken a 3D scene capture, we can then view and manipulate this data in a common modeling or animation package, such as Maya. This module allows us to extract 3D models from any program that displays them using OpenGL.

## 5 Capabilities and Limitations

Intercepting OpenGL and reconstructing a declarative scene data representation has obvious advantages over hand-crafting new renderers within each application. The interception mechanism allows us to implement new rendering styles that can be applied across a wide range of applications. By constructing a declarative representation of the scene geometry from the OpenGL stream, we enable a broad class of rendering algorithms that either require data in this form or are simplified by its availability.

The non-invasiveness of the interception approach does have its cost over hand-tuned renderers built right into applications. The most obvious is performance. HijackGL requires an application's data to pass through the OpenGL interface and be reconstructed into geometry before being sent to the renderer. A renderer that can directly access the application's data structures clearly has less overhead. Despite this performance cost, HijackGL is still capable of providing interactive performance on complex geometry. For example, our unoptimized HijackGL prototype is able to achieve more than 20 frames per second on a standard Quake III Arena benchmark. It is difficult to characterize the performance implications of our technique because it depends on the performance of the underlying graphics system. For instance, on an application that does not stress the graphics system, our overhead may account for an order-of-magnitude or more decrease in performance. For applications that are limited by the speed of the graphics system, our overhead may be insignificant. We have seen both these extremes and examples that lie in the middle. We have encountered very few applications that are made too slow to be interactive by our system.

The interception approach has limitations in its generality and resulting quality as well. The information available from the OpenGL stream is limited. As discussed in Section 3.4, we have no knowledge of why an application is drawing what it is drawing. For example, if an application chooses to display its menu or control panel by painting a texture onto a polygon (a common practice in many games), we have no way to distinguish this from scenes that we would like to stylistically render. Similarly, we have no way to differentiate semantically different objects that should be treated differently, or to apply internal application data in making rendering decisions. For example, in a shooter game such as that shown in Figure 4, we cannot treat the enemy characters differently from the scenery, nor can we change the characters' color based on the state of their AI, effects that would be easy if we were to change the applications source code.

Another class of limitations of our technique is that we must decode the OpenGL stream, which is a sequence of instructions designed to create a particular image, not just to relay the scene geometry. Clever programmers use a wide variety of tricks to create their visual effects. In essence, the interceptor must de-stylize the application before re-stylizing it. At worst, a programmer might use the OpenGL machinery to perform computations completely unrelated to rendering, such as the hardware assisted path planning of [Lengyel et al. 1990].

One potential route to resolving these issues is to adopt a "mildly invasive" strategy. We would still use an OpenGL interception mechanism, but provide OpenGL extensions to allow an application to give hints about its intent as well as to provide a faster path to providing the scene geometry.

A final category of limitations stems from the incompleteness of our prototype. While HijackGL implements enough of OpenGL to allow for many interesting applications and renderers to be created,

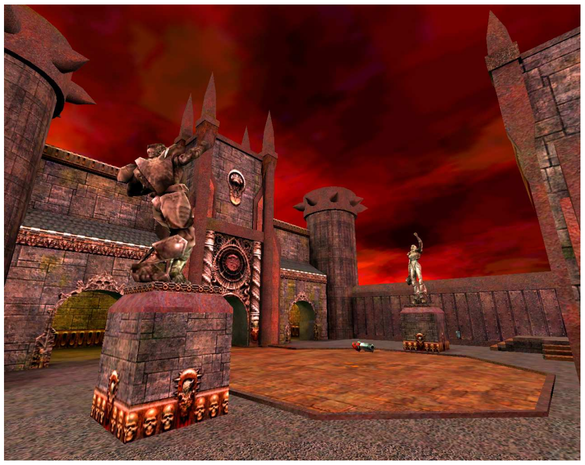
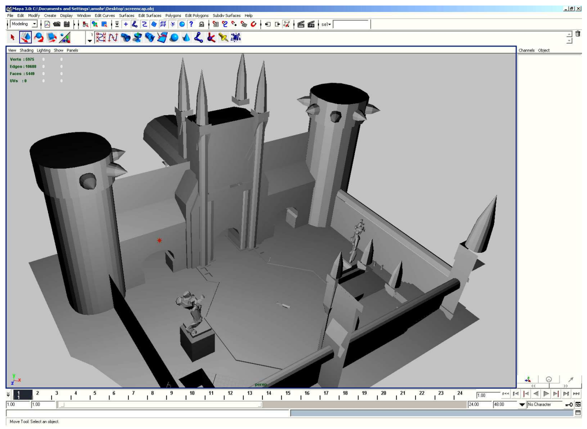



**Figure 3:** On the left is a scene from the Quake III Arena demo. On the right is the geometry of this scene captured as a model and loaded into Maya.

it does not track all of the OpenGL state or recreate all of its functionality. In principle the entire OpenGL state could be tracked, although given the complexity of the OpenGL state machine this would be difficult to implement and efficiently store over time such that the current state for each primitive could be recovered.

While the interception approach has its limitations, re-engineering all existing applications with all desirable renderers is also an impractical approach. Using our interception mechanisms and geometric reconstruction techniques we have created a system that can connect a wide variety of stylized rendering algorithms to a variety of existing applications.

Even within the limitations of interception, there is still much to be done. Our prototype must be made more robust, complete and efficient. Many additional rendering styles, such as pen-and-ink, are possible given only the geometric information we have. Other output mechanisms, such as writing geometric data for high-quality rendering is possible, as are debugging aids and analysis tools. Providing more information than just geometry, either through heuristics or mildly invasive hinting, will open up even more opportunities for extensions.

Visual style changes are useful for many applications. For example, a visualization application may render in a fashion that is inadequate to make the data comprehensible to the scientist. With our system, the renderer could effectively be replaced with one explicitly designed to help visualize some specific data. A different example comes from accessibility software. Many programs are not written with visually impaired people in mind. A non-invasive extension built using our techniques might increase contrast or emphasize shapes in such a way that enables visually impaired people to use programs they could not earlier. Other uses of visual style change include prototyping new renderers with existing applications, examining level-of-detail or culling algorithms in action, or simply putting new twists on old computer games. With our interception-based approach, changing the visual style of an existing application is possible and practical.

## Acknowledgments

Christopher Herrman wrote the HijackGL cartoon renderer described in Section 4.4 and shown in Figure 6.

This research is supported by an NSF Career Award "Motion Transformations for Computer Animation" CCR-9984506, and a Wisconsin University-Industrial Relations (UIR) Grant. Additional support provided by Microsoft and the Wisconsin Alumni Research Foundation. Equipment donations provided by IBM, NVIDIA and Intel, and software donations from Microsoft, Intel, Alias/Wavefront, SoftImage, Discreet and Pixar.

## References

BREMER, D., AND HUGHES, J. F., 1998. Rapid approximate silhouette rendering of implicit surfaces.

GEUZAINE, C., 2001. GL2PS: an OpenGL to Postscript printing library. computer software. `http://www.geuz.org/gl2ps`.

HERTZMANN, A., AND PERLIN, K. 2000. Painterly rendering for video and interaction. *NPAR 2000 : First International Symposium on Non Photorealistic Animation and Rendering* (June), 7–12.

HERTZMANN, A., AND ZORIN, D. 2000. Illustrating smooth surfaces. *Proceedings of SIGGRAPH 2000* (July), 517–526.

HUMPHREYS, G., ELDRIDGE, M., BUCK, I., STOLL, G., EVERETT, M., AND HANRAHAN, P. 2001. Wiregl: A scalable graphics system for clusters. *Proceedings of SIGGRAPH 2001* (August), 129–140. ISBN 1-58113-292-1.

HUMPHREYS, G., 2001. Chromium. Computer Software.

IBM, 1998. Zapdb. Computer Software.

INTEL, 1997–2000. Intel graphics performance toolkit. Computer Software.

LAKE, A., MARSHALL, C., HARRIS, M., AND BLACKSTEIN, M. 2000. Stylized rendering techniques for scalable real-time 3d animation. *NPAR 2000 : First International Symposium on Non Photorealistic Animation and Rendering* (June), 13–20.

LENGYEL, J., REICHERT, M., DONALD, B. R., AND GREENBERG, D. P. 1990. Real-time robot motion planning using rasterizing computer graphics hardware. *Computer Graphics (Proceedings of SIGGRAPH 90) 24*, 4 (August), 327–335.

LITWINOWICZ, P. 1997. Processing images and video for an impressionist effect. *Proceedings of SIGGRAPH 97* (August), 407–414.

MOHR, A., AND GLEICHER, M. 2001. Non-invasive, interactive, stylized rendering. *2001 ACM Symposium on Interactive 3D Graphics* (March), 175–178.

PEERCY, M. S., OLANO, M., AIREY, J., AND UNGAR, P. J. 2000. Interactive multi-pass programmable shading. *Proceedings of SIGGRAPH 2000* (July), 425–432.

PRAUN, E., HOPPE, H., WEBB, M., AND FINKELSTEIN, A. 2001. Real-time hatching. *Proceedings of SIGGRAPH 2001* (August), 579–584.

REYNOLDS, C., 1999–2002. Stylized depiction in computer graphics. http://www.red3d.com/cwr/npr/.

SELIGMANN, D. D., AND FEINER, S. 1991. Automated generation of intent-based 3d illustrations. *Computer Graphics (Proceedings of SIGGRAPH 91) 25*, 4 (July), 123–132. ISBN 0-201-56291-X. Held in Las Vegas, Nevada.

SGI, AND DUNWOODY, C., 1996. The opengl stream codec. http://trant.sgi.com/opengl/docs/Specs/glsspec.txt.

SGI, AND MILES, J., 1997. gltrace. http://reality.sgi.com/opengl/gltrace/.

STRAUSS, P. S., AND CAREY, R. 1992. An object-oriented 3d graphics toolkit. *Computer Graphics (Proceedings of SIGGRAPH 92) 26*, 2 (July), 341–349. ISBN 0-201-51585-7. Held in Chicago, Illinois.

UPSTILL, S. 1989. *The Renderman Companion: A Programmers Guide to Realistic Computer Graphics*. Addison-Wesley.

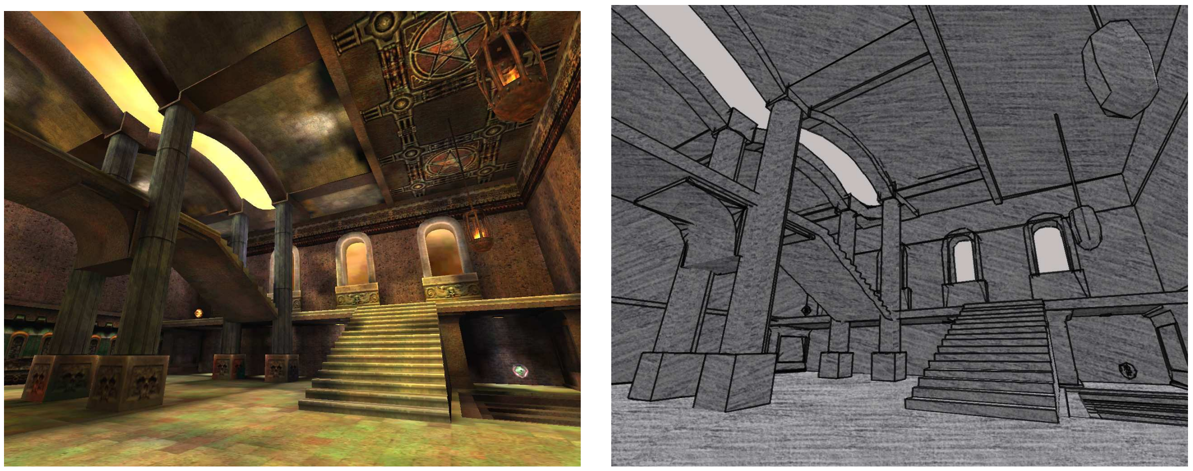


**Figure 4:** HijackGL's pencil sketch renderer running on id Software's Quake III Arena demo.

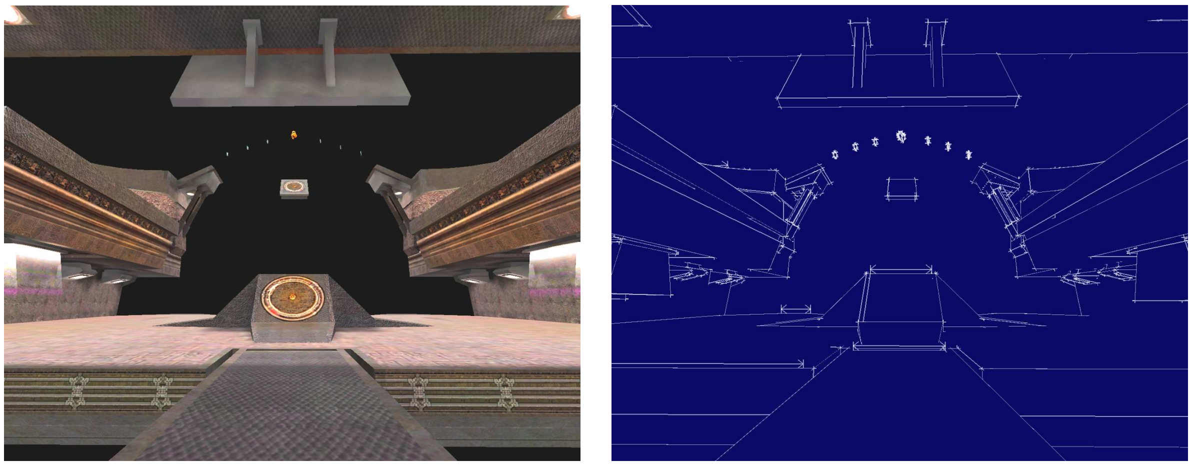


**Figure 5:** A blueprint renderer applied to Quake III Arena.

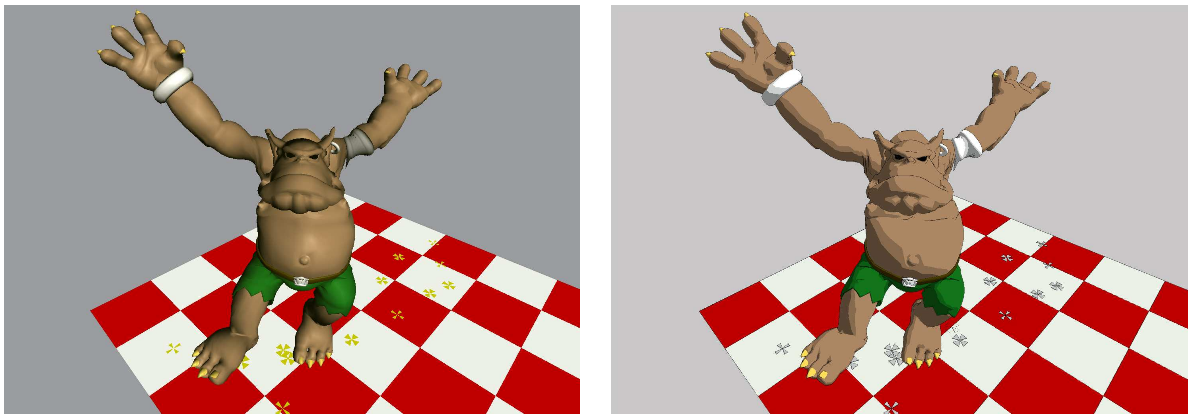


**Figure 6:** A cartoon renderer shown on a character in a research animation system.