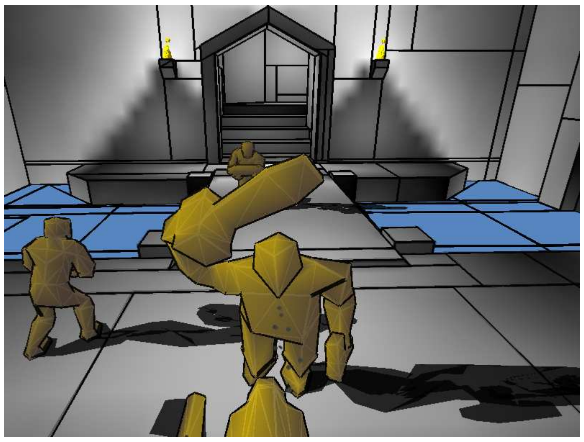
# Technical Quake [*]

Michael Batchelder
*mbatch@cs.mcgill.ca*

Kacper Wysocki
*kacper@cs.mcgill.ca*



## Abstract

*The basic goal of digital rendering is to closely approximate a projection of a 3D scene onto a 2D surface in some rendering style. Classically, the style of choice has been photorealistism. However, more and more research is being done in the realm of non-photorealistic rendering. This is an area ripe with new possibilities - new ways to consider visual scenes, new ways to look at the world, new ways to share information. One particular non-photorealistic style, dubbed technical illustration, can be used to draw out important details of a scene. This style renders silhouettes and creases of models in a clear fashion, which leads to better comprehension of model shape. It moves away from realistic lighting models which can add ambiguity to a scene. The style also favours single hard shadows as well as shading, which clearly conveys shape. This style has been explored by many within the last decade. In this paper we attempt to render the 3D first-person shooter video game Quake [ID Software 1992] in a technical style. We first explore the possibility of gathering neighbouring polygon information for the edges in a model, in order to render silhouettes and various crease styles in a view-dependent fashion, similar to [Raskar 2001]. We then implement two different technical shading models as well as two different shadow models described by [Gooch et al. 1999]. Finally, we explore a few new ideas - blending creases and silhouettes with model colour, and fading creases and silhouettes with distance. These ideas have not yet been mentioned in literature to date that we are aware of. We feel they are potentially useful in interactive 3D environments rendered in the technical illustration style.*

[*]McGill University COMP-767 Non-Photorealistic Rendering

## 1 Introduction and Motivation

Traditionally, technical illustrations are drawn by hand to draw out certain details for the viewer. Artists have the ability to shade, contrast, and highlight information as they see fit. Unfortunately these illustrations are, by their very nature, single instances of 2D approximations of a 3D scene. When multiple views of a model are needed, multiple illustrations must be drawn. This is both time consuming and expensive. Attempting to make a video in this technical rendering style would be even more expensive. However, the advent of computer modeling opens up the possibility of automatically rendering sufficient views of 3D model, no more than the cost than a single image. Videos and real-time interactive 3D systems in which the user can rotate and explore a 3D scene are even possible, giving the user far more information than any set of static illustrations. The problem, is developing rules for detailing technical illustrations which can be automatically applied while still retaining the aesthetically pleasing results of a traditional artist.

Literature exists detailing various styles that can be used to enhance the level of shape information in 3D scenes. It is universally understood that artists enhance important edges which consist of silhouettes and surface discontinuities, such as creases. These edges are inherently included in a digital 3D model and therefore, computers can be used to find and render these special edges. Edges can be drawn with unique colours to differentiate between them, such as black for silhouettes and white for creases. In addition, crease edges can be drawn with different widths or colours based on whether they are jutting out of the model like a mountain ridge or sinking into a model like a valley [Raskar 2001].

Artists also use non-realistic shading to convey shape and curvature. By shading across a model from light to dark, computers can achieve similar results. This allows a user to rotate a model, effectively observing the shape; as the model rotates, any particular part of the model is rendered from light to dark. Seeing the same part of a model under varied lighting can provide far more insight than a static image.

Photorealistic shadows generally do not provide much useful shape information about an object. Artists will often draw shadows with clear edges that don't just approximate the objects silhouette, but mirror it exactly. This allows the viewer to gather information about parts of the model that aren't even visible, depending on lighting direction. Imagine a robot standing in front of you, with a light source above and to the right. The shadow that is spilled on the floor to the left will contain information about the shape of the back of the robot: the silhouette the viewer *would* see if they were looking at a profile of the robot. This is useful even in an interactive system where a user can see objects from any angle. Because the user is seeing more information at any given instance in time, there is less need to alternate between views.

## 2 Related Work

### 2.1 Technical Illustration

Technical illustration techniques have been explored by a number of researchers in recent years. [Gooch et al. 1999] implemented a real-time interactive system which renders lines, shading, and shadows in a technical style. They draw silhouettes in black and creases in white. They outline three shading methods, building on their previous work from [Gooch et al. 1998]. The first, is a cool-to-warm diffuse shading for matte objects. This method linearly interpolates from a cool colour (blue) to a warm colour (yellow) based on the surface normals of a model. This cool-to-warm shift causes cool areas of the model to recede while warm areas advance, accentuating the model's shape. The second method of shading is for metallic objects. It simulates the traditional artistic method of using alternating light and dark bands to represent anisotropic reflection resulting from the milling of a metal object. The final shading method is a modification of the cool-to-warm shift that introduces an element of light splash-back, which is in the opposite direction of the light source. This method gives a dramatic effect that can sometimes communicate more shape information. Gooch et al also explore three styles of non-photorealistic shadows. These methods are meant to describe more clearly shape information of parts of the object that are not necessarily in view. Their best approach is the simplest: they render one hard shadow that does not fade at all. This results in clear edges as long as the background colour is sufficiently different from that of the shadow. However, they also explored painting a shadow with a hard penumbra and a hard umbra which is a nice compromise between a photo-realistic approach and an approach richer in information. Their final style, soft shadows, which we do not explore, is not particularly useful in technical illustration because the *softness* of the shadow effectively removes the shape information gained from a hard shadow.

[Raskar and Cohen 1999] introduce a novel approach to drawing silhouettes. They show how back-facing polygons can be "fattened" so that their edges stick out from behind front-facing polygons. This is done by extending each edge of back-facing polygons with a quadrilateral [Raskar 2001], or by simply pushing these back-facing polygons forward enough that their edges stick out in front of front-facing polygons. The width of these silhouettes can be controlled by using the view vector and the surface normals to calculate the 2D width of the extensions on the view plane. The width of the resulting silhouette is controlled by the width of the quadrilateral, in relation to the viewing angle. In the case of pushing back-facing polygons forward, width is controlled by how far the polygons are pushed forward.

Raskar's later paper on the subject [Raskar 2001], which uses the quadrilateral method for silhouettes, outlines a new algorithm for classifying creases as ridges or valleys based on the normals of the two surfaces which the edge connects. He suggests rendering ridges with one width while rendering valleys with another in order to differentiate between them, giving better shape information to the viewer.

[Lake et al. 2000] suggest the idea of rendering silhouettes in a style other than the simple black line. They extend the work of [Markosian et al. 1997] in rendering edges with textures, in order to map curved textures to silhouettes to simulate the curve of edges better than the primitive polygons that a model is built with. We observed from this work that other styles of edges are possible, such as coloring based on shading.

## 2.2 Quake and NPRQuake

The video game Quake [ID Software 1992] is a first-person shooter style game created by ID Software. A version of the game which uses OpenGL [Woo et al. 1999] for it's 3D rendering is available as open source software, thanks to the generosity of ID Software, and this version has been extended by [Mohr et al. 2002] to handle dynamic loading of rendering styles. [Mohr et al. 2002] implemented three non-photorealistic styles in Quake: Sketch, Blueprint, and Brush. Sketch renders a pencil-sketch style which gives a very nice effect. The blueprint style draws white lines on a blue background similar to architectural blueprints, as it's name would suggest. The final style, Brush, is intended to give a paintbrush style but unfortunately doesn't give very good results.

[Ilie 2003] used [Mohr et al. 2002]'s abstraction of the rendering code to create a toon-style renderer that included an implementation of [Raskar 2001]'s silhouettes and creases as well as hard shadows and toon shading.

## 3 Approach

We chose the Quake engine to implement our technical illustration methods because the source was available and the game itself provided an environment and models that we could use to test and compare our results. The availability of dynamic render loading in [Mohr et al. 2002] was useful as well since it provided a nicely modularized framework for quickly introducing different rendering styles in Quake.

### 3.1 Edges

Our first approach was to find silhouette edges as in [Gooch et al. 1999], which are edges $\vec{e}$ such that:

$$(\vec{n}_1 \cdot (\vec{v} - \vec{e}))(\vec{n}_2 \cdot (\vec{v} - \vec{e})) \leq 0,$$

where $\vec{v}$ is a vertex on the edge, and $\vec{n}_i$ are the outward facing surface normals of the two neighbouring surfaces. Furthermore, creases can be defined as edges where the dot product of unit adjacent surface normals is higher than a pre-specified user threshold. This threshold controls how steep a ridge or valley must be before it is considered a crease.

#### 3.1.1 Recovering neighbouring information

Unfortunately, Quake stores each frame of its models as a ”polygon soup”, that is, series of independent triangle strips and fans which consist of independent points in model space. Such a model format does not contain neighbouring information for edges, and can therefore not be used to test for silhouettes.

Our approach was to attempt to recover this edge information by creating a three-level hash structure at runtime. We do this by employing a three-dimensional hash using the two vertices of an edge as the hash key. The hash value is a structure containing the normal vectors of the planes adjacent to the key edge. Each time a model is drawn, it's name is looked up in a hash table containing a table of frame numbers. Then, the frame number is looked up in this second-level hash table. If the frame number does not exist, a new edge hash table is created for this frame by visiting every polygon of the model, adding edges and surface normals to this third-level hash table as we encounter them. If an edge already exists in the hash table, this signifies that the current polygon is a neighbour of a previously encountered polygon, and its normal is added to the hash value of this edge.

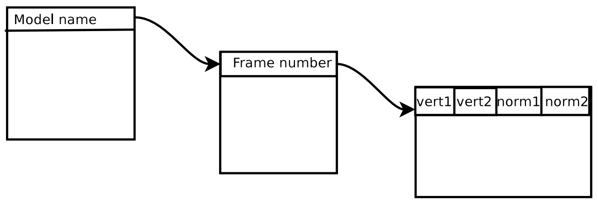


Figure 1: Custom three-level data structure for storing edge neighbouring information.

Once this structure was in place, we were surprised that creating the edge tables on the fly did not significantly slow down the rendering. However, we were also disappointed in finding that the Quake models contain degenerate polygons (polygons with colinear vertices), and polygons that should be neighbours but do not share an edge geometrically, possibly due to the edge of one polygon being slightly offset from its neighbours edge. We also found that there were some edges that connected a front-facing polygon to an overlapping back-facing polygon with a normal going in the reverse direction of its neighbour. Because we were unable to get *clean* neighbouring information we were unable to properly employ the techniques of [Gooch et al. 1999] or [Raskar 2001]. The results were unsatisfying at best, which can be seen in figure [2].

### 3.1.2 Recovering the camera position

In order to properly render the width of silhouettes and creases using the method of [Raskar 2001], the camera position and direction is required. Silhouette and crease edges (quadrilaterals attached as extensions to polygon edges) are not oriented towards the camera as billboards are. Because of this, edges will appear, visually, to have different widths depending on the viewing angle and direction. However, the camera direction and position can be used to calculate the proper width of edges based on their orientation.

However, due to limitations in the abstraction of the rendering calls in [Mohr et al. 2002], we were unable to discover a direct way of accessing the camera position or viewing direction. In order to use this information for edge rendering, we attempted to recover the camera vector by inverting the OpenGL modelview matrix. Then, the camera position $v$ is given by:

$$\vec{0}M^{-1} = v,$$

where $M$ is the OpenGL modelview matrix, and the camera direction is specified similarly. This follows from the fact that the classical approach in OpenGL is to have the camera position always be the origin in world-space, and the camera is always directed in the $z$-direction. This approach did not yield a useful camera vector, however, possibly due to the way Quake uses the modelview matrix. ID Software originally implemented its own rendering engine, and later ported Quake to OpenGL to support a growing user base with accelerated OpenGL hardware. The orientation of the Quake coordinate system is thus different from the regular OpenGL coordinate system, and the mapping is not documented.

We were therefore unable to get a useful camera vector.

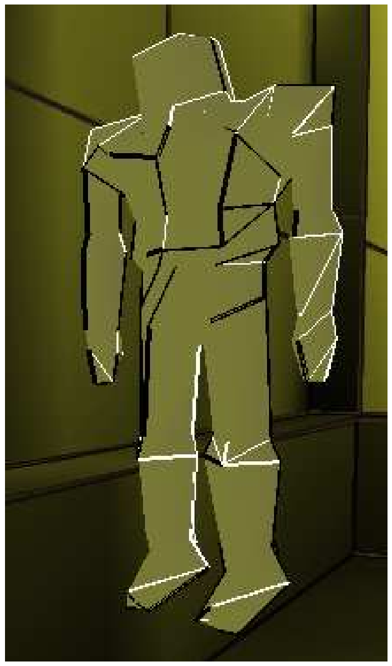


Figure 2: Flawed models and an incorrect camera vector yield unsatisfying creases and silhouettes

### 3.1.3 Silhouettes

Without proper edge neighbouring information or the correct camera position or direction we were not able to find silhouettes in the manner proposed by [Raskar 2001]. Instead of pushing back-facing polygons forward as in the second diagram of figure [3] or rendering quadrilaterals as in the third diagram of figure [3] we observed that we could adapt Raskar's approach to simply draw thick polygon edge lines for back-facing polygons. The effect of a thick line is sort of like a rectangular billboard. Because lines, width-wise, are centered along the line connecting the line's two end points, the line will always have a visual thickness half that of what is rendered, as in the final diagram of figure [3], and this thickness does not vary with viewing angle as do the other approaches. This form of silhouette render is not only simple but also efficient. It does not require any calculations of dot products like the other approaches and it does not introduce more polygon primitives as in the quadrilateral extension approach.

### 3.1.4 Creases

Without adjacency information, our chosen method for crease detection became unfeasible. With silhouettes

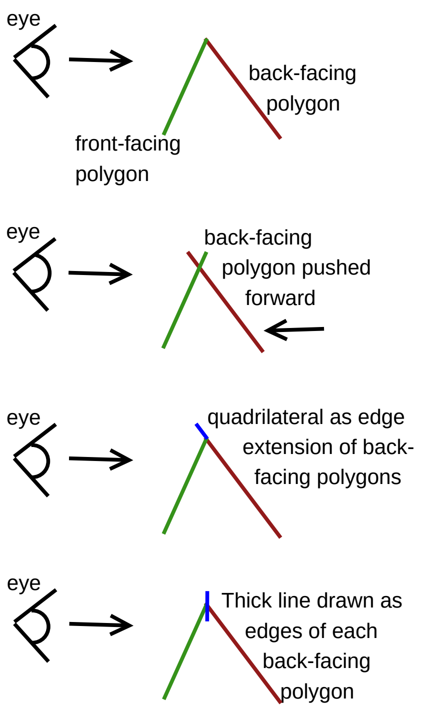

eye
back-facing polygon
front-facing polygon
eye
back-facing polygon pushed forward
eye
quadrilateral as edge extension of back-facing polygons
eye
Thick line drawn as edges of each back-facing polygon

Figure 3: In the top diagram, the normal rendering is shown with a back-facing polygon in red and a front-facing polygon in green. The second diagram shows the effect of pushing a back-facing polygon forward so it's edge sticks out in front of the front-facing polygon's edge. The third diagram, the back-facing polygon is extended with a quadrilateral, in blue. The fi nal diagram shows our simple approach of drawing thick edge lines (in blue) for all back-facing polygons.

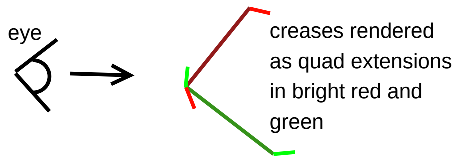


Figure 4: Raskar's approach is shown in this diagram, with a ridge being rendered as quadrilaterals (bright green and bright red) extended at an angle off of two front-facing polygon's joining edge. As the angle between the two polygons approaches 180 degrees, the quadrilateral extensions become hidden behind the polygons.

we were able to exploit the back-facing polygons to achieve our goals. With creases, this is not possible. In order to classify a crease properly as a ridge or a valley, surface normals of adjacent polygons are required to find the angle between the two polygons, as in figure [4]. Since we did not have this information we instead draw polygon boundary lines for all polygons. Unfortunately, this approach looses much of the benefits we were hoping for. Detail of model shape is not as clear because all edges are drawn. Most disappointing is the inclusion of edges joining two polygons on the same plane. These edges introduce a sense of division, geometrically speaking, when in fact there is none.

### 3.1.5 Less Jarring Edges

Creases and often silhouettes can sometimes look jarring. The choice of white for creases is quite common in technical illustration but we observed that this often *chops up* the models too much as opposed to conveying shape, especially when the underlying model colour differs greatly from white. In fact, as the model moves farther away from the viewer the white creases can often complicate the scene as seen in figure [5]. We introduce the idea of blending the colour of silhouettes and creases with the underlying model colour, an approach we did not find any mention of in the literature. This creates a less jarring visual style while maintaining the detailed shape information provided by silhouettes and creases.

We accomplish this by blending the crease colour (white) with the underlying shaded colour of the model at each vertex. In order to control the level of blending, we use a user-definable parameter which specifies how much white to add. We call this the *white addition value*. This value is added to each RGB colour compo-

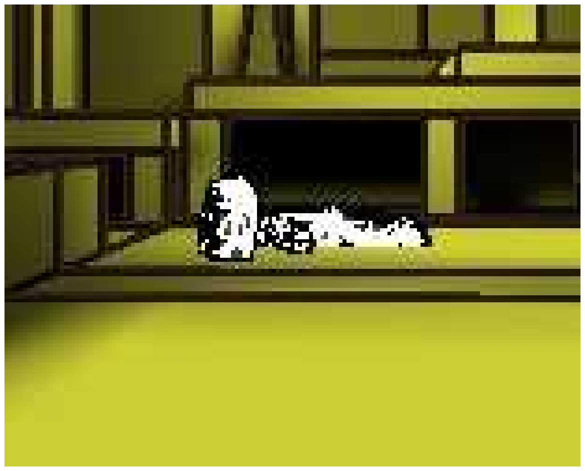


Figure 5: White creases can be too jarring as models move farther away from the viewer.

nent of the colour at a given point. In it's simplest form, this parameter is 1 and all creases will always be white. Choosing a lower value, such as 0.1, will retain more model colour and less crease colour. OpenGL then interpolates the colour of the edge at any point based on the colours assigned to it's two endpoints as seen in figure [12].

Even when crease colours were blended with the underlying model colour we still observed that models far away did not convey their shape very well. Creases become highly dense on the model surface the farther away it became, complicating the scene. We also observed that the simple cool-to-warm shading discussed in the next section communicated a decent amount of shape information by itself from far away. Using these realizations we introduced the idea of fading the colour of silhouettes and creases into the underlying model colour as a model moves farther away from the camera. This prevents the "cluttered" look of a model that is far away with lots of creases being rendered into a tiny 2D volume.

We implemented depth blending of edge colour with model colour using a user-definable parameter which we called the *blend threshold*. This parameter specifies the distance at which edges should fade completely into the model colour. At any given point the distance of a model to the viewer is calculated and the ratio of this distance over the *blend threshold* distance defines a percentage of the *white addition* value to add to the underlying model colour. This produces a very nice effect that keeps models clean as they move farther away yet

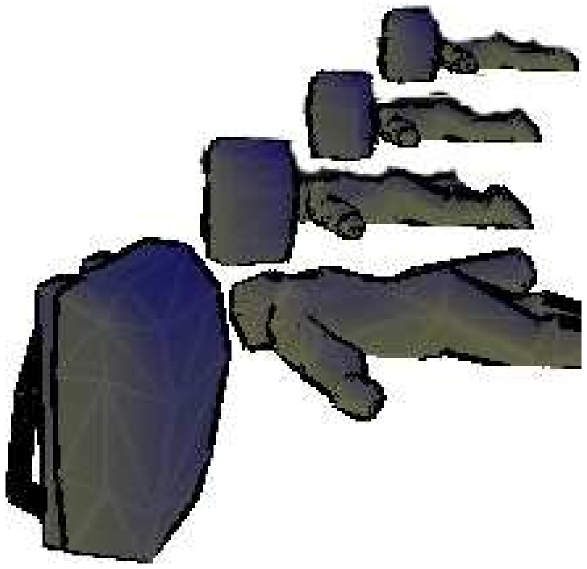


Figure 6: As the distance to the object increases, silhouettes and edge lines are blended with the object colour. Edge lines are clear and distinct when viewed up close, and do not clutter the model when viewed from far away.

still allows for clear edges when up close to a model, as seen in figure [6].

## 3.2 Shading

We implemented the technical illustration shading model as described in [Gooch et al. 1998]. Traditional diffuse shading sets surface irradiance $I$ of a point proportional to the angle between the light direction and the surface normal:

$$I = k_d k_a + k_d max(0, \vec{l} \cdot \vec{n})$$

where $k_a$ is the ambient illumination, $k_d$ is the diffuse illumination, $\vec{l}$ is the unit vector in the direction of the light source and $\vec{n}$ is the unit surface normal at that point. We cannot add edge lines in this model, because highlights would be lost in the well-lit areas while silhouettes would be lost in dark regions of the object. [Gooch et al. 1998] propose a shading model based on the perception that cool colours recede while warm colours advance. This shading model increases shape comprehension by restricting the range of colours available for shading so as to not interfere with the colour of edge and silhouette lines. The surface irradiance $I$ is calculated under this model by blending between two

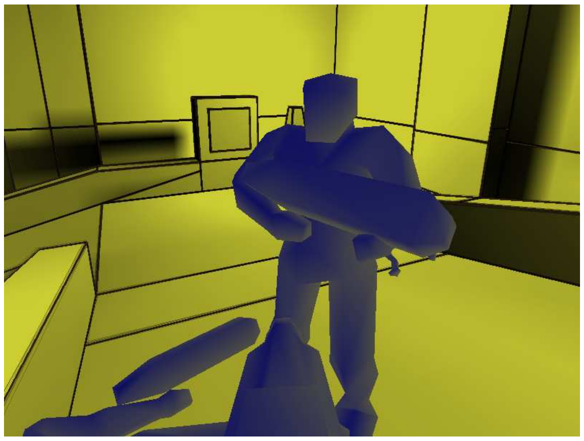


Figure 7: The object shaded with a cool-to-warm transition.

colours $k_{cool}$ and $k_{warm}$ according to:

$$I = (\frac{1+\vec{l}\cdot\vec{n}}{2})k_{cool} + (1 - \frac{1+\vec{l}\cdot\vec{n}}{2})k_{warm}$$

The model is rendered with approximately constant luminance tone, as shown in figure [7]. This lends for subtle shape information that does not require a large dynamic range, but renders the object in rather unnatural colours.

## 3.3 Splash-back

[Gooch et al. 1999] describe an adaptation of the above cool-to-warm shading that simulates the more dramatic shading effects sometimes used by artists. The effect is achieved when the reflected light from the left of the object produces a back-splash of light opposite the direct lighting source. We achieve this effect by adding the following multiplier to the cool-to-warm irradiance equation:

$$(\alpha|\cos\theta| + (1-\alpha))^p$$

Here, $\alpha$ and $p$ are free parameters that we have set to 0.76 and 0.78 respectively, as recommended in [Gooch et al. 1999]. The effect is quite pronounced, as seen in figure [8].

## 3.4 Shadows

Shadows are only drawn in technical illustration when they do not occlude detail. Objects typically do not self-shadow in this model, and shadows are only cast onto the ground plane. Shadows can help convey shape information about parts of the object that are otherwise occluded from view, as shown in figure [9]. It is however not as important for the shadow to be entirely accurate or realistic. We have implemented two types of shadow in Quake: the single hard shadow, and a soft shadow with a hard umbra and a hard penumbra.

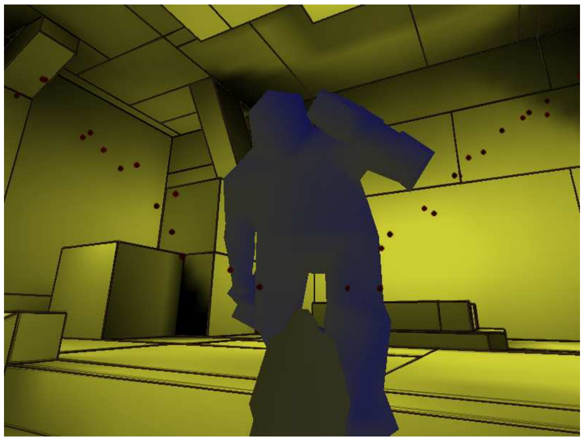


Figure 8: The splash-back term in the irradiance equation results in a more dramatic shading effect.

The shadow shape is achieved by projecting each polygon of the model onto the ground plane in a solid colour, in the direction opposite to the light source. Quake scenes are always lit using a single light, so the direction a shadow is cast is computed without much difficulty. The shadows are achieved by drawing three projections of the model onto the ground plane: The first two are drawn with a negative and a positive offset in the y direction, respectively, and with a lower alpha value, thus approximating the penumbra. Subsequently, the third projection is drawn on top of the first two, with a higher alpha value. It is scaled down a factor, and models the umbra. In our implementation, the colour, offset(figure [13]), scaling(figure [12]) and alpha value(figure [11]) of the shadow is configurable, and the drawing of the penumbra can be switched off in order to model a single hard shadow.

## 4 Results

We tested our rendering library on an Intel P4 2.66GHz with 512MB of RAM and a nVidia GeForce4 Ti 4200 graphics card. The Quake game itself has three time demos from various 3D environments and a larger demo,

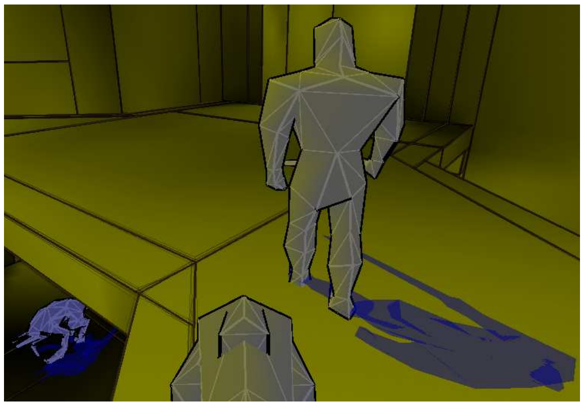


Figure 9: Shadows supply more information with one viewpoint. We can see from the shadow that the model is holding a staff, even though the staff is entirely occluded by the body of the model.

*BigAss1*, is obtainable from the Internet which is a much better stress-test. We use these four demos to evaluate the performance of our renderer.

Our initial attempts at recovering edge adjacency information, though unfruitful, did give some insights into the speed of the Quake engine. Building an edge adjacency data structure for each model's frame that we encounter on the fly, as well as edge retrieval from this data structure at the time of crease rendering slows the game down by 25% when measured in frames per second, on average. This is clearly a bit of a performance hit but considering the frame rates we were able to achieve, it did not present a problem:

| Demo1 | Demo2 | Demo3 | BigAss1 |
|---|---|---|---|
| 78.2 fps | 83.2 fps | 72.4 fps | 71.0 fps |

If the edge information could be properly gathered, it is quite clear from these numbers that detecting and rendering ridges and valleys in their own styles, as well as detecting silhouettes, would not be too much of a performance bottleneck at all. It is unfortunate that this was not possible since our results would surely have been much better had this worked.

Our method for silhouette rendering was a simplification of [Raskar 2001]'s approach which draws back-facing polygon edges with thick lines. The results of this method are quite pleasing, as seen in figure [10]. Creases, however, were not particularly impressive because we were unable to discern between proper ridges and valleys and just normal polygon edges. Our experimentation with colour blending edges with their underlying model colour improved things somewhat by reducing the jarring effects of rendering all edges regardless of classification. Our use of a distance metric to determine colour blending levels nullified the *cluttering* artifacts introduced in models rendered far from the viewer.

The cool-to-warm shading technique of [Gooch et al. 1998] proved to be very nice and easy to implement. The simplicity of the algorithm makes for good performance as well as visually appealing results as can be seen in figure [7]. The addition of [Gooch et al. 1999]'s newer splash-back term in the cool-to-warm shading also gave interesting results but ultimately proved to be less useful in conveying shape information since the resulting renderings were darker and more uniform in color across the models.

The shadow model described in [Gooch et al. 1999] as shown in figures [12], [11] and [13], provide more shape information in the scene, by essentially showing the viewer a profile of the object shape. The implementation is conceptually simple as it is just a projection of the model onto a plane. We feel that our approach serves its purpose well, and the soft shadowing technique generates visually pleasing results.

Nevertheless, this approach is not intended to produce geometrically accurate results. Shadows appear to float when the object is elevated from the surface, for example if the object is positioned on a step and its shadow is cast off the step, and shadows do not contact properly. Instead of casting shadow on a wall, the shadow will disappear into the wall. Still, this shadow model does not intend to be accurate, but merely to convey shape that would otherwise be obscured. Even in an interactive environment this may save the user from moving between several views excessively to gain adequate comprehension of the object.

Our final timing benchmarks with all of effects turned on including silhouette and edge rendering, shading, edge colour blending with distance, and shadows is as follows:

| Demo1 | Demo2 | Demo3 | BigAss1 |
|---|---|---|---|
| 98.7 fps | 96.3 fps | 90.9 fps | 87.5 fps |

These numbers include the optimizations we made to our code which take advantage of faster OpenGL calls as outlined in [Blythe et al. 1999]. In all cases, we made our best attempt to write concise and optimized code in

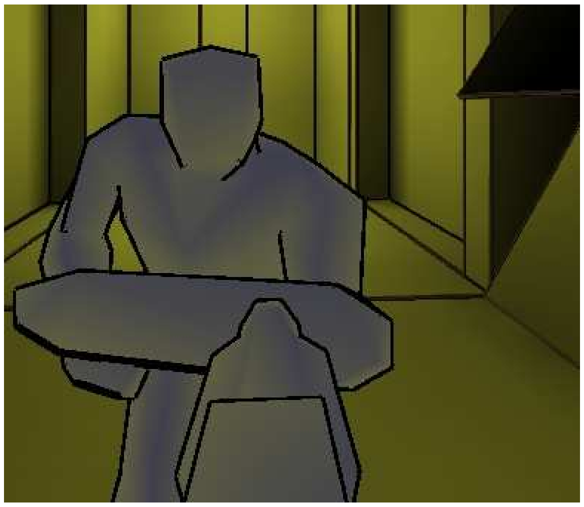


Figure 10: Our method of rendering silhouettes by drawing back-facing polygons with thick edge lines that peek out from behind front-facing polygons.

order to eek out every last frame per second possible.

## 5 Future Work

The technical illustration style for Quake could still be extended with many improvements. The edge adjacency information for models could be properly computed by either using improved models that are free from inconsistencies, rounding vertex locations so that more polygon edges coincide, or weeding out degenerate polygons. The camera position and direction could be recovered by further exploring the Quake source code and the OpenGL mapping. The neighbouring information coupled with the camera information could then be used to implement ridges and valleys as described by [Raskar 2001].

Furthermore, the Quake walls and sprites may be modified for a more technical look. Environment and texture maps could be employed to render the scene in a style that is a compromise between the technical and the more realistic method of rendering. Texture colour could be blended with cool-to-warm shading to sacrifice some shape clarity for object surface detail that is otherwise not represented by the surface shape.

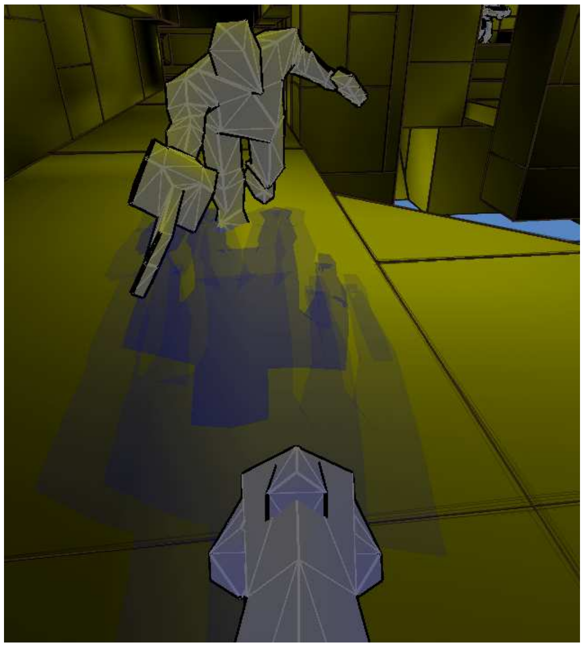


Figure 11: A lower alpha value can be used to create the suggestion of a shadow

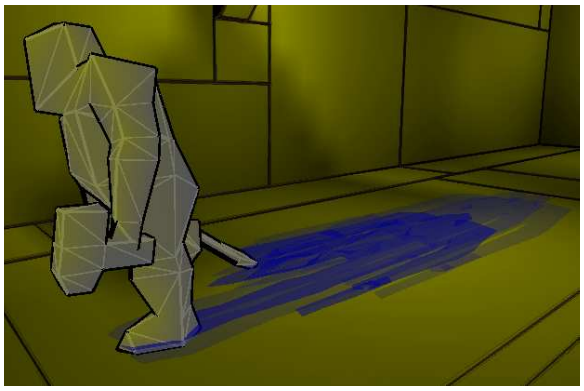


Figure 12: Shadows are drawn here with a large scale factor, creating a somewhat gloomy artistic effect.

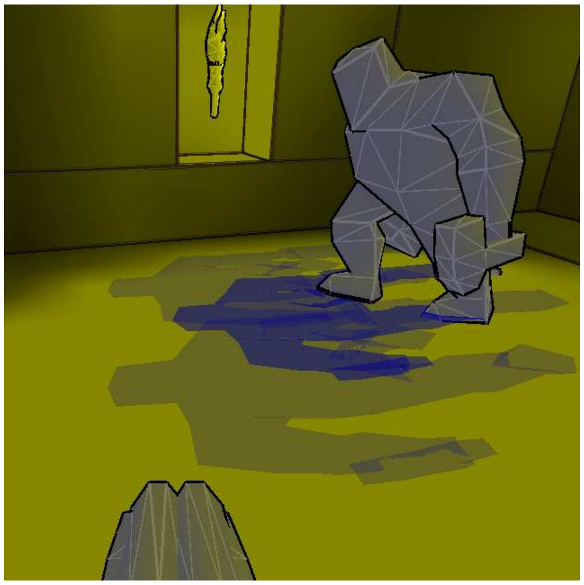


Figure 13: The penumbra is drawn here with a large offset. The shadow gives the appearance of being cast by two lights: the torch visible on the wall, and a light behind and on the right of the camera, outside the viewing frame.

## 6 Conclusions

We have shown that better shape information can be conveyed through the use of shading and edge rendering even on basic models represented as ”polygon soups”. Even with no camera direction or position, and no edge adjacency information for the models it is possible to render scenes that communicate more overall detail than that of photorealistic approaches. We have explored a novel approach to handling silhouettes and creases that allows for blending of edge colour with the underlying shaded model colour. This gives results that are less jarring then the classical black silhouettes and white creases, yet maintains an appropriate level of shape detail. Finally, we outline an approach to fade creases and silhouettes of a model based on the distance the model is from the viewer, which is novelly pimpin‘.

## References

BLYTHE, D., GRANTHAM, B., MCREYNOLDS, T., AND NELSON, S. R. 1999. Advanced Graphics Programming Techniques Using OpenGL. *SIGGRAPH ’99 Course*.

GOOCH, A., GOOCH, B., SHIRLEY, P., AND COHEN, E. 1998. A non-photorealistic lighting model for automatic technical illustration. In *Computer Graphics*. ACM Siggraph ’98 Conference Proceedings.

GOOCH, B., SLOAN, P.-P., GOOCH, A., SHIRLEY, P., AND RIESENFELD, R. 1999. Interactive technical illustration. ACM SIGGRAPH, 31–38. ISBN 1-58113-082-1.

ID SOFTWARE. 1992. Quake. `http://www.idsoftware.com`.

ILIE, A. 2003. Non-photorealistic rendering techniques for a game engine. `http://www.cs.unc.edu/~adyilie/comp238/Final/Final.htm`.

LAKE, A., MARSHALL, C., HARRIS, M., AND BLACKSTEIN, M. 2000. Stylized rendering techniques for scalable real-time 3d animation. In *Non-Photorealistic Animation and Rendering 2000 (NPAR ’00)*.

MARKOSIAN, L., KOWALSKI, M. A., TRYCHIN, S. J., BOURDEV, L. D., GOLDSTEIN, D., AND HUGHES, J. F. 1997. Real-time nonphotorealistic rendering. *Proceedings of SIGGRAPH 97* (August), 415–420. ISBN 0-89791-896-7. Held in Los Angeles, California.

MOHR, A., BAKKE, E., GARDNER, A., HENNMAN, C., AND DUTCHER, S. 2002. NPRQuake. `http://www.cs.wisc.edu/graphics/Gallery/NPRQuake/`.

RASKAR, R., AND COHEN, M. 1999. Image Precision Silhouette Edges. In *Proc. 1999 ACM Symposium on Interactive 3D Graphics*.

RASKAR, R. 2001. Hardware support for non-photorealistic rendering. *2001 SIGGRAPH / Eurographics Workshop on Graphics Hardware* (August).

SCHREINER, D., SCHREINER, D. E., AND SHREINER, D. 1999. *OpenGL Reference Manual: The Offi cial Reference Document to OpenGL, Version 1.2*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

WOO, M., DAVIS, AND SHERIDAN, M. B. 1999. *OpenGL Programming Guide: The Offi cial Guide to Learning OpenGL, Version 1.2*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.