Stylized Rendering Techniques For Scalable Real-Time 3D Animation

Adam Lake Carl Marshall Mark Harris† Marc Blackstein
Graphics Algorithms and 3D Technologies Group (G3D)
Intel Architecture Labs (IAL)
†University of North Carolina at Chapel Hill

"We're searching here, trying to get away from the cut and dried handling of things all the way through—everything—and the only way to do it is to leave things open until we have completely explored every bit of it."

–Walt Disney

Abstract

Researchers in nonphotorealistic rendering (NPR) have investigated a variety of techniques to simulate the styles of artists. Recent work has resulted in methods for pen-and-ink illustration, pencil sketching, watercolor, engraving, and silhouette edge rendering. This paper presents real-time methods to emulate cartoon styles. We also present variations on a texture mapping technique to achieve real-time pencil sketching. We demonstrate our method of inking silhouettes, material and mesh boundaries, and crease edges. In addition, we present techniques for emphasizing motion of cartoon objects by introducing geometry into the cartoon scene. The rendering system is integrated with an animation system and a runtime multi-resolution mesh (MRM) system to achieve scalability, ensuring real-time performance on any platform. Such solutions allow us to take advantage of evolving hardware in order to make nonphotorealistic animation and rendering achievable on low- and high-end consumer platforms. All of the techniques described can be applied to models created with standard modeling tools and require no additional mark-up information from the modeler.

CR Categories and Subject Descriptors: I.3.3 [Computer Graphics]: Picture/Image Generation; I.3.5 [Computer Graphics]: Three-Dimensional Graphics and Realism – Color, Shading, Shadowing, and Texture.

Additional Key Words: real-time nonphotorealistic animation and rendering, silhouette edge detection, cartoon rendering, pencil sketch rendering, stylized rendering, cartoon effects.

1 Introduction

Recent advances in consumer-level graphics card performance are making a new domain of real-time 3D applications available for commodity platforms. Since the processor is no longer required to spend time rasterizing polygons, it can be utilized for other effects, such as the computation of subdivision surfaces, real-time physics, cloth modeling, realistic animation and inverse kinematics.

{adam.t.lake, carl.s.marshall, marc.s.blackstein}@intel.com
† harrism@cs.unc.edu

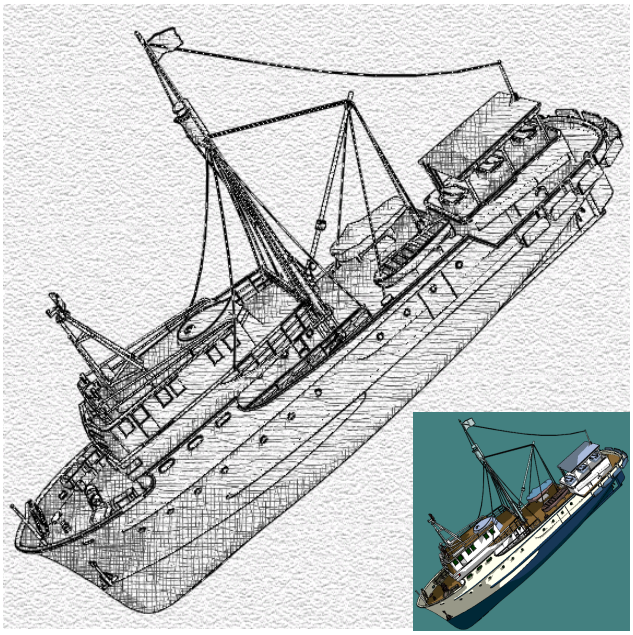

Figure 1: An example of a pencil sketch style rendering. Inset: the same model rendered in the cartoon style.

Another use of the graphics processor and CPU is to create a stylistic look for the rendered imagery. In this paper we will introduce our painting and inking system, a nonphotorealistic rendering system that runs in real time on today's PC platforms. In addition, it is integrated with a multi-resolution mesh system that can be used to achieve guaranteed frame rates and optimal use of processing resources in a system that scales both to new hardware and to models of increasing complexity. Because of the rapidly increasing demand for greater visual fidelity, it is critical to integrate such solutions into any real-time graphics system that needs to run across machines with varying levels of performance.

Uses: Nonphotorealistic rendering is capable of broadening our ability to communicate thoughts, emotions, and feelings through computers. Artists have learned that by making images look less photorealistic they enable audiences to feel more immersed in a story [McCl93]. Our primary interest is to enable the use of real-time stylized rendering to create a more compelling storytelling experience, whether that be through dynamic images in an on-line book, interactive technical illustrations, or immersive cartoon experiences. Another use is the rapid prototyping of cartoon animations, coloring, and shot angles before allocating valuable artistic talent for painting and inking cels in an animation environment.

Results: In this paper we introduce new real-time algorithms for cartoon shading, pencil sketching, and silhouette edge detection and rendering. We also present a new technique for generating motion lines to emphasize motion in 3D cartoon renderings. It is important to note that the system does not require any additional mark-up information from the author. *These shading techniques*

require only per-vertex positions, normals, material properties, texture coordinates, and connectivity information. We integrate these new rendering techniques with a character animation system and a multi-resolution mesh to provide scalability.

Organization: Our real-time rendering architecture is composed of two core components. Section 2 discusses previous work on these components. The *Painter*, which is used for determining shading information to 'fill' the polygons, is discussed in Section 3. Section 4 presents the *Inker,* highlights our silhouette edge detection methods and gives details on a curvature-driven approach for rendering silhouette edges that has yielded pleasing results. Section 5 presents a method for creating an effect used by animators to indicate real-world characteristics like object motion. Finally, we conclude with a discussion of the performance characteristics and avenues for future work in real-time stylized rendering techniques. We use *stylistic rendering* and *nonphotorealistic rendering* interchangeably. A video accompanies the paper that demonstrates our methods.

2 Previous Work

First, we discuss previous work on real-time and non real-time shading methods. We then discuss previous work on silhouette edge detection and rendering.

2.1 Previous Work In Stylistic Rendering

Recent years have seen strong interest in real-time NPR methods. Our variation of the standard rendering equations to achieve cartoon shading is similar to a variation presented in [Gooc98] that creates a warm-to-cool transition for technical illustration. [Rade99] achieves a variety of NPR effects by varying the Gooch warm-to-cool shading method with each surface. Our cartoon shading uses a similar modification to the lighting equations, but also utilizes hardware accelerated texture mapping to simulate the limited color palette cartoonists use for painting cels. [Deca96] uses a combination of multi-pass rendering and image processing techniques to create cartoon-style images. Decaudin's thresholding of surface color via the value of $n \cdot l$ is similar to our approach. Example applications that use NPR for rendering free-form design in a modeling system are [Igar99] and [Zele96]. Since their focus is on user interface design and not real-time animation, they do not include multi-resolution mesh capability or animation support. [Digi97], [Ment99], and [Ligh99], are commercial ray-tracing packages that support stylized rendering. Each supports cartoon shading but does so via ray tracing and does not take advantage of fast texture mapping hardware. [View98] implements several styles for rendering, and [Meta99] allows animated models to be rendered in a cartoon style. Our system renders animated models in multiple styles and is integrated with a multi-resolution mesh system that improves scalability.

2.2 Previous Work In Silhouette Edge Detection (SED)

SED is a technique that has been used in NPR and technical illustration for many years. [Mark97] provides a real-time rendering SED implementation that is built upon Appel's hidden-line algorithm [Appe67]. Markosian's algorithm uses a technique to find silhouettes that deliberately sacrifices accuracy for speed, but is limited by the fact that it does not take advantage of current PC graphics hardware to resolve visibility using the z-buffer. Raskar et al demonstrate several image-space metrics to find the silhouette edges of polyhedral models while trying to maintain a specified pixel thickness of the lines used to render the silhouettes [Rask99]. On current generation systems this method is not suitable due to slow frame buffer reads.

[Gooc99] demonstrates several ways to find the silhouettes of a model. In that work, three methods are presented: use of a Gauss map, use of `glPolygonOffset` with a scale factor, and a straightforward method of testing every edge. The Gauss map introduces inconsistent line thickness of silhouettes but creates interesting effects. `glPolygonOffset` moves a polygon a small distance toward the eye to force coincident z-values apart [Woo99], and requires rendering the model twice, a potentially significant performance cost on consumer hardware. Our method provides a fast and accurate system to find the important edges of a mesh while maintaining frame-to-frame coherence. We use a straightforward technique for SED, in which each edge of a model is checked to determine if it is one of the important edges.

3 Stylistic Shading

We call our shader the Painter due to its similarity to the cel animator's process of *painting* an already *inked* cell. A content author can set the rendering style based on the look and feel she is trying to achieve via the rendering API. By varying the shadow and highlight parameters, the number and styles of textures used, and the weighting factors for our shading calculations, we are able to produce a variety of styles.

3.1 Cartoon Shading

Cartoon characters are intentionally "two-dimensional". Animators deliberately reduce the amount of visual detail in their drawings in order to draw the audience into the story and to add humor and emotional appeal. [McCl93] demonstrates that reducing detail in the features of a character – usually facial – allows a wider audience to easily relate to the character. Rather than shading the character to give it a three-dimensional appearance, the cartoonist typically uses solid colors that do not vary across the materials they represent. Often the artist will shade the part of a material that is in shadow with a color that is a darkened version of the main material color. This helps add lighting cues, cues to the shape and context of the character in a scene, and even dramatic cues. The boundary between shadowed and illuminated colors is a hard edge that follows the contours of the object or character. We refer to this technique as *hard shading,* and examples are shown in Figures 3, 16, 17, and 18.

Our system allows a 3D model to be drawn to look just like a traditional 2D cartoon, and to be viewed interactively. The technique relies on texture mapping and the mathematics of traditional diffuse lighting.

Rather than smoothly interpolating shading across a model as in Gouraud shading [Gour71], our hard shading technique finds a transition boundary and shades each side of the boundary with a solid color. The lighting equation used to calculate the diffuse lighting at the vertices for both smooth shading and our cartoon shading is shown in Equation 1.

$$C_i = a_g \times a_m + a_l \times a_m + (Max\{\overline{L} \cdot \overline{n}, 0\}) \times d_l \times d_m$$

Equation 1.

Here, C_i is the vertex color, a_g is the coefficient of global ambient light, a_l and d_l are the ambient and diffuse coefficients of the light source, and a_m and d_m are the ambient and diffuse coefficients of the object's material. $\overline{L}$ is the unit vector from the light source to the vertex, and $\overline{n}$ is the unit vector normal to the

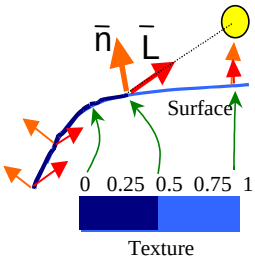

Figure 2: Generation of texture coordinates from $\overline{L} \cdot \overline{n}$. In this case, the shadow boundary occurs at the point where $\overline{L} \cdot \overline{n}$ equals 0.5.

Figure 3: Olaf rendered using cartoon shading.

surface at the vertex. $\overline{L} \cdot \overline{n}$ results in the cosine of the angle formed between the two vectors.

The math for our cartoon shading technique is essentially the same. Instead of calculating the colors per vertex, however, we create a texture map of a minimal number of colors (in most cases only two – one for the illuminated color and one for the shadowed color). The main color of the texture map is calculated by replacing the dot product term from equation 1 with a value of one (which is equivalent to an angle of zero degrees, and normally happens when the light is directed right at the vertex). The shadow color of the texture map is calculated by replacing the dot product term in Equation 1 with a value of zero (so that it is made up of only ambient illumination). The resulting texture is a one-dimensional texture map, and in the simplest case, it has only two texels – one for each color. This texture is computed once per material and stored ahead of time.

The boundary between lit and shadowed colors on the model depends, just as the color calculated in smooth shading, on the cosine of the angle between the light and normal vectors. At each frame, we calculate $Max\{\overline{L} \cdot \overline{n}, 0\}$ for each vertex, and use these per-vertex values as texture coordinates for the precomputed one-dimensional texture map.

Figure 2 demonstrates how the light position and normal direction are used to index into the one-dimensional texture map. Note that as $Max\{\overline{L} \cdot \overline{n}, 0\}$ crosses 0.5, the index into the texture map chooses a different color. The transition between the illuminated and shadowed regions may appear jagged if viewed closely. 3D graphics APIs provide texture-filtering modes that can be helpful in this case, depending on the model and the viewing conditions. One mode chooses the nearest texel to a pixel ("nearest"), while another interpolates among a set of texels nearest to the pixel ("linear"). We find that using the nearest filter mode results in acceptable results, but a jagged color transition can be smoothed using the linear filter. This mode smoothes the color boundary, but the smooth transition may be too wide if the polygons on which the transitions occur are too large in screen space. The preprocess and runtime portions of our cartoon shading algorithm follow.

ALGORITHM Cartoon Shade

Preprocess:

1. Calculate the illuminated diffuse color for each material:
   $C_i = a_g \times a_m + a_l \times a_m + d_l \times d_m$.
2. Calculate the shadowed diffuse color:
   $C_s = a_g \times a_m + a_l \times a_m$
3. For each material, create and store a one-dimensional texture map with two texels using the texture functionality provided by a standard 3D graphics API. Color the texel at the u=1 end of the texture with C_i and the texel at the u=0 end of the texture with C_s.

Runtime:

1. Calculate the one-dimensional texture coordinate at each vertex of the model using $Max\{\overline{L} \cdot \overline{n}, 0\}$, where $\overline{L}$ is the normalized vector from the vertex to the light source location, $\overline{n}$ is the unit normal to the surface at the vertex, and $\overline{L} \cdot \overline{n}$ is their vector inner (dot) product.
2. Render the model using a standard graphics API with lighting disabled and one-dimensional texture maps enabled.

Notice that we must do a dot product at each vertex for every frame. While current processors are fast enough to handle this math at interactive rates on scenes with reasonable complexity, hardware lighting could potentially be used to speed these computations. Since 3D graphics accelerators with hardware lighting compute $\overline{L} \cdot \overline{n}$ per vertex anyway, a hardware pathway between the lighting and texturing subsystems and an associated automatic texture-coordinate generation mode would be beneficial for cartoon shading. A variety of shading techniques, both photorealistic and nonphotorealistic, could also benefit from such hardware features [Heid99]. Our texture-mapped method is equivalent to per-pixel color thresholding. Without texture mapping we would need either hardware-supported per-pixel procedural shading or a thresholded variation of hardware Gouraud shading to achieve the same fast results.

3.2 Variations On Cartoon Shading

The cartoon shading technique described in section 3.1 can be used to render in a variety of different styles. The choice of colors alone has a strong effect on the appearance or style of the cartoon. The Painter provides the user with the ability to override the automatic computation of texture colors so that she may assign custom colors. By using higher-resolution texture maps, additional effects can be created. In the case of a two-color texture, using a higher-resolution texture can provide flexibility in the location of the shadow boundary. If an eight-texel texture is used, for example, with seven of the eight texels set to the shadow color and one to the illuminated color, most of the character will appear in shadow, with small portions illuminated. A dark comic book feel can be achieved in this way by setting the shadow color to black or to the background color, and the illuminated color to something much brighter. An example of this, in the spirit of a style demonstrated in [Hoga81], is shown in Figure 16. A look similar to *double-source lighting*, also discussed by Hogarth, can be created in a similar manner by setting the colors at both ends of a texture map to an illuminated color, and the colors in the middle of the texture to a shadowed color.

Another effect that we have implemented is a *shadow-and-highlight* style. This is similar to standard hard shading, but uses

three colors instead of just two when building the one-dimensional texture. We use a higher resolution texture with eight or sixteen texels in which half of the texels are set to the shadow color. At the illuminated end of the texture, one or two of the texels are set to a highlight color, while the rest of the texels are set to the illuminated material color. The highlight color can be chosen automatically by multiplying the material color by a *highlight factor.* Shadow-and-highlight shading is shown in figures 16 and 17. These highlights are view-independent, "diffuse" highlights. To render a specular cartoon highlight, we add a second texture using multitexturing that is blended with the first texture and whose texture coordinates come from a view-dependent specular computation rather than the diffuse computation shown above. Multiple simultaneous textures can be rendered quickly with current multitexturing PC hardware. Multitexturing is the capability provided in most new graphics hardware to map two or more textures onto a polygon in a single rendering pass.

The Painter also allows the use of color gradient textures. The user simply specifies the texture resolution and the starting and ending colors of the texture, and the Painter calculates a gradient to fill the texture. This technique can be used as an alternative to the Phong shading provided by the graphics API, by using a high resolution gradient texture (128 or 256 texels with linear interpolation) with the shadowed color at one end and the illuminated color at the other. In fact, the texture-mapped shading provides a spectrum of approximations to diffuse illumination, with single-color flat shading (one texel value) at one end of the spectrum, and smooth diffuse shading at the other. By providing the user with just a few parameters, our Painter offers a large variety of stylized and cartoon rendering effects.

3.3 Pencil Sketch Shading

In this technique, we extend the texture mapping method of the cartoon shading algorithm to handle two-dimensional textures. As with cartoon shading, we use the calculation of $\overline{L} \cdot \overline{n}$ per vertex. Instead of selecting texels in a one-dimensional texture, the algorithm uses $\overline{L} \cdot \overline{n}$ to select a texture of appropriate density. The textures are created by randomly selecting from the pencil strokes in Figure 6, and placing them on the texture with random spacing to avoid coherence (Figure 7). Regions receiving less light have a higher density of pencil strokes. In regions of highest density (lowest light), we combine pencil strokes in the horizontal and vertical directions. We use multitexturing to render the paper texture onto the model to make it look sketched on paper. If the model is scaled, the paper texture has to be scaled proportionately to maintain coherence with the background paper texture. As in the cartoon shader, the user may choose a non-linear mapping between $\overline{L} \cdot \overline{n}$ values and the density of the chosen texture. For example, $\overline{L} \cdot \overline{n}$ values from 0 to 0.5 might map to one texture while values from 0.5 to 1 map uniformly to the remaining textures. In the cartoon shader we were able to rely on the texture mapping hardware and graphics API functionality to determine the boundary between shadowed and illuminated regions. Here, a subdivision algorithm is required to break any polygon that requires more than one texture applied. We group the polygons based on which texture needs to be applied and render all polygons receiving the same texture at once to minimize API overhead. The results of this technique can be seen in Figures 1, 4, and 20.

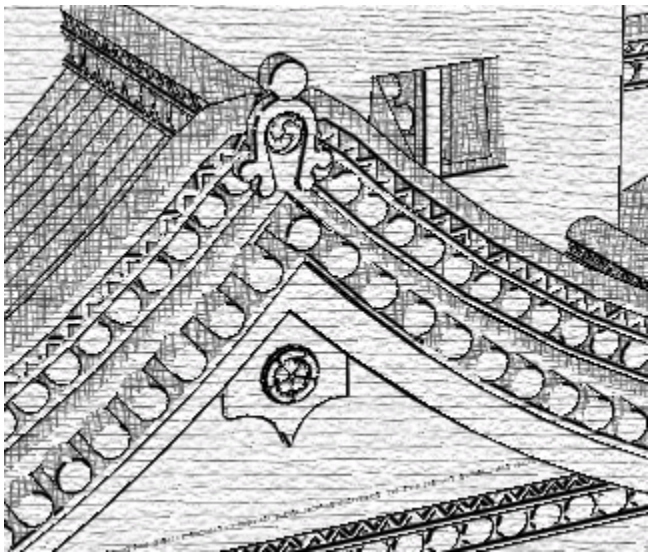

Figure 4: A pencil-sketch rendering of a 3D castle.

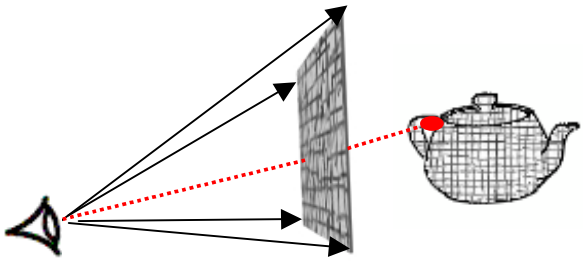

Figure 5: To generate texture coordinates, each texture is projected from the viewport onto the model.

Since the $\overline{L} \cdot \overline{n}$ values calculated above are used to choose *textures* rather than *texels*, we must generate the texture coordinates. One method is to 'project' a texture onto the model through the viewport and use normalized *(x, y)* device coordinates as the *(u, v)* texture coordinates, as shown in Figure 5. This can be done for each frame or once as a preprocessing step.

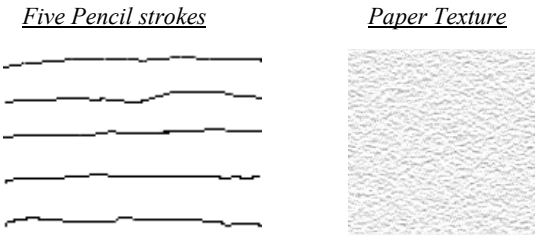

Figure 6: The pencil strokes and paper texture combined for the pencil sketch shader.

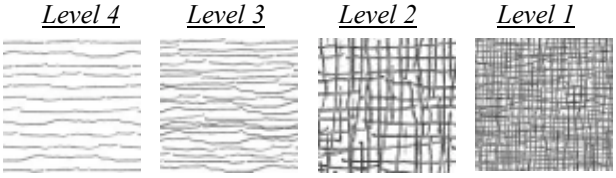

Figure 7: Textures used for pencil sketch shader. As polygons face away from the light source, lower level textures are used.

ALGORITHM Pencil Sketch

Preprocess:

1. Read in a set of pencil strokes. These are bitmap images as in Figure 6.
2. Construct a set of *n* two-dimensional textures by randomly selecting
3. pencil strokes and arranging them uniformly along the v direction with u spacing inversely proportional to density. The highest-density textures may combine orthogonal pencil strokes, and the lowest-density textures may be blank. Select the starting *v* position randomly to avoid coherence in the texture.

Runtime:

1. Draw the background using an orthographic projection of a large quadrilateral texture mapped with the paper texture.
2. For each vertex, calculate $\overline{L} \cdot \overline{n}$ where $\overline{L}$ is the light vector and $\overline{n}$ is the vertex normal and use this value to quantize into one of *n* bins. The bin corresponds to the texture that will be associated with this vertex. Back facing polygons are not submitted. For example, in a system with three bins we would have the intervals [0,a], (a, b], and (b, 1.0] where a and b are adjustable values with the restriction a<b and 0<=a, b<=1.0.
3. Build *n* face lists from the set of polygons. For each polygon, if each vertex belongs to the same bin then append the polygon to the appropriate face list. Otherwise, subdivide the polygon using linear interpolation (see the following paragraph for details).
4. If using runtime *(u, v)* determination, determine position on the viewport and the associated texture coordinate for the vertex.
5. For each set of face lists, use multitexturing to render the polygons using the appropriate pencil texture, from Figure 7, and paper background, from Figure 6.

In the case where the vertices of a face belong to different bins, we subdivide the face into discrete triangles. To accomplish this, we linearly interpolate along each of the edges to find new, temporary vertices whose $\overline{L} \cdot \overline{n}$ value is exactly the border between two adjacent bins. For example, if v_1 is in bin *i* and v_2 is in bin $i + j$, we create *j* interpolated vertices along the edge v_1v_2. For each new vertex, there will exist a vertex on a different edge that "splits" the same two bins. By connecting corresponding vertices, we have divided the triangle into subregions, each belonging to a different texture bin.

The viewport mapping technique preserves the hand-drawn feel of the textures, and thus is well suited for capturing static images. However, during animations, this mapping technique causes a "shower door effect" in which the model appears to be swimming through the texture [Meie96]. In such circumstances, it may be desirable to fix the texture coordinates during preprocessing, thereby trading a flat appearance for smooth animation sequences.

4 Stylistic Inking

The Painter enables a variety of imaginative rendering styles. For some artisitic styles, such as cartoon rendering, the shading provided by the Painter is incomplete. Most cartoon drawings accent the details and silhouettes of characters with ink lines. Our Inker complements the Painter by detecting and rendering the important edges of a polygonal model. These edges can be rendered using traditional straight line segments or using texture-based lines.

The primary technology in the Inker is silhouette edge detection (SED). A silhouette is an edge shared by one front- and one back-facing polygon. Artistically, the silhouette of an object or character helps to define space that helps to separate the character from the rest of a scene while also providing intensity, weight, and emotion [Hoga81]. The Inker detects and draws the silhouettes of a 3D model in real time. In addition to silhouettes, we find edge lines that mark other key features of a model, such as crease and border edges. A *crease* edge is detected when the dihedral angle between two faces is greater than a given threshold. *Border* edges are those that lie on the edge of a single polygon, or that are shared by two polygons with different materials.

4.1 Silhouette Edge Detection

An edge is marked as a silhouette edge if a front-facing and a back-facing polygon share the edge as in figure 8. We find silhouettes in each frame by taking the dot products of the face normals of the two faces adjacent to an edge with the viewing vector (see Equation 2), and comparing the product of these two dot products with zero. If the result of this computation is less than or equal to zero, the edge is a silhouette edge and it is flagged for rendering. The Inker detects and draws important edges using the following algorithm. The result is shown in Figure 18.

ALGORITHM SED

Preprocess:

1. Allocate memory for an edge list including an edge flag for each edge to indicate whether it is a border, crease, or silhouette edge.
2. Iterate over the faces of the model and create a unique edge list using a hash table. The hash function for an edge is the sum of the two vertex indices of the edge.
3. Set border flag for edges with only one neighboring face or two neighboring faces with different materials.
4. For non-deformable geometry, set crease flag for edges for which the dihedral angle between the two neighboring faces is greater than the crease threshold.

Runtime:

1. If necessary, calculate face normals.
2. For deformable meshes (i.e., animation), detect crease edges from face normals and set crease flags.
3. Detect silhouette edges via Equation 2 and set silhouette flags.
4. Traverse the edge list and render edges whose edge flag is set.

The visibility of important edges is solved via the z-buffer, as provided by the graphics API. The Inker operates most optimally on static geometry that is not animated or deformable, since the edge list – including crease and border edges – can be created at authoring or load time. To Ink non-static geometry, we must re-detect crease edges every frame.

4.2 Rendering Important Edges

Silhouettes are rendered using line segments, the width of which can be adjusted according to lighting parameters [Gooc99], a distance metric, or a user-defined parameter. As an object moves away from the eye, the silhouette line width can be reduced with increasing distance. Silhouettes can be rendered using any color or texture map. We have observed that ink lines in cartoons are traditionally black or a darker shade of the material color.

$$(faceNormal_1 \bullet eyeVect) * (faceNormal_2 \bullet eyeVect) \le 0$$

Equation 2.

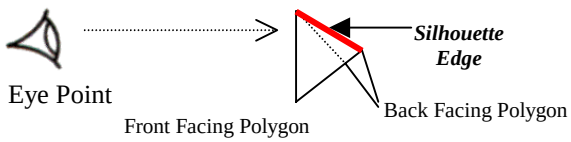

Figure 8: Silhouette edge detection. A silhouette edge is an edge between a front-facing and a back-facing polygon.

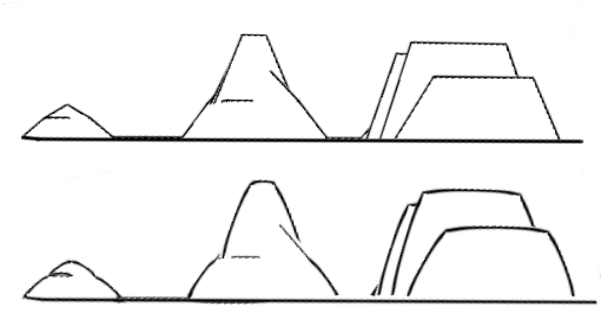

Figure 9: Top: a simplified terrain model rendered with basic line segments. Bottom: the terrain model rendered using curvature-driven textures.

Figure 10: Strokes used for stylized silhouette edges

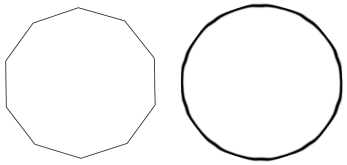

Figure 11: Left: a set of silhouettes as determined from the model of a 3D sphere. Right: the result using curvature-driven texture strokes.

4.3 Stylized Silhouette Edges

In addition to straight ink lines, the Inker allows texture mapping of important edges of a mesh to add artistic styles to edge lines as first shown in [Mark97]. We extend the method of rendering edges with textures used by Markosian et al to support textures that follow the curvature of the edges. To illustrate this method, we show a 2D "terrain" model in Figure 9 composed of many edges, where each edge is rendered once as a straight line segment, and again as a curved texture. Figures 11 and 12 demonstrate the technique on other 3D models.

We use three textures to implement curvature-driven textured edges. These textures are a straight stroke, a "rightward" stroke, and a "leftward" stroke as shown in Figure 10. A straight texture is used if the edge $\bar{E}_1$ forms an angle of d degrees or less with its successor edge $\bar{E}_2$; otherwise either a "leftward" or "rightward" texture is selected, as shown in Equation 3. We calculate the cosine of the angle d from the dot product of the vector along edge $\bar{E}_1$ with the vector along edge $\bar{E}_2$. In order to efficiently determine the successor of an edge, we create a graph whose vertices correspond to the vertices of the model, and insert edges into the graph as silhouette edges are found by the SED algorithm.

$$\bar{E}_1 \bullet \bar{E}_2 = \begin{cases} \leq -\cos(d) & \text{then apply } \textit{left} \text{ texture} \\ -\cos(d) < 0 < \cos(d) & \text{then apply } \textit{straight} \text{ texture} \\ \geq \cos(d) & \text{then apply } \textit{right} \text{ texture} \end{cases}$$

Equation 3.

Once all edges have been added, we perform a depth-first search on the graph to find the successor of each silhouette edge. This takes O(|E| + |V|) time, but in practice, |E| will be proportional to |V| because a silhouette edge will usually have at most two neighboring edges. Thus, the average cost of finding the successor of one edge will be O(1). The number of textures used is variable, and more complex texture-selection metrics can be applied.

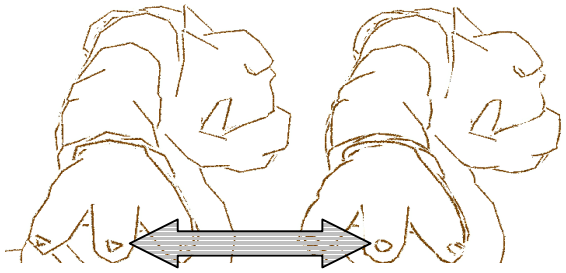

Figure 12: Left: a straight texture map applied to silhouettes of Olaf. Right: shows the curvature-driven texture maps applied to Olaf. Notice that the curvature of the fingernails is maintained.

In order to texture an edge, we must first construct a quadrilateral on which to map a texture. Quadrilaterals are oriented to face the eye. We apply the texture previously selected for each edge to the quadrilateral representing the edge. Figure 12 shows our curvature-driven texture mapping applied to a model using a brown crayon texture. Notice that in the areas of high curvature, in particular the fingernails, the effects of this technique are visible. We find that this style of edge drawing presents a more pleasing, higher-detail appearance even to models with a fairly small number of polygons. For edges that are small we can increase the width of the quadrilateral. This might be the case if the object is translated far from the eye. Also, various artistic effects can be achieved by randomizing the width of the quadrilaterals.

While we have found this technique to be suitable for models rendered only as silhouettes with polygons colored the same as the background color, it is not effective when also rendering interior polygons with their original material color. Several new problems are introduced, including the overlapping of the textured quadrilaterals and the polygons of the mesh. Offsetting the textured quads away from the mesh introduces gaps between the quad and mesh that must be filled. In addition, when the quads project to sharp angles on the view plane, gaps in the silhouette edges may result. From a distance or with a highly-tessellated mesh these problems are not so disturbing. More robust and general solutions should be pursued, including the use of 2D image-based stylized silhouette edge rendering techniques.

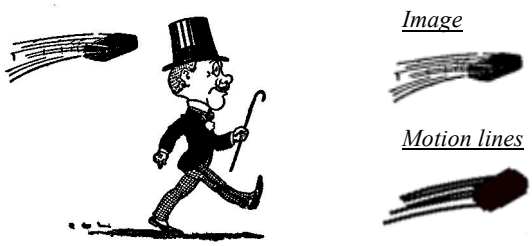

Figure 13: Left: an artist's depiction of a brick flying towards a man [Lutz20]. Top right: the artist's image of a brick flying. Bottom right: motion lines generated using our technique.

5 Motion Lines

Many cartoons introduce nongeometric information into the scene in order to indicate that a character or object is moving quickly, a surface is glossy, an impact has occurred, and other effects. Motion lines are used in cartoons to show rapid motion or for artistic style [Lutz20]. Figure 13 shows a brick flying toward a character walking down a street. The lines drawn behind the brick convey the sense that the brick is in motion. Without these motion lines, there would be no context for the brick in the scene. [Hsu94] and [Masu99] show previous examples of techniques for these types of effects. [Hsu94] describes a two dimensional approach that uses the difference in position of an object in two different frames to construct two dimensional motion lines using triangle slivers that originate on the back side of the model. They also describe a technique to interpolate using a Catmull-Rom spline with intermediate positioning of the objects in the scene. [Masu99] uses an algorithm similar to ours and outputs a vector oriented description of the surface for rendering.

Our technique keeps track of the translation of an object from frame to frame and draws motion lines for the object. Each motion line is a collection of line segments with associated parameters for length, width, color, starting vertex position, and a visibility flag. A set of motion lines is used to give the appearance of motion in a cartoon scene. The number of lines can be varied depending on aesthetic preference, but we find that fewer than ten works well in practice. Finally, we need to determine which objects in the scene need motion lines. One method is for the content author to tag objects in the scene, another is to test each object to determine if it has traveled over a threshold velocity.

In a preprocess, we randomly select *n* vertices of the 3D model for which motion lines will be drawn. We allocate and initialize a circular buffer of length that is at least that of the longest motion line. Parameter values that we have found work well in practice are black line color, a line width of one pixel, and a length of 20 line segments for all motion lines. At runtime we store in the circular buffer a translation vector corresponding to the translation of the object in the current frame, and increment the pointer into the circular buffer by one. For each motion line, we iterate over all locations in the circular buffer, wrapping around to the beginning of the buffer. Starting from the initial vertex position stored with the motion line, at each step of the iteration we add the translation vector in the buffer to the previous vertex position, and render a line segment between the previous and new positions.

Scalability using a Multi-Resolution Mesh Representation of Geometry

Processor Class	Gouraud	Painter	Painter and Inker
Pentium®	2,598/1,697	2,282/1,525	934/740
Pentium II®	13,456/7,402	12,820/7,081	6,920/4,011
Pentium III®	16,760/8,764	16,760/8,764	11,162/6,226

Table 1. To maintain 30 fps the MRM system was used to control the number of polygons in the pipeline. To give an idea of the performance that can be achieved, the table shows the faces and vertices that can be rendered and still maintain 30 fps on each class of machine (faces/vertices). The model used was the duck in Figure 18.

6 Integration With Animation and Multi-Resolution Mesh

Many systems have been created to address animation in a stylized rendering architecture, including [Rade99] and [Lass87]. In our system, no new information from the modeler is necessary beyond that required for traditional computer animation. We export from a 3D-modeling tool a bones-based representation of the animation to which we apply standard key frame animation [Digi00a].

Our Multi-Resolution Mesh (MRM) system is a plug-in for a 3D authoring tool based on the work by Garland and Heckbert [Garl97]. The MRM system exports a file consisting of a base mesh and an ordered list of vertex additions and deletions for the MRM runtime system [Digi00b]. Each addition or deletion is an update record in the file. A constant frame rate can be achieved by using update records from the MRM file to dynamically change mesh resolution.

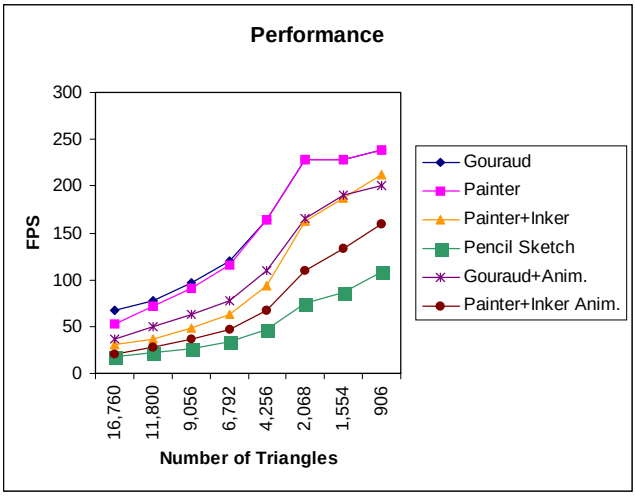

Figure 14: A comparison of stylized rendering and Gouraud shading with and without animation on a high-end Pentium III® PC.

7 Scalability

Since today's consumer hardware is equipped with various types of processors, graphics accelerators, and drivers, we have integrated our NPR techniques with MRM so that they can scale in real time to maintain a constant frame rate while maximizing visual fidelity. We use the term *scalability* to mean the ability to render models with our NPR techniques at interactive frame rates on a range of low- to high-end platforms. In Table 1, we show that we can remove from or add vertices to a 3D mesh to change the level of detail of the mesh at run time and maintain 30 fps. Figure 14 shows a comparison of the performance of our stylized rendering algorithms against that of Gouraud shading with and without animation on a high-end Pentium III® performance level PC. We are in no way indicating benchmarks of a particular processor, graphics card, or any other device.

8 Conclusions and Future Work

In this paper we have introduced techniques for cartoon shading and pencil-sketch rendering for real-time animation. We have also discussed our implementation of silhouette edge detection and rendering. Finally, we have introduced motion lines as a way to simulate motion in a cartoon rendered animation.

In the future we would like to find a solution to the "shower door" problem that results when we animate scenes rendered with the pencil sketch technique described in Section 3.3. Also, we would like to integrate our motion lines into a cartoon world with other traditional cartoon effects and cartoon characters rendered with our Inker and Painter. Another area of future work is in handling the issues of stylized silhouette rendering presented in Section 4.3.

There are many "by-hand" techniques that have been used by artists and animators for years that will elicit interesting research for methods of automated simulation in years to come. We are excited by the prospect of providing artists with new media and tools for storytelling and immersive experiences. We have presented a snapshot of our research and development. In the future we will continue to explore the arena of automated methods for stylized rendering and animation.

9 Acknowledgements

Thanks to Michael Rosenzweig, Mike Mesnier, Stephen Junkins, Jason Plumb, and Dan Johnston for support, ideas, and suggestions, and David Hostetler for speedy coding. We would also like to thank Bruce Gooch, Lee Markosian, and Matt Cutts for discussions and ideas, and John Hughes for suggestions on the paper. Finally, thanks to Keith Feher for the duck model and Gary Barclay for the Toon background. The material contained herein may be used for informational purposes only. Any third party brands and names are the property of their respective owners. No license, expressed or implied, by estoppel or otherwise, to any Intel intellectual property is granted by this document. This document is provided “as is” without warranty, including liability or warranties relating to fitness for a particular purpose, merchantability, or infringement of any patent, copyright, or other intellectual property right.

References

[Appe67] Arthur Appel. The notion of quantitative invisibility and the machine rendering of solids. Proceedings of 22nd National Conference, ACM, Thompson Book Company, Washington D.C., Academic Press, London. Pages 387-393. 1967.

[Deca96] Philippe Decaudin. Rendu de scènes 3D imitant le style «dessin animé». Rapport de Recherche 2919, Institut National de Recherche en Informatique et en Automatique. 1996.

[Digi97] Digimation. *The Incredible Comic Shop*. 150 Jamies Drive East, Suite 140, St. Rose, LA. 70087. 1997.

[Digi00a] Digimation. *Animate RT*. 150 Jamies Drive East, Suite 140, St. Rose, LA. 70087. 1999.

[Digi00b] Digimation. *MultiRes 2*. 150 Jamies Drive East, Suite 140, St. Rose, LA. 70087. 1999.

[Fole96] James Foley, Andreis Van Dam, Steven Feiner, and John Hughes. *Introduction to Computer Graphics: Principles and Practice*. Addison Wesley, 1996.

[Garl97] Michael Garland and Paul S. Heckbert. SurfaceSimplification Using Quadric Error Metrics. In Proceedings of ACM SIGGRAPH 97, pages 209-216. 1997.

[Gooc98] Amy Gooch, Bruce Gooch, Peter Shirley, and Elizabeth Cohen. A Non photorealistic Lighting Model for Automatic Technical Illustration. In *Proceedings of ACM SIGGRAPH 98*, pages 447-452. 1998.

[Gooc99] Bruce Gooch, Peter-Pike J. Sloan, Amy Gooch, Peter Shirley and Richard Riesenfeld. Interactive Technical Illustration, IEEE Symposium on Interactive 3D Graphics, pages. 31--38 April 1999.

[Gour71] H. Gouraud. Continuous Shading of Curved Surfaces. IEEE Trans. On Computer Graphics. Pages 623-629. 1971.

[Heid99] Wolfgang Heidrich and Hans-Peter Seidel. Realistic, Hardware-accelerated Shading and Lighting. In *Proceedings of ACM SIGGRAPH 99*, pages 171-178. 1999.

[Hoga81] Burne Hogarth. Dynamic Light and Shade. Watson-Guptill Publications. New York. 1981.

[Hsu94] Siu Chi Hsu and Irene H. H. Lee. "Drawing and Animation using Skeletal Strokes". In *Proceedings of ACM SIGGRAPH 94*, pages 109-118. 1994.

[Igar99] Takeo Igarashi, Satoshi Matsuoka and Hidehiko Tanaka. Teddy: A Sketching Interface for 3D Freeform Design. In *Proceedings of ACM SIGGRAPH 99*, pages 409-416. 1999.

[Kowa99] Michael Kowalski, Lee Markosian, J.D.Northrup, et al Art-Based Rendering of Fur, Grass, and Trees. In *Proceedings of SIGGRAPH 99*, pages 433-438. 1999.

[Lass87] John Lasseter, Principles of Traditional Animation Applied to 3D Computer Animation. In *Proceedings of ACM SIGGRAPH 87*, pages 35-44. 1987.

[Ligh99] Lightwork Design. *Kazoo.* Lightwork Design Ltd. Rutledge House. 78 Clarkehouse, Shefield S10 2LJ, United Kingdom. 1999.

[Lutz20] E. G. Lutz. *Animated Cartoons*. Applewood Books. Massachusetts. 1920.

[Mark97] Lee Markosian, Michael Kowalski, Samuel Trychi, Lubomir Bourdev, Daniel Goldstein, and John Hughes. Real-Time Nonphotorealistic Rendering. In *Proceedings of ACM SIGGRAPH 97*, pages 113-122. 1997.

[Masu99] Maic Masuch. “Speedlines: Depicting Motion in Motionless Pictures.” ACM SIGGRAPH 99 Technical sketch.

[Ment99] Mental Images. *Mental Ray*. GmbH & Co. KG. Fasanenstrasse 81, D-10623 Berlin, Germany.

[Meta99] Metacreations Poser. http://www.metacreations.com/products/poser4/. 6303 Carpinteria, CA 93013, 1999.

[McCl93] Scott McCloud. *Understanding Comics*. Harper Collins Publishers, New York. 1993.

[Meie96] Barbara Meier. Painterly Rendering for Animation. In *Proceedings of SIGGRAPH 96,* pages 477-484, 1996.

[Rask99] Ramesh Raskar and Michael Cohen. Image Precision Silhouette Edges. Symposium on Interactive 3D Graphics 99.

[Rade99] Paul Rademacher. View-Dependent Geometry. In *Proceedings of ACM SIGGRAPH 99*, pages 439-446. 1999.

[Reyn99] Craig Reynolds. *Stylized Depiction in Computer Graphics*. http://www.red.com/cwr/painterly.html/.

[Sous99] Mario Costa Sousa and John W. Buchanan. Computer Generated Graphite Pencil Renderings of 3D Polygonal Models. . In *Computer Graphics Forum*, pages 195-207. Eurographics ’99 Conference issue.

[Thom81] Frank Thomas and Ollie Johnston. *The Illusion of Life: Disney Animation.* Hyperion, New York. 1981.

[View98] Viewpoint Data Labs. *Liveart 98*. Orem, UT, 1998.

[Wink94a] Georges Winkenbach and David H. Salesin. Computer Generated Pen-and-Ink Illustration. In *Proceedings of ACM SIGGRAPH 94*, pages 91-100. 1994.

[Woo99] Mason Woo, Jackie Neider, Tom Davis, and Dave Shriener. *OpenGL Programming Guide, 3rd Edition*. Addison Wesley, 1999.

[Zele96] R. Zeleznik, K. Herndon, and J.F. Hughes. Sketch: An interface for sketching 3d Scenes. In *Proceedings of SIGGRAPH 96,* pages 163-170. 1996.

Figure 15: A cartoon-shaded duck that uses our Painter and Inker in a cel animation.

Figure 16: Olaf rendered using different cartoon shading styles for each material of the 3D mesh.

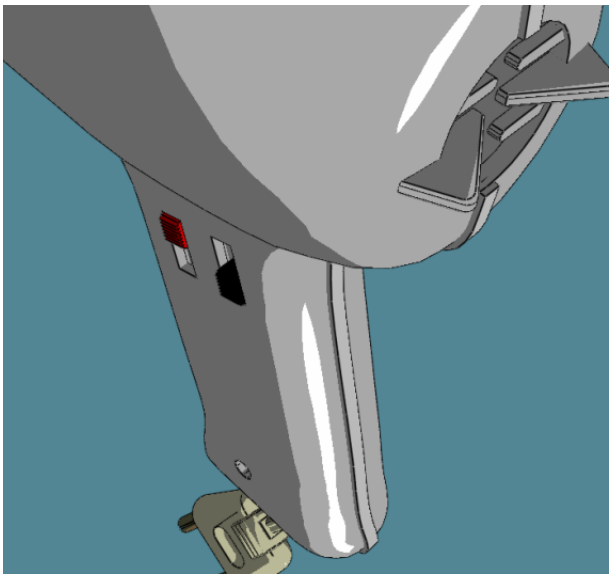

Figure 17: A cartoon-shaded hair dryer using shadow-and-highlight shading and ink lines.

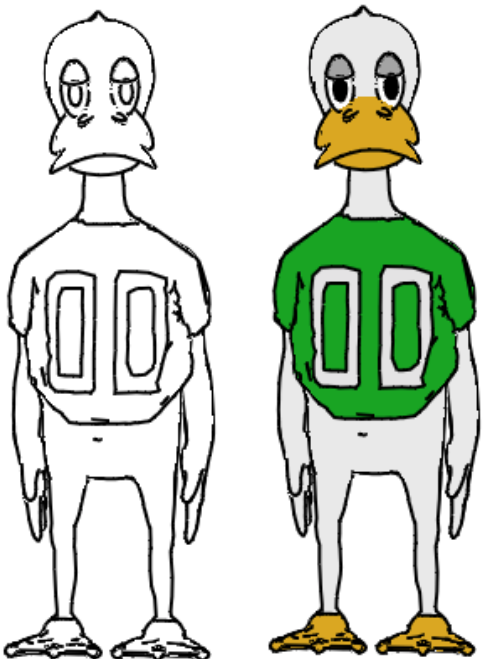

Figure 18: Left: an image rendered using the Inker and background-colored faces, to simulate a cel before painting. Right: an inked and painted cel, using the cartoon shading from Section 3.

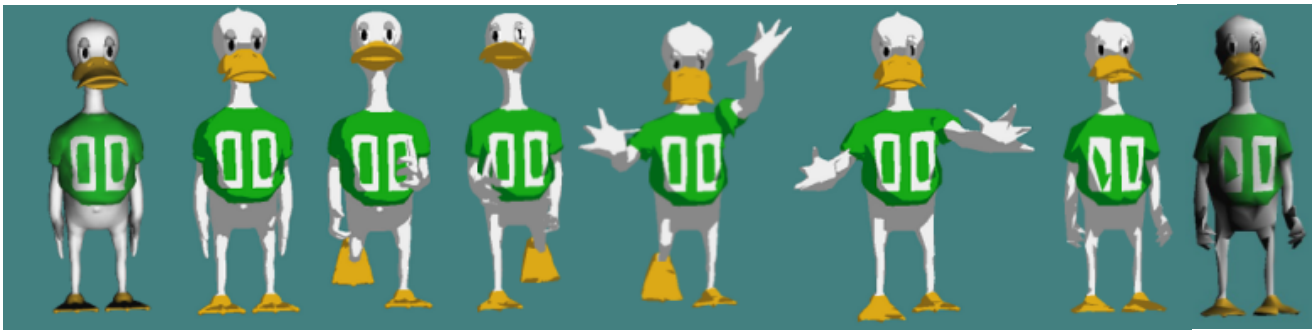

Figure 19: An example animation sequence that uses our cartoon shading technique and runs in real time on a PC. The left and right images are Gouraud shaded. From left to right the number of polygons was decreased at runtime to increase frame rate. The leftmost image has 16,000 polygons, and the rightmost has 1,000.

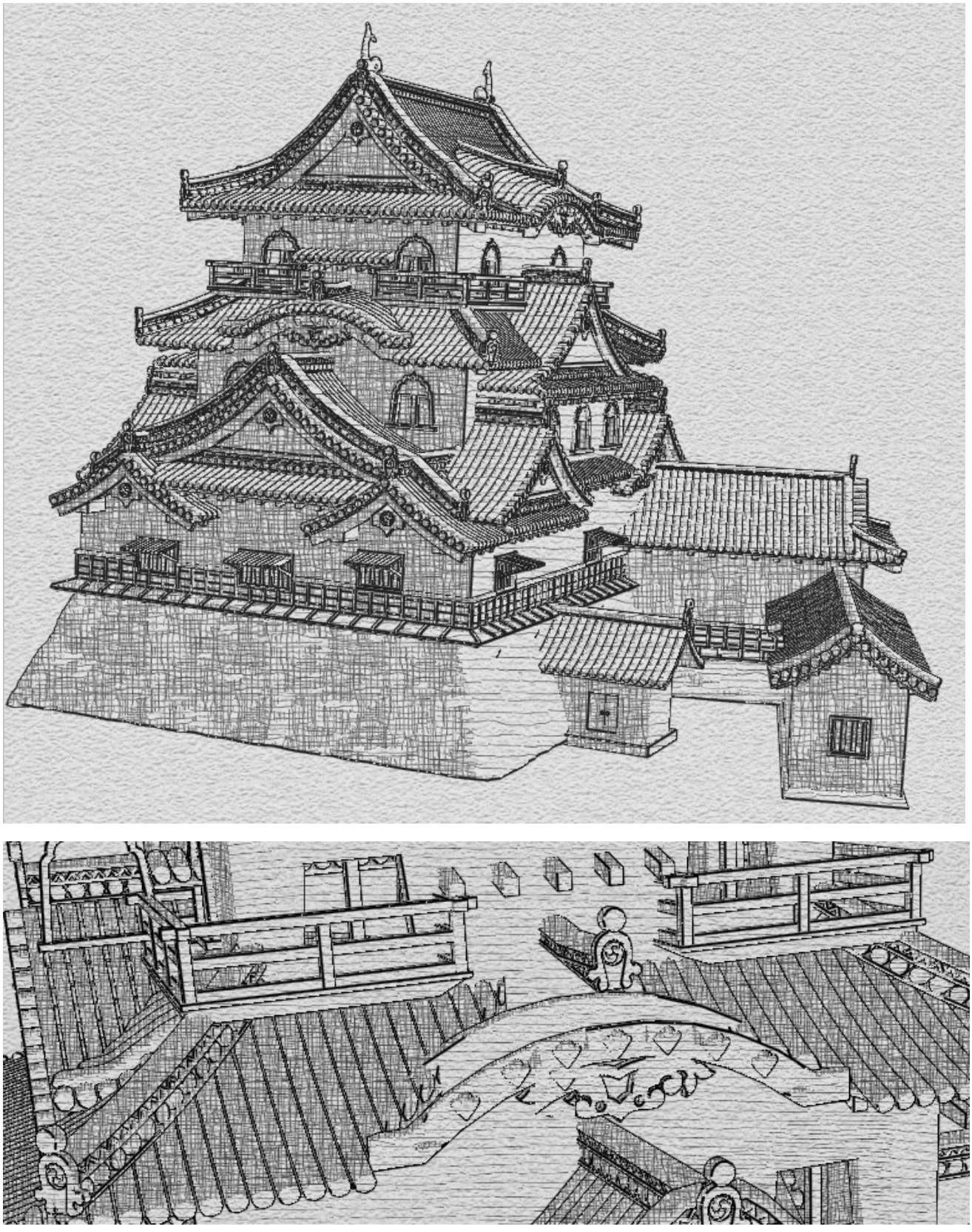

Figure 20: Images rendered using our pencil sketch algorithm and Inker.