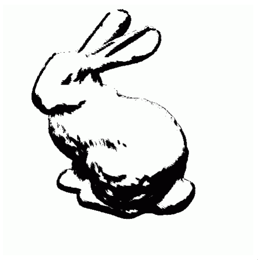
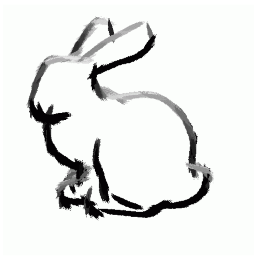


View-Dependent Particles for Interactive Non-Photorealistic Rendering



Derek Cornish[1], Andrea Rowan[2], David Luebke[2]
[1] Intrinsic Graphics [2] University of Virginia

Abstract

We present a novel framework for non-photorealistic rendering (NPR) based on view-dependent geometric simplification techniques. Following a common thread in NPR research, we represent the model as a system of particles, which will be rendered as strokes in the final image and which may optionally overlay a polygonal surface. Our primary contribution is the use of a hierarchical view-dependent clustering algorithm to regulate the number and placement of these particles. This algorithm unifies several tasks common in artistic rendering, such as placing strokes, regulating the screen-space density of strokes, and ensuring inter-frame coherence in animated or interactive rendering. View-dependent callback functions determine which particles are rendered and how to render the associated strokes. The resulting framework is interactive and extremely flexible, letting users easily produce and experiment with many different art-based rendering styles.

Key words: Non-photorealistic rendering, artistic rendering, view-dependent simplification, view-dependent particles

1 Introduction

Non-photorealistic rendering, or *NPR*, has become an important and impressive branch of computer graphics in recent years. Most NPR techniques attempt to create images or virtual worlds visually comparable to renderings produced by a human artist. Several artwork styles have been explored in the NPR literature, such as pen and ink [9,10], painting [1,6], informal sketching [5], and charcoal drawing [7]. To date, most published NPR algorithms focus on a specific artistic style, or a closely related family of styles. For example, Meier describes a technique for creating animations evocative of impressionistic painting [6], and Markosian creates cartoon-style renderings reminiscent of Dr. Seuss [4]. Underlying these highly varied artistic effects, however, are several recurring themes common to most NPR techniques. The framework presented in this paper began as an attempt to identify and unify these common algorithmic threads:

Strokes and particles: Human artwork typically consists of multiple separate *strokes*, ranging from dabs with a paintbrush to streaks of charcoal to lines drawn with pen or pencil. Most NPR algorithms therefore cast rendering as a process of selecting and placing strokes. These strokes are usually placed and rendered with some randomness to imitate the unpredictability of the human artist, but this randomness can introduce flicker when the resulting images are animated. To eliminate this flicker, Meier [6] introduced the idea of associating strokes with *particles* defined on the surface of the object to be rendered. Since the strokes are associated with actual locations in space, they move smoothly across the screen in a visually pleasing manner as the viewpoint shifts. Many NPR systems have since incorporated this idea; notable examples include work by Markosian and Hughes [4] and by Kaplan et al [12].

Orientation fields: For the full range of expressive effect, an artist or programmer must have control over how strokes are oriented as well as where they are placed. Salisbury et al used a user-specified vector field defined on an image to guide the orientation of pen-and-ink strokes [9], while others rasterize normal and curvature information from a 3-D model [6][8]. This use of orientation fields to guide stroke drawing is another common thread running through multiple NPR approaches.

Screen-space density: Careful attention to screen-space density of strokes is yet another common theme. For example, concentrating strokes most densely on the silhouette of an object can suggest a great deal of complexity with relatively few strokes (e.g., [4][7]). Too many strokes throughout the image can create a cluttered effect, while too few strokes may fail to convey the underlying shape. In styles such as pen-and-ink, stroke density also controls tone, so that too many strokes will create a darker drawing with a completely different look [9][10]. Associating strokes with particles as described above does not solve this problem, since the screen-space particle density increases as objects recede into the distance.

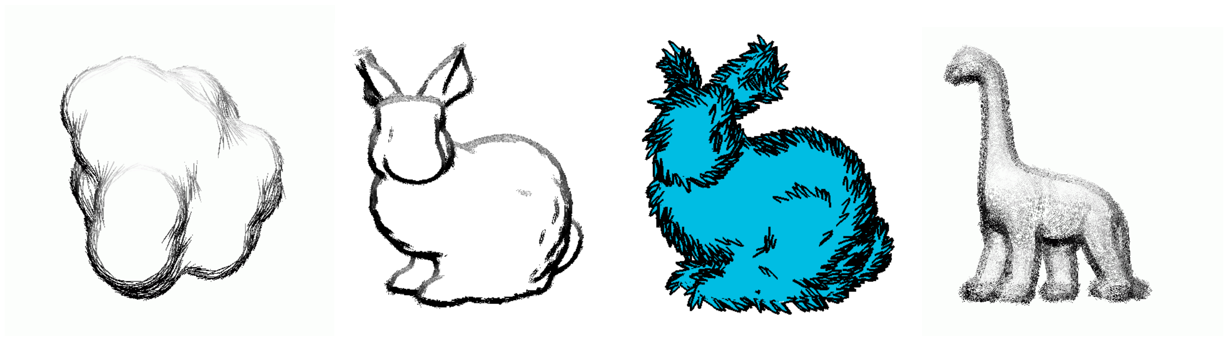


Figure 1: Several examples of different styles that can be created in our interactive NPR system. From left to right: a sketched molecule, a pencil bunny, a furry bunny after Markosian [4], and a chalky dinosaur.

1.1 Contribution

These themes—representing strokes with particles, guiding rendering with orientation fields, and careful attention to screen-space density—recur in NPR algorithms varying widely in architecture and artistic style. Our primary contribution is a system for non-photorealistic rendering that addresses all three issues, providing a framework with which users can easily create a whole spectrum of stylistic effects. The key to our system is a new representation for the particle field, which we call *view-dependent particles*. View-dependent particles provide an efficient multiresolution structure for fine-grained control over the placement of strokes, and can be generated from any polygonal model. The multiresolution nature of the structure provides efficient rendering at all scales, allowing densely populated scenes containing tens or hundreds of thousands of particles. Arbitrary scalar or vector fields may be defined over the particles to describe attributes, such as color or orientation, which affect stroke rendering.

Our system is designed for interactive rendering. The view-dependent particle system is adjusted dynamically and continuously as viewing parameters shift, using the underlying multiresolution structure to enhance interactivity. We use a multi-stage rendering process designed to take advantage of hardware acceleration. The resulting system can produce compelling non-photorealistic imagery of polygonal models in many varied artistic styles at interactive rates.

2 Overview of the algorithm

View-dependent particles are inspired by and built upon algorithms for view-dependent polygonal simplification. These algorithms address many issues relevant to non-photorealistic rendering, such as regulating the screen-space density of polygons and finding visual silhouettes. As such, they provide an ideal springboard for a system to manage particles in an NPR algorithm. Our NPR framework uses *VDSlib* [13], a public-domain view-dependent simplification package that supports user-defined simplification, culling, and rendering criteria via callback functions. We represent the object to be rendered as a densely sampled polygonal model; the vertices of this model form the highest resolution of the view-dependent particle system.

Rendering a frame in our algorithm comprises up to four stages:

1. **Adjust particles:** In this stage the view-dependent particles are adjusted for the current view parameters. The *active set* of particles is traversed and a user-supplied callback function decides whether to fold, unfold, or leave unchanged each particle. *Unfolding* a particle adds its children to the active set, creating more local detail in that region of the model; *folding* a particle removes its children from the active set, reducing local detail. It is at this stage that the user controls the density and placement of strokes in the final image. For example, a user callback might mediate screen-space density of strokes by folding particle clusters that project to a small portion of the screen, or add contour strokes by unfolding only particles that lie on a silhouette.

2. **Render polygons (optional):** For many NPR styles the interior of the object must be rendered in some fashion. For example, if pen-and-ink strokes are to be depth-buffered, so that particles on the far side of the object do not generate visible strokes in the final image, the polygons of the object should be rasterized into the depth-buffer. Other effects, such as "cartoon rendering", require rendering the object interior, perhaps with a single uniform color as in Figure 6. In this stage the user may optionally render the simplified polygonal mesh whose vertices are the active set of particles. A user callback specifies how to render these polygons. In our cartoon rendering mode, for example, the polygon rendering callback would disable lighting, enable depth buffering, set the color, and render all polygons for the object.

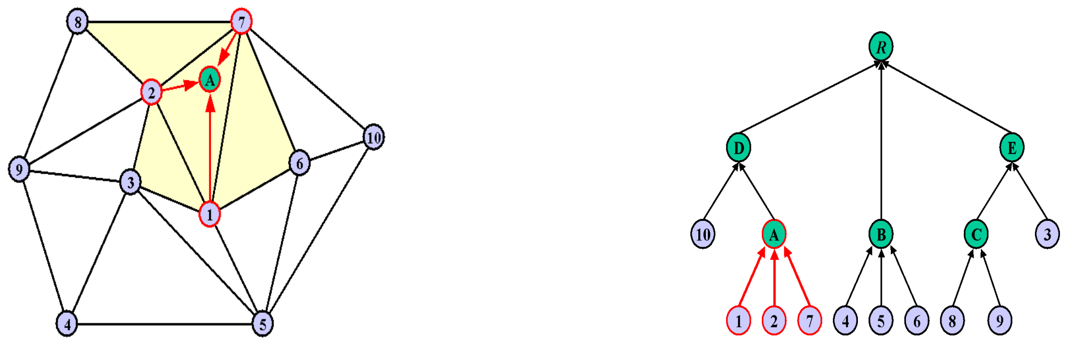


Figure 2: A vertex merge operation on a polygonal mesh, and the associated hierarchy. Here vertices 1, 2, and 7 merge to form the single vertex A, eliminating the shaded triangles. We associate particles with each vertex, which in turn guide the placement of strokes. In practice, we usually choose the position of A to be the same as one of its children to increase frame-to-frame coherence across merge operations.

3. **Transform/light particles:** In this stage the particles are partially rendered, in that they are transformed and clipped to screen-space, with lighting and other calculations optionally applied. The transformed particles are not rasterized. Instead, particle data is accumulated into a buffer that will be guide stroke rendering in the next stage. We elected to implement this stage using OpenGL feedback mode. This allows the user to write custom callbacks that "render" the particles in a familiar intuitive fashion, while exploiting any hardware acceleration of transformation, lighting, etc. By rendering an offset vertex, vectors associated with the particles (such as normals or an orientation field) can be transformed into screen-space to guide stroke rendering.

4. **Render strokes:** In the final stage, the screen-space particle data is used to guide the rendering of strokes into the image. Again, a user-defined callback performs the rendering, parsing the feedback buffer to extract the particle position as well as any color or vector data processed in the third stage. The stroke is then rendered, typically in 2-D at the particle's transformed screen-space position, using whatever method is appropriate for the desired artistic effect. For example, in painterly rendering strokes may be rendered as partially-transparent textured polygons, while in pen-and-ink rendering strokes might be rendered as line segments. If the optional second stage was used to render the underlying polygonal model, the same buffer is used for the final image. For example, strokes can outline a filled object, or the depth buffer can prevent rendering occluded strokes.

3 Implementation Details

As mentioned above, our implementation of view-dependent particles is built on VDSlib, a view-dependent algorithm from the polygonal simplification literature. Polygonal simplification, also known as *level of detail (LOD) management*, is a well-known technique for increasing rendering speed by reducing the geometric detail of small, distant, or otherwise unimportant portions of the model. View-dependent simplification techniques for general polygonal models are comparatively new, with several algorithms proposed by researchers in recent years [2,3,11]. All of these algorithms share the same underlying mechanism for reducing complexity: a hierarchy of *vertex merge* operations that progressively cluster vertices of a triangulated model [Figure 2]. Each operation replaces multiple vertices with a single representative vertex, in the process removing some triangles from the model. By selectively applying and reversing vertex merge operations, the underlying model may be represented with greater detail in some regions (for instance, near the viewer or on object silhouettes) than others. Of course, this requires the algorithms to adjust the simplification continuously as the viewpoint moves. View-dependent simplification algorithms are thus designed to run on highly complex models at interactive rates. By defining our view-dependent particles as the vertices of a dense polygonal mesh, we can take full advantage of the speed and generality of view-dependent simplification.

Each node in a view-dependent simplification hierarchy either represents a vertex from the original full-resolution model or a vertex created by a series of vertex merge operations. Similarly, in our system the nodes form a hierarchy called the *particle tree*. Each node represents a particle; leaf nodes are attached to a vertex of the original polygonal model, while internal nodes represent the result of particle merge operations. Internal nodes in polygonal simplification typically represent a sort of average of all vertices below them in the hierarchy; however, for better frame-to-frame coherence we simply pick one of the merged particles to represent the rest. Replacing a collection of sibling particles with their parent in the particle tree has the effect of locally simplifying the model, using fewer particles and ultimately fewer strokes to render that portion of the object. The first stage in our rendering process performs this local simplification, traversing the particle tree and applying a user-supplied callback function to determine which nodes should be merged, or *folded*, and which should be *unfolded*. Since the simplification changes every frame, the distribution of particles can account for view-dependent factors, such as the distance of particles from the viewer or which particles lie on the silhouette.

Since the particles are in fact vertices of a polygonal mesh, the fold and unfold operations also affect the underlying surface model. Layering our system over a view-dependent polygonal simplification algorithm allows us to track that underlying model, rendering the polygons if necessary for the desired artistic effect. For example, the surface might be rendered as a single solid color for cartoon rendering, or textured with a crosshatch pattern for a pencil-sketch effect. For many effects, a depth buffer of the surface may be wanted to eliminate particles that should be occluded. Our optional second stage enables all these effects by rendering the polygons of the current simplification using a custom user callback. In short, the first and second stages consist of simplifying and rendering a polygonal model with user-specified criteria. While those criteria are quite unlike those used for traditional rendering, both operations are supported directly by the view-dependent simplification library.

The third stage uses OpenGL feedback mode to transform and clip the particles. It can also be used to apply lighting calculations and project vector attributes (such as normal or orientation vectors defined on the particles) into screen space. Such vectors are rendered in feedback mode as 3-D line segments using GL_LINE; the projected 2-D vectors can be then used to orient strokes tangent or perpendicular to the surface. Primitives rendered in feedback mode are not rasterized. Instead, the transformed, projected, clipped, and lit primitives are stored, with all relevant information, as a sequence of tokens in a buffer. The final stage will parse this buffer, using the contents to place and render the final strokes. As usual, a callback function allows the user complete flexibility.

The use of feedback mode during the third stage is open to question. Certainly all of the geometric operations performed during this stage could be performed on the host CPU, and the results used immediately to place the strokes on the final image. This would effectively merge the third and fourth stages, and eliminate the need for intermediate storage of the feedback results. We felt, however, that rendering the particles in feedback mode would provide two main advantages: efficient use of graphics hardware and a flexible, familiar interface for the user. Rendering particles in feedback mode allows us to exploit hardware transform, lighting, and clipping acceleration. Graphics hardware is typically deeply pipelined, operating most efficiently when the pipeline is kept full. Rendering all particles in a continuous stream into a feedback buffer followed by rendering all strokes continuously to the framebuffer helps maximize graphics hardware performance. Rendering particles directly with OpenGL also increases ease of use, enabling the user to experiment with different rendering strategies quickly and painlessly. Incorporating lighting, for example, would be tedious to implement programmatically, but this is easily done in OpenGL.

4 Results

Here we show images spanning several very different rendering styles, and briefly explain the various callback functions used to produce each effect. It should be emphasized again that the system is fully interactive. The effects and models shown ran at frame rates ranging from 5-20 Hz on an SGI Onyx^2 with InfiniteReality graphics.

1.2 The models

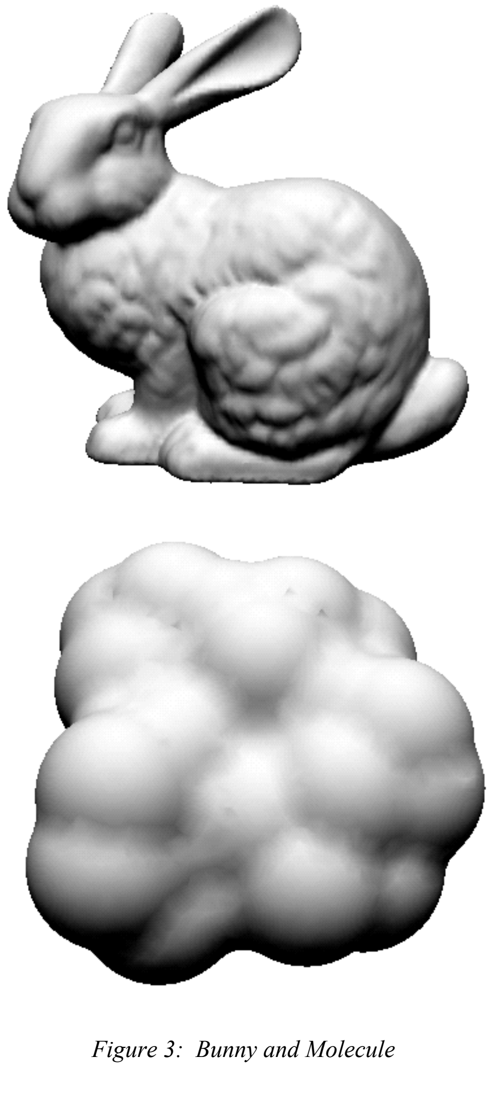


Figure 3: Bunny and Molecule

Bunny is the familiar Stanford bunny, containing approximately 69,451 triangles and 34,834 vertices. *Molecule* is an accessibility isosurface for a simple molecule, containing 7,344 triangles and 3,675 vertices. Both models contain interesting curvature and silhouettes that illustrate the various rendering styles well. *Dinosaur* [Figure 1] has 47,904 triangles and 23,984 vertices.

1.3 Painterly

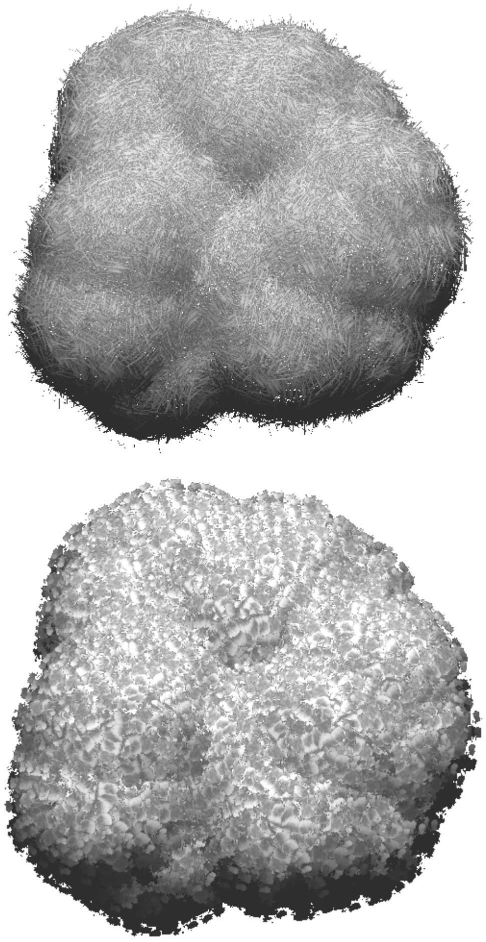


Figure 4: On top, a painterly rendering of Molecule with long, thin strokes. Below, the same model rendered with round, dabbed strokes and a different brush texture.

For these renderings:

- Stage 1: View-dependent particles were left unsimplified, at the full original resolution.
- Stage 2: Triangles of model rendered into depth buffer to prevent drawing occluded strokes in stage 4.
- Stage 3: Particles rendered in feedback mode with lighting enabled. A 3-D orientation vector was also rendered and projected to screen-space. The resulting 2-D vector was used to align the strokes. This orientation vector was set randomly and stored with each particle, so that the stroke orientation would remain constant from frame to frame.
- Stage 4: Strokes rendered as texture-mapped quadrilaterals, at the depth returned from feedback mode for the particle. The texture used for the top is long and thin stroke, while on the bottom a short, almost circular dab was used.

1.4 Sketched

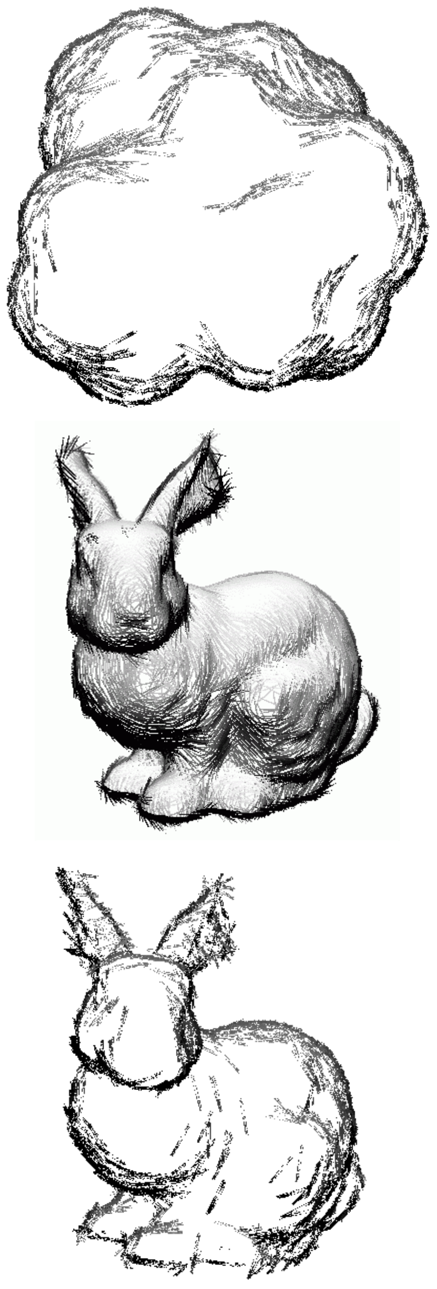


Figure 5: Two quite different effects with a somewhat pencil-sketch feel. Top and bottom: Molecule and Bunny drawn with short choppy silhouette strokes. Middle: Bunny drawn with very thin, elongated strokes roughly aligned.

On the top and bottom, particles were simplified in stage 1 to moderate density, and removed from non-silhouette regions. In stage 3, the surface normal was transformed into screen-space for use as an orientation vector. Stage 4 rendered the strokes with a stroke texture scanned from a crayon mark, oriented by the transformed normal to appear tangent to the surface.

In the middle, particles were only slightly simplified in stage 1, with only slight preference given to silhouette particles. Stage 3 again transformed the surface normal, which was then slightly perturbed. In stage 4, a single very long, thin stroke aligned with the transformed normal was rendered for each particle.

1.5 Cartoon drawing

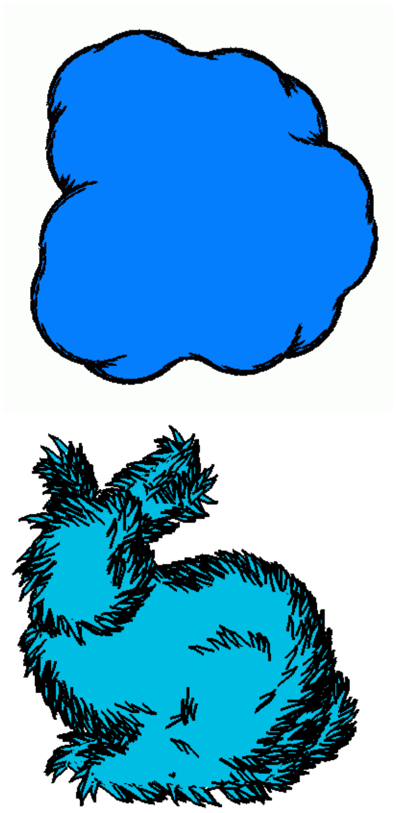


Figure 6: Two cartoon-like drawing effects, with solid interior shading and strong silhouette strokes (top) and Markosian-style graftals (bottom).

Both these styles are identical until stage 4. In stage 1, particles not on the silhouette were simplified. Stage 2 rendered the underlying triangles a solid color, with flat shading and lighting disabled. Stage 3 transformed the particles and the surface normal into screen space, but did not perform any lighting calculations. For the effect on the top, stage 4 rendered those particles lying exactly on the silhouette with strong black horizontal strokes, aligned by the transformed normal to appear roughly tangent to the surface. For the Dr. Seuss-inspired effect on the bottom, particles were rendered with *graftals* in the manner of Markosian and Hughes [4], but without using a desire image. Instead we rely on the view-dependent simplification of particles and simple screen-space binning to avoid rendering too many graftals in a single area of the screen.

1.6 Charcoal smudge

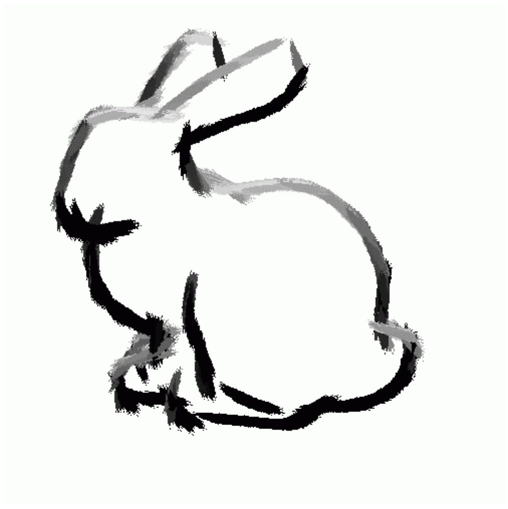


Figure 7: A charcoal effect using just a few strong overlapping silhouette strokes.

Here the particle tree was simplified drastically in stage 1, leaving only a few particles near the silhouette. These particles were transformed and lit in stage 3, along with an orientation vector aligned with the surface normal. The effect was completed by rendering with long, thick brushstrokes in stage 4.

1.7 Two-tone

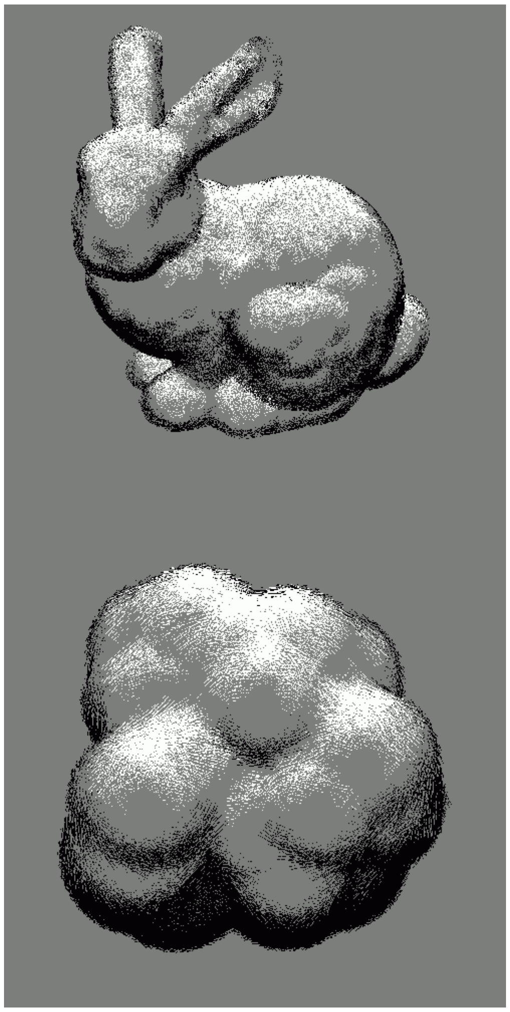


Figure 8: A two-tone effect in which light and dark strokes are drawn on a neutral gray background.

Following an image by Markosian and Hughes [4], these two images use dark and light strokes to bring shape out of a neutral background. Stage 1 does not appreciably simplify the particle tree. In stage 2, the underlying polygons are rasterized to the depth buffer but not the color buffer. Stage 3 transforms and lights the particles. Stage 4 examines the intensity of the lit particles in the feedback buffer. Particles over an upper intensity threshold are rendered as solid white strokes, while particles below a lower threshold are rendered as solid black strokes. All other particles are discarded.

1.8 India ink

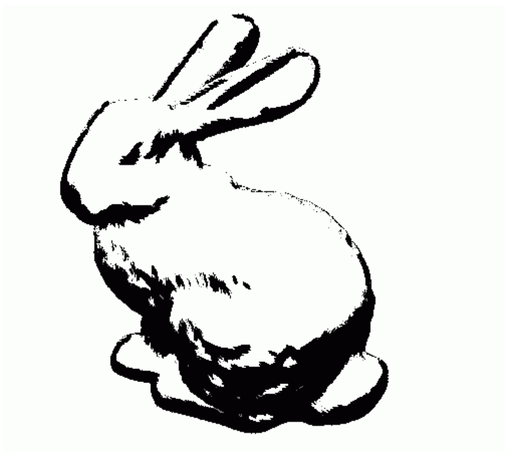


Figure 9: An India-ink effect using only strong, black strokes densely placed in regions of darkness.

This style is very similar to the two-tone effect described above, but uses only black strokes on a white background. Stage 1 does not simplify the model at all, so the particle tree is very densely represented. Stage 3 transforms and lights the particles. In stage 4, particles below an intensity threshold are rendered as solid black strokes; other particles are ignored. The thickly placed strokes create solid black regions for a strong, stylized effect.

5 Conclusions and Future Work

We have demonstrated a novel framework for non-photorealistic rendering that unifies several common threads running through the NPR literature: particle systems to guide stroke placement, scalar and vector fields to guide stroke rendering, and careful attention to screen-space density of strokes. Our system introduces *view-dependent particles*, a multiresolution representation inspired by and built on view-dependent simplification algorithms. A multi-stage rendering process with a flexible plug-in architecture gives the user great control over the placement and rendering of strokes, without sacrificing graphics acceleration. The resulting system supports rendering in many highly varied styles at interactive rates.

The greatest strength of our system is its extensibility. We have found that extending the system to add additional rendering styles is typically quite straightforward; the callbacks to generate many of the effects demonstrated in this paper were developed in less than an hour. There are certainly many interesting avenues for future work in simply exploring techniques and frameworks for new styles. For example, many algorithms use a *desire image* produced by rendering the model to regulate the screen-space density of strokes. New particles are placed where the desire image indicates they are most needed, and adding the particle reduces the "desire" of that part of the image. None of our current styles use this technique, but integrating a desire image and comparing the results seems straightforward and quite interesting. For now, we have gotten satisfactory results using simple spatial binning [Figure 6].

We would also like to experiment with multi-layer painterly rendering techniques such as [6], which more closely emulate the process a human painter follows. Better support for pen and ink rendering also seems worth investigating. The long, expressive strokes typical to this medium are difficult to produce in our current model, where a single stroke is associated with each visible particle. One possibility might be to thread strokes across multiple particles, adjusting the stroke as individual particles come in and out of view.

There is scope for more work on ensuring temporal coherence of strokes. By attaching the strokes to object-space particles in the fashion of Meier [6], we achieve good frame-to-frame coherence while the LOD is constant. However, the view-dependence of our particles can work against us, since they may be folded away or unfolded into additional particles. In practice, we have found only slight flicker introduced by this process, but we might eliminate that flicker entirely by fading in over several frames the strokes associated with newly created particles. This should be simple to implement by adding an age field to each particle.

A clear limitation of our current approach is the correlation between particles and vertices in the original model. This requires us to use densely sampled polygonal models, to ensure even coverage of particles on the surface. Placing particles only at vertices also prevents us from approaching the model too closely, since the vertices and strokes will grow apart and eventually "tear open" the illusion of a solid surface. One solution we are exploring is to procedurally place "temporary" particles on triangles, so that as the triangle grows larger more particles are scattered across its surface. These particles would be rendered from the triangles and accumulated into the same feedback buffer used for vertex particles during the third stage. Another limitation is the inherent assumption that the image is comprised of many short strokes. For certain styles it is important to have e.g. a single long contour stroke. One possibility we are investigating is the use of particles as control points to guide long strokes rendered as splines or "snakes".

References

[1] Aaron Hertzmann. Painterly Rendering with Curved Brush Strokes of Multiple Sizes. In *SIGGRAPH 98 Conference Proceedings*, pp. 453-460. ACM SIGGRAPH, July 1998.

[2] Hugues Hoppe. Smooth View-Dependant Level-of-Detail Control and it Application to Terrain Rendering. In *SIGGRAPH 97 Conference Proceedings*, pp. 189-198. ACM SIGGRAPH, August 1997.

[3] David Luebke and Carl Erikson. View-Dependant Simplification of Arbitrary Polygonal Environments. In *SIGGRAPH 97 Conference Proceedings*, pp. 199-208. ACM SIGGRAPH, August 1997.

[4] Lee Markosian and John F. Hughes. Art-based Rendering of Fur, Grass, and Trees. In *SIGGRAPH 99 Conference Proceedings*, pp. 433-438. ACM SIGGRAPH, August 1999.

[5] Lee Markosian, Michael A. Kowalski, Samuel J. Trychin, Lubomir D. Bourdev, Daniel Goldstein and John F. Hughes. Real-time Nonphotorealistic Rendering. In *SIGGRAPH 97 Conference Proceedings*, pp. 415-420. ACM SIGGRAPH, August 1997.

[6] Barbara J. Meier. Painterly Rendering for Animation. In *SIGGRAPH 96 Conference Proceedings*, pp. 477-484. ACM SIGGRAPH, August 1996.

[7] Ramesh Raskar and Michael Cohen. Image Precision Silhouette Edges. In *1999 Symposium on Interactive 3D Graphics.* pp. 135-140. ACM SIGGRAPH, 1999.

[8] Takafumi Saito and Tokiichiro Takahashi. Comprehensible Rendering of 3-D Shapes. In *SIGGRAPH 90 Conference Proceedings*, pp. 197-206. ACM SIGGRAPH, August 1990.

[9] Michael P. Salisbury, Michael T. Wong, John F. Hughes, and David H. Salesin. Orientable Textures for Image Based Pen-and-Ink Illustration. In *SIGGRAPH 97 Conference Proceedings*, pp. 401-406. ACM SIGGRAPH, August 1997.

[10] Georges Winkenbach and David H. Salesin. Computer Generated pen-and-ink illustration. In *SIGGRAPH 94 Proceedings*, pp. 91-100. ACM Press, July 1994.

[11] Julie Xia and Amitabh Varshney. Dynamic View-Dependant Simplification for Polygonal Models, *Visualization 96*.

[12] Matthew Kaplan, Bruce Gooch and Elaine Cohen, Interactive Artistic Rendering. *Non-Photorealistic Animation and Rendering 2000* (NPAR '00), Annecy, France, June 5-7, 2000.

[13] VDSlib is available at *http://vdslib.virginia.edu.*