Real-Time Animated Stippling

Oscar Meruvia Pastor, Bert Freudenberg, and Thomas Strothotte
Otto-von-Guericke University of Magdeburg

We present an approach to produce animations of 3D models using stippling as a rendering style. Our technique ensures frame-to-frame coherence as the model moves and changes over time.

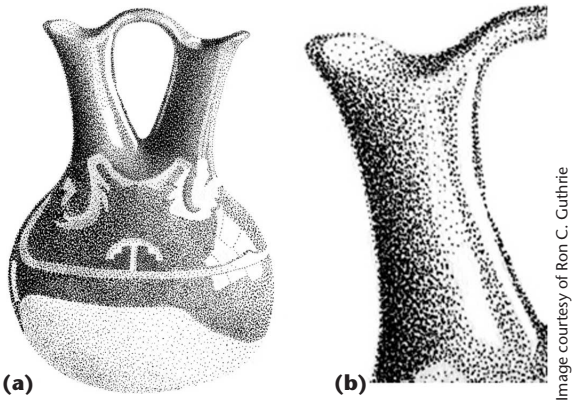
We define *stippling* as the process of rendering an image using dots. You can render almost any imaginable model using stipples. In the natural sciences or in archaeology, scientific illustrators use stippling in combination with other rendering techniques to convey an object's shape, texture, and surface material.

Figure 1 shows a vase rendered using stippling. A closer look at the image reveals that the darker areas are more densely stippled than the lighter areas, and that the stipples have a constant size. Stippled objects do not scale well—especially when you drastically reduce the image's size or view it from a distance—because the stipples become too small and blend with each other. The stippling process requires that you judge the proper scale and spacing of individual stipples because the density of the dots conveys both shape and tone.

The original work for computer-generated stippling typically focuses on creating high-quality renditions of 2D images at high resolutions. These renditions are normally one-time productions intended for printed media. Image-based approaches[1,2] provide interactive and automatic tools to obtain high-quality renderings for single renditions (see Figure 2). These techniques have focused on evenly distributing input dots using Voronoi relaxation.

Several issues make animated stippling a complex problem for graphics researchers. The first issue relates to how the animated stipples should behave. Because the stippling technique typically produces single images at a certain scale, it's not clear how the stipples should react to scaling and changes in shading when they exist in a moving 3D environment. Ideally, we would create stippled renditions that could be arbitrarily scaled, but this is an elusive goal. Another issue relates to maintaining even distributions as the viewpoint, illumination, and viewing distance change—or even as the model itself changes.

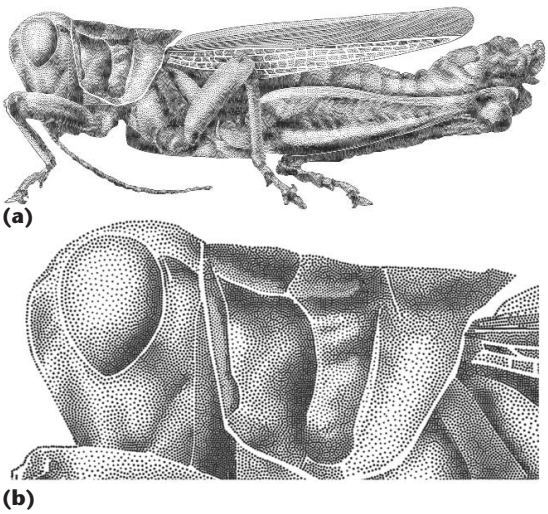
Our contribution to this area is in integrating exist-



(a) (b)

Image courtesy of Ron C. Guthrie

1 Two stippling examples: (a) Ron C. Guthrie's *Indian Pottery* and (b) a close-up from the same image showing the individual stipples.



(a) (b)

Image courtesy of Oliver Deussen

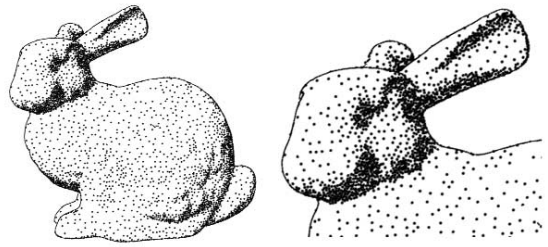
2 Stippling technique using Voronoi relaxation: (a) a stippled image of a grasshopper and (b) a detail of the grasshopper's head.[1-2]

Published by the IEEE Computer Society

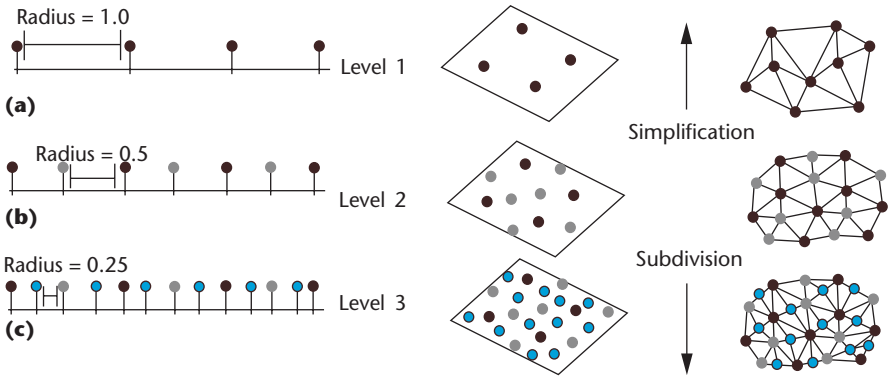
ing work in point hierarchies[3] in a general framework that produces view-dependent, frame-coherent animations in the stippling style for static and animated 3D models at any resolution.

Frame-coherent stippling

In animation, you cannot use stippling by putting together a sequence of independently rendered images without introducing noise. While artists produce animations in the hatching style by redrawing each frame and pasting them together in a sequence, it would be useless to do this with stippling because stipples would randomly appear and disappear over the course of the animation. This effect could even become annoying, depending on the amount of noise the stipples generated. For this reason, we advocate frame-coherent stippling at the level of each individual stipple.



3 (a) The famous Stanford Bunny in the stippling style obtained by frame-coherent stippling. (b) Detail of the bunny's head.



4 Point hierarchies for (a) 1D, (b) 2D, and (c) 3D cases. Points at the lower levels of the hierarchy have smaller radius values, which determines their relevance in the hierarchy. For 3D models, we create a continuous level of detail using mesh simplification and subdivision.

The stipples should behave like a texture on the surface of the model. In addition, the stippling density should smoothly adapt to changes in illumination. It should increase when shading becomes darker and decrease when shading becomes lighter. Thus, the stipples would blend in or out of the images by means of a change in point size. We also want to keep appropriate spacing between rendered stipples, avoiding the formation of regular patterns or irregular grouping of stipple dots as much as possible.

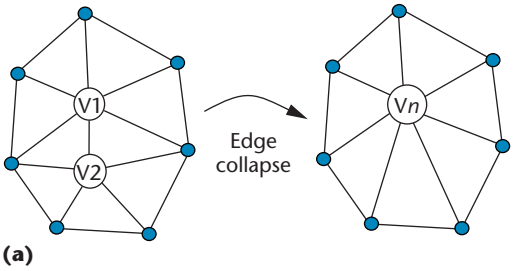
We implement frame-coherent animated stippling by defining a point hierarchy that we fix to the surface of a model and by using a rendering algorithm that employs this point hierarchy to produce stippled renditions. To achieve frame coherence, we take the concept of particle systems from painterly rendering, where particles, graftals,[4] or geograftals[5] are fixed on the surface of 3D models. In principle, we consider each input model's vertex to be a particle that indicates a potential stipple location. Because each point attaches to a specific location on the surface of the model, points move along with the model as the model moves in an animated scene. This technique provides the frame-coherence effect at the stipple level.

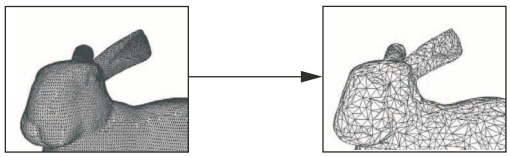
In addition, we control the stipple distribution on the model's surface so that it dynamically adapts to changes in shading, letting dark areas fill with more stipples than light shaded areas (see Figure 3). We use the point hierarchy to determine which points should appear or disappear first from the images. Stippling is not scalable per se, but 3D models are, so we also use the point hierarchy to control the stipple density according to changes in scale and viewing distance(see Figure 4).

We generate the point hierarchy in the same way that vertex hierarchies for mesh simplification and level of detail are generated.[6,7] We refine the edges of a mesh and produce a vertex hierarchy as a result. Other research[3] has developed the idea of applying mesh simplification in a view-dependent real-time system for rendering strokes. We extend this idea by adding a mesh subdivision stage to generate more stipples when needed. This extension lets us use models with low complexity and render them in nonphotorealistic styles. In addition, we include a randomization stage to improve point-distribution quality.

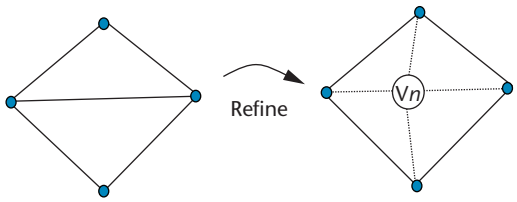
We also decouple the mesh simplification from the rendering process, eliminating the need to perform real-time mesh simplification. After each simplification and each refinement step, we assign the resulting vertex to a list of neighbors that we use to decide which stipples should be included in a particular rendition. Previous research presented the initial approach for frame-coherent stippling in which models are refined as part of an offline animation.[8] Our technique extends this approach to include animated models and real-time rendering.

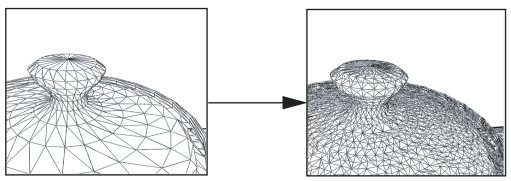
When we look at existing work for 2D stippling and try to extrapolate it to 3D models, we wonder whether it's possible to obtain appropriately spaced particle or stipple distributions (such as those obtained in image-based stippling) while providing frame coherence at the particle level. Other researchers have pursued frame coherence on the image plane[9] instead of the object space. The results of this approach show that by enforcing frame coherence in this way, stipple particles float on the surface of an object as it moves or as shading changes. This effect conveys a vibrating look to the animations and is different from the effect that we want to achieve, where





5 (a) Edge-collapse operation used in mesh simplification. (b) Wireframe view of the original Stanford Bunny model and the same model after a series of simplification steps.





6 (a) Refinement operation used in mesh subdivision. (b) Wireframe view of the original teapot model and the same model after a series of refinement steps.

particles move along with the model as the model moves.

Recent improvements in texture-mapping hardware permit real-time frame-coherent rendering in pen-and-ink styles using stroke textures. When rendered at an appropriate resolution, stroke textures can emulate stippling and adapt to changes in illumination at a given viewing distance. The problem with textures, however, is that it's difficult to control the stipple shape so that it's always projected as a circular dot on the screen. We avoid this problem by drawing each stipple explicitly with point primitives.

Point-set hierarchy

When stippling, it's important to obtain regular point distributions on the final rendition. Artists create stippled drawings by placing some groups of dots in a region of interest and then adding dots until they achieve the desired tone.[10] Our point hierarchy takes into account stipple spacing when adding and removing stipples. Our system places new stipples at locations roughly in the middle of existing stipples located at the top of the hierarchy. When we remove stipples from a model's surface, stipples at the bottom of the hierarchy are the ones that vanish first.

Figure 4 illustrates how we distribute stipples in the higher levels of the hierarchy and add new stipples between existing ones. Figure 4 also shows how we assign distance values to the stipples according to their hierarchy level. This value helps determine whether a stipple should be rendered. In our system, we create a hierarchy of vertices in 3D space to represent stipple locations on the model's surface. Depending on the viewing distance and the number of polygons in the input model, the number of vertices in the input model might not be enough to cover dark areas. To fill these areas, our system generates more vertices on the surface of the model by mesh subdivision.

In other cases, the number of vertices in the input model rises so high that we must discard many of them to produce a light shading tone. To discard vertices from a highly tessellated model, we perform mesh simplification. To ensure that we obtain the appropriate level of detail at most viewing ranges, we mix mesh simplification and mesh subdivision to provide seamless levels of detail regardless of input model resolution (see Figure 4c). The vertices down the hierarchy fill the space between existing vertices, so new stipples always come up to fill uncovered regions of the canvas until we achieve the desired tone.

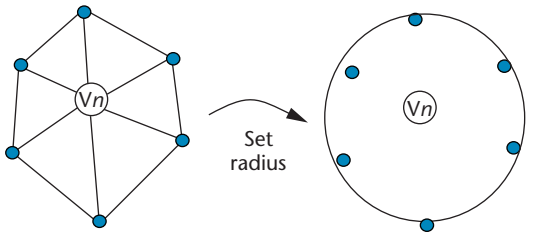
To generate the point hierarchy, our system computes a connectivity graph, applies randomization on the vertices of the input mesh, and then performs simplification and subdivision on the input mesh.

The connectivity graph comes from the input polygonal mesh and contains information about the connections between the model's vertices, edges, and faces. We use this information for randomization, mesh simplification, and subdivision.

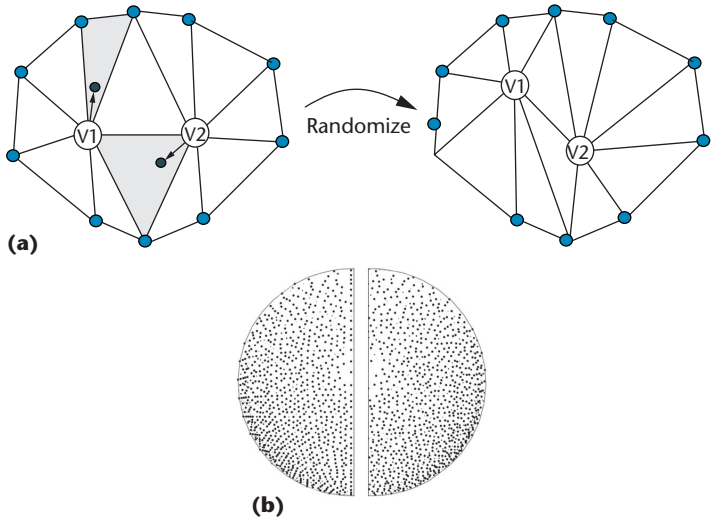
For mesh simplification, we create a vertex hierarchy by applying a series of edge-collapse operations until the model is simplified to a few vertices. The operator for mesh refinement is a variant of the edge-collapse operator[6] where one of the vertices is removed (see Figure 5). By performing mesh simplification, we can render models of complex geometry (such as horses, bunnies, or dragons) with only a few stipples by using the vertices at the top of the hierarchy. We generate a hierarchy by subdividing an input 3D model until we reach the desired number of vertices in the model or when the longest edge in the refined model falls under a certain threshold in object space.

For mesh subdivision, an edge-split operator creates a point around the middle of two vertices on the edge we want to split (see Figure 6). At each refinement step, we subdivide the longest edge in object space. Each vertex obtained by subdivision indicates the location of a new stipple. Figure 6 shows a wireframe view of a teapot before and after mesh refinement.

During rendering, we use a relevance function that draws stipples depending on the desired darkness at the vertex and the screen-space distances between the vertex and a group of relevant neighbors. We save the list of relevant neighbors for a vertex after applying either an edge-split or an edge-collapse operator. In addition, we compute a radius value as the average of the distance to the relevant edges, which is used for real-time rendering. Figure 7 shows the relevant edges and the definition of the radius for a given node.



7 **We save the vertices connected to a point affected by an edge collapse or an edge split in a list of relevant neighbors of the resulting vertex. We then determine the radius associated with the point.**



8 **(a) Application of the randomize operator on two vertices of an input mesh. (b) On the left, we show the original point distribution; on the right, we show the point distribution after randomization.**

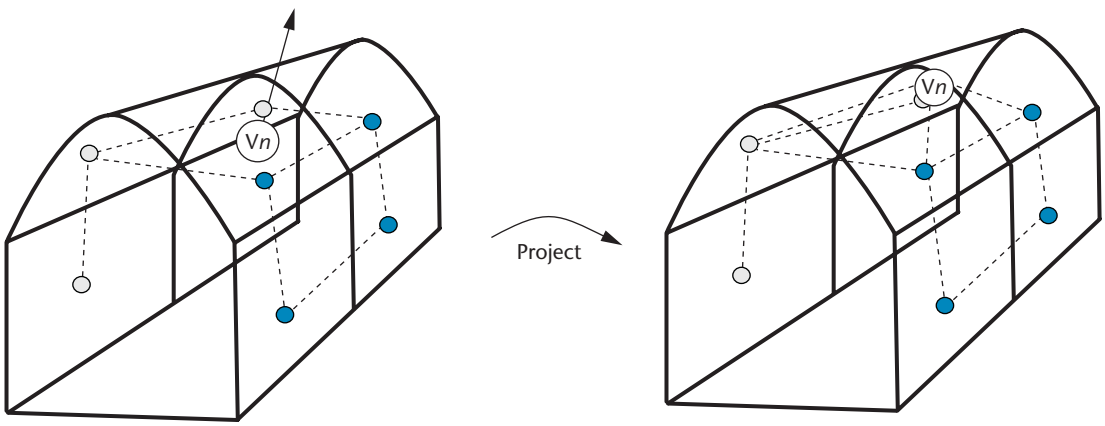
Because all vertices in the point hierarchy lie on the input model's surface, we can define a mapping between each vertex in the hierarchy and the polygon in the input model where the vertex lies. We define this mapping using the barycentric coordinates of the refined vertex with respect to its corresponding face in the model (which we call the *host face*). We then save this information in the vertex and assign each vertex a normal by interpolating the normals of the neighboring vertices (for gouraud shading) or by consulting the normal of the host face (for flat shading). We use this mapping for stippling animated models.

After the mesh simplification and subdivision stages, each point has a position in 3D space, a list of relevant neighbors, a radius value, the barycentric coordinates, and a normal vector. We store this information in an array, used later for rendering. The distribution of vertices in most input models is quite regular, which becomes noticeable in the form of linear patterns that appear when we render each vertex as a stipple. To reduce the presence of these patterns, we apply randomize and project operations to the vertices of the input model and to the vertices generated by mesh subdivision.

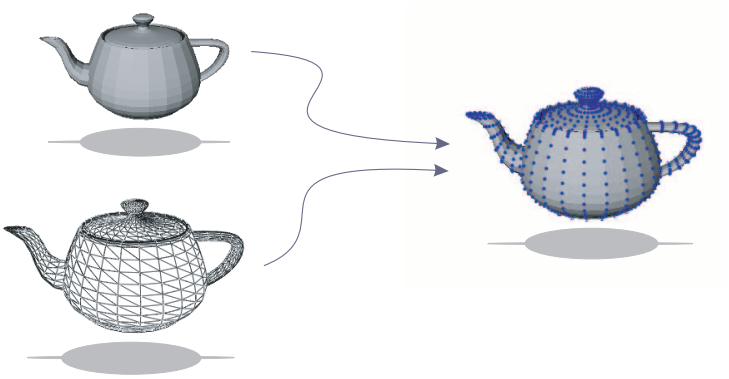
The randomize operator receives the vertex to be moved and the set of faces connected to the vertex. Figure 8a shows how we select a neighboring face at random and how we displace the vertex to a random position within that face. Figure 8b illustrates the effect of the randomize operator on the overall stipple distribution on a sphere. The composition shows the same half of the model before and after randomization.

The vertices generated by subdivision or displaced by randomization do not lie on the model's surface if the surface is not planar. The mesh that connects the randomized vertices has a different geometry than the input model mesh, as their vertices do not coincide. We project these vertices on the surface of the original model using the projection operator shown in Figure 9.

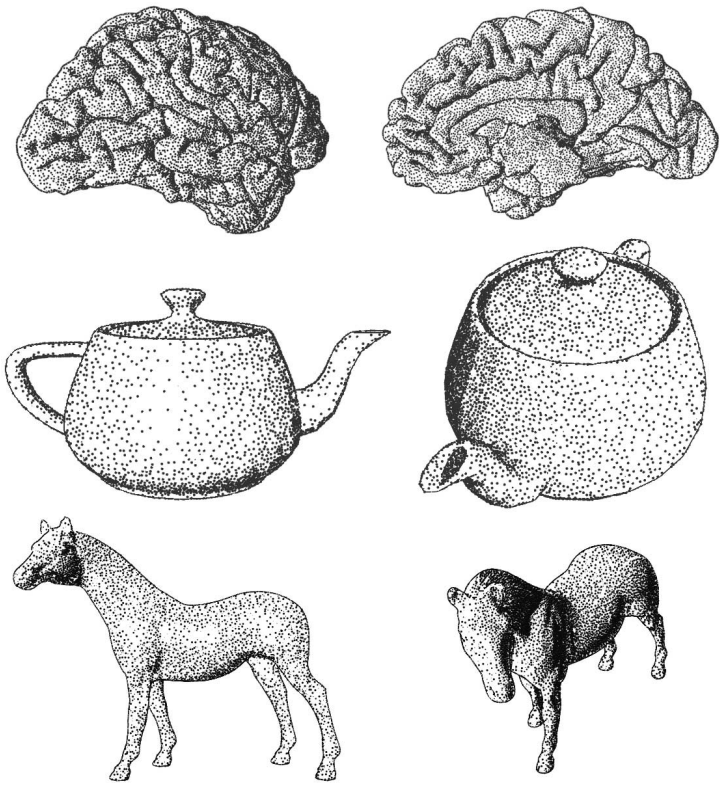
The projection operator defines a ray that departs from the input vertex toward the model's surface. We test the ray for intersection with the faces in the neighborhood of the input vertex and select a point that intersects with one of these faces and that lies closest to the input vertex. One way to avoid projection is to simplify and subdivide using the input geometry and the randomize operator to determine the position of the stipples without actually modifying the input mesh.



9 **The projection operator takes a randomized vertex and displaces it to the surface of the input model. In this illustration, black lines represent the input model and dashed lines represent the particle mesh.**



10 We obtain the frame-coherent stippling effect by rendering a set of points on the surface of the 3D model on top of the original model and smoothly varying the point size. Here, we have the teapot model and its vertices rendered as stipples on top of the model.



11 Frames from our stippled renditions of static models showing changes in lighting and viewpoint.

Point set hierarchy

We obtain a stippled rendition by rendering a 3D model and a set of points in 3D space on the surface of the model using the z-buffer for hidden-surface removal, and setting a small offset so that the faces of the model lie behind the point primitives. We obtain the frame-coherent effect by smoothly varying the point size across frames. We can disable color writes when drawing on the z-buffer to save bandwidth. We can obtain a more striking effect by using colors that differ from the background, as Figure 10 illustrates.

For the actual stippling, we traverse the complete point hierarchy and decide which stipples to draw and determine their point sizes by applying a rendering test on each potential stipple. First, we determine for each point a threshold value that lets it show up in the image. We define the threshold value as a function of lighting, viewing angle, target darkness, and viewport size. We then compare the screen space projection of every edge between the input point and its list of neighboring nodes of the vertex against the dynamically computed threshold.

If a connected edge falls below the threshold, we set the stipple size to zero. If the length of the shortest edge exceeds the threshold by a factor of N times, we set the point to a user-defined maximum size. If the length of the shortest edge is somewhere between the threshold and $N \times threshold$, we interpolate its size between these two values in a scale from zero to the maximum user-defined point size. The value N determines the smoothness of the transition from zero to full size. Because spatial criteria drive mesh simplification and subdivision, points appear or vanish roughly in the middle of existing points when rendering occurs. We also considered defining the stipples as set or unset after exceeding a certain threshold and smoothly varying the point size as a function of time. But this solution produced lag in shading, which became more apparent as the model moved rapidly.

Figure 11 shows frames taken from the stippling animations of static models. We computed silhouettes using the geometry information and the viewing angle to draw an edge as a 3D line if it connects a front- and back-facing face. The complete animations are available at http://isgwww.cs.uni-magdeburg.de/~oscar/. The preprocessing stage takes about 5 minutes for the system to create a 60,000-point hierarchy for the teapot model, which includes mesh simplification and subdivision. For the brain model, it takes about 18 minutes to create a 144,000-point hierarchy. Rendering a model with 60,000 points takes about 2.5 seconds. On average, using an SGI Onyx2 Infinite Reality computer with two 195-MHz processors, we can produce 700 frames of an offline animation in 1 hour for a model that has 40,000 points.

The sample animations show smooth transitions between frames and how stipple dots emerge and disappear between existing dots during rendering, yielding an interesting visual effect, as sand particles emerging and disappearing from the surface of the model. The stipples remain attached to the surfaces as the model moves. Having a fixed amount of stipples available for a model means that some areas lose darkness after the maximum number of stipples for that region has been used. To guarantee shading, it's possible to refine the model as needed during interaction, but this is only recommended for offline rendering because of the overhead implied when refining a highly tessellated mesh.

Hardware-accelerated rendering

Rendering a stippled drawing from the point hierarchy is a time-consuming task. You can subject the rendering algorithm to parallelization and process it with programmable vertex hardware.[11] Such hardware lets

the user control the size of each submitted vertex with a vertex program. Only data submitted with the vertex is accessible in the vertex program. That means we cannot literally transcribe the stipple-rendering algorithm as described in the previous section because it would require access to each vertex's neighbors. For this reason, we developed a simplified version of the algorithm that yields satisfying results.

The algorithm's main purpose is to reduce the calculations based on each neighbor's distance to a single scalar value (the stipple radius). The vertex program computes a threshold value based on distance, slope, and lighting. We vary the point size based on the difference of threshold and radius to achieve a smooth introduction and fading with the same function used for conventional graphics pipelines. For maximum efficiency, we store the vertex array in video memory. Using this version of the algorithm, we can display a model with 120,000 stipples at 60 frames per second using an Nvidia GeForce 4 card. You can view the video showing real-time stippling of several models at http://computer.org/cga/cg2003/g4toc.htm.

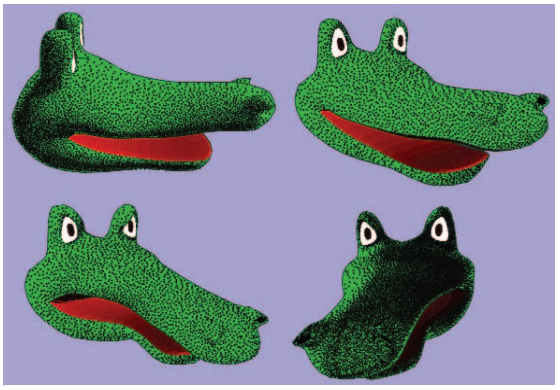
Stippling animated models

Most nonphotorealistic techniques applied to models in 3D space work well for static models, but researchers have done little work in applying nonphotorealistic styles to animated models. The challenge lies in scaling and defining how nonphotorealistic particles should adapt to changes in a mesh's shape. We have found that stippling as a rendering style is well suited to producing computer animations, because we can use the point hierarchy as an elastic texture attached to the surface of the model (see Figure 12). The main requirement for producing animated stippling is to start with a mesh that has a fixed number of vertices. You then have the mesh's vertices change in position over time to produce the animation sequence.

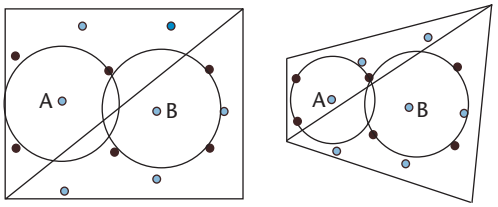
To attach the stipples to the surface of the mesh during the animation, we recompute the position of each point in the hierarchy at each frame using barycentric coordinates. In the barycentric coordinate system, we define the position of the points within the surface of a triangle with respect to the triangle's vertices, which is convenient for stippling because we can move the vertices of a triangular mesh in 3D space and then recompute the positions of the stipples as a function of these vertices. As a result, the points on the surface of the mesh move along with the triangles as the mesh is distorted (see Figure 13).

Because there are several stipples per face, each point in the stipple set contains an index to the face of the model on which the point lies (the host face for that stipple) and the barycentric coordinates of the point within that face. At each animation step, we recompute the point coordinates using the barycentric coordinates and the vertex positions of the host face. With this technique, the stipples behave like an elastic texture printed on the model's surface.

If we don't touch the stipple radii during the animation, the overall point density becomes a function of the mesh distortion. That is, the point density increases on those regions of the model that shrink and decreases in those regions that expand, which is interesting for illustrating mesh deformation. However, to keep the stippling as a function of shading and scale, as originally postulated, we need a more elaborate solution to compensate for the distortion's effect. In regions that shrink, we should draw fewer stipples, and in regions that expand, we should draw more stipples—assuming the illumination conditions and the user viewpoints remain constant while the distortion takes place.



12 **Several sample frames from our animation *Upsetting the Crocodile*.**



(a) **(b)**

13 **(a) Points A and B have a radius value that is the average distance to their relevant neighbors. (b) Two points with their new average radius after the distortion. Because the distance to their neighbors has changed after the transformation, the values for the radius of the stippled points also change, but in different proportion for each point.**

To allow stipple particles to adapt to the new polygon shape while maintaining appropriate shading and scale, we compute the radius of each stipple as the average of a small set of neighboring points for each frame. We define the set of neighboring points when we generate the point set. By defining the neighboring points in barycentric coordinates, the position and the radius of the stipples can adapt to mesh deformations. The point density varies proportionally to the distance from the stipple to the original neighboring points, as Figure 13 illustrates.

Ideally, more stipples should be generated when the surfaces are expanded and some should be eliminated when the surfaces shrink. However, this puts additional overhead on the rendering process, which is not necessary if all the stipples potentially needed are generated in a preprocessing stage.

Future directions

Our implementation makes use of several techniques that we could individually optimize to improve the system efficiency and the overall renderings. For instance, we could improve the point distribution by using an algorithm to redistribute the vertices on the surface of the input model as part of preprocessing. We could also try different approaches for mesh simplification to improve the selection of vertices while constructing the point hierarchy.

It would be worth determining the visibility of the original model's faces in the first pass and rendering only stipples belonging to visible faces. A simple visibility test would be back-face elimination for closed objects. This test should reduce the number of stipples by roughly one half. An even more aggressive method would be to employ the occlusion test provided by some graphics hardware. Both options would require breaking up the model into parts with associated stipple sets.

If all polygons in a certain part are back facing or invisible, the part's stipples do not have to be submitted to the graphics pipeline. We could also use a screen-based level-of-detail approach to render only the relevant points in the hierarchy according to the projection of the input polygons on the model's surface.

We also plan to develop an efficient algorithm that will fill specific areas of the model with stipple particles when the user zooms in and exhausts the point hierarchy. ■

Acknowledgments

We thank Lourdes Peña Castillo for reviewing drafts, the members of the Institute for Simulation and Graphics at the Otto-von-Guericke University of Magdeburg—especially Nick Halper—for providing feedback toward the development of this technique, and Stefan Schlechtweg for his comments on this work. We also thank Oleg Veryovka for pointing out the feasibility of using vertex programmable hardware for real-time rendering, Bernd Eckardt for implementing the vertex program for real-time rendering, Thomas Witzel for providing the brain model, and Oliver Deussen and Ron C. Guthrie for their stippled renditions. This work was partially supported by a scholarship of the state of Sachsen-Anhalt, Germany.

References

1. O. Deussen et al., "Floating points: A Method for Computing Stipple Drawings," *Computer Graphics Forum*, vol. 19, no. 3, 2000, pp. 40-51, http://www.eg.org/EG/CGF/volume19/issue3.
2. A. Secord, "Weighted Voronoi Stippling," *Proc. 2nd Int'l Symp. Nonphotorealistic Animation and Rendering*, ACM Press, 2002, pp. 37-43.
3. D. Cornish, A. Rowan, and D. Luebke, "View-Dependent Particles for Interactive Nonphotorealistic Rendering," *Graphics Interface*, June 2001, pp. 151-158, http://www.graphicsinterface.org/proceedings/2001/158/.
4. L. Markosian et al., "Art-Based Rendering with Continuous Levels of Detail," *Proc. 1st Int'l Symp. Nonphotorealistic Animation and Rendering*, ACM Press, 2000, pp. 59-66, http://doi.acm.org/10.1145/340916.340924.
5. M. Kaplan, B. Gooch, and E. Cohen, "Interactive Artistic Rendering," *Proc. 1st Int'l Symp Nonphotorealistic Animation and Rendering*, ACM Press, 2000, pp. 67-74, http://www.cs.utah.edu/npr/papers.html.
6. H. Hoppe, "Progressive Meshes," *Proc. Siggraph 96*, ACM Press, 1996 pp. 99-108, http://doi.acm.org/10.1145/237170.237216.
7. D. Luebke, "Developer's Survey of Polygonal Simplification Algorithms," *IEEE Computer Graphics and Applications*, vol. 21, no. 3, May/June, 2001, pp. 24-35.
8. O. Meruvia and T. Strothotte, "Frame-Coherent Stippling," *Eurographics 2002 Short Paper Proc.*, 2002, http://isgwww.cs.uni-magdeburg.de/~oscar/.
9. A. Secord, W. Heidrich, and L. Streit, "Fast Primitive Distribution for Illustration," *Proc. Eurographics Workshop on Rendering*, Eurographics, 2002, pp. 215-226.
10. E.R.S. Hodges, *The Guild Handbook of Scientific Illustration*, Wiley Europe, 1988.
11. E. Lindholm, M. J. Kilgard, and H. Moreton, "A User-Programmable Vertex Engine," *Proc. Siggraph 01*, ACM Press, 2001, pp. 149-158.

Oscar Meruvia Pastor *is a PhD student at the Otto-von-Guericke University of Magdeburg in Germany. His research interests include nonphotorealistic animation and rendering and human–computer interfaces. Meruvia received a BSc in computer science engineering from the Instituto Tecnológico y de Estudios Superiores de Monterrey (ITESM), Mexico and an MS in computer science from the University of Alberta, Canada.*

Bert Freudenberg *is a PhD candidate at the Otto-von-Guericke University of Magdeburg. His research interests include real-time rendering, nonphotorealistic computer graphics, and interactive educational environments. Freudenberg received a Diplom in computer science from the Otto-von-Guericke University of Magdeburg.*

Thomas Strothotte *is professor at the Otto-von-Guericke University of Magdeburg, where he is the chair of Graphics and Interactive Systems. His research interests include smart graphics, image-text coherence, rendering illustrations, and virtual communities. Strothotte received a PhD in computer science from McGill University.*

Readers may contact Oscar Meruvia Pastor at the Dept. of Simulation and Graphics, Otto-von-Guericke University of Magdeburg, Postfach 4120, D-39016 Magdeburg, Germany; oscar@isg.cs.uni.magdeburg.de.