Real-Time Animated Stippling

To appear on the IEEE Computer Graphics and Applications
July/August 2003
Special Issue on Non-Photorealistic Rendering

Oscar Meruvia Pastor Bert Freudenberg Thomas Strothotte
Department of Simulation and Graphics
Otto-von-Guericke University of Magdeburg
{oscar,bert,tstr}@isg.cs.uni-magdeburg.de

***Abstract*— Stippling is an artistic rendering technique where shading and texture is given by placing points or stipples on the canvas until the desired darkness is achieved. Computer-generated stippling has focused on producing high quality 2D renditions for print media, while stippling of 3D models in animations has received little attention. After describing current advances in stippling for print media and real-time rendering, we present an approach to produce animations of 3D models using stippling as a rendering style. In our approach, we ensure frame-to-frame coherence as the model moves and shading changes over time, by attaching stipple particles to the surface of the model. We present a point hierarchy, used to control the stipple density during rendering, and solutions for rendering animated and static models using conventional and vertex-programmable graphics hardware.**

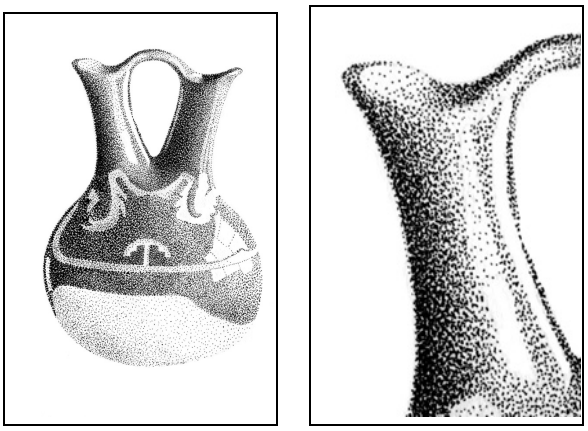

Fig. 1
ON THE LEFT, "INDIAN POTTERY" BY RON C. GUTHREY. ON THE RIGHT, A DETAIL FROM THE SAME IMAGE, WHERE THE INDIVIDUAL STIPPLE DOTS CAN BE MORE CLEARLY APPRECIATED.

I. INTRODUCTION

Stippling is a painting technique where the artist renders an image using single dots, also called stipples. Almost any model from nature and human manufacture can be rendered using stipples. In scientific illustration (for example in the natural sciences or in archaeology), stippling is used in combination with other rendering techniques to convey the shape, texture and surface material of the objects being depicted. Figure 1 shows a vase that is rendered by an artist using the stippling style. A closer look at the image reveals that darker areas are more densely stippled than lighter areas, and that stipples have more or less a constant size. Stippled renditions do not scale well, specially when the image is drastically reduced or looked at from a distance, because the stipples become too small and blend with each other in the image. While stippling, the artist must take care to judge the proper scale and spacing of individual stipple dots, since the density of the dots conveys both shape and tone.

The original work for computer-generated stippling focuses on the creation of high-quality renditions of 2D images at high resolutions. These renditions are normally one-time productions intended for printed media. Image-based approaches such as that of Deussen et al. [2] and Secord [10] provide interactive and automatic tools to obtain high quality renderings in the stippling style for single renditions (see Figure 2). These techniques have focused on the even distribution of a number of input dots using Voronoi relaxation.

In this article we describe our approach to produce computer animations of 3D models in the stippling style. There are several issues that make stippling for animations a hard topic for graphics researchers. The first issue is a conceptual one, and refers to the question of how the stipples in an animation of a stippled object should behave. Since the stippling technique is originally meant to produce single images at a certain scale, it is not clear how the stipples should react to scaling and changes in shading through illumination. Ideally, we would like to obtain stippled renditions of a model which can be arbitrarily scaled, but this can be an elusive goal. Another important issue for stippling is how to maintain even point distributions as the viewing parameters (viewpoint, illumination, viewing distance) change, or even as the model itself changes. Our contribution to this area is the introduction of a point hierarchy that can be used within a rendering procedure to produce view-dependent, frame-coherent animations in the stippling style for static and animated 3D models[1].

In the next section we describe our approach to obtain frame-coherent stippling. In Section III we review related

[1]The complete animations are available under http://isgwww.cs.uni-magdeburg.de/~oscar/

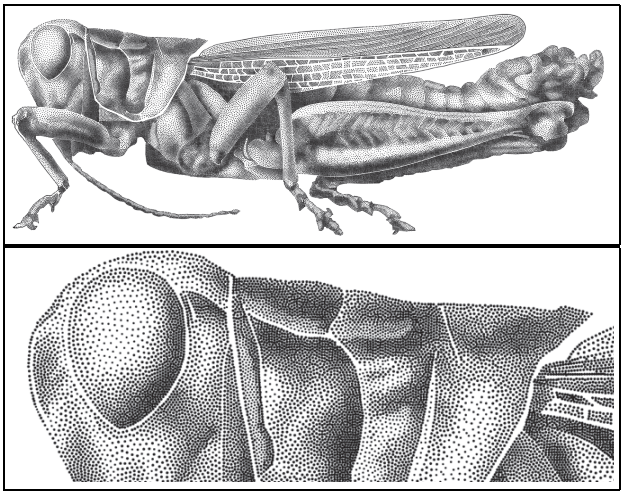

Fig. 2
ON THE TOP, A STIPPLED IMAGE PRODUCED BY VORONOI RELAXATION, ON THE BOTTOM A DETAIL OF THE GRASS-HOPPER'S HEAD (IMAGES BY DEUSSEN ET AL.[2])

work on computer-generated stippling. Section IV explains in detail how the point hierarchy is created. Section V describes how the point set is used for rendering in the stippling style in conventional graphics hardware, the hardware-accelerated implementation, and stippling of animated models. In Section VI we discuss our results and the open problems.

II. FRAME-COHERENT STIPPLING

As a starting point we may notice that stippling and most other artistic drawing styles cannot be used in an animation by simply putting together a sequence of independently rendered images without introducing noise during the animation. While artists produce animations in the hatching style by redrawing each frame and pasting them together in a sequence, it would be useless to try the same technique with stippling because we would have stipples randomly appearing and disappearing from the image. This effect would not be appealing to the viewer and it could even become annoying after a short while, depending on the amount of noise perceived. This is why we advocate the implementation of stippling in a frame-coherent way at the level of each individual stipple. Under this view, a stipple should be attached to the surface of the model and should move along with the model, in other words, it should behave like a texture on the surface of the model. In addition, the stippling density should smoothly adapt to changes in illumination, i.e. it should increase when shading becomes darker and decrease when shading is lighter. Since stipples are not allowed to move, stipples can only blend in or out of the images. Finally, we want to keep appropriate spacing between rendered stipples, trying to avoid the formation of regular patterns or irregular grouping of points as much as possible.

Our frame-coherent stippling system is implemented by defining a point hierarchy which is fixed to the surface of a model and a rendering algorithm that uses this point hierarchy to produce stippled renditions in a way that satisfies the conditions previously mentioned.

To achieve frame-coherence, we have taken the concept of *particle systems* from *painterly rendering* and *artistic rendering*, where particles, *graftals* [7] or *geograftals* [4], are fixed on the surface of 3D models. In principle, we consider each vertex of the input model as a particle that indicates the location of a potential stipple. Because each point is attached to a specific location on the surface of the model, points move along with the model as the model is moved in the scene. This provides the frame-coherence effect at the stipple level.

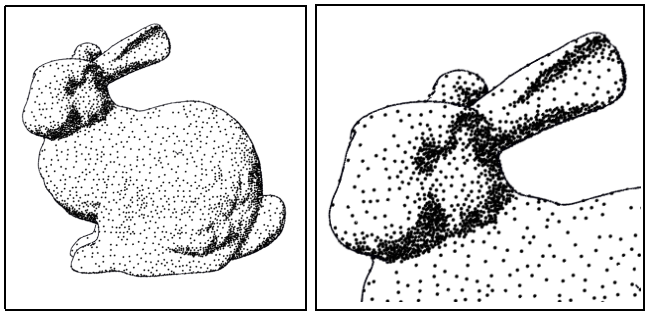

Fig. 3
ON THE LEFT, THE STANFORD BUNNY IN THE STIPPLING STYLE, OBTAINED BY FRAME-COHERENT STIPPLING. ON THE RIGHT, A DETAIL OF THE BUNNY'S HEAD.

In addition, we control the stipple distribution on the surface of the model so that it dynamically adapts to changes in the shading, allowing dark areas to be filled with more stipples than light shaded areas (see Figure 3). We do this by creating a hierarchy of points which is used to determine which points should appear or disappear first from the images. Stippling is not scalable per se, but 3D models are, so we use the point hierarchy to control the stipple density according to changes in scale and viewing distance as well (see Figure 4).

The point hierarchy is generated in the same way as vertex hierarchies for mesh simplification and Level-of-Detail [3], [6] are, i.e., the edges of a mesh are subsequently refined and a vertex hierarchy is produced as a result. In fact, Cornish et al. [1] introduced the idea of applying mesh simplification in a view-dependent real-time NPR system and we incorporated this as part of our system. Vertices which are lower in the hierarchy, appear last and vanish first than vertices which are higher in the hierarchy. We extended this idea by adding a mesh subdivision stage which we use to generate more stipples when needed. After each simplification and each refinement step, the resulting vertex is assigned a list of neighbours (alternatively, a relevance value) that we use to decide which stipples should be included in a particular rendition. The initial approach for frame-coherent stippling was presented in [8], where a model was refined as needed as part of an off-line animation. In this article, we have extended this approach to include animated models and real-time rendering.

III. RELATED WORK

The first stippled renditions were presented by Winkenbach and Salesin for parametric surfaces using randomness to

distribute the points on the surface of a model. Deussen et al. [2] and Secord [10] obtain high-quality stippled images using dithering and relaxation of Voronoi diagrams and taking greyscale images as input (recall Figure 2). While these images are visually attractive and the stipple dots are carefully distributed, it is not possible to use these approaches to build a noise-free animation sequence, because each frame is obtained by iterative Voronoi relaxation of existing particles and the stipple distribution obtained in one frame is not guaranteed to correspond with the one obtained in the next frame. A question that arises when we look at existing works for 2D stippling and try to extrapolate it to 3D models is whether it is 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. An interesting proposal in this respect is that of Secord et al. [11], where frame-coherence is pursued on the image plane, not in object space (i.e. the rendering particles are not attached to the model's surface, but to the image). Their results show that by enforcing frame-coherence in this way, stipple particles float (or swim) on the surface of the 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 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 be used to emulate stippling and adapt to changes in illumination at a given viewing distance. The problem with textures, however, is that it is hard to control the stipple shape in such a way that it is always projected as a circular dot on the screen. We avoid this problem by drawing each stipple explicitly with point primitives.

IV. Generating the Point Set Hierarchy

In this section we describe the properties of the point set hierarchy, and how it is generated. In section V, we describe how the hierarchy is used to produce stipple renderings using the point hierarchy.

When stippling, it is important to obtain regular point distributions on the final rendition. According to Hodges [9], artists create stippled drawings by first placing some groups of dots in a region of interest and then filling in until the desired tone is achieved. Our point hierarchy is defined in such a way that spacing of stipples is taken into account when adding and removing stipples: new stipples (which are inserted lower in the particle hierarchy) are placed at locations roughly in the middle of existing stipples. Alternatively, when stipples are removed from the surface of a model, stipples at the bottom of the hierarchy are the ones that vanish first.

Figure 4 illustrates how stipples in the higher levels of the hierarchy are sparsely distributed, and how new stipples are added between existing ones. In the left side of the image, we can observe how distance values can be assigned to the stipples according to their hierarchy level. This distance is a key value used to determine whether a stipple should be rendered or not.

In our system we follow the strategy proposed by Hodges by creating a hierarchy of vertices in 3D space which represent

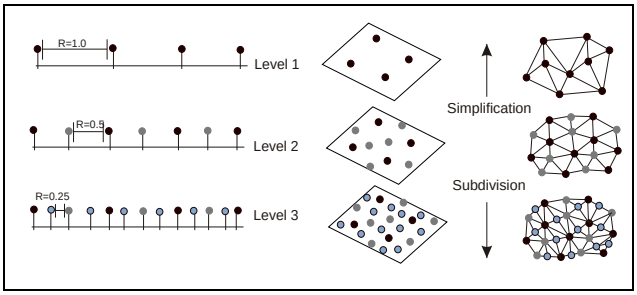

Fig. 4

Point hierarchies for the one- (left), two- (middle) and three dimensional cases (right). Points at the lower levels of the hierarchy have smaller radius values, which determines their relevance in the hierarchy. For 3D models, a continuous level of detail is created using mesh simplification and subdivision.

stipple locations on the surface of the model. Depending on the viewing distance and the number of polygons in the input model, the number of vertices of the input model might not be enough to cover darks areas. To fill these areas, more vertices are generated on the surface of the model by mesh subdivision. In other cases, the number of vertices in the input model is so high that we have to 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 the appropriate level of detail is obtained at most viewing ranges, we mix mesh simplification and mesh subdivision in a preprocessing stage to provide seamless levels of detail regardless of the resolution of the input model (see Figure 4, right). Vertices at the top of the hierarchy are the initial group of dots spatially distant from each other; the vertices down the hierarchy fill-in the space between existing vertices, so that new stipples always come up to fill-in uncovered regions of the canvas until the desired tone is achieved.

To generate the point hierarchy the system does the following steps:

1) Compute a connectivity graph to operate on the input polygonal mesh.
2) Apply a randomize phase on the vertices of the input mesh to reduce the presence of regular patterns in the stipple distribution (see Section IV-E).
3) Perform mesh simplification on the input mesh, creating a hierarchy for the vertices in the input mesh.
4) Perform mesh subdivision on the input mesh, up to a desired level of detail, or a desired point count, expanding the existing point hierarchy with the new vertices.

A. Setting up the connectivity graph

As a setup stage, a connectivity graph based on the input polygonal mesh is created, which contains information about the connections between vertices, edges and faces of the model, as well as the vertices positions in object space. This information is used for mesh simplification, subdivision, randomization and vertex projection. We consider each vertex

to be a particle for potential rendering, so the initial vertex distribution is the collection of vertices of the input mesh.

B. Mesh Simplification

In 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 that we use for mesh refinement is a variant of the *edge collapse* (ecol) introduced by Hoppe [3] where one of the vertices is removed (see Figure 5, top).

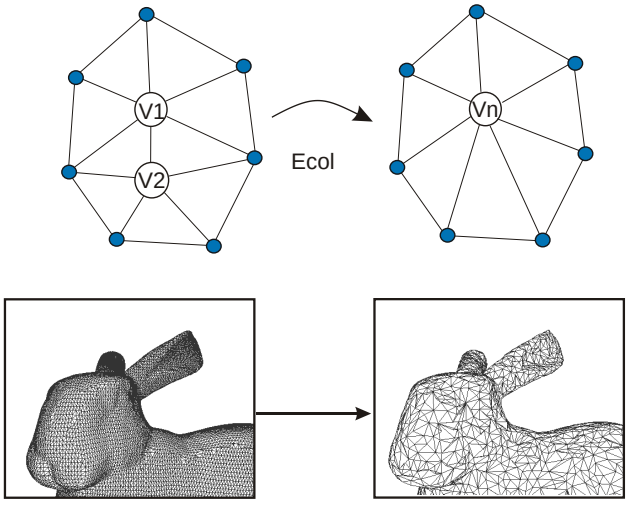

Fig. 5

TOP: THE EDGE COLLAPSE OPERATION USED IN MESH SIMPLIFICATION [3]. BOTTOM: WIREFRAME VIEW OF THE ORIGINAL BUNNY MODEL (LEFT) AND THE MODEL AFTER A SERIES OF SIMPLIFICATION STEPS (RIGHT).

By performing mesh simplification we can take models of complex geometry (like horses, Stanford bunnies or dragons) and render them with few stipples by using the vertices at the top of the hierarchy (see Figure 5, bottom).

C. Hierarchical Subdivision

We generate a hierarchy by subdividing (refining) an input 3D model iteratively until the desired number of vertices in the model has been reached, or when the longest edge in the refined model falls under a certain threshold in object space.

The operator that we use for mesh subdivision is an edge-split that creates a point around the middle of two vertices of the edge to be split (see Figure 6, top). At each refinement step, the longest edge in object space is subdivided. We always take the longest edge to avoid creating extremely thin triangles, which would appear if only a specific region of a model is refined. Each vertex obtained by subdivision indicates the location of a new stipple, and its neighbours are included in the list of relevant neighbours for use in rendering. Figure 6 (bottom) shows a wireframe view of the teapot before and after mesh refinement.

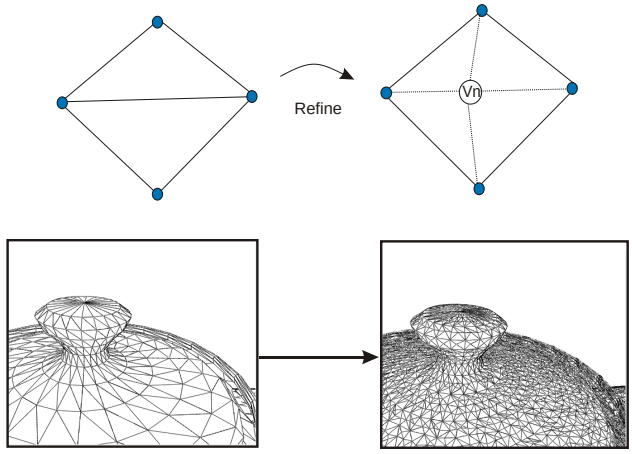

Fig. 6

TOP: THE REFINEMENT OPERATION USED IN MESH SUBDIVISION. BOTTOM: WIREFRAME VIEW OF THE ORIGINAL TEAPOT MODEL (LEFT) AND THE MODEL AFTER A SERIES OF REFINEMENT STEPS (RIGHT).

D. Defining the Point Set Hierarchy

Since a stipple covers an area of influence delimited by the stipples in its neighbourhood [2], [10], we define a relevance function where a particle is drawn depending on the desired darkness at the vertex and the screen-space distances between the vertex and a group of relevant neighbours (see section V-A). The list of relevant neighbours for a vertex is saved after applying either the edge-split or the edge-collapse operators. 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.

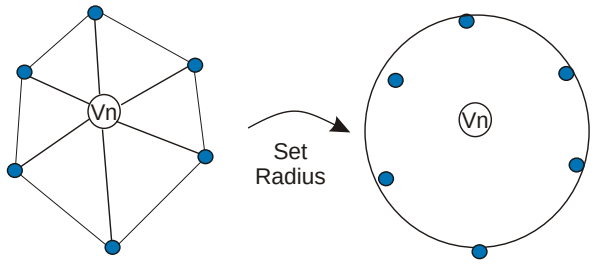

Fig. 7

THE VERTICES CONNECTED TO A POINT AFFECTED BY AN EDGE COLLAPSE OR AN EDGE SPLIT ARE SAVED IN THE LIST OF RELEVANT NEIGHBOURS OF THE RESULTING VERTEX, AND DETERMINE THE RADIUS ASSOCIATED WITH THE POINT.

Since all vertices in the point hierarchy lie on the surface of the input model, we can define a mapping between each vertex in the hierarchy and the polygon in the input model where the vertex lies. We do this by defining the barycentric coordinates of the vertex with respect to the corresponding face in the model (which we call the host face), and save this information

in the vertex. This is used for animated stippling. Last but not least, each vertex is assigned a normal which can be obtained either by interpolating the normals of the neighbouring vertices (for gouraud shading) or by consulting the normal of the host face (for flat shading). In sum, after the mesh simplification and subdivision stages have taken place, each point has the following information:

- Position in 3D space.
- Indexed list of relevant neighbours or the radius value for real-time rendering.
- Barycentric coordinates and index of the host face in the input model.
- Normal vector.

Using this scheme, points with shorter edges (or shorter radius) are drawn only after points with larger edges (or larger radius) have been drawn, assuming these points lie on an evenly shaded surface. If this is not the case, the rendering algorithm determines for each point the required distance that will allow it to show up in the image, depending on the desired darkness at the stipple's position. Since both mesh simplification and subdivision are driven by spatial criteria (closest edges are simplified first, and new points are placed roughly in the middle of existing points), points down the hierarchy appear or vanish between existing points when rendering.

E. Improving the stipple distribution

The distribution of vertices in the original model is most of the times quite regular. This becomes noticeable when we render each vertex as a stipple and is a problem because this introduces linear patterns which are not desirable from an esthetic point of view. In addition, our subdivision operator also tends to generate linear point distributions, since it creates vertices along existing edges of the model.

To reduce the presence of these patterns, "randomize and project" operations are applied to the vertices of the input model before proceeding to mesh simplification and to the vertices generated by mesh subdivision.

1) Randomization: The randomize operator receives as input the vertex to be moved and the set of faces sharing the vertex. The vertex is displaced within the region enclosed by these faces, so we first select a neighbouring face at random and then we displace the vertex to a point in this face which falls within a range that covers a small region (user-defined) between the input vertex and the other vertices of the face (see Figure 8, top). The region cannot be the whole face, because we could displace the vertex to a position too close to the other vertices and we would generate thin polygons, which we want to avoid in general.

Figure 8 (bottom) illustrates the effect of the randomize operator on the overall stipple distribution. The image on the left shows a point distribution obtained by applying only the subdivision operator on a sphere, where some stipples are arranged in linear distributions. The image on the right shows the stipple distribution obtained when the randomize operator is applied after each subdivision.

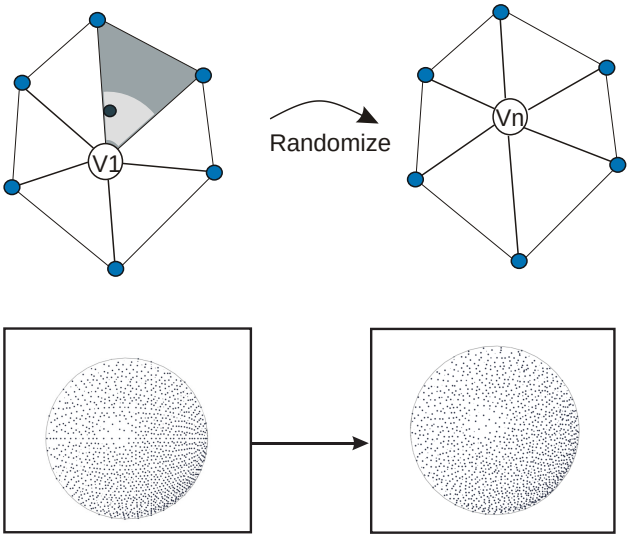

Fig. 8

TOP: THE RANDOM OPERATOR DISPLACES THE INPUT VERTEX TO A NEW LOCATION WITHIN THE NEIGHBOURING FACES. BOTTOM: AN EXAMPLE OF THE STIPPLE DISTRIBUTION ON A SPHERE BEFORE AND AFTER RANDOMIZATION.

2) The Projection Operator: A randomized vertex should lie on the surface of the input model after randomization, because we perform hidden surface removal using the z-buffer. However, the mesh that connects the randomized vertices has a different geometry than the mesh of the input model (because their vertices do not coincide) so we need to project the randomized vertices to the surface of the original model. The only case where a randomized or a newly created vertex is guaranteed to lie on the surface of the input model is when all the vertices of the polygon fan around it are coplanar and lie on the surface of the model. Otherwise, the vertex has to be projected back on the surface using a projection operator.

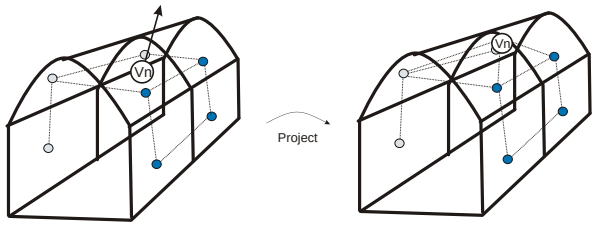

Fig. 9

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 THIN LINES REPRESENT THE PARTICLE MESH.

The projection operator (see Figure 9) defines a projection ray departing from the input vertex towards the direction of the sum of normals of the vertices connected to the input vertex.

After that, the faces of the input model which are connected to the neighbor vertices are tested for intersection with the projection ray. If more than one intersection is found, we select the one that lies closest to the input point and which is not invalid. An invalid projection is one that creates a fold in the particle mesh, which can occur if the vertex is projected past an edge that connects two neighbours of the input vertex.

V. Using the Point Set Hierarchy

A. Rendering in conventional graphics pipelines

A stippled rendition is produced using a 3D model and a set of points in 3D space. We render the model with a small offset behind the point primitives and use the z-buffer for hidden-surface removal. The buffer is initialized by rendering the original object's faces in a pass before the actual stippling. Color writes can be disabled for this pass, to save bandwidth. However, it may be desirable for objects to have a base color differing from the background. This can easily be done by using an arbitrary color for this pass. Figure 10 illustrates this approach. On the top left, we see the teapot model rendered in the flat shading style, On the bottom left, we see the model rendered in the wireframe style. On the right side of the image, we have the teapot model and the vertices of the teapot rendered as stipples on top of the model.

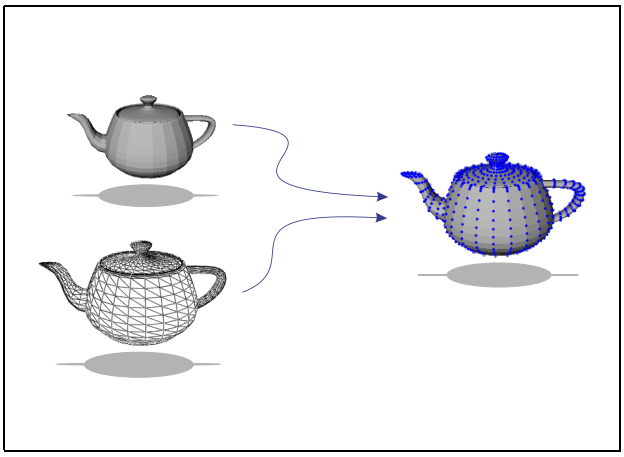

Fig. 10

THE FRAME-COHERENT STIPPLING EFFECT IS OBTAINED BY RENDERING A SET OF POINTS IN 3D SPACE ON THE SURFACE OF A 3D MODEL, USING THE z-BUFFER FOR HIDDEN-SURFACE REMOVAL AND SETTING A SMALL OFFSET WHERE THE FACES OF THE MODEL LIE A BIT BEHIND THE POINT PRIMITIVES FROM THE VIEWER'S POINT OF VIEW.

For the actual stippling, we traverse the complete point hierarchy and decide which stipple should be drawn by applying a rendering test on each potential stipple. The rendering test works as follows: The screen-space projection of every edge in the list of neighbouring nodes is measured in pixels and compared against a dynamically computed threshold value. If a connected edge falls below the threshold value, the stipple is unset. If all edges exceed the threshold, the stipple is set to be drawn. The threshold value is computed for each particle as a function of the illumination model, the normal of the vertex and the viewing parameters.

In our rendering system we have stipples of variable point size which goes from zero to a user-defined maximum point size. We use OpenGL smoothing for points, blending and multipass antialiasing to ensure that stipples are smoothly introduced in or removed from the images during animation. The size of the stipple is defined as a function of the threshold, the radius of each individual stipple and a user-defined fade-in factor which defines how fast a stipple fades in or out of the image. We also considered defining the stipples as being set or unset after a certain threshold is exceeded, and smoothly varying the point size across frames to fade-in or out the stipples. This solution however produced a lag in shading, which became more apparent as the model moved rapidly, and was thus discarded.

In Figure 11 we present frames taken from the stippling animations of static models, where models are shown under different conditions of lighting, viewing and model orientation. The complete animations are available under 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, which includes mesh simplification and subdivision. For the brain it takes about 18 minutes to create a 144,000 point hierarchy, which is entirely spent on simplification. Rendering a model with 60,000 points takes about 2.5 seconds. In average, 700 frames of an off-line animation are produced in one hour for a model with 40,000 points on a SGI Onyx2 Infinite Reality, with 2 195MHz MIPS R10000 processors and 900MB RAM.

The animations presented show smooth transitions between frames, and how stipple dots emerge and disappear between existing dots, yielding an interesting visual effect, as if sand particles emerged or disappeared from the surface of the model, but on the other hand, they remain attached to the surfaces as the model moves.

Having a fixed amount of stipples available for a model has the drawback that some areas loose darkness (i.e. the stipple density on the image decreases) after the maximum number of stipples for that region has appeared, which occurs when the user zooms at the model. If it is required that shading is always guaranteed, it is possible to refine the model further during interaction, but this is only recommended for off-line rendering, because of the overhead implied by having to refine a highly tessellated mesh.

B. Hardware-accelerated rendering

Rendering a stippled drawing from the point hierarchy is a rather time-consuming task, as described in the previous section. On the other hand, the rendering algorithm can be subject to parallelization. In this section, we describe an implementation that can be processed in parallel by programmable vertex processing hardware [5].

Such hardware allows the user to control the processing of each submitted vertex with a *vertex program*. However, each vertex is processed independently of the others. Only data submitted with this vertex is accessible in the vertex program.

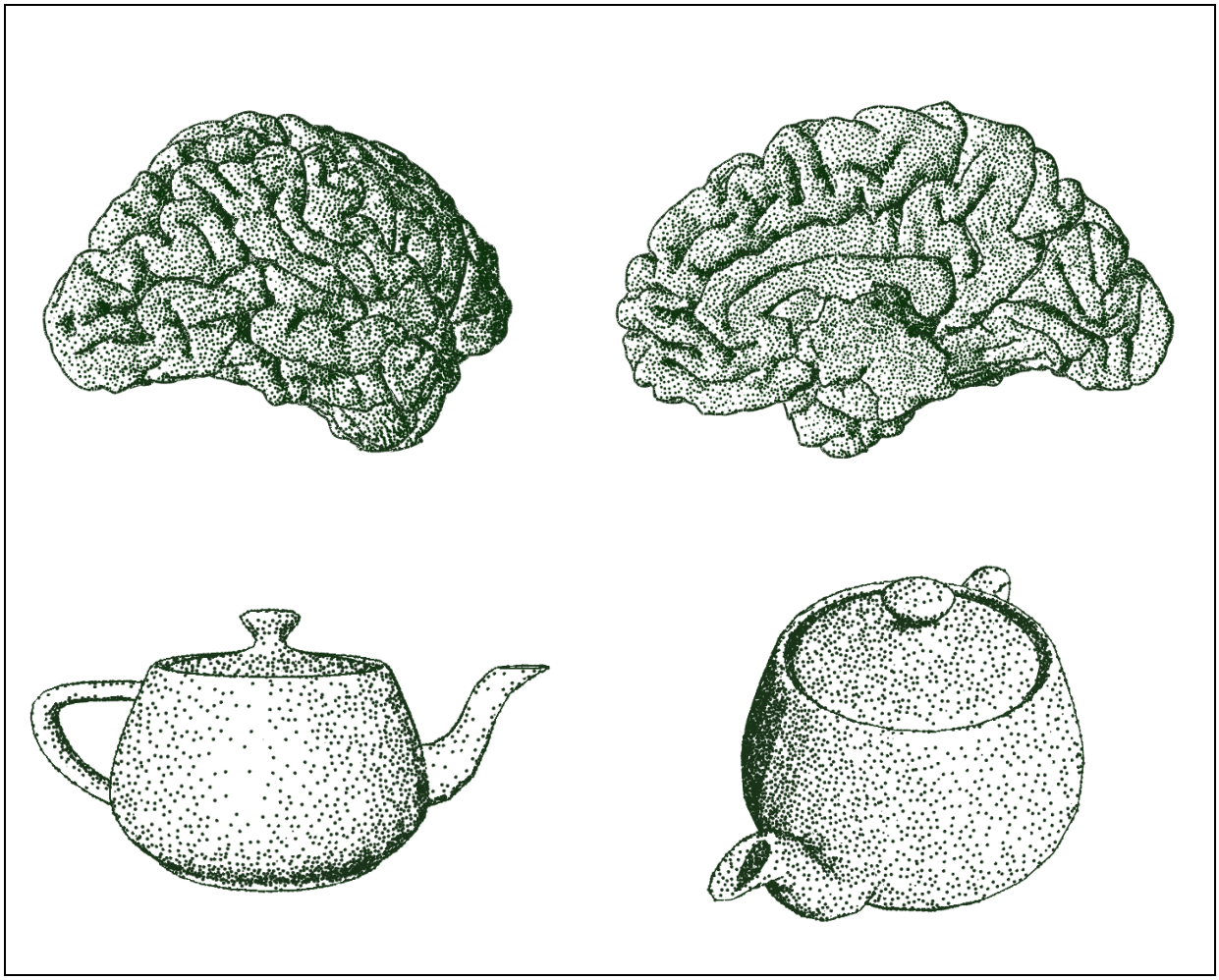

Fig. 11
FRAMES FROM OUR STIPPLED RENDITIONS OF STATIC MODELS, SHOWING CHANGES IN LIGHTING AND VIEWPOINT.

That means we can not use a literal transcription of the stipple rendering algorithm as described in the previous section, because it would require access to each vertex' neighbours. For this reason, we developed a simplified version of the algorithm which still yields satisfying results.

The algorithm's main idea is to reduce the calculations based on neighbour's distances to a single scalar value, the stipple *radius* computed as the Point Set Hierarchy is created, as described in Section IV-D. This value is stored for each vertex, along with its position and normal. In the vertex program, a *threshold* value is computed 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 of stipples with the same function used for conventional graphics hardware (see previous Section).

For maximum efficiency, we store the vertex array in video memory. An optimization we have not yet employed is limiting the number of potential stipples tested for each object. This could be achieved by estimating the number of stipples based on the area of the object's screen-space projection. However, the depth-complexity of the object would have to be taken into account, too. Furthermore, it is difficult to estimate the actual lighting so the darkest tone would have to be assumed.

Using this version of the algorithm, we can display a model with 120,000 stipples at 60 fps on a NVIDIA GeForce 4.

C. Stippling Animated Models.

Most of the non-photorealistic techniques which are applied to models in 3D space work well for static models, but relatively few work has been done to apply non-photorealistic scenes for animated models. The challenge lies again in scaling and in defining how non-photorealistic particles or strokes should adapt to changes in the shape of a polygonal mesh. We have found that stippling as a rendering style is well suited for producing computer animations, because the point hierarchy described above can be used as an elastic texture which is attached to the surface of the model (see Figure 14). In the following discussion we describe animated stippling, a technique that allows us to use the point hierarchy for animated models.

For mesh morphing we take two polygonal meshes of the same topology with vertices at different locations in object space. These meshes are used to construct an animation by

transforming one mesh into the other. The animation is typically constructed by performing linear interpolation between the two vertex sets of the key meshes for every frame of the animation. Alternatively, an animation can be created by having as input a polygonal mesh and a time varying vector field which indicates the direction of displacement for the vertices in the mesh. Regardless of the technique chosen to produce the animation, the main requirement for producing animated stippling is to have a mesh with fixed connectivity (topology) while the vertices of the mesh (or a subset of the vertices of the mesh) change in position over time, changing the shape of the model.

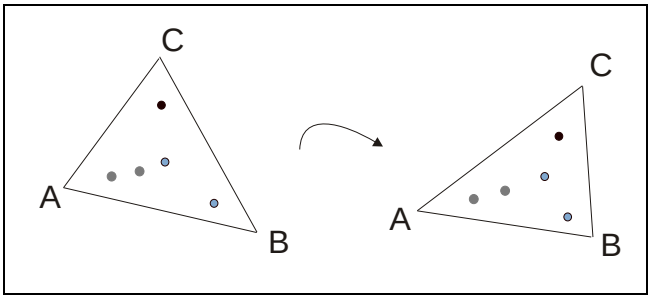

Fig. 12

THE POINTS ON THE SURFACE OF THE TRIANGLE ARE DEFINED USING BARYCENTRIC COORDINATES, THAT IS, THEY ARE DEFINED RELATIVE TO THE VERTICES OF THE TRIANGLE. AS A RESULT, THE POINTS ON THE SURFACE MOVE ALONG WITH THE TRIANGLE AS THE POSITION OF ITS VERTICES CHANGES.

To attach the stipples to the surface of the mesh during the animation, we redefine the positions of the points in the hierarchy using barycentric coordinates (see Figure 12). In the barycentric coordinate system the position of the points contained on the plane of a triangle are defined with respect to the vertices of the triangle, which is very convenient for stippling, because we can move the vertices of the triangular mesh on 3D space, and then recompute the positions of the stipples as a function of these vertices.

Since normally there are several stipples per face, each point in the stipple set contains an index to the face of the model where the point lies (the host face for that stipple) and the barycentric coordinates of the point within that face. At each animation step, the point coordinates are recomputed using the barycentric coordinates and the vertex positions of the host face. The effect achieved is that stipples behave like an elastic texture printed on the models surface. In addition, each particle's normal has to be updated after each deformation step, using either gouraud shading or flat shading.

During morphing, the polygons are normally not proportionally distorted, and this has an effect on the overall stipple distribution. If the stipples radii are left untouched during the animation, the overall point density becomes a function of the mesh distortion that occurs during morphing, i.e. 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 per se. However, if the stippling is to be kept as a function of shading and scale,

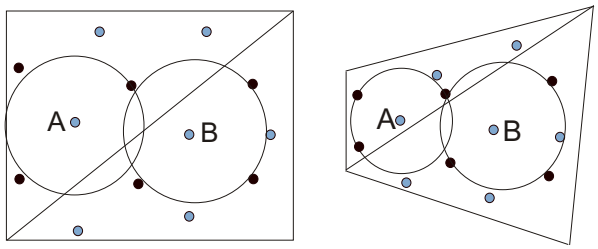

Fig. 13

ON THE LEFT, TWO SAMPLE POINTS A AND B ARE GIVEN A RADIUS VALUE WHICH IS THE AVERAGE DISTANCE TO ITS RELEVANT NEIGHBOURS (THE DARK POINTS). ON THE RIGHT, WE OBSERVE THE TWO POINTS WITH THEIR NEW AVERAGE RADIUS AFTER THE DISTORTION HAS TAKEN PLACE. SINCE THE DISTANCE TO THEIR NEIGHBOURS HAS CHANGED AFTER THE TRANSFORMATION, THE VALUES FOR THE RADIUS OF THE STIPPLED POINTS ALSO CHANGE, BUT IN DIFFERENT PROPORTION FOR EACH POINT.

as originally postulated, a more elaborate solution is required which compensates the effect of the distortion. In regions that shrink, less stipples should be drawn and in regions that expand, more stipples should come up, assuming illumination conditions and viewpoint are kept constant while the distortion takes place.

To allow stipple particles to adapt to the new polygon shape while maintaining an appropriate shading and scale, we compute the radius of each stipple as the average of a small set of neighbouring points (2 to 4 points) for each frame. The set of neighbouring points is defined when the point set is generated. (see Section IV). By defining the neighbouring 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 neighbouring points, as illustrated in figure 13. While this introduces an additional computational effort during rendering, this step ensures that the stipple hierarchy originally computed is kept consistent as deformations take place. Ideally, more stipples should be generated when the surfaces are expanded, and some could be eliminated when the surfaces shrink. This, however puts additional overhead to the rendering process, which is not necessary if all the stipples that are potentially needed are generated in a preprocessing stage. Videos of stippling for animated models can be observed at http://isgwww.cs.uni-magdeburg.de/~oscar/.

VI. CONCLUSIONS

We have presented a system to produce frame-coherent stippled drawings of 3D polygonal models adapted for real-time interaction and for producing animations. Our system ensures frame-coherence at the stipple level, i.e. every stipple shown in a frame of a video sequence appears in a corresponding location on the next frame. This is achieved by creating a point hierarchy where the points are fixed on the surface of the input model. The point hierarchy is created by applying mesh subdivision and mesh simplification to the input model to obtain seamless levels-of-detail. Higher levels-

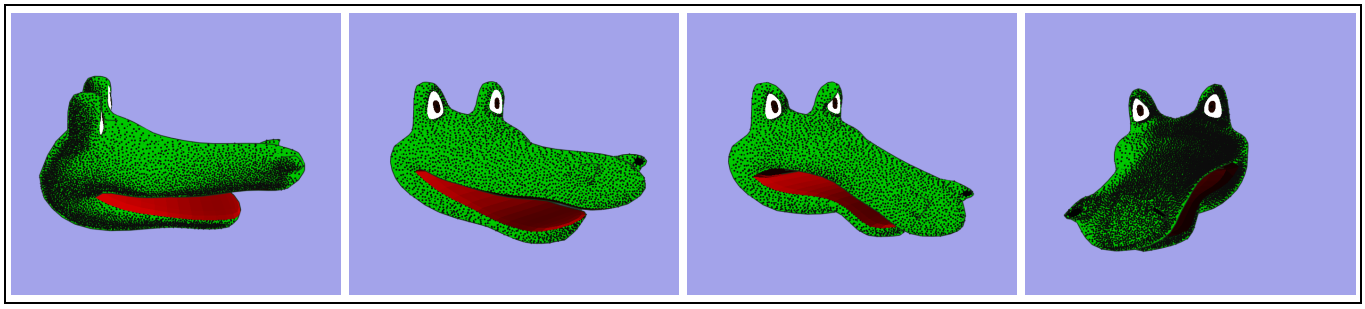

Fig. 14
FRAMES FROM OUR ANIMATION "UPSETTING THE CROCODILE".

of-detail, produced by mesh subdivision, are used to fill-in the darker regions of the model. Lower levels of detail, produced by mesh simplification, permit removal of vertices at levels-of-detail lower than that of the input model and are used to stipple lighter shaded regions in the image. In the system presented, new stipples always fill-in the spaces between existing stipples, and smoothly blend-in or fade out of the image according to illumination and viewing parameters. Randomization and projection operators are used to improve the distribution of the stipples and ensure these are always mapped to the surface of the input model, respectively. In addition, the point hierarchy is used to produce frame-coherent stippled animations of polygonal models by setting the points in barycentric coordinates (i.e., as a function of the polygonal mesh). For static models, the point hierarchy is used in a vertex program that allows the real-time rendering of models in the stippling style. The point hierarchy can be used to render models in new styles related to pointillist rendering and view-dependent particle systems.

Our system makes use of a number of techniques which can be individually optimized to improve the overall aspect of our renditions and the system's efficiency. For instance, the point distribution can be improved by using an algorithm that redistributes the vertices on the surface of the input model as part of preprocessing, and different approaches for mesh simplification can be tried to improve the selection of vertices while constructing the point hierarchy. Another optimization worthwhile investigating is 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 should reduce the number of stipples tested by roughly one half. An even more aggressive method would be to employ the occlusion test provided by some graphics hardware (e. g., the GeForce 3). Both options would require to break 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 at all. A screen-based level of detail approach can also be used to determine which points should be rendered according to the projection of the input polygons on the surface of the model, as mentioned in Section V-B. As future work, we want to develop an efficient algorithm to fill in specific areas of the model with stipple particles when the user zooms at the model and the point hierarchy has been exhausted.

ACKNOWLEDGMENTS

The authors would like to thank Lourdes Peña Castillo for reviewing drafts, the members of the Institute for Simulation and Graphics at the University of Magdeburg, specially Nick Halper for providing important feedback towards 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. Guthrey for their stippled renditions. This work was partially supported by a scholarship of the state of Sachsen-Anhalt, Germany.

REFERENCES

[1] D. Cornish, A. Rowan, and D. Luebke. View-dependent particles for interactive non-photorealistic rendering. In *GI 2001*, pages 151–158, June 2001. http://www.graphicsinterface.org/proceedings/2001/158/.

[2] O. Deussen, S. Hiller, C. van Overveld, and T. Strothotte. Floating points: A method for computing stipple drawings. *Computer Graphics Forum*, 19(3):40–51, 2000. http://www.eg.org/EG/CGF/volume19/issue3.

[3] H. Hoppe. Progressive meshes. In *SIGGRAPH 96 Conference Proceedings*, pages 99–108. ACM, 1996. http://doi.acm.org/10.1145/237170.237216.

[4] M. Kaplan, B. Gooch, and E. Cohen. Interactive artistic rendering. In *Proceedings of the first international symposium on Non-photorealistic animation and rendering*, pages 67–74. ACM Press, 2000. http://www.cs.utah.edu/npr/papers.html.

[5] E. Lindholm, M. J. Kligard, and H. Moreton. A user-programmable vertex engine. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 149–158. ACM Press, 2001.

[6] D. Luebke. Developer's survey of polygonal simplification algorithms. In *IEEE Computer Graphics & Applications (May 2001).*, pages 24–35, May 2001. http://www.cs.virginia.edu/~luebke/publications.html.

[7] L. Markosian, B. J. Meier, M. A. Kowalski, L. S. Holden, J. D. Northrup, and J. F. Hughes. Art-based rendering with continuous levels of detail. In *Proceedings of the first international symposium on Non-photorealistic animation and rendering*, pages 59–66. ACM Press, 2000. http://doi.acm.org/10.1145/340916.340924.

[8] O. Meruvia and T. Strothotte. Frame-coherent stippling. In *Eurographics'2002 Short Paper Proceedings.*, 2002. http://isgwww.cs.uni-magdeburg.de/~oscar/.

[9] E. R.S. Hodges. *The Guild Handbook of Scientific Illustration.* Wiley Europe, 1988.

[10] A. Secord. Weighted voronoi stippling. In *Proc. of the 2nd. international symposium on Non-photorealistic animation and rendering*, pages 37–43. ACM Press, 2002.

[11] A. J. Secord, W. Heidrich, and L. Streit. Fast primitive distribution illustration. In *Proc. of the Eurographics Workshop on Rendering*. Eurographics, 2002.