The Use of Points as a Display Primitive

Marc Levoy
Turner Whitted

Computer Science Department
University of North Carolina
Chapel Hill, NC 27514

Abstract

As the visual complexity of computer generated scenes continues to increase, the use of classical modeling primitives as display primitives becomes less appealing. Customization of display algorithms, the conflict between object order and image order rendering and the reduced usefulness of object coherence in the presence of extreme complexity are all contributing factors. This paper proposes to decouple the modeling geometry from the rendering process by introducing the notion of points as a universal meta-primitive. We first demonstrate that a discrete array of points arbitrarily displaced in space using a tabular array of perturbations can be rendered as a continuous three-dimensional surface. This solves the long-standing problem of producing correct silhouette edges for bump mapped textures. We then demonstrate that a wide class of geometrically defined objects, including both flat and curved surfaces, can be converted into points. The conversion can proceed in object order, facilitating the display of procedurally defined objects. The rendering algorithm is simple and requires no coherence in order to be efficient. It will also be shown that the points may be rendered in random order, leading to several interesting and unexpected applications of the technique.

1. Introduction

In classical image synthesis, objects may be modeled using a variety of geometric primitives. Examples are lines, curves or polygons in two dimensions and patches or polyhedra in three dimensions. At the simplest level, rendering consists of converting this geometry into a two-dimensional array of pixels for viewing on a raster display. As the visual complexity of computer generated scenes continues to increase, the use of classical modeling primitives as display primitives becomes less appealing. Three problem areas can be identified:

1. Customized rendering algorithms. The introduction of a new geometric primitive traditionally requires the development of two algorithms, one for modeling objects using the primitive and one for displaying them. The second step is unfortunate because it requires customizing existing display algorithms for each new primitive.

This paper proposes to decouple the modeling geometry from the rendering process by introducing the notion of a *meta-primitive.* This new entity mediates between objects modeled using traditional geometry and the display pipeline.

Rendering then consists of converting objects from their geometric description into the new format, which is then displayed using a standardized algorithm. Figure 1 illustrates the concept. Of course, such an approach is only worthwhile if developing an algorithm to convert traditional geometries into the meta-primitive is easier than developing an algorithm to display them directly.

2. Object order vrs. image order rendering. Procedurally defined objects have become very popular in the past few years. Fractals and solids of revolution are examples of this class. The problem of customization in the rendering algorithm becomes even more acute in these cases. Since the geometry of procedurally defined objects is derived by executing a sequential algorithm, it would be convenient to display them in the order in which they are computed. This could be called *object order* rendering. In order to compute correct visibility and filtering at each pixel, most display algorithms prefer that the image be computed in *image order.* In other words, the image is constructed pixel by pixel. All objects that might contribute to a given pixel are evaluated at the time that pixel is rendered. Ray tracing and scanline algorithms fall into this class. The Z-buffer [4] and more recently the A-buffer [3] do not. Researchers have invested considerable effort in resolving the conflict between object-driven and image-driven approaches using such techniques as bounding boxes [12] and image space decomposition [14].

For this reason, we have chosen to focus on a meta-primitive that can be rendered in object order. This facilitates applying the results to the growing class of procedurally defined objects. Of course, such an approach is valid only if correct visibility and filtering can be computed for all pixels even though rendering is performed in object order.

3. Complexity vrs. coherence. Geometrically defined curves and surfaces are an efficient means for describing man-made environments. Recently, researchers have begun to explore phenomena that are not readily modeled using classical surface elements. Terrain, foliage, clouds and fire are examples in this class. As complexity increases, coherence decreases. Beyond some threshold level, the time saved through the evaluation of coherence does not justify its expense. When the number of polygons in the object exceeds the number of pixels on the screen, coherence becomes almost useless.

This paper is addressed toward the display of scenes containing a high level of image complexity, particularly those in which the number of polygons approaches or exceeds the number of pixels on the screen. For this reason, we have chosen a meta-primitive that can be rendered fast and without coherence.

The meta-primitive that has been chosen is zero-dimensional points. Researchers have already shown that points are a good primitive for modeling intangible objects such as smoke or fire. This paper will show that points are considerably more versatile than that. It will first be demonstrated that a discrete array of points arbitrarily displaced in space using a tabular array of perturbations can be rendered as a continuous three-dimensional surface. One can think of this as geometric modeling without geometry. It will then be demonstrated that a wide class of geometrically defined objects, including both flat and curved surfaces, can be converted into points. The conversion can proceed in object order, facilitating the display of procedurally defined objects. The rendering algorithm is simple and requires no coherence in

order to be efficient. It will also be shown that the points may be rendered in random order, leading to several interesting and unexpected applications of the technique.

2. Prior work

The use of points to model objects is not new. Csuri et al [7] used points quite successfully to model smoke. In a similar vein, Blinn has used points to model the shading of clouds [2] and Reeves to model fire [13] and trees [15]. All of these efforts treated classes of objects that could not be modeled using classical geometries. This is not, however, a necessary restriction. Geometric subdivision algorithms may ultimately yield points as Catmull [4] and Rubin and Whitted [14] have observed. Since raster displays are discrete and of a finite resolution, points can be used to model anything which can be displayed. This is one of the primary reasons that points were selected as the meta-primitive for this paper. It should be pointed out while the points used in this paper are superficially similar to the particles used by Reeves, the extension of the algorithm used in displaying particles to the algorithm used in making points appear as continuous opaque surfaces is not trivial.

From a mathematical perspective, points are discrete, zero-dimensional entities. When a large number of points are arrayed in tabular form, they become textures and begin to take on some of the characteristics of surfaces. Until the mid-1970's, the complexity of computer-generated images was directly proportional to the complexity of the geometric database defining them. One of the most important single advances in the realism of computer imagery came with the development of texture mapping, first published by Catmull [4]. By associating a tabular two-dimensional array of intensity values with each geometric primitive, it allowed the visual complexity to far exceed the shape complexity. Blinn's bump mapping technique generalized texture mapping by using a tabular array to locally modify surface orientation [1]. The shape, however, remains unchanged. This causes the surface to appear three-dimensional at all points except along the silhouette edge, where it still appears flat.

Several researchers, including Dungan [8] and Fishman and Schacter [9], have explored the use of tabular arrays to provide z-height for the modeling of three-dimensional terrain. For the restricted case of a flat plane displaced in one direction only, the visible surface problem can be solved rather trivially as described by Fishman and also by Max [11]. More recently, Cook [6] has demonstrated the use of displacement maps in his Bee Box but has not described how they are rendered. This paper takes texture mapping one step further and demonstrates how the entire geometry can be reduced to the transformation and display of points arrayed as textures.

Object order rendering has received little attention in the literature. A particularly simple method of rendering parametric surfaces is to select sample points in parameter space and individually transform them to screen space [10]. The two-pass algorithm for texture mapping [5] is an efficient implementation of the same idea and one that carefully treats the problem of aliasing. If the texture space to image space mapping is many-to-one, however, the two-pass algorithm becomes messy and loses some of its appeal. The important aspect of both of these methods is that the geometry is not rendered directly, but rather supports the transformation of points from a canonical texture space to screen space.

The only hidden-surface algorithm that allows object order rendering is the Z-buffer [4]. Carpenter's A-buffer [3] is an extension of this basic technique that supports correct filtering of polygonally defined objects. The algorithm in this paper is similar in concept to an A-buffer, but stripped down and modified for use with zero-dimensional points.

3. Overview of the method

The approach proposed in this paper requires two separate but equally important algorithms. The first converts geometry into points and the second renders those points. Figure 1 shows their relationship. The rendering pipeline is the more difficult of the two and demands that the conversion algorithm produce certain information about a geometry in order for it to be rendered. For this reason, the rendering algorithm will be described first and the conversion algorithm second. A simple example will be used at first to illustrate the operation of the rendering pipeline. Its extension to more complex shapes will then be developed and the conversion algorithm presented.

The example used is a flat rectangular polygon represented as a 170-by-170 array of points. The shading of each point is multiplied first by the pin stripe texture shown in figure 6a to enhance its readability, then by the reflectance map shown in figure 6b. Finally, the z-coordinate of each point is perturbed upwards using the discrete height field shown in figure 6c. The goal of the rendering pipeline is to take this array of points and render them on a raster display in such a manner that they appear as a continuous, three-dimensional surface. This implies that the following three criteria must be satisfied:

(i) The texture in the interior of the array must be properly filtered.

(ii) The edge of the array must be properly anti-aliased.

(iii) The array must completely obscure its background.

Two issues make this a difficult problem. In the first place, no constraint is placed on the nature of the spatial perturbation. The example shows only z-displacement, but x or y displacement are equally possible. Secondly, the image must look correct regardless of the order in which the points are rendered. The figures in this paper were rendered in random order. The resulting surface thus appears to ‘sparkle’ in much like a ‘Scotty, beam me up’ fade-in from the Star Trek television series.

4. The rendering pipeline

4.1. Definition of a point

Let a *source point* be defined as a 7-tuple:

(x, y, z, r, g, b, α)

Each of the values in the 7-tuple are called *attributes.* The *x, y* and *z* values are called *spatial attributes* and the rest are called *non-spatial attributes.* Any attribute is fair game for perturbation, as we shall see later.

Let an *initial grid* be defined as an:

> evenly spaced, rectangular two-dimensional lattice of source points bearing the parametric coordinates *u* and *v*.

At this stage, we assume that $x = u$ and $y = v$. The grid could therefore easily be stored as a texture (i.e. in a two-dimensional array without *x* or *y* values.)

4.2. Selection of a point

Each iteration of the rendering process consists of selecting an arbitrary point from the initial grid and sending it through the rendering pipeline. The rendering order may be sequential in some parametric space, may be governed by the execution of a procedure or may be random.

4.3. Perturbation of points

Let a *perturbation* be defined as any operation which changes any of the attributes associated with a point. Any number of perturbations may applied to the randomly selected point. The non-spatial attributes may be perturbed in any manner desired so long as the resulting values fall within the range of the computer representation being used. The spatial attributes can only be perturbed within reasonable bounds or the surface defined by the grid becomes discontinuous. These bounds will be discussed in detail later.

4.4. Transformation and clipping

Let M be defined as a 4-by-4 transformation matrix that includes perspective projection. The transformation step consists of performing the usual matrix multiplication of $[x,y,z,1]$ by M followed by the perspective divide. The division of z by w is suppressed so that z-clipping can be performed. The clipping of a point consists simply of comparing the transformed x, y and z coordinates of the point against a three-dimensional frustum of vision and rejecting the point if it falls outside the boundaries of the frustum.

4.5. Computing the density of points in image space

In order to properly filter a texture, the contribution of each source point to each display pixel must be roughly proportional to its distance from the center of the pixel. This is accomplished by assuming that each pixel is overlain by a filter function that is highest at the center of the pixel and drops off toward zero at increasing distances. A radially symmetric Gaussian filter is used in this implementation. The contribution of each source point to each pixel is determined by computing the distance from the point to the center of the pixel, then weighting that distance according to the Gaussian function.

Since the density of source points in close proximity to the center of the pixel varies with the viewing transformation, the sum of the contributions may vary. In conventional texture mapping, this problem is readily solved by summing the weights computed for each pixel and then dividing the accumulated color by this sum. If the pixel being rendered lies along the edge of the texture as shown in figure 2a, fewer source points contribute to it. This will cause the sum of the weights to vary. It is a simple matter to determine if the edge of the texture passes through a given pixel and to compute its coverage. This coverage can then be used to control blending between this texture and other surfaces that lie behind it.

When a spatially perturbed texture is rendered, silhouette edges may occur at any place in the interior of the texture as shown in figure 2b. Since the algorithm permits the rendering of surfaces for which no underlying mathematical description of their curvature exists, it is

impossible to predict these foldover points. Consequently, variations in the sum of the contributions in a display pixel may result either from variations in the density of the source points or from partial coverage along silhouette edges. In order to discern between these two cases, it is necessary to pre-normalize the contributions. This forces them to sum to unity. If they do not sum to unity despite this pre-normalization, it is because the texture only partially covers the pixel. The sum of the contributions is then precisely equal to the coverage and may be used to control blending between surfaces. The problem therefore becomes one of predicting the density of source points in a neighborhood surrounding the current transformed source point before rendering begins. This density can then be used to compute a normalizing divisor for the weights.

Specifically, let:

$$\mathbf{p}_0 = (x_{0,}\ y_{0,}\ z_0)$$

be a source point after perturbation but before transformation. Let:

$$\mathbf{p}_u = (x_{u,}\ y_{u,}\ z_u)$$

$$\mathbf{p}_v = (x_{v,}\ y_{v,}\ z_v)$$

be two additional points spaced one unit away from $\mathbf{p}_0$ in *u* and *v* respectively. These three points can be used to form two unit vectors:

$$\mathbf{u} = (x_u - x_{0,}\ y_u - y_{0,}\ z_u - z_0)$$

$$\mathbf{v} = (x_v - x_{0,}\ y_v - y_{0,}\ z_v - z_0)$$

These unit vectors give the orientation of the perturbed but untransformed surface in a small neighborhood around the source point.

Now let:

$$\mathbf{p}_0' = (x_0', y_0', z_0')$$

$$\mathbf{p}_u' = (x_u', y_u', z_u')$$

$$\mathbf{p}_v' = (x_v', y_v', z_v')$$

be the coordinates of these same three points after the application of the viewing transformation *M* and let:

$$\mathbf{u}' = (x_u' - x_0', y_u' - y_0', z_u' - z_0')$$

$$\mathbf{v}' = (x_v' - x_0', y_v' - y_0', z_v' - z_0')$$

be the corresponding transformed unit vectors. Figure 3a and 3b show these two unit vectors before and after the application of the viewing transformation.

The assumption is made that the surface is continuous and differentiable. The transformed unit vectors can then be interpreted as a tangent plane passing through the point $\mathbf{p}_0'$ and approximating the surface in a small neighborhood around the point as shown in figure 3c. Since the unit vectors have now been transformed into screen space, the *z* coordinate can be dropped. The resulting two-dimensional unit vectors now represent the relationship of the transformed source grid to the display pixel grid. It is obvious (and can be proven geometrically) that the the density of source points in screen space is inversely proportional to the area of parallelogram formed by the unit vectors as shown in figure 3d. This area is

simple to compute. It is equal to the absolute value of the determinant of the Jacobian matrix:

$$A = \left| \det \left[J_{\mathbf{F}}(\mathbf{p}_0') \right] \right| = \left| \det \begin{bmatrix} x_u' - x_0' & x_v' - x_0' \\ y_u' - y_0' & y_v' - y_0' \end{bmatrix} \right|$$

Knowing the density of source points, it is possible to compute the proper normalizing divisor for any source point subject to any viewing transformation. This in turn insures that the sum of the contributions for any display pixel will sum to unity in the interior of the texture and will sum to the coverage along the edges.

A tangent plane is of course only a local approximation of a general surface. If either the perturbations or the effect of perspective distortion are severe, the true density of points will differ from the computed density. This is manifested in the final image by fluctuations in computed coverage. If the contributions sum to less than unity, the surface will not completely obscure its background. It has essentially 'pulled apart'. This error can be characterized numerically by sudden changes in the determinant of the matrix computed above. For two adjacent source points $\mathbf{p}_0$ and $\mathbf{q}_0$, the error is:

$$\varepsilon = \left| \frac{\det \left[J_{\mathbf{F}}(\mathbf{p}_0') \right]}{\det \left[J_{\mathbf{F}}(\mathbf{q}_0') \right]} - 1 \right|$$

If ε grows too large, the opacity of the surface degrades and artifacts result. On the other hand, high values of ε are an indication that the spatial resolution of the initial grid is insufficient to handle the high frequencies present in the perturbation function. The solution is either to low-pass filter the perturbation function or to increase the spatial resolution of the initial grid.

4.6. Selecting the proper filter radius

It would help to re-examine the procedure up to this point. The algorithm is taking a point and a tangent plane and transforming them into image space. The result is a point in image space and a scalar measurement of the area which the point would cover if it were a surface element instead of a point. The position of these points in image space is not anchored in any way to the location of the display sample points in the image plane. We have tried to devise a method which simultaneously filters the components of the resulting image, reconstructs a continuous image function and computes display sample values. We lack a thorough analysis of this method of constructing an image from fuzzy points and are instead guided by experience with texture mapping. The effective radius of a point must be a function both of the source density and the display sample density since at least two filters are being implemented in the same function. When the viewing transformation calls for minification (many source points to one display pixel), the filter must be large enough in order to avoid aliasing of the source function. In the case of magnification (many display pixels to one source point), the filter must be large enough to avoid aliasing of the reconstruction. As a practical matter, the effective radius decreases as source density increases, but reaches a minimum radius that depends on display sample resolution.

Since an ideal filter is infinite in extent, the number of source points that contribute to each display pixel remains the same regardless of the alignment between the source and display grids. In practice, filter functions are assumed to be zero beyond a small neighborhood around the pixel. This sudden cutoff causes the number of source points that contribute to a given display pixel and hence the sum of the contributions to vary slightly as the grids are shifted with respect to one another. If the texture is rendered in image order, the normalization process described above adjusts for these variations quite naturally and they cause no trouble. When a texture is rendered in object order, any variation in the sum of the contributions is interpreted as a partial coverage as described above. This error manifests itself as errors in computed coverage that change as the alignment between the grids changes. The problem can be alleviated by extending the Gaussian filter out slightly further than usual. The small contributions by these extra source points tend to cancel out the effect of the discontinuity in the filter function, making the filter more invariant to shifts in the grid alignment.

4.7. Hidden-surface elimination

As many researchers have pointed out, a standard Z-buffer hidden surface algorithm cannot produce images with correct anti-aliasing. The difficulty arises as follows. There are two possible ways in which the contribution of source points to display pixels might take place:

1. *Blending.* If the contents of the Z-buffer for a given display pixel contains a piece of the same surface as the incoming contribution, the two should simply be added together. Specifically:

$$color_{new} = color_{old} + (color_{incoming} \times weight_{incoming})$$

Information about the overall transparency versus opacity of the surface, which is contained in the α attribute, is treated exactly like *color* in a blending calculation. (When a surface folds over and obscures itself, it forms two separate surfaces from the standpoint of hidden-surface removal. This special case is treated later on.)

2. *Visibility.* If the contents of the Z-buffer for a given display pixel contains a piece of a different surface from the incoming contribution, then a depth comparison must be performed. Let us suppose that the incoming contribution is in front of (obscures) the surface stored in the z-buffer. In this case, the two should be merged together as follows:

$$color_{new} = color_{old} \times (1 - \alpha_{incoming}) + color_{incoming} \times \alpha_{incoming}$$

This visibility calculation only works if the incoming color and α are completely known (i.e. if all blending that will be required has already been performed). One cannot intersperse blending calculations with visibility calculations. The logical solution, as demonstrated by Carpenter in his A-buffer algorithm [3], is to gather together all contributions to a given surface, performing blending calculations whenever possible to consolidate surface fragments, and delaying the visibility calculation until all contributions have been made.

In the case of a point rendering pipeline, the surface fragments have no geometry (as they do in Carpenter's algorithm), only a weight. Contributions from each surface are pigeonholed into unique bins and blending is performed to keep the information in each bin to a minimum. When all surfaces have been rendered, visibility calculations are performed between bins. This final step implements the hidden surface elimination. Figure 4 illustrates the two types of computations and the order in which they are performed. It should be noted that if all surfaces being rendered are opaque, two bins suffice to properly render any environment. Slight errors will still occur in pixels where multiple edges coincide, but these errors are seldom visible.

As noted earlier, the sum of the weights in a particular bin is not always unity after a surface has been completely rendered. In particular, display pixels along the edge of a texture will be left with contributions that sum to only a fraction of unity. In this case, the following additional calculation is performed prior to the visibility calculations:

$$\alpha_{bin} = \alpha_{bin} \times \sum weight_{bin}$$

This adjusts the overall transparency versus opacity of the texture according to the reduced extent to which it covers a display pixel coincident with its edge. This new α value is now ready for use in the visibility calculations.

The only remaining difficulty is determining whether or not a contribution is from the same surface as that contained in a particular bin, in which case a blend should be performed, or from a different surface, in which case a depth comparison should be performed. Surface identification tags work fine on flat polygons but not on surfaces that may obscure themselves. The solution taken in this algorithm is to blend only those points that are *contiguous* (in the sense that the shortest path between them passes along the surface rather than through space). Points that are separated from each other by space are resolved using a depth comparison. Each bin has associated with it a range of depths. If the transformed *z* coordinate of an incoming contribution falls within a certain tolerance of the depth of a given bin, the contribution is blended in and the range of depths in the bin is extended to include both its old contents and the new contribution. Otherwise, the incoming contribution is tentatively placed into a separate bin. As more contributions arrive, some blending will inevitably occur and the depth ranges of some bins will grow. Whenever the depth ranges of two bins are found to be overlapping, again within a certain tolerance, their contents are combined and the result is placed in a single bin.

As an example, suppose that the surface is as shown in Figure 5. Suppose further that point #1 is the first to arrive. It will be stored in bin #1. If point #2 is the next to arrive, its depth does not fall within the allowed tolerance of the depth in bin #1. It is therefore placed into bin #2. Sooner or later, point #3 will arrive. Its depth matches that of bin #1 within the tolerance and so it is blended in. The resulting depth range of bin #1 now overlaps with the depth range of bin #2 and so all the contributions are combined and the result placed into bin #1. In this manner, the stack of contributions grows and shrinks dynamically as the rendering progresses. By the end, there should be exactly as many bins as there are non-contiguous surfaces. Visibility calculations are then performed between the remaining bins.

The required depth tolerance z_{tol} depends on the separation between the transformed *z* coordinates of adjacent source points. A rough approximation that works reasonably well is:

$$z_{tol} = \left| \frac{1}{(z'_u - z'_0) + (z'_v - z'_0)} \right| \times \varepsilon$$

As it turns out, yet another test must be performed before two contributions may be blended together. As a curved surface folds over and obscures itself, it forms two separate surfaces from the standpoint of hidden-surface removal. Despite the fact that points just barely on the front-facing side of the fold are contiguous with points just barely on the back-facing side, they should not be blended together. This situation can be detected by comparing the transformed surface normals at two such points. If both face the same way, they can be blended. If one is front-facing and the other is back-facing, or vice versa, a depth comparison is performed. The surface normal for a small neighborhood around each point is computed by taking the cross product of $\mathbf{u}'$ and $\mathbf{v}'$:

$$\mathbf{w}' = (x'_w, y'_w, z'_w) = \mathbf{u}' \times \mathbf{v}'$$

Since these unit vectors have already been transformed into display space, the sign of z'_w tells whether a surface faces front or back.

5. Conversion of geometries into points

Having followed the rendering pipeline through from beginning to end using a simple geometry, we are now in a position to state minimum conditions that any geometry must meet in order to be handled by this algorithm:

(i) It must be possible to break the surface into points.

(ii) The surface must be continuous and differentiable in a small neighborhood around each point.

(iii) In order to find the determinant of the Jacobian, it must be possible to find two non-collinear vectors that both lie on a tangent plane that locally approximates the surface at the point. These two vectors can also be used to find the surface normal vector which is required by the foldover detector.

These are the only conditions a surface must meet. In particular, nothing is required about the spacing or distribution of the points. They may be spaced evenly in texture space, evenly in image space, or neither. They may be parametrically derived, randomly derived, etc. The sphere described in the next section was defined by loading texture and displacement maps into rectangular arrays whose u and v coordinates were then re-interpreted as θ and ϕ for mapping into polar coordinates. Points were sparse around the equator and dense at the poles, but the determinant of the Jacobian matrix gave the necessary filter sizing and normalization coefficients.

What kinds of geometry does this allow and what kinds does it prohibit? It obviously allows flat or perturbed polygons. It also allows spheres. The mathematics for handling spheres can be trivially extended to handle any conic section. In fact, the algorithm can handle any parametrically defined surface. Although it may be inefficient, it is always possible to produce a set of parametrically spaced points to feed the algorithm. It is less clear whether the algorithm can handle fractals. One of the major features of fractals are their lack of local derivatives. They are rough at all scales. Extension to three-dimensional density maps are also possible, but the hidden-surface processing performed by the algorithm would have to be modified.

6. Implementation and experience

The algorithm presented in this paper has been implemented in the C language on a VAX-11/780. The photographs accompanying this paper were produced using this implementation. Figure 7 shows the result of passing two copies of the example array from figure 6 through the rendering pipeline. The texture on the surface appears correctly filtered, the edges exhibit correct anti-aliasing and each fold of the surface completely obscures whatever is behind it. The stretching and distortion of the array caused by the height field perturbation have no effect on the apparent continuity and opacity of the surface. Figure 8 shows the result of adding a ripple perturbation to the surface of a sphere. Note that the resulting surface is fully three-dimensional and that the ripples may be seen along the silhouette edge.

Random order rendering has been mentioned several times in this paper. It is finally time to suggest a use for it. The authors envision (but have not implemented) an interactive image synthesis system that operates in the following manner. The user sits in front of a raster display. Vector outlines of polygons or surfaces move in real-time as the user selects a viewpoint or manipulates a geometry using a joy stick. When the user releases the stick, the surfaces in the environment are rendered in random order. This gives the user an immediate although noisy impression of the image. As the user waits, the signal-to-noise ratio increases and the image 'sparkles' in. As an example, figures 9a and 9b show the image from figure 7 after 50% and 75% of the points have been rendered. The user may of course decide based on the hazy image viewed thus far that more manipulation is required and resumes manipulation of the joy stick. Image synthesis stops immediately and the real-time vector outlines appear once again. The advantage of this approach over traditional rendering algorithms is clear. Rather than appearing top-down or surface-by-surface, all portions of the image appear simultaneously. User comprehension of the partially formed image is enhanced.

A simple yet effective way to render a texture in random order is as follows. A list of u and v parametric coordinates is compiled. This list is then randomly permuted. Finally, coordinate pairs are selected by moving sequentially through the list. This insures that the rendering is random, yet that no point is rendered twice. The list can be computed in advance and re-used each time a surface is rendered. If the list is treated as cyclic and the starting index is made a random function of a surface identification tag, then the same list can be used for all surfaces in the environment.

No attempt has been made to optimize the code in this implementation. In fact, the Gaussian weighting function is currently being computed for every contribution. The use of lookup tables would increase its speed several-fold. Timing statistics are therefore not presented. The computational complexity of this algorithm is identical to classical texture mapping. One weighting computation or table lookup is required for each contribution made by a source point to a display pixel. The non-zero portion of the Gaussian is slightly larger in this algorithm than in classical texture mapping for reasons already explained. This increases the number of contributions slightly. Based on informal timing tests and general experience with lookup tables, the authors estimate that this method could be made to operate at about 50% to 75% the speed of conventional texture mapping algorithms.

There are two issues that the authors have not yet completely resolved. The first is a more rigorous characterization of the sensitivity of the algorithm to extreme perturbations. In other words, it would be advantageous to know in advance whether a set of points is too severely perturbed to be rendered successfully as a continuous surface. Tests have indicated some sensitivity to high frequencies.

Secondly, numerical errors that arise from the finite extent of the Gaussian filter, while they are not objectionable in simple static scenes, might pose problems during animation or in scenes involving recursive visibility calculations. It is suspected that a more sophisticated filter shape might yield weighting calculations that are more tolerant of the spatial cutoff.

7. Conclusions

This paper has addressed the problems that increasing image complexity has brought to geometric modeling. The notion of decoupling the modeling geometry from the rendering process has been proposed by introducing the notion of points as a meta-primitive. Several shapes have been converted from geometry into points and the continuity and opacity of the original geometry is preserved. The extension of the technique to more complex geometries has been discussed.

The major advantages of this approach are:

(i) A standardized rendering algorithm can be used to display any geometry. Customization for each new modeling primitive is not necessary.

(ii) Geometries can be rendered in object order. This is particularly advantageous in the case of procedurally defined objects.

(iii) The algorithm is capable of rendering as surfaces arrays of points that have no underlying geometry. This provides a simple solution to the bump mapping silhouette problem.

(iv) The point meta-primitive is simple and requires no coherence in order to be rendered efficiently.

As a final note, the capability to render the points in random order suggests that the rendering of each point is independent from the rendering of any other point. This suggests a parallel or hardware implementation.

8. References

[1] BLINN, JAMES T., ‘‘Simulation of Wrinkled Surfaces,’’ *Computer Graphics,* Vol. 12, No. 3, August, 1978, pp. 286-292.

[2] BLINN, JAMES F., ‘‘Light Reflection Functions for Simulation of Clouds and Dusty Surfaces,’’ *Computer Graphics,* Vol. 16, No. 3, July, 1982, pp. 21-29.

[3] CARPENTER, LOREN, ‘‘The A-buffer, an Antialiased Hidden Surface Method,’’ *Computer Graphics,* Vol. 18, No. 3, July 1984, pp. 103-108.

[4] CATMULL, EDWIN E., *A Subdivision Algorithm for Computer Display of Curved Surfaces,* Dissertation, University of Utah, Salt Lake City, December 1974.

[5] CATMULL, EDWIN E. AND SMITH, ALVY RAY, ‘‘3-d Transformation of Images in Scanline Order,’’ *Computer Graphics,* Vol. 14, No. 3, July, 1980, pp. 279-285.

[6] COOK, ROBERT L., ‘‘Shade Trees,’’ *Computer Graphics,* Vol. 18, No. 3, July, 1984, pp. 223-231.

[7] CSURI, C., HACKATHORN, R., PARENT, R., CARLSON, W. AND HOWARD, M., ‘‘Towards an Interactive High Visual Complexity Animation System,’’ *Computer Graphics,* Vol. 13, No. 2, August, 1979, pp. 289-298.

[8] DUNGAN, WILLIAM, JR., ‘‘A Terrain and Cloud Computer Image Generation Model,’’ *Computer Graphics,* Vol. 13, No. 2, August, 1979, pp. 143-150.

[9] FISHMAN, B. AND SCHACHTER, B., ‘‘Computer Display of Height Fields,’’ *Computer and Graphics,* Vol. 5, 1980, pp. 53-60.

[10] FORREST, A. R., ‘‘On the Rendering of Surfaces,’’ *Computer Graphics,* Vol. 13, No. 2, August, 1979, pp. 253-259.

[11] MAX, NELSON L., ‘‘Vectorized Procedural Models for Natural Terrain: Waves and Islands in the Sunset,’’ *Computer Graphics,* Vol. 15, No. 3, August, 1981, pp. 317-324.

[12] KAJIYA, JAMES T., ‘‘New Techniques for Ray Tracing Procedurally Defined Objects,’’ *Computer Graphics,* Vol. 17, No. 3, July 1983, pp. 91-102.

[13] REEVES, WILLIAM T., ‘‘Particle Systems - A Technique for Modeling a Class of Fuzzy Objects,’’ *Computer Graphics,* Vol. 17, No. 3, July, 1983, pp. 359-376.

[14] RUBIN, STEVEN M. AND WHITTED, TURNER, ‘‘A 3-Dimensional Representation for Fast Rendering of Complex Scenes,’’ *Computer Graphics,* Vol. 14, No. 3, July 1980, pp. 110-116.

[15] SMITH, ALVY RAY, ‘‘Plants, Fractals and Formal Languages,’’ *Computer Graphics,* Vol. 18, No. 3, July, 1984, pp. 1-10.

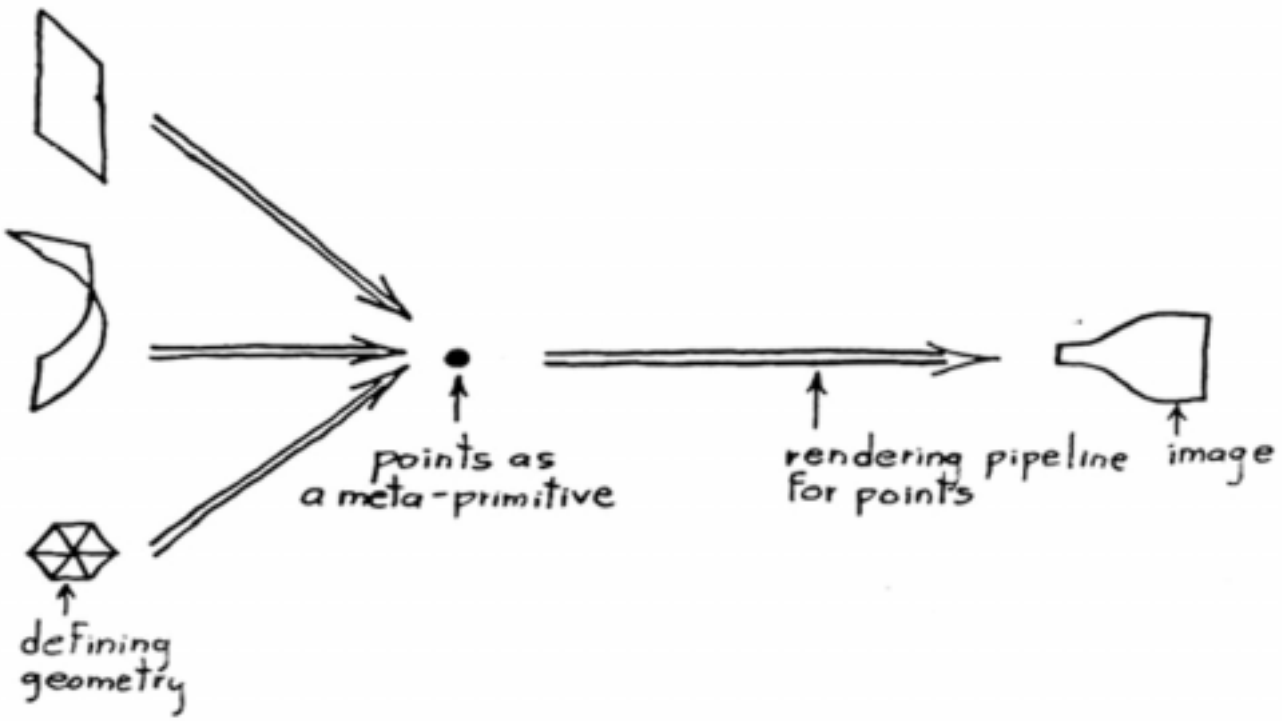

Figure 1: Overview of algorithm

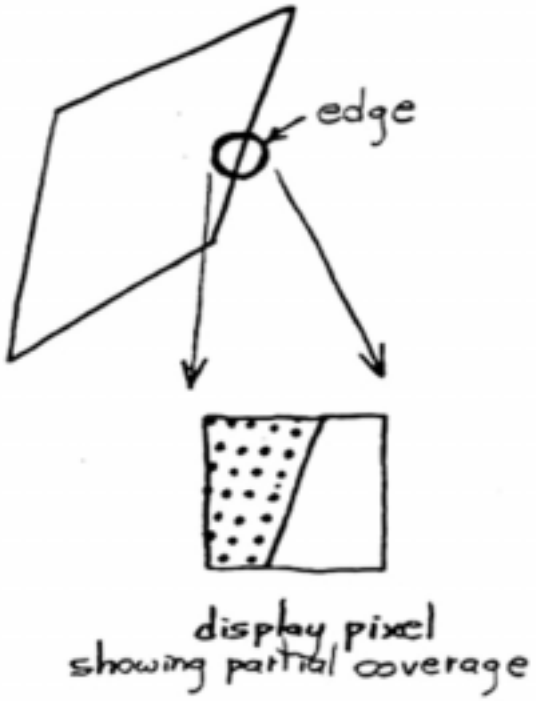

Figure 2a: Edge of texture

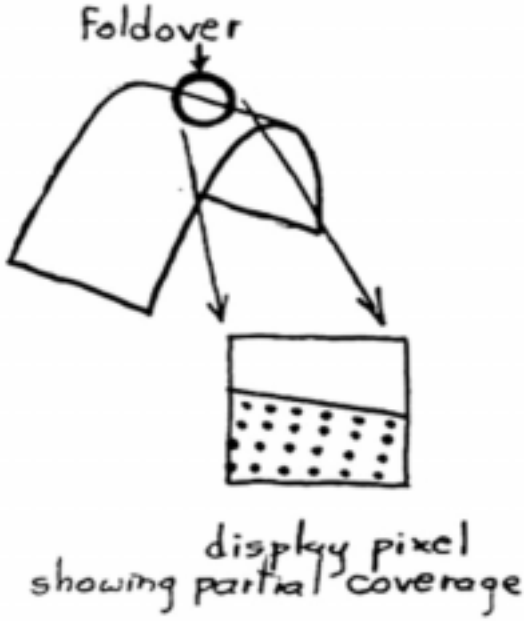

Figure 2b: Foldover

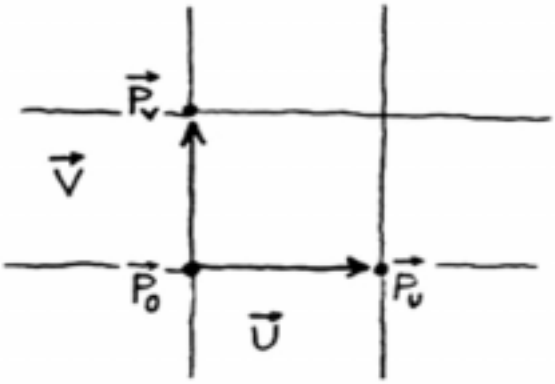

Figure 3a: Unit vectors in u-v space, perturbed but not transformed

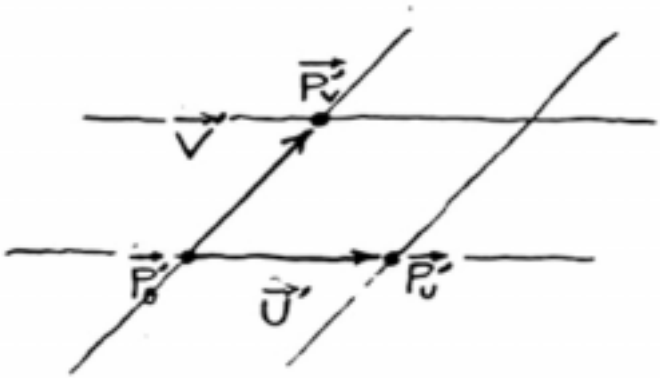

Figure 3b: Unit vectors in perturbed and transformed u-v space

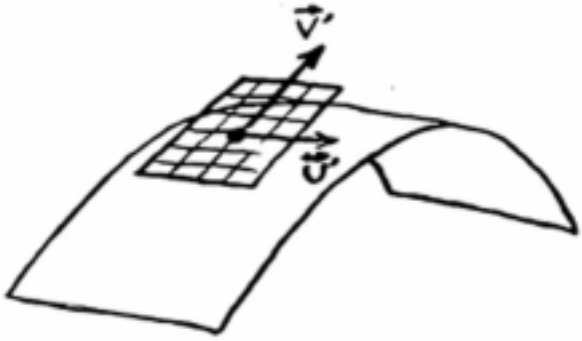

Figure 3c: Tangent plane to surface in small neighborhood

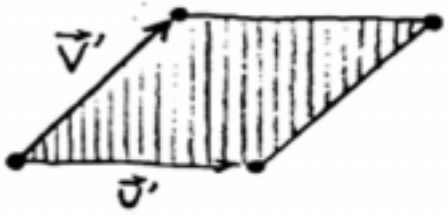

Figure 3d: Area of parallelogram gives density of source points

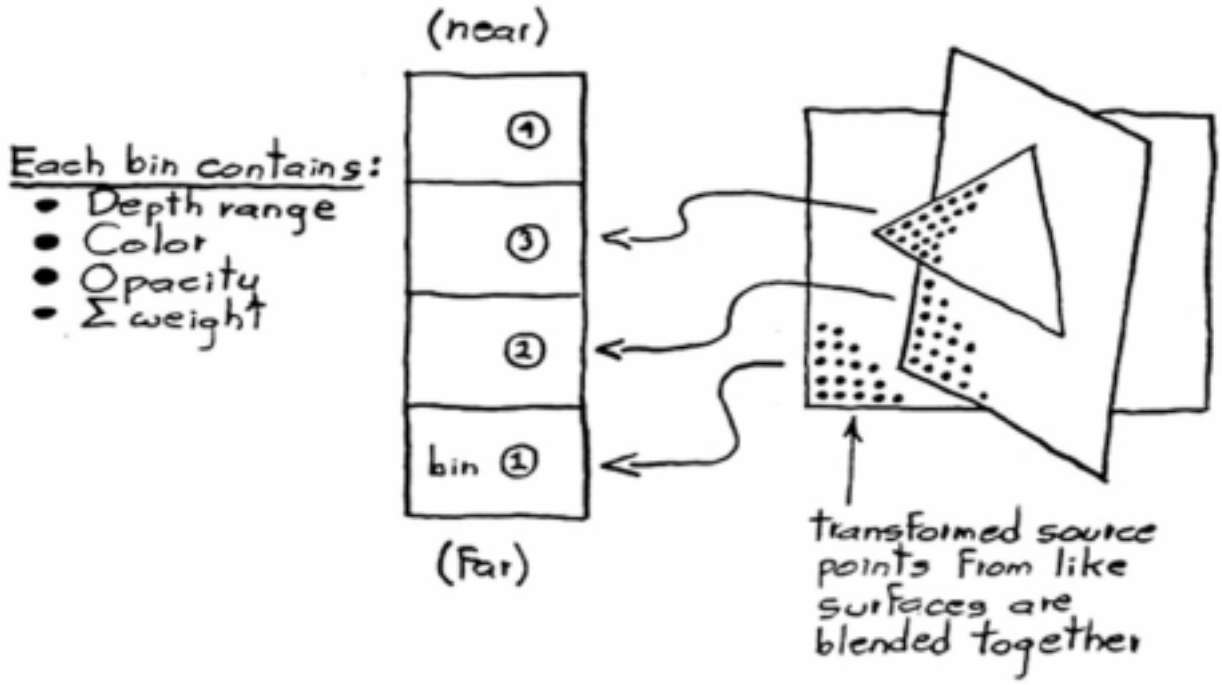

Figure 4a: Blending calculations

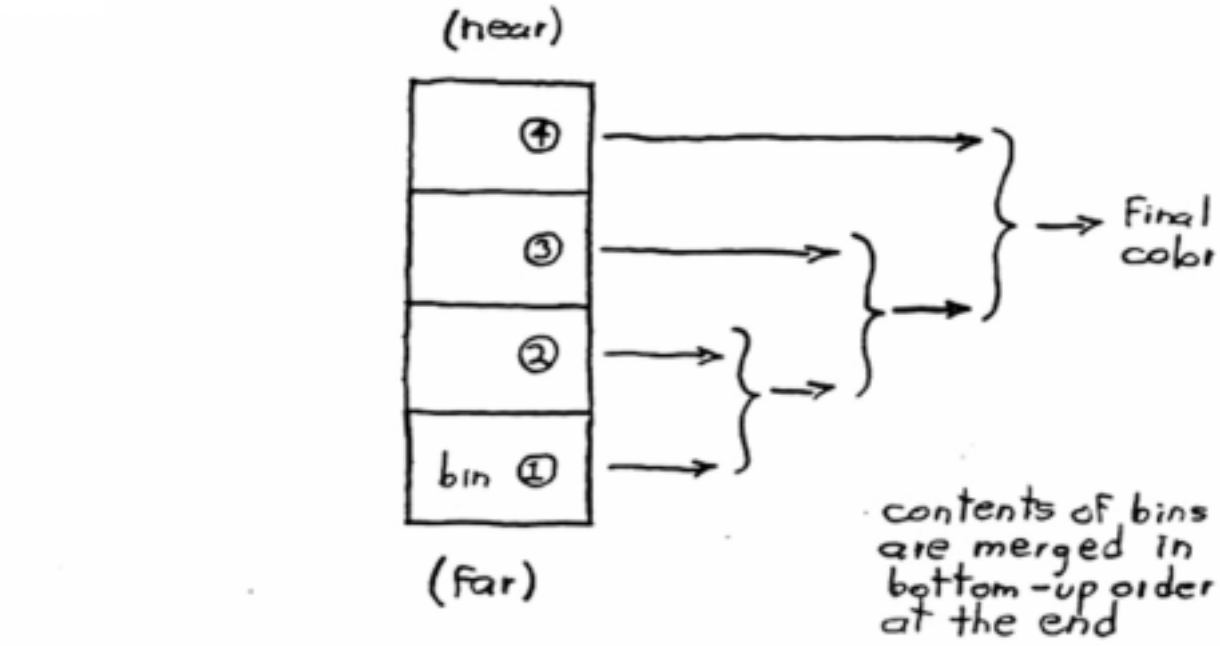

Figure 4b: Visibility calculations

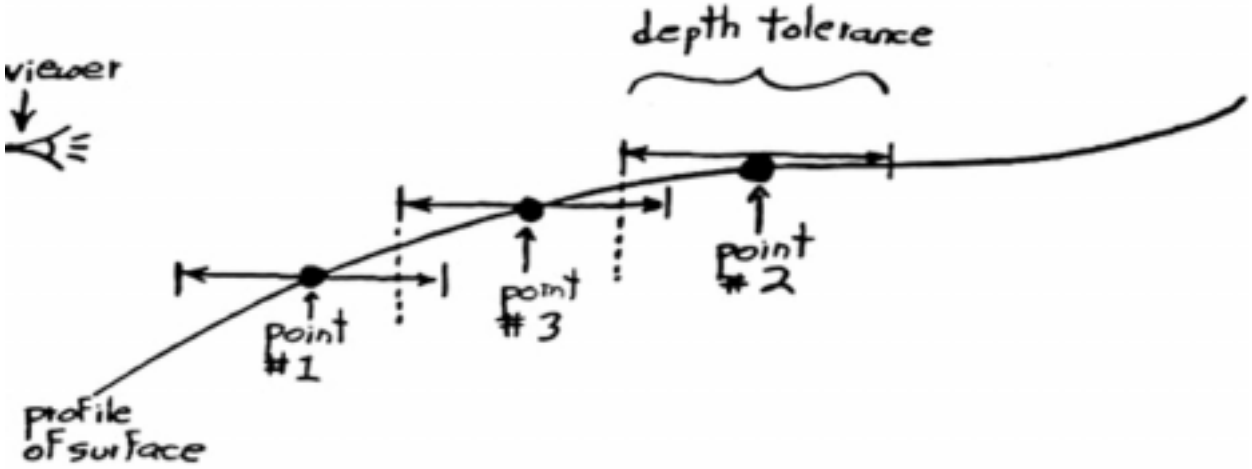

Figure 5: Depth comparisons with tolerance

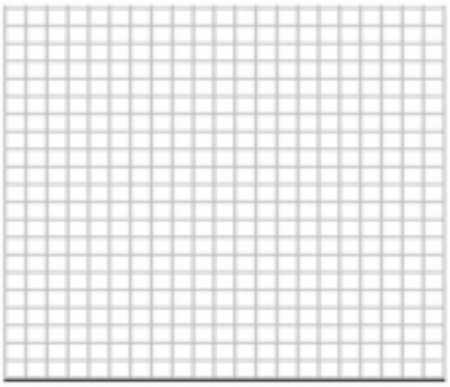

Fig 6a

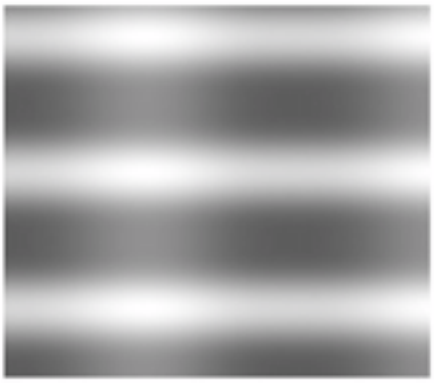

Fig 6b

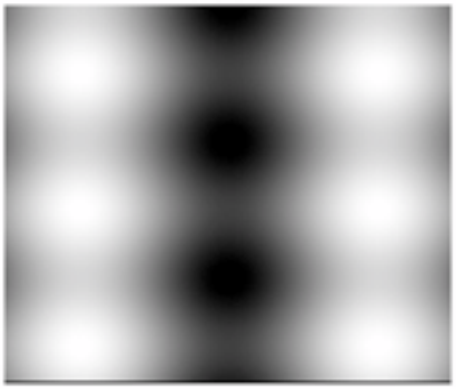

Fig 6c

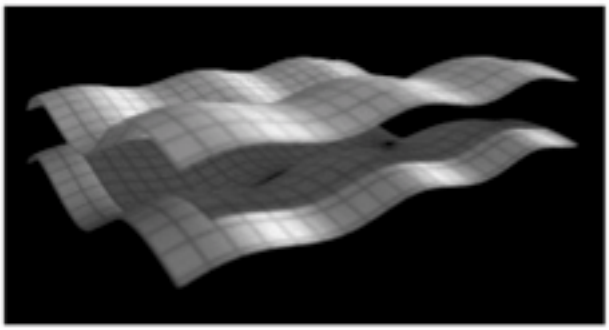

Fig 7

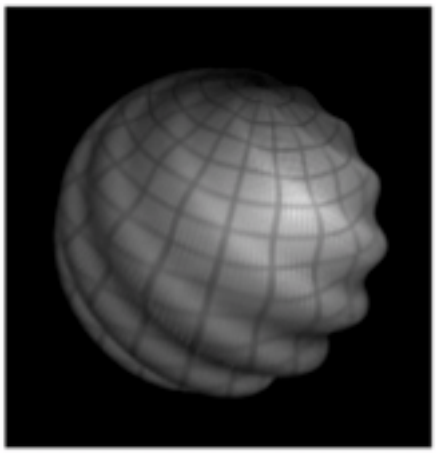

Figure 8

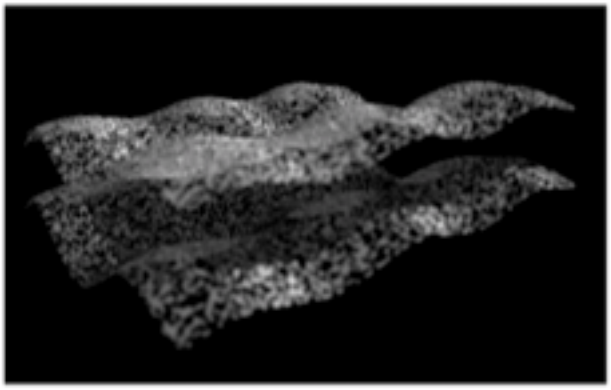

Figure 9a

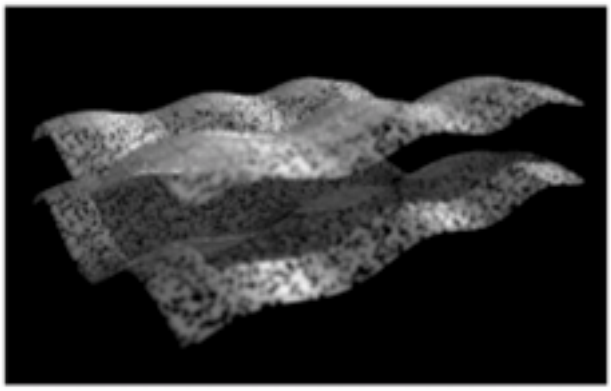

Figure 9b