The Visual Computer

Data structure for *soft* objects

Geoff Wyvill [1], Craig McPheeters [2], and Brian Wyvill [2]

[1] Department of Computer Science, University of Otago, Box 56, Dunedin, New Zealand
[2] Department of Computer Science, University of Calgary, 2500 University Drive N.W. Calgary, Alberta, Canada, T2N 1N4

We introduce the concept of *soft* objects whose shape changes in response to their surroundings. Established geometric modelling techniques exist to handle most engineering components, including 'free form' shapes such as car bodies and telephones. More recently, there has been a lot of interest in modelling natural pheomena such as smoke, clouds, mountains and coastlines where the shapes are described stochastically, or as fractals. None of these techniques lends itself to the description of *soft* objects. This class of objects includes fabrics, cushions, living forms, mud and water. In this paper, we describe a method of modelling such objects and discuss its uses in animation. Our method is to represent a *soft* object, or collection of objects, as a surface of constant value in a scalar field over three dimensions. The main technical problem is to avoid calculating the field value at too many points. We do this with a combination of data structures at some cost in internal memory usage.

Key words: *Soft* objects – Geometric modelling – Computer animation

The *Graphicsland* project group (Wyvill 1985a) at the University of Calgary has developed an organised collection of software tools for producing animation from models in three dimensions. The system allows the combination of several different kinds of modelling primitive (Wyvill et al. 1985b). Thus polygon based models can be mixed freely with fractals (Mandelbrot 1983, Fournier 1982) and particles (Reeve 1983) in a scene. Motion and camera paths can be described, and animation generated. Note that we do not include the use of a two dimensional 'paint' system. Our objective is always to construct views of a full three dimensional model.

An important class of objects in the everyday world is *soft*. That is, the shape of the object varies constantly because of the forces imposed on it by its surroundings. A bouncing ball is a simple example: as it strikes the ground, it flattens. The smoothly covered joints of animals change shape with seamless continuity, and liquids mould themselves to their surroundings and even break into separate droplets. Even apparently rigid objects deform in some circumstances. Trees, for example, bend in the wind.

To date, there seem to have been few attempts to model *soft* objects for computer graphics. Possibly, this is because *soft* objects are less important in engineering. But it is also true that much effort in computer graphics has been directed to producing still pictures and you cannot tell that an object is *soft* until it moves. Clouds (Gardner 1985) and particles (Reeve 1983) come close, but there is nothing in either of these papers which deals with the interaction of particles with surrounding objects.

We have been experimenting with a general model for *soft* objects which represents an object or collection of objects by a scalar field. That is a mathematical function defined over a volume of space. The object is considered to occupy the space over which the function has a value greater than some threshold so the surface of the object is an isosurface of the field function. That is a surface of constant function value within the space considered.

This is not a new idea. The technique is described by James Blinn, and used to create models of molecules (Blinn 1982). He also suggests other applications and describes a direct rendering technique using an elegant set of sorted lists. A similar technique has been used for some years in the LINKS project at the University of Osaka (Nishimura 1985). Ken Perlin has used a modification of Blinn's method to represent 'stochastic' shapes

The Visual Computer (1986) 2:227–234

(Perlin 1985). We, however, are interested in simpler shapes which we can *move convincingly*.
By suitable choice of field function, we can represent a wide variety of shapes conveniently and in principle, any shape somehow. For this paper we concentrate on simple functions based on proximity to given data points. We achieve animation by specifying the motion of these key points, without otherwise altering our function.
Strictly speaking, the word 'field' refers to a particular set of values distributed across space at some time. Thus the value of 'field' at some point $\langle x, y, z\rangle$ is found by evaluating the function $f(x, y, z)$. In what follows we use the words "function" and "field" more or less interchangeably.
In this paper we describe our choice of function and the data structure which enables us to construct the surface quickly. In the companion article in this issue of the Visual Computer (Wyvill 1986a), we describe some of the techniques for controlling the animation. This article is a revised version of a paper presented at the CG Tokyo conference (Wyvill 1986b). The only substantial revision concerns the data structure for the field calculation which has been improved. This article stands alone and can be undestood without reference to the earlier version.

2 Space function or *field*

We want to construct a function which will enable us to represent arbitrary shapes when we plot the iso-surfaces. The function is therefore going to depend on a set of given key points. We assume that the key points are independent. That is, we treat them like particles in a cloud. They are all alike and the field value at a point in space depends only on the proximity of key points. (These assumptions are arbitrary. One can easily imagine a function of key points which includes some knowledge about relations between points. The points could be ordered, for example, and a function value calculated by interpolation between adjacent point values.)
For our models we want the field to be continuous. For some models, we may use hundreds, thousands or even tens of thousands of key points. So it is important that we do not have to inspect every key point whenever we wish to calculate a field value. Therefore we use a function which is not influenced by any point beyond a certain distance away. This distance is known as the radius of influence, R. Again, arbitrarily, we make the field value due to several nearby points equal to the sum of the values due to each point. This is the very simplest way to combine the effect of many points and it seems to work well enough.
By definition, the contribution to the field made by any key point beyond its radius of influence, R, is zero. The contribution at the position of the point itself will be some maximum value (We use 1.0) and we would like to arrange that the field drops smoothly to zero at R. If we express this contribution to the field as a function C of r, the distance from the key point, these requirements can be expressed in terms of the values of the function at the point, $r=0.0$, and at the radius of influence $r=R$.

$$\begin{aligned} C(0.0)&=1.0 \qquad C'(0.0)=0.0 \\ C(R)&=0.0 \qquad C'(R)=0 \end{aligned} \tag{1}$$

where C' is the derivative of C with respect to r. These conditions are sufficient to define a unique cubic function for C:

$$C(r)=2\frac{r^3}{R^3}-3\frac{r^2}{R^2}+1 \tag{2}$$

The field at any point $\langle x, y, z\rangle$ is then the sum of $C(r)$ calculated for each key point within R. This field function turns out to be quite satisfactory but a little slow because calculation of r requires a square root. We eliminate the need for this by using a cubic in r squared and adjusting the coefficients to make it approximate the original function C. A cubic in r squared is guaranteed to have $C'(0.0)=0.0$, so we can add another condition: $C(r)=f$, for some chosen values r, f. We use $r=R/2$ and $f=0.5$ and this is close enough to the original function to make no difference in practice. Our field function is thus:

$$C(r)=a\frac{r^6}{R^6}+b\frac{r^4}{R^4}+c\frac{r^2}{R^2}+1 \tag{3}$$

where the values of a, b, c are found by solving Eq. (1) together with the condition $C(R/2)=0.5$. Approximately:

$$\begin{aligned} a&=-0.444444 \\ b&=\ \ 1.888889 \\ c&=-2.444444 \end{aligned} \tag{4}$$

This function is shown graphically in Fig. 1.
Blinn used an exponential function for his field based on the known field of electron density

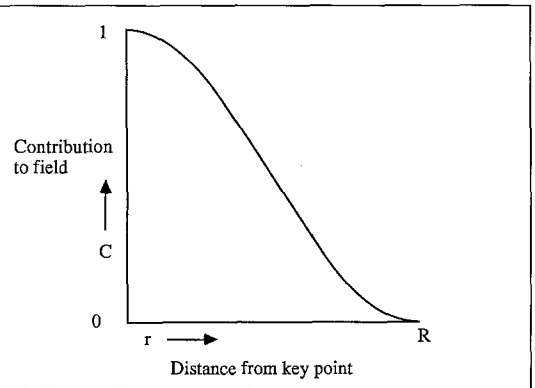


Fig. 1. Field contribution as a function of distance

around an atom (Blinn 1982). Our function is similarly shaped and has the desirable property of dropping to zero at the radius of influence, R. It is also very cheap to calculate, needing only three additions and five multiplications.

3 Defining the iso-surface

Having established a definition of the field we must choose a field value for the iso-surface. Clearly the field due to a single key point will be symmetrical about the point and any iso-surface in that field will be a sphere. Suppose we choose a function value, *magic*, and plot the iso-surface connecting all points whose field value equals *magic*. Now consider the field due to two key points in the same place. It is still symmetrical and an iso-surface for value *magic* in this field will be a sphere of larger radius than the iso-surface in the field due to one point. We have chosen *magic* so that this larger sphere has exactly twice the volume of the other.

This choice is intended to suit the modelling of liquids, to provide a reasonable effect when two droplets merge. Other choices are possible as are other functions for the field. Finding functions appropriate to particular applications is a research project in its own right. For our purposes, the above is used throughout.

4 Producing the surface

The surface defined in this way, by a collection of data points is very general. It is not even necessarily connected and in order to make a picture, we first convert it to a more tractable form. We have chosen to use a simple polygon mesh for this purpose.

We construct the mesh in two distinct stages. Imagine that the part of space occupied by the surface is divided by a three dimensional grid into small cubes. First we find all the cubes which are intersected by the surface and then we construct the polygons in each cube.

To find the cubes intersected by the surface without scanning the whole of a large three dimensional grid, we take advantage of the knowledge that all our key points are enclosed by some part of the surface. For each key point, starting at the nearest grid point, we calculate the field at a succession of adjacent grid points along one axis until we encounter a point whose field value is less than *magic*. This point and the previous one form the endpoints of one edge of a cube which is intersected by the surface. This process gives us a set of 'seed' cubes such that every disconnected component of the surface intersects at least one seed cube.

Each seed cube shares faces with six neighbours. Starting at the seed cubes, we examine each cube's neighbours to see whether or not it is intersected by the surface. If a neighbour is intersected then we look at its neighbours and so on until all of the cubes intersected by the surface have been found. This completes the first stage.

In the second stage, we have only to deal with cubes which are intersected by the surface. For each cube we have eight values which are the field values at its vertices. From these we construct a set of polygons which are part of the iso-surface. The previous stage has sorted out all the cubes intersected by the surface, so when we put these polygons together we have a representation of the surface.

To complete the description we must explain the data structure used in the first stage and the logic used in the second.

5 Data structure

There are two distinct problems each handled by a structure using a hash table.

5.1 Fast evaluation of field values

The first problem is to be able to calculate the field value at any point efficiently. This is solved using the structure shown in Fig. 2.

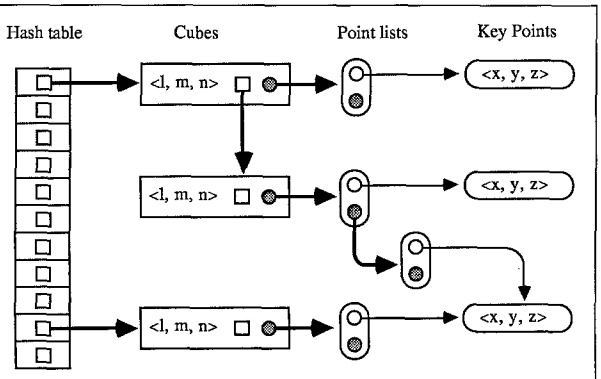


Fig. 2. Data structure for efficient calculation of field value

The volume of space which represents the whole scene is divided into cubes of side S. Each of these cubes is represented by a record which heads a linked list of pointers to key points. The list has pointers to just those key points close enough to the cube to affect the field within it. Thus there can be many pointers to a given key point, especially if it has a large radius of influence R. The key points themselves are represented by triples of floating point values, $\langle x, y, z\rangle$ together with the cubic coefficients for fast evaluation of the field function. These coefficients are calculated from the radius of influence during setting up. Other properties related to colour and texture are also stored here.

Only non-empty cubes are represented and they are accessed by means of a hash table. The table entries are pointers to cube records. Each record contains a triple $\langle l, m, n\rangle$, where $\langle l*S, m*S, n*S\rangle$ represents the 'low' vertex of the cube. The 'low' vertex is the bottom-south-west corner, or, more formally, the vertex of lowest $\langle x, y, z\rangle$.

The hash address is calculated from the $\langle l, m, n\rangle$ triple. This is used as an index in the table. The table entry contains a pointer to a linked list of cube records. To access all the key points in a given cube, we first find the list of records from the hash table. Then we search down the list until we find a cube which matches our values of $\langle l, m, n\rangle$. This group contains the pointer to the appropriate list of key points. The majority of hash table entries are empty or point to a list of length one. In practice, there is very little searching down these lists.

The choice of S depends on the expected density of key points. If S is too large, many unnecessary points may be inspected when we calculate the field. If it is too small, the data structure gets too big. Ideally the program itself ought to find the optimal value. At present we set it by hand. In the case where there are many key points with the same radius R, we set $S=R$. Please note that the cubes referred to in this section are not the same as the little cubes used to build the polygon mesh.

5.2 The cubic net

Our second problem is to avoid recalculating field values as we find the cubes which intersect the surface. This is done with a second hash table, Fig. 3. We are only interested in calculating the field at grid points which are the vertices of our little cubes. It is convenient, therefore, to represent these points by integers i, j, k where $i*d, j*d, k*d$ are the actual co-ordinates of the point and d is the grid spacing. Each vertex is represented by a quintuple $\langle i, k, j, f,$ done$\rangle$ and these are linked in a list for access through the hash table. 'f' is the field value for the vertex and the meaning 'done' is explained below.

To access the quintuple representing a given vertex a hash code is calculated from its i, j, k values. This is used as an index in the table and the table entry contains a pointer to a linked list of vertices. This list must be searched to find the particular vertex. When a vertex is referred to for the first time, the search fails. In this case the field value, f is calculated and a new vertex is linked in to the list. Each vertex is shared by as many as eight cubes but its field value need be calculated only once. Subsequent references will find it stored in the structure.

To trace all the cubes intersected by the iso-surface, it is necessary to mark those cubes which have already been processed. For this purpose,

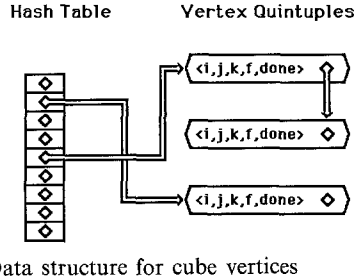


Fig. 3. Data structure for cube vertices

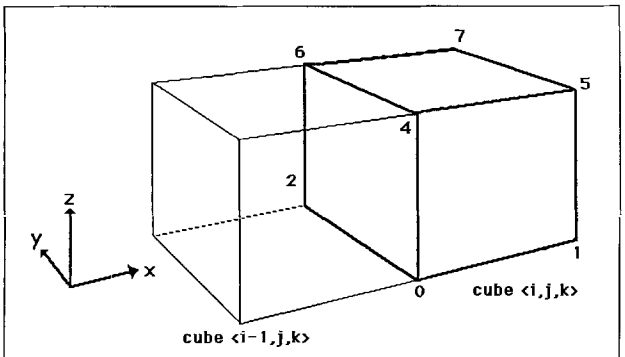


Fig. 4. Neighbouring cubes

each vertex is also considered to represent the cube of which it is the 'low' vertex. The flag 'done' within the vertex is made **true** to indicate that this cube has been dealt with.

The algorithm for finding all cubes on the surface from a seed cube can now be described. Observe the numbering of the cube vertices in Fig. 4. The low vertex of the cube is numbered 0 and this is the vertex $\langle i, j, k\rangle$. Consider the four vertices 0, 4, 6, 2. These are shared by the cube $\langle i-l, j, k\rangle$. If the field values of these vertices are all greater than the iso-surface value or if they are all less, then the surface does not pass through the face 0, 4, 6, 2. If some of the values are greater and some less than the iso-surface value, the surface does pass through the face and the cube $\langle i-l, j, k\rangle$ must be processed if this has not already been done. To process a cube, first its 'done' flag is made **true** and then a pointer to the low vertex of the cube is entered in a queue. More precisely:

```
begin Set seed cube's done flag to true.
    Add seed cube to the queue.
    while queue is not empty do
        begin Remove one cube from the queue.
            for each face of cube do
                begin if surface intersects face then
                    begin select neighbour cube for that face.
                        if neighbour's done flag is not true then
                            begin Set neighbour's done flag to true.
                                Add neighbour to queue.
                            end
                    end
                end
            Pass vertices and values for cube to second stage.
        end
end
```

5.3 The hash functions

Most of the space is empty. That is to say that the cubes which intersect our *soft* object's surface represent a small fraction of the total. Similarly, of the group cubes described in 5.1, those which contain key points are few compared with the number of such cubes in the region of space we are dealing with. If this were not so, we would find no advantage in using a hash table. We would simply store points in an array. So our hash function must have an even distribution for points which are geometrically close together. In both of our tables, the hash function maps a triplet to an address. For a table of size $l*m*n$ we map $\langle i, j, k\rangle$ to:

$$(\mathrm{rem}(i, l)*m+\mathrm{rem}(j, m))*n+\mathrm{rem}(k, n) \qquad (5)$$

where $\mathrm{rem}(x, y)$ is the remainder when x is divided by y. We use $l=m=n=16$ and the function is calculated quickly by logical operations. A more sophisticated function might be appropriate for some applications but we have found no reason to change it.

6 Generating the polygons

In this part of the process, we are given the field values at the vertices of a cube in the mesh and we must construct polygons which represent the part of the iso-surface which intersects that cube.

Firstly we find points which approximate the intersection of the iso-surface with the edges of the cube. Then we connect these to make the polygons. To avoid ambiguity we refer to these points as 'intersections'. The word 'vertex' is reserved for vertices of the cube. A vertex whose function value is greater than *magic*, we call hot, and a vertex whose function value is less than or equal to *magic* we call cold. Vertices whose field value is exactly *magic* present minor complications. (We have to take special action to avoid generating polygons of zero size.)

Suppose two adjacent vertices p, q have field values f_p, f_q. If q is hot and p is cold, then:

$$\frac{\text{magic}-f_p}{f_q-f_p} \qquad (6)$$

is taken as the distance from p to the intersection of the iso-surface with pq. Although this 'linear interpolation' is not the true intersection, it is

much cheaper to calculate and, provided the cubes are small enough that the polygonal approximation to the surface is reasonable, it is good enough. Note that this calculation is consistent across adjacent cubes which share edges.

The process by which we connect these points is fairly complex, so we start with an example. Figure 5A shows a cube with only one hot vertex. Clearly the intersecting iso-surface is approximated by a single triangle. In Fig. 5B, there are two adjacent hot vertices and we have a quadrilateral. In Fig. 5C, there are two non-adjacent hot vertices and the polygon is a hexagon which encloses both of them. Why do we choose this hexagon rather than two triangles (Fig. 5D)? In effect we are electing to link the two hot vertices and separate the two cold ones. Whether to link the hot or cold vertices in this case is decided by examining the values of the field. The value at the centre of the face is approximately the mean of the four vertex values. If this value is greater than *magic* we link the hot vertices.

Each intersection is uniquely associated with one edge of the cube. So we can label an intersection by the pair of vertices $\langle p, q \rangle$ between which it lies. The possible configurations of one face of the cube are shown in Fig. 6. When the number of hot vertices in this face is four or zero, no polygon edges are created: cases A, B. When the number of hot vertices is one or three, a single edge is created: cases C, D. For two hot vertices we have

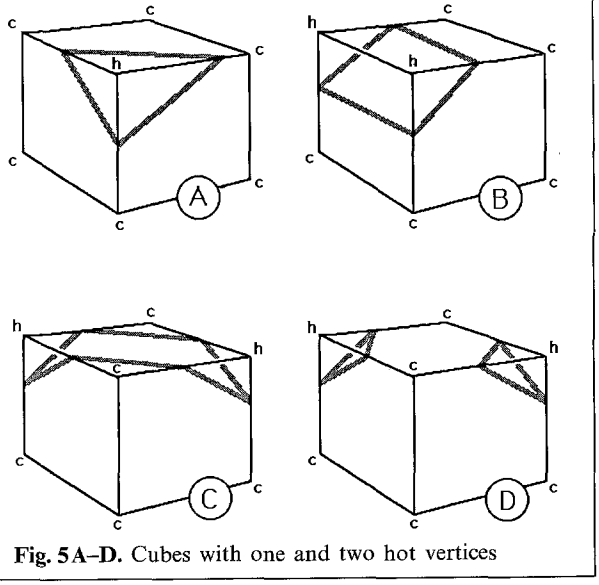


Fig. 5A–D. Cubes with one and two hot vertices

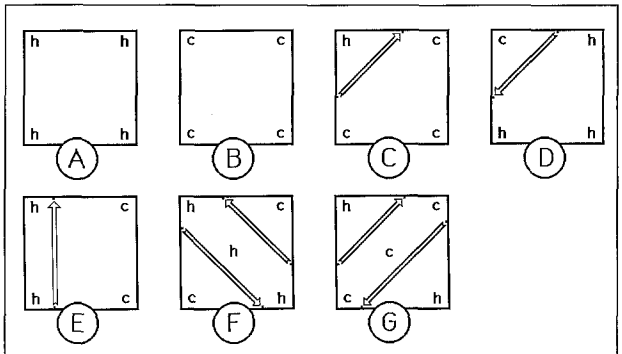


Fig. 6A–G. The seven different cases for connecting intersections

case E: create one edge; case F: create two edges linking hot vertices; and case G: create two edges linking cold vertices. These created polygon edges are represented by ordered pairs of intersections. That is pairs of vertex pairs. In Fig. 6, the order is indicated by the arrows.

These polygon edges are stored in an array, indexed by the first intersection. The second intersection is always the same as the first intersection of some other edge, so we can form a polygon by tracing the natural successors of each edge until we return to the edge we started at.

The algorithm follows:

```
for each edge of cube, <p,q> do
    if p is hot and q is cold or p is cold and q is hot
        then create intersection <p,q>;
for each face of cube do create edges according to Fig. 6;
while edges remain do
    begin start:=any edge;
          polygon:={start};
          remove start from edge array;
          next:=successor of start;
          while next <> start do
              begin polygon := polygon + {next};
                    remove next from edge array
              end;
          output polygon
    end
```

The polygons are not, in general, planar. So we divide them into triangles by connecting each intersection to a centroid as follows:

Given an ordered set of points (polygon vertices):

$$p_i = \langle x_i, y_i, z_i \rangle, \quad 0 \leq i < n \tag{8}$$

The centroid is:

$$C=\left\langle \frac{1}{n}\sum_{i=0}^{n-1} x_i, \frac{l}{n}\sum_{i=0}^{n-1} y_i, \frac{l}{n}\sum_{i=0}^{n-1} z_i \right\rangle \tag{9}$$

For $n>3$ we can divide the polygon into triangles:

$\langle p_{i-1}, p_i, C\rangle, \quad 0<i\leq n-1$

and

$\langle p_{n-1}, p_0, C\rangle$ (10)

This method of triangulation doesn't work for polygons in general but it seems to be alright for polygons generated by this algorithm.

7 Animating the objects

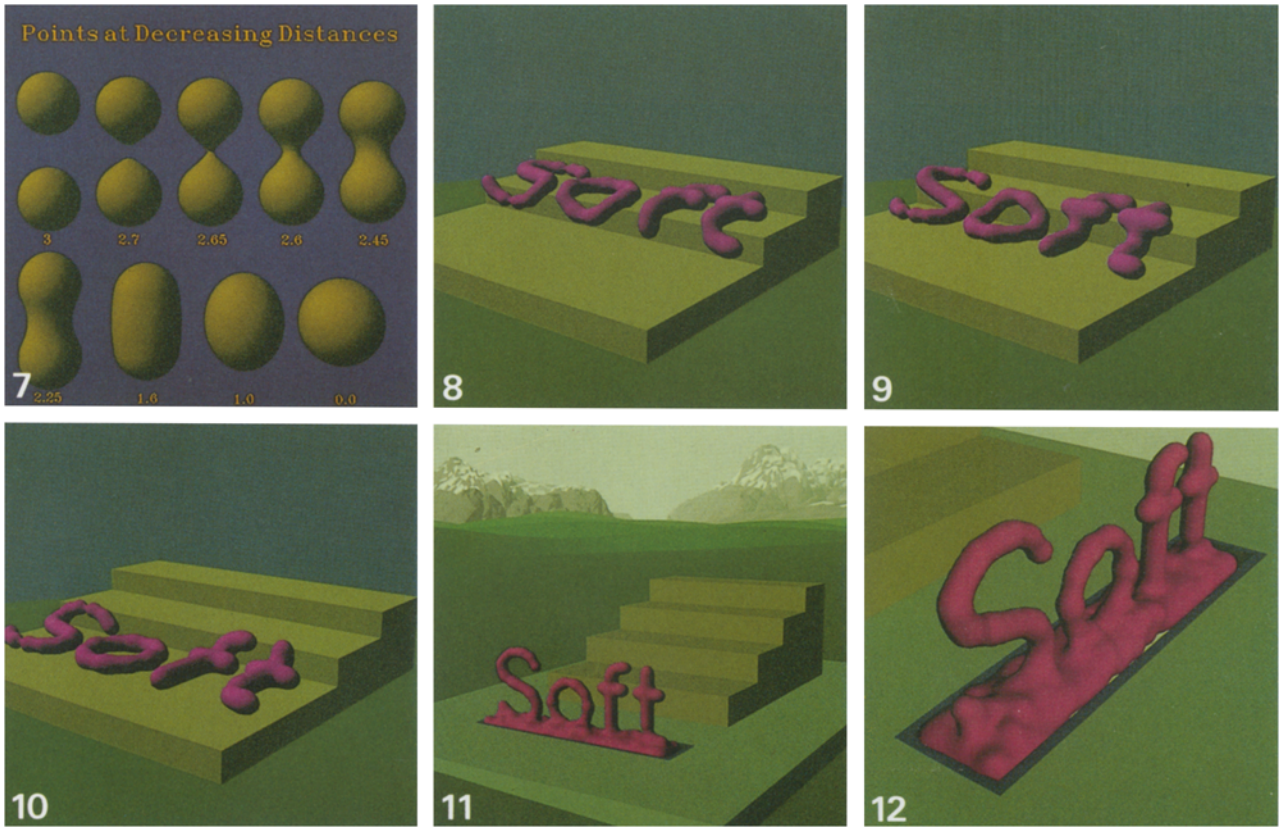
The shape of a *soft* object is entirely determined by the positions of the key points. So we describe its motion and shape changes by moving key points only. This is discussed in more detail in the companion article (Wyvill 1986a). Here we describe only a few examples.

7.1 Examples of application

Figure 7 shows two droplets merging in stages. This is quite pleasing in animation. The droplets are modelled by single key points with $R=2.0$. The distance, r, between the two key points is shown for each stage.

Figures 8–10 are frames from an animation of a *soft* object sliding down steps. The object is roughly shaped as the letters "Soft" and these distort smoothly as they slide down.

Figures 11 and 12 show the letters "Soft" rising from a trough of bubbling *soft* material. The background in Fig. 11 features fractal mountains and rolling hills. These are part of the solid scene and show how the *soft* objects have been incorporated into the *Graphicsland* system.



Figs. 7–12. For explanation see text

8 Discussion

Our modelling technique has proved to be a convenient tool in computer animation. So far, we have concentrated on one very simple method of making our field functions and there are many other possibilities which are worthy of investigation. The field can be generated by arbitrary functions permitting the modelling of mathematical surfaces. Our field function can be modified by extra terms which are not related to the key points. This can be used to make objects deform in special ways. For example, if we add a surface held to a negative field value, objects can be made to vanish bit by bit as they approach the surface, and to reappear on the other side. If we make a 'hole' in the surface, an object approaching the hole, appears to squeeze through it.

We have represented our surfaces by polygon patches. This is because the *Graphicsland* system already offered us versatile display tools for this kind of model. It would clearly be a good idea to use bicubic or conic patches instead and we are planning some experiments to do this. We have not followed Blinn's technique (Blinn 1982) of rendering the surface directly, because we wanted to produce objects which we could easily incorporate into *Graphicsland*. Also, our models can contain a very large number of key points and most of them are not near to any surface. This is particularly true of models of liquids. The polygon generating algorithm is particularly efficient in this case because the 'hidden' key points do not have to be known to the rendering algorithm.

We are, however, experimenting with a ray-tracer which renders the field directly. In this case we use our surface following algorithm to construct a boundary of cubes which contains the iso-surface.

Even when we have a superior rendering technique for these surfaces, an efficient generator of polygon patches will be useful. *Graphicsland* does not yet offer interactive graphical development of animation, because it takes too long to render the scenes. We are introducing a new feature to produce very fast rendering of scenes at greatly reduced quality. For this purpose we expect to continue to need a polygonal representation.

Our *soft* objects are reasonably quickly generated. The letters and contents of the trough in Fig. 11 took about 8 min per frame on a VAK 11/780 computer to generate the polygons. This is much less than the rendering time for 512 by 512 pixels using a z-buffer algorithm.

Acknowledgements. The JADE project at the University of Calgary has been particularly supportive. This work and JADE is supported by the Natural Science and Engineering Research Council of Canada.

References

Blinn J (1982) A Generalization of Algebraic Surface Drawing. ACM Transactions on Graphics 1:235–256

Fournier A, Fussel D, Carpenter L (1982) Computer Rendering of Stochastic Models. CACM 25:371–384

Gardner G (1985) Visual Simulation of Clouds. SIGGRAPH 85 Computer Graphics 19 (3):297–303

Nishimura H, Hirai M, Kawai T, Kawata T, Shirakawa I, Omura K (1985) Object Modeling by Distribution Function and a Method of Image Generation. Journal of papers given at the Electronics Communication Conference '85, vol. J68-D No 4 (in Japanese)

Mandelbrot B (1982) The Fractal Geometry of Nature. W.H. Freeman, San Francisco

Perlin K (1985) An Image Synthesizer. SIGGRAPH 85 Computer Graphics 19 (3):287–296

Reeves W (1983) Particle Systems – A Technique for Modeling a Class of Fuzzy Objects. ACM Transactions on Graphics 2:91–108

Wyvill BLM, McPheeters C, Garbutt R (1985a) A Practical 3D Computer Animation System. The BKSTS Journal 67:328–332

Wyvill BLM, McPheeters C, Novacek M (1985b) Specifying Stochastic Objects in a Hierarchical Graphics System. Proceedings of Graphics Interface 85, Montreal, pp 17–20

Wyvill BLM, McPheeters C, Wyvill G (1986a) Animating Soft Objects. The Visual Computer 2:235–242

Wyvill G, McPheeters C, Wyvill BLM (1986b) Soft Objects. Advanced Computer Graphics. Proceedings of Computer Graphics Tokyo 86, pp 113–128