Computing Random Fields

N. E. WISEMAN AND S. NEDUNURI*
Computer Laboratory, University of Cambridge, Corn Exchange Street, Cambridge CB2 3QG
**Now at Hewlett Packard Laboratories, Filton Road, Stoke Gifford, Bristol BS12 6QZ*

A program that moves over a 2- or 3-dimensional mesh performing refinements by moving vertices or introducing new ones is a common enough matter. If the data structure for the mesh persists across the entire calculation its management is simple. However, it may be that the mesh data are needed only transiently, and in this case the maintenance of a (possibly enormous) data structure across the whole calculation seems very space-inefficient. This point takes on extra strength when the mesh is supposed to model some random field, such as a mountain range, cloud formation or meadow plain (generated no doubt by a method related to Mandelbrot's fractal curves) when the only use of the mesh is to make a picture for display. Existing methods for dispensing with the need for the whole mesh across the whole calculation complicate the program considerably. This paper proposes the use of co-routines to control the independent update of all edges depending on a changing vertex. In this way facets are passed out as soon as they are complete and can then be displayed and disposed of when convenient. Only visibility issues dictate how long a piece of data survives.

Received May 1985

1. INTRODUCTION

Suppose that a surface is to be modelled by generating a mesh that clothes it and that this mesh is built by successively refining a coarser one. The edges of the mesh will enclose facets of surface and the refinement will cease when these facets are either sufficiently small or sufficiently close to the desired surface.

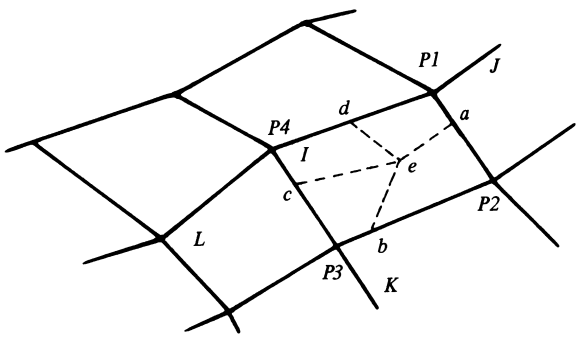

Figure 1. Mesh refinement

In Fig. 1 a portion of such a mesh is shown and facet *I* having vertices *P1*, *P2*, *P3*, *P4* is about to be refined. New vertices are introduced at the points marked *a*, *b*, *c*, *d*, *e* by applying some formula to the mesh edges (it does not matter here what this formula is). If the new facets are sufficiently good they can be issued to the consumer, otherwise further refinements take place. Eventually the program moves on to other areas in the mesh, returning at various times to neighbouring facets *I*, such as *J*, *K* or *L*.

On these return visits the refined facet *I* must be seen – in particular, the new vertices which were introduced into its boundary edges must be used as part of the refinement of the present facet. There are really only two ways to ensure this. Either they have to be recomputed by some scheme guaranteed to come up with the same values each time (not trivial if they depend on some random variable) or they have to be stored somewhere in case they are needed for future visits. Neither seems ideal, for recomputing them is likely to be expensive in time and storing them can be expensive in space. Ingenuity must be brought to bear, so that the expense is not unreasonably high. Two previously published approaches will be described. Both are for use in making a mesh with vertices that depend on random variables.

Fournier, Fussel and Carpenter[1] adopted a method of recomputing the values on each encounter, making the random number seed some unique function of the edge coordinates to ensure repeatability. It was also necessary to average the normals of the facets meeting on that edge to establish the direction of perturbation repeatably. An algorithm which initially offered considerable simplicity and elegance suffered badly from these acts, although of course something had been necessary to cure the difficulty mentioned above. Coquillart[2] takes the opposite approach. He stores the values in a data structure specially contrived to enable old data to be stripped out when no longer needed – a form of heap with explicit capture and release of space for storing the values in question. The amount of store used is economized by this, but again at a cost in algorithm simplicity. The method reported here also stores the values for later re-use, but with very little visibility to the programmer of the storage structures in use. The idea is basically to regard every facet as being computed concurrently and to pass new refinements of existing edges to all consumers as they are calculated. Of course there is no need actually to compute things in parallel, and a fleet of co-routines is adopted instead. Edges are never visited more than once, so there can be no problem with consistency, and directly all the consumers of a fresh piece of data have done their work the data disappear.

2. MESH GENERATION

A mesh that clothes a patch of surface may be represented as a set of vertices and joining edges that delimit a number of plane facets each chosen to be close enough to the surface. Refinement of the mesh corresponds with tightening the interpretation of 'close enough'. It can be done by reducing the overall spacing of a regular mesh

or replacing facets by smaller ones only where needed. Some surfaces can be represented as a function of position on a 2-dimensional rectangular grid which gives the projection of the surface from that position. The projection is orthogonal to the grid. In that case refining the mesh consists of refining the rectangular grid in the plane until the projected surface is sufficiently well defined. Note in this case that not all surfaces will refine properly (or at all) and that multiple projections of a single position are disallowed. The resulting surface is called a *Projected Mesh*. The methods proposed in this paper will in general work for both clothing and projected meshes, although the program details will be different.

There are many ways in which the mesh data could be generated. The most straightforward idea is probably that of Fig. 1, illustrating recursive subdivision of a single quadrilateral into a mesh of the desired fineness. At each level of recursion the procedure inserts a cross-shaped set of four new edges into the quadrilateral passed to it. If the desired resolution has been reached the procedure then returns, otherwise each quadrilateral formed by inserting the new edges is passed down to four distinct recursive calls to the same procedure. The following code shows how it works (it makes the data, but does nothing with them).

```
// p1, p2, p3, p4 is the input quadrilateral
// index is a control on level of recursion
let refine(p1, p2, p3, p4, index) be
$(  // make 4 new vertices by refining existing edges
    let a = fun(p1, p2)
    let b = fun(p2, p3)
    let c = fun(p3, p4)
    let d = fun(p4, p1)
    // and a new vertex by refining existing vertices
    let e = another.fun(p1, p2, p3, p4)

    unless index = 0
    $(  refine(p1, a, e, d, index-1)
        refine(a, p2, b, e, index-1)
        refine(e, b, p3, c, index-1)
        refine(d, e, c, p4, index-1)
    $)
$)
```

The procedure does not communicate with its brothers of course, which give rise to the consistency problem addressed by this paper – there will be two independent refinements of each edge at every level of recursion, with the result that the final mesh may not be properly connected. An even more contorted result could be obtained if the recursive call depended on the closeness of match of the subdivided facet with the desired surface, rather than the degree of subdivision.

Another method is to regard the quadrilateral mesh as being delimited by struts running in one direction and rungs in the other. The program scans along the struts and rungs performing a recursive or iterative subdivision of each to achieve the desired accuracy. The idea is shown below using recursive subdivision:

```
let refine(p1, p2, p3, p4, index) be
$(  let strut(p, q, r, s, strutindex, rungindex) be
    $(  let strutrung(u, v, rungindex) be
        $(  let e = fun(u, v)
            unless rungindex = 0
            $(  strutrung(u, e, rungindex-1)
                strutrung(e, v, rungindex-1)
            $)
        $)
        let d = fun(p, q)
        let b = fun(r, s)

        test strutindex = 0
        then strutrung(d, b, rungindex)
        or
        $(  strut(p, d, r, b, strutindex-1, rungindex)
            strut(d, q, b, s, strutindex-1, rungindex)
        $)
    $)

// and a similar declaration for rung and rungstrut to
// cross the mesh in the other direction

    strut(p1, p4, p2, p3, index, index)
    rung(p1, p2, p4, p3, index, index)
$)
```

This seems no better. The code is less easy to understand and the only points where strut and rung vertices must coincide are at their beginnings and ends (that is each strut(rung) must start(end) on a rung(strut)). The same problem with consistency arises. The problem will only go away if new mesh vertices remain available until all neighbouring edges have used them. It could be thought of as a scope rule matter – data are arranged to have a scope just sufficient for the intended task (if too small the values must be recomputed, if too large then space to store them is being wasted). For a single-threaded program moving over a multiply connected structure like a quadrilateral mesh the necessary scope cannot be achieved with local variables. We thus have to adopt some form of off-stack storage for holding the values or use a multi-threaded program. Taking the former approach, we could try persistent data, such as a database or a file. In-core storage on a heap, in a manner such as that used by Coquillart, is better. The other idea, of using a multi-threaded program, is more unusual. The programming for a multi-threaded *struts-and-rungs* solution is extremely easy and the mesh data survive exactly as long as they are wanted (and no longer). True parallelism is not necessary of course, and co-routines are used below, enabling the same idea to be implemented with better operating efficiency.

The basis of the algorithm can be explained as follows. A co-routine is launched on each strut with the task of moving along it, emitting coordinates at suitable intervals. It adopts some method of subdivision to refine each strut into as many parts as are needed, and is assumed to be pure code, so that a single copy is sharable over all struts. Each instance has to emit each new coordinate to the main routine which makes rungs. The algorithm driving each strut should preferably take no information about the other struts or the rungs that result from each step which it takes. The general form is seen below:

```
// create a batch of coroutines and initialize them
for i = 0 to struts
$(  strutvec!i := cocreate(strut, stacksize)
    // set up initial arguments in pkt
    // and start each strut off
    cocall(strutvec!i, pkt)
$)
```

```
// go across the struts, making a chain of rungs on each step
for j = 0 to rungs
$(  new.vertex := cocall(strutvec!0, 0)
    for i = 1 to struts
    $(  old.vertex := new.vertex
        // now collect another vertex from the next strut
        new.vertex := cocall(strutvec!i,0)
        rung(old.vertex, new.vertex)
    $)
$)
```

and the strut co-routines go:

```
// invent a vertex P
// if sufficiently refined then
// make the next bit of strut that uses P
// and issue P to the rung maker with:
cowait(P)
// recurse, iterate or return as appropriate
```

The program sweeps across the mesh making at each step a segment of every strut and the rung that joins these segments together. The struts grow in the individual instances of the strut co-routines, and the rungs are put in by the calling procedure, which is released incrementally as the strut co-routines advance. In a simple program that draws meshes, the rung procedure would just draw a segment of rung, and the strut co-routines would each draw along their strut to each new vertex as it is generated. Figures 2, 3 and 4 show examples of meshes drawn in this manner. The first is a projected mesh for

$$z = f(x+d, y+d)+f(x+d, y-d)+f(x-d, y+d) \\ +f(x-d, y-d)$$

where

$$f(x, y) = A/(B+x*x+y*y)$$

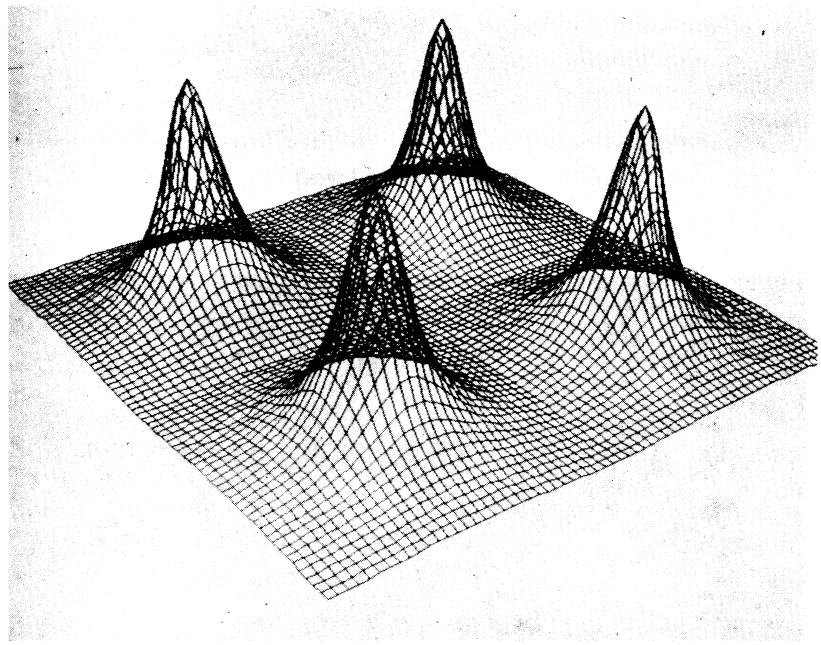

Figure 2. Mesh for a simple function

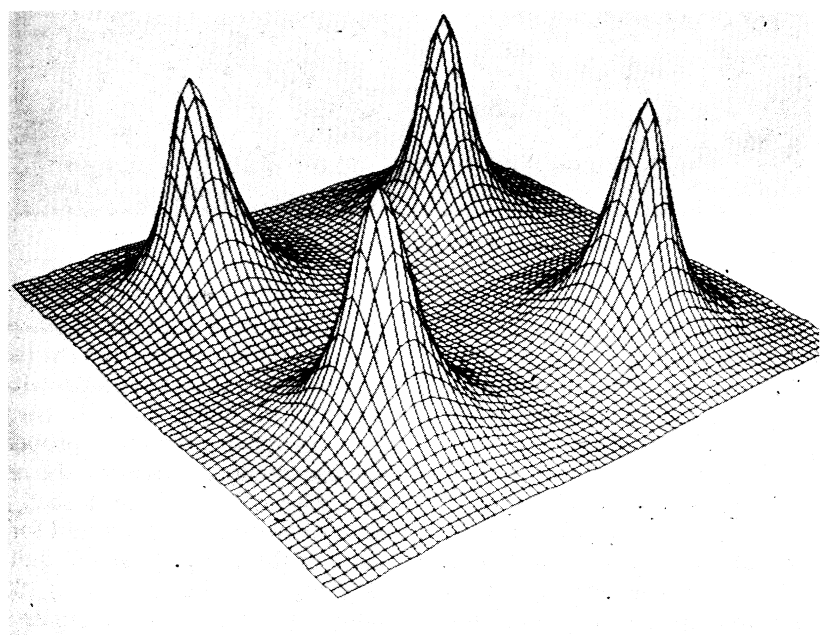

Figure 3. The same mesh with hidden lines removed

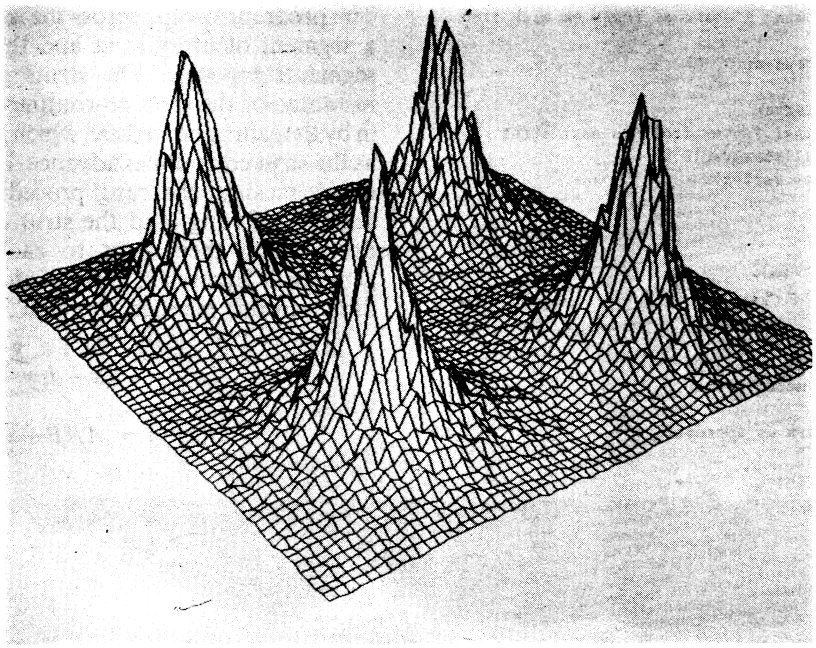

Figure 4. Fractal mesh

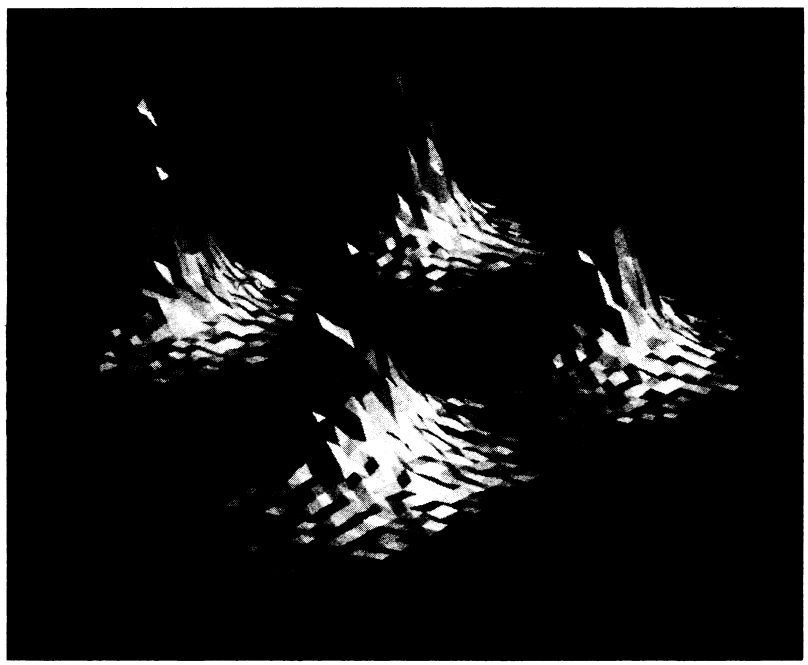

Figure 5. Shaded fractal mesh

and the next is the same mesh with hidden lines removed, using the painter's algorithm and filling each facet with background colour. Fig. 4 shows the hidden-line version of a form of fractal mesh in which each strut is separately refined from the original base function (z). The refinement of the base function mesh (we call it 'fractalising' but only with apologies!) is done by a method closely similar to that described in Ref. 1. Observe that the mesh remains properly connected, with no complication arising from the perturbation of its vertices. Shaded versions using the growing silhouette method are shown in Figs 5 and 6. In this case the sweep direction is left to right, front to back, and the viewing position must be such that the sweep never obscures itself. This restriction arises from the fact that the scope rule (if we may call it that) has no knowledge of the requirements for facet visibility, but only ensures vertex survival for long enough to guarantee a properly connected mesh. For the use intended, however, the restriction is not serious, and self-shadowing effects such as are shown in Fig. 6 scenes can be easily produced for a single light source in front of the sweep. Scenes of quite high complexity have been made by the method – a 100,000-facet image takes about 30 minutes to compute and display with self shadowing, and 10 minutes without, on an M68000-based microcomputer.

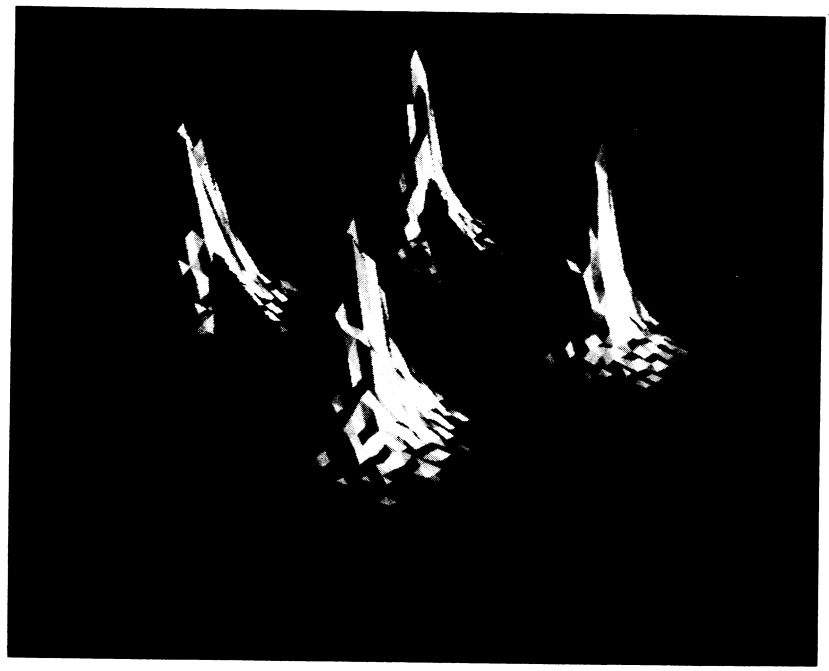

Figure 6. Self-shadowing shaded fractal mesh

3. CONCLUSIONS

The method described combines economy of storage and simplicity of code. With increasing interest in special-purpose display processors it may find its place in simplifying the process of mesh display by being implemented in hardware. For implementation in software, we have observed that multi-threaded programs are frequently simpler and may even be more efficient than their single-threaded counterparts. Simpler programs are easier to get working.

REFERENCES

1. A. Fournier, D. Fussel and L. Carpenter, Computer rendering of stochastic models. *CACM* **25**, (26) 371–384, (1982).
2. S. Coquillart, Displaying random fields. *Computer Graphics Forum* **4**, (1), 11–19, (1985).
3. D. P. Anderson, Hidden line elimination in projected grid surfaces *ACM Transactions on Graphics*, **1**, 274–288, (1982).
4. E. Catmull and J. Clark, 'Recursively generated B-spline surfaces on arbitrary topological meshes. *Computer-Aided Design* **10**, (6), 350–355, (1978).
5. D. Doo & M. Sabin, Behaviour of recursive division surfaces near extraordinary points. *Computer-Aided Design* **10**, (6), 356–360 (1978).