Grid Tracing: Fast Ray Tracing for Height Fields

F. Kenton Musgrave

Department of Mathematics
Yale University

ABSTRACT

A fast algorithm for ray tracing height fields is presented. The algorithm employs a modified Bresenham DDA to traverse a two dimensional array of height values. At each cell the altitude of the ray is compared with the heights of the four corners of the cell; ray/object intersections need only be calculated when the altitude of the ray is in the range of those heights. The average number of ray/object intersections performed is about two per ray; the two objects tested for intersection are located in $O(\sqrt{N} - 1)$ time where N is the number of height values in the field.

KEYWORDS AND PHRASES: Ray tracing, fractal, DDA, height field, surface tessellation, stochastic surface, 2D grid, Linda, parallel processing.

November 3, 1990

Grid Tracing: Fast Ray Tracing for Height Fields

F. Kenton Musgrave

Department of Mathematics
Yale University

1. INTRODUCTION

"Grid tracing" is a fast technique for ray tracing height fields. A height field is a dataset composed of a two dimensional array of altitude values, which describe a surface. Grid tracing is essentially a fast method for testing bounding volumes, which takes advantage of the spatial coherence and intrinsic ordering of a height field. The problem that inspired this work is that of ray tracing fractal terrains,[12, 21, 5, 6] specifically those composed of a large number of triangles which tessellate the surface. In a naive ray tracing algorithm, the time required to ray trace a scene is determined largely by the number of primitives in the scene; most of the computing time in ray tracing is spent on calculating ray/object intersections.[18, 23] Many schemes to reduce the number of objects which are to be intersected have been devised.[3, 7, 8, 10, 11, 13, 18, 19, 22] Grid tracing is not dissimilar to the 3DDDA scheme used by Fujimoto[7] to reduce the number of ray/object intersections, yet it is a new and more efficient approach, though it is limited to rendering surfaces which can be described as two dimensional height fields. In practice, "grids" are perhaps best categorized as a new primitive object for ray tracing; as such they can be incorporated into general ray tracing systems such as that of Syder et. al.[19]

When ray tracing fractal terrains, memory constraints are a significant concern as the sheer number of objects used to represent the surface of the terrain (upwards of a million triangles) can easily overwhelm the memory capacity of any commonly available computer. Thus procedural definitions for fractal terrains have been developed.[4, 9, 14] These schemes do not attempt to compute the actual location of the surface until a ray enters the immediate vicinity of the surface. As the algorithm determining the morphology of the surface is deterministic, no information describing the primitive polygons which compose the surface need be stored. Thus efficient memory usage is exchanged for increased computation time (many steps of the recursive algorithm determining the surface location must be repeated often if their results are not stored). An advantage of these schemes is that the level of detail in the representation of the surface may be dynamically changed for the purposes of animation. Drawbacks of procedural schemes include increased computation time and difficulty of implementation.[20]

The approach presented here takes advantage of the coherence and regularity of two dimensional height fields to get a significant speed advantage over naive algorithms which consider a large number of objects for ray intersections. In contrast to procedural algorithms, in this approach the entire height field is computed just once and stored, prior to rendering. Therefore no computations need be repeated. While the level of detail in the field cannot be immediately changed for a dynamically changing point of view, the precomputed height field can, however, be rendered with a great savings in computation time.

This idea for ray tracing a height field, as opposed to a procedurally defined surface, was generated out of necessity. Procedural definitions of terrain surfaces rely on some knowledge of the shape of the projected area covered by the polygons in the patch of terrain, for use in determining ray/surface intersections. As part of ongoing research with B. B. Mandelbrot, polygon subdivision schemes for generating fractal terrain models have been developed which, while starting with a simple convex polygon such as a triangle or hexagon, evolve into terrain patches with fractal edges. This happens because the polygon subdivisions used do not "nest" neatly*, as does the common scheme of subdividing an equilateral triangle into four equilateral triangles by joining the midpoints of the edges. Because the successive stages of subdivision do not nest, the shape of the area covered by a polygon after subdivision is difficult to determine. What the shape is, depends on the subdivision scheme being used and on the number of levels to which the recursive subdivision will be performed. For these reasons, a procedural definition of our terrain models was deemed impractical; instead this new and more general algorithm was developed.

2. THE ALGORITHM

This is how the grid tracing algorithm works: A two dimensional array of altitude values is traversed in an arbitrary direction by a ray, using a modified *DDA* (*Digital Differential Analyzer*) algorithm. The array is thought of as composing a *grid* of small square *cells* (corresponding to the pixels being illuminated by a DDA algorithm). Each cell has associated with it the altitudes of the four corners of the cell. As the ray traverses the array of cells, the altitude of the ray at each cell is compared to the four altitudes associated with the cell. Ray/surface intersection need only be checked when the altitude span of the ray over the extent of the cell intersects the interval of altitude defined by the lowest and highest of the four altitudes associated with the cell. For a ray traveling **above** the surface, the condition can be stated:

$$Min(ray_{z_{near}}, ray_{z_{far}}) \le Max(h_{i,j}, h_{i,j+1}, h_{i+1,j}, h_{i+1,j+1}) \qquad (1)$$

where $ray_{z_{[near\,far]}}$ represents the altitudes of the end points of the ray segment within the cell and the $h_{m,n}$ represent the altitudes of the height field H at the four corners of the i,j^{th} cell. As the surface of the terrain within a cell can be represented with exactly two triangles (splitting the square diagonally), the ray/surface intersection test consists of two ray/triangle intersection tests. Only rays grazing past the surface will fail the intersection tests; most rays will incline directly into the surface at the first cell where surface intersection is tested.

Advantages of this algorithm are manifold. First, only the height field need be calculated and stored as the model. The actual polygon descriptions (e.g., the plane equations of the triangles) need only be calculated (and optionally, stored) when an intersection is tested for. This can save both time and space in the

* For an explanation of the "nesting" issue, see the article by Mandelbrot in Peitgen.[15]

creation of the terrain model, as polygons which are not visible in the rendering are never fully described. Second, the grid traversal can be accomplished with the use of a modified Bresenham DDA algorithm. The Bresenham algorithm is a highly optimized, fast algorithm which uses only floating point addition* in determining the height of the ray and the next cell along the path of the ray. Third, the algorithm is general. Any two dimensional array of scalar data (i.e., an image) may be interpreted as a height field and ray traced with this algorithm. Fourth, the algorithm performs ray/object intersections in $O(\sqrt{N}-1)$ time, with a very small constant multiplier. Fifth, this algorithm is nicely amenable to stepwise implementation and optimization. Thus the implementation can be tested and debugged incrementally.

3. IMPLEMENTATION

Implementation of this algorithm can be accomplished in the following stepwise fashion, where each step represents a distinct programming and verification task:

1) Calculate the x,y,z coordinates of the intersection of the ray with the global bounding box of the height field.
2) Implement a standard DDA to traverse the grid in x,y coordinates until the ray exits the bounding box.
3) Modify the DDA to identify **all** cells traversed by the ray.
4) Modify the DDA to track the z (altitude) values of the endpoints of the ray within each cell.
5) Test the z values of the ray at each cell against the altitudes at the four corners of the cell. If this test indicates that the ray passes between those altitudes at that cell, proceed with the steps below; if not, go to the next cell.
6) Create two triangles (only the plane description is needed) from the four altitude values at the corners of the cell.
7) Intersect the ray with the two triangles which tessellate the surface within the cell.
8) Include an inverse skewing and scaling transformation for the ray, so the triangles may be equilateral rather than right triangles.

These steps implement the basic algorithm; many optimizations can subsequently be made. Note that the optional step 8) turns the height field patch, in world space, from a square to a diamond shape, but does not affect the shape of the square cells or the DDA traversal.

---

* The Bresenham DDA most widely known is an integer algorithm. The version used for our purposes is not the integer Bresenham DDA, but rather a slightly less optimal floating point version. A simpler alternate scheme could use an ordinary integer DDA for the traversal, but would need to check one cell to either side of the ray path for possible intersections due to imprecision in tracking that path.

3.1. DDA Algorithm

The DDA is the heart of this algorithm. The traversal of the grid with the DDA is where most of the cpu time is spent. There are two fundamental modifications to a line-drawing DDA which must be made for the purposes of this algorithm. First, the DDA must identify *all* cells traversed by the ray; a normal DDA identifies only one cell per unit travel along the driving axis. Second, the DDA must produce floating point intersection points with cell boundaries, rather than just identifying the integer *x,y* coordinate label of a cell. Care must be taken that cells are identified in the order in which the ray passes over them in its flight from point A to point B. The second required modification makes a pure integer Bresenham DDA unusable for our purposes. The DDA we have used is an extension of the first Bresenham DDA presented in Rogers.[17]

The modifications to the DDA are extensive; a sample of the inner loop for rays of a limited range of slope across the grid is provided in Appendix A. We note that the number of comparisons to be performed at each cell (i.e., in the inner loop of the DDA) should be minimized. Naively, at each cell we must check six exit-of-bounding-box conditions for the top, bottom, and the four sides of the bounding box, and compare the four altitudes of the cell corners against the z values of the two ray endpoints at the cell boundaries. Careful construction of the DDA can reduce the number of comparisons, based on information about the path of the ray (e.g., noting that the ray is traveling in the positive x and y directions, and negative z direction can eliminate three exit tests). Such optimization will trade increased quantity of code to be written and executed, for a faster running time, as is typical of a DDA algorithm.

An optimization we have implemented involves forming a secondary $(n-1)$ by $(n-1)$ grid A:

$$[\ A_{i,j}\ \mid\ a_{i,j} = Max(h_{x,y}, h_{x+1,y}, h_{x,y+1}, h_{x+1,y+1}),\ 1 \le x,y \le n,\ 1 \le i,j \le n-1\ ]$$

where $h_{i,j}$ is the altitude of the i,j^{th} position in the height field H and n is the size of one side of the square height field. This grid A stores the maximum of the four altitude values for each cell, at the minimum x,y address of the cell. By thus precomputing the right hand side of inequality (1) for all cells, we can reduce that inequality to

$$ray_{z_{\min}} \le a_{i,j} \tag{2}$$

using *a priori* knowledge of the ray's z direction to track $ray_{z_{\min}}$ without comparisons. Simply stated, we need only check if the minimum z value of the ray is less than the maximum altitude of the cell. Note that this approach constrains our point of view to be above the surface of the patch; a similar approach with inverted *min* and *max* would suffice for rays coming from below the surface.

3.2. Intersection Tests

The ray/triangle intersection tests performed at a cell consist of:

1) Finding the line/plane intersection for the ray and triangle plane.
2) Clipping the intersection point to the extent of each triangle in the cell.

The first step is a standard operation in ray tracing; the second step is very fast. For the lower left hand triangle of the cell it can be expressed:

$$\begin{aligned}
&\text{if } (intersection_x \geq cell_{x_{\min}}) \;\; \&\& \\
&\quad (intersection_y \geq cell_{y_{\min}}) \;\; \&\& \\
&\quad (intersection_x + intersection_y \leq cell_{x_{\max}} + cell_{y_{\max}}) \\
&\qquad\quad then \;\; intersected = TRUE\,;
\end{aligned}$$

where $intersection_{[xy]}$ is the x or y coordinate of the line/plane intersection point and $cell_{[xy]_{[min\ max]}}$ is the the minimum or maximum of the x,y coordinates of the four corners of the cell. Note that the latter are integers. The first two tests check the intersection point against the extent of the (square) cell. The third test checks the intersection against the diagonal of the cell, which diagonal divides the square into two triangles.

3.3. Memory Requirements

This algorithm is memory intensive. Memory resources are the major limiting factor in its use, as opposed to computational power or time. We have rendered scenes containing nearly 3 million *virtual triangles** (a height field of dimensions 1217^2); there is simply no easy, compact way to store that amount of data. Several approaches have been pursued to limit memory use.

The height field is a vector of altitude values. As mentioned above, this vector may be quite large. Therefore, the data type used for elements of the vector has a direct, linear impact on the memory requirements of a vector of given size. In the C language on the system we used, a variable of type *double* requisitions 8 bytes, a *float* takes 4 bytes, an *int* takes 4 bytes, and a *short* takes 2 bytes. To decrease memory requirements, one should use no more bits than are necessary to encode altitudes to the required precision. Our current implementation uses the type *float* for storing the $h_{i,j}$ and $a_{i,j}$ altitude values; we do not believe that this is an optimal use of memory space, but it is easy.

The only essential data that must be stored is the height field H and, optionally, the A grid. When ray/surface intersections are tested, additional data describing the planes of the two triangles in a cell is generated. This data may or may not be stored. If all such data is stored, it may cause the host machine to run out of available memory; if it is not stored, it must be recomputed at each test. Recomputing this data at each test is undesirable, as some triangles (particularly those in the foreground when the eye is close to the surface) may be intersected by many rays, thus indicating storage of the triangle plane data.

Our initial implementation of the *plane* data structure uses four *doubles* or 32 bytes (clearly not optimal). With approximately 3 million triangles, this implementation can exceed the memory capacity of the Encore Multimax machine, which has 64MB (megabytes) main memory and 115MB total virtual memory; this indicates discarding the plane data rather than storing it. Our solution to this

* We refer to the triangles composing the tessellation of the surface as *virtual triangles* because they do not exist as entities, until they are needed for ray/surface intersection tests.

problem has been to implement a circular queue of *plane* structures. Thus we store only the number of *plane* structures in this queue (two to four times the length of a scanline in our implementations). When a new *plane* structure is created it uses a pointer to a *plane* structure in this circular queue, and the pointer to the *plane* structure which previously occupied that position in the queue is set to *NULL*. Another approach, for ray *casting*, is to store just two temporary *plane* structures with the x,y coordinates of the cell in which they reside. By checking the x,y coordinates before recalculating the plane data, coherence of a triangle on a given scanline can be used to eliminate repeated calculations; recalculation is only necessary for each scanline on which the triangle is tested for intersection.

The efficacy of these caching schemes will be strongly affected by two factors: the point of view or eye point relative to the grid, and reflection, refraction, or shadow ray propagation. If the eye point is everywhere distant from the surface, all triangles may be pixel- to sub-pixel size and will therefore not be intersected by more than a few rays. Storing only one cell's plane data will often be futile if shadow, reflected, or refracted rays are propagated. Thus the hit rate of either caching scheme can vary widely; other caching schemes may prove more efficient.

3.4. C-Linda Implementation

The initial implementation of the grid tracing algorithm was an extension of the Optik[2] ray tracer of the University of Toronto's Dynamic Graphics Project. The algorithm was developed under the C-Linda[1] programming language on an Encore Multimax parallel computer. The Linda language (a minimal extension of the C language) allows dynamic allocation of the 18 National Semiconductor 32332 CPUs of our Encore Multimax. The Multimax is a MIMD parallel machine; each processor can concurrently execute one process, or Linda "task".

As it is difficult at best to take advantage of spatial coherence in ray tracing algorithms, we simply divide the job of rendering an image into subtasks of rendering single scanlines (as opposed to rendering some more compact area, e.g., a square). This scanline approach has the advantage of easy checkpointing: as each processor completes a scanline, the "supervisor" process collects that data and writes it to nonvolatile storage (a file) as soon as all previous scanlines are available and have been collected and written to storage. Thus if the fifth scanline is the first to be completed, it is cached until the preceding four scanlines are completed; then all five are written consecutively to the image file. If the process dies when the image is partially completed, all that is lost is those scanlines upon which work has been started, but which have not been written to nonvolatile storage.

Although the incipient scanline tasks are stored in a theoretically unordered "tuple space" in the Linda system, the implementation of this space is of course an ordered list, and this virtual ordering in practice can be exploited to assure that the tasks will be executed by the "worker" processes in roughly the same order as the task "tuples" were output into the tuple space by the supervisor process. Thus the task tuples describing the entire image (e.g., 1024 scanline tasks) can be output to tuple space before rendering is begun, and the image can still be expected to rendered in an orderly fashion, such as top-to-bottom. Again, this facilitates checkpointing of the program. An alternative scheme is to assign every k^{th}

scanline to processor or Linda worker k. This scheme risks a single processor falling behind and thereby increasing temporary storage required for scanlines by the supervisor process, and puts the stored scanlines at risk of loss in case of catastrophic failure of the rendering program.

The Encore Multimax supports shared memory for the 18 processors. This facilitates the rendering of large height fields, because the height field data can be stored in shared memory rather than being duplicated many times. The shared memory feature also makes the combined real memory for all 18 processors available to all processors and tasks. This is important when rendering a height field of several megabytes in size as it allows the entire height field to be stored in main memory, thus avoiding page faults which can severely degrade performance of the algorithm.

A concern in the use of shared memory is contention among the various processors for memory access cycles to the data stored in shared in shared memory. We have not carefully analyzed the memory contention situation, but we have seen no evidence that this has been a significant impediment to the efficiency of the algorithm. The fact that a large proportion of the rendering time for our images is spent on independent work such as evaluation of procedural textures may help to alleviate potential memory contention problems, but subjective evaluation of simple tests of rendering without such textures still indicates no significant contention problem.

Overall, our assessment is that the Linda/Multimax combination presents an optimal environment for the ray tracing of very large model descriptions.

4. TIME COMPLEXITY

The worst case time complexity of this algorithm is $O(\sqrt{N}-1)$ where N is the number of height values in the field. Note that $N=n^2$, where n is the size of a side of a square height field grid. The time is linear in the number of rays cast at the virtual screen. Each ray will traverse the grid with a number of operations proportional to, in the worse case, $2(\sqrt{N}-1)$ for a ray crossing the grid diagonally without intersecting the surface. The average case is closer to $0.5\,C\,(\sqrt{N}-1)$ operations, where C is the worst case constant number of operations per cell traversed, and the 0.5 scalar is due to the expectation that the ray will intersect the surface, on average, halfway across the grid.

The number of virtual triangles t in a height field is

$$t \;=\; 2(N-2\sqrt{N}+1) \;=\; 2(n-1)^2,$$

as the number of cells c in the grid is

$$c \;=\; N-2\sqrt{N}+1 \;=\; (n-1)^2$$

and there are two virtual triangles per cell. The DDA algorithm insures that the first intersection found will be the closest, therefore the search for ray/object intersections can be terminated at the first intersection. As most rays will intersect the surface in the first cell for which the line/plane intersection test is performed, the average number of ray/object intersection tests for any N, t or c is approximately two. The worst case is $4(\sqrt{N}-1)$ for a ray that skims through a trough that runs diagonally across the grid (quite unlikely for a stochastic surface).

The low number of ray/object intersection tests is the real advantage of this algorithm. Naive ray tracing algorithms have a time complexity proportional to the number of primitive objects in the scene; most of their time is spent in the relatively expensive ray/object intersection tests. In this algorithm the time spent calculating ray/object intersections is small and constant,* and the time complexity is dominated by the relatively inexpensive search for candidate objects to be intersected with the rays. Note that the preprocessing overhead of this algorithm is small as well. The height field is calculated just once (as opposed to procedural algorithms). The spatial coherence of the height field obviates potentially costly preprocessing required by some spatial subdivision algorithms, for sorting primitives and assigning them to the appropriate cells in a grid or hierarchy.

It is the inner loop of the DDA traversing the grid which dominates the time complexity of this algorithm. As this inner loop involves a small number of inexpensive operations, we claim that the constant scalar C to the $O(\sqrt{N}-1)$ time complexity is small and that the algorithm is quite efficient. In our use of this algorithm to ray trace fractal mountain patches, the execution time of our renderer is dominated by the evaluation of the procedural Perlin solid textures[16] used to simulate water, treelines and snowlines, and clouds. Thus a typical 1024 by 1280 resolution image of a fractal patch described by a height field of size 407^2 (329,672 virtual triangles) ray traced at 1 ray/pixel may take 24 hours of cpu time on a single National Semiconductor 32332 microprocessor with fast floating point support. Of this time, only about 8 hours are spent in ray tracing and determining lighting for the surface; most of the balance is spent evaluating the procedural textures at ray/surface intersection points. Generation of the height field by polygon subdivision typically takes only a few minutes on a machine such as a Sun 3/60, and while it may take some tens of minutes to generate a large height field, the time is still considerably less than the time required for rendering.

5. FUTURE DIRECTIONS

Some foreseeable applications of the grid tracing technique are the interpretation of procedural textures (such as Perlin's "marble") as height fields for novel surface geometries, the rendering of fourier synthesized height fields,[13,21] and the decomposition of complex three dimensional objects into two dimensional patches to be rendered as height fields. The first two applications may lead to many novel surfaces through the precise control of spatial frequencies available in fourier and procedural textures. The last might prove useful for rendering a limited class of complexly textured, closed three dimensional objects (perhaps a rough, sandblasted wooden teapot, repleat with bumpy profile?)

Problems with the grid tracing technique include spatial frequency clamping. Distant portions of the height field rendered with a perspective projection will have arbitrarily high spatial frequencies, leading to aliasing. This aliasing will be particularly objectionable in animated sequences where it will cause scintillation on distant surfaces. As yet unexplored solutions to this problem include adaptive refinement of the grid and low-pass filtering of the height values as a function of distance from the view point.

---

* This is what Kaplan[10] calls "constant time" for his spatial subdivision scheme.

The grid tracing algorithm is amenable to many fine optimizations, mostly in the DDA. Other more significant global optimizations are possible. For instance, Mastin and Watterburg[13] have suggested a quad tree approach to ray tracing height fields which is reminiscent of Kajiya's procedural algorithm[9] for stochastic surfaces. In Mastin's approach, the ray altitude is checked against a quadtree of bounding boxes of decreasing volume which enclose a rectilinear set of subdivisions of the surface (i.e., a square grid is subdivided into four smaller square grids, recursively). The data structure required to store such a tree would be about 35 percent larger than that storing the simple height field; this would be prohibitive for the sizes of height fields which we render with grid tracing. However, a small set of bounding boxes, whether in a quadtree structure or not, could speed up the grid tracing process considerably by breaking the bounding box for the the entire height field into several smaller, tighter bounding boxes. These smaller bounding boxes could be incorporated into a grid or quadtree, and could each contain a grid. By creating a hierarchy of grids or a quadtree/grid hierarchy, the advantages of a hierarchical data structure can be combined with the efficiency of the DDA grid traversal. Such a scheme would be very similar to that required for an adaptive level-of-detail solution to the aliasing problem in distant sections of the grid, in which the resolution of the grid in distant areas would be lower than that in areas close to the view point. Note that by making the grids smaller through a hierarchy, page faults can be decreased. Also, the time complexity of the overall rendering algorithm may be affected favorably. If, for example, the height field were placed in a hierarchy of 16 by 16 grids, the time complexity would go to $O(\sqrt{\log_{16} N})$.

This algorithm could be implemented as an integer algorithm, with some work. For example, the bounding box of the entire grid could be scaled in z (height) to the full range of the integers, thereby facilitating tracking of the ray altitude with an integer DDA. As mentioned above, the traversal of the grid can be accomplished with an ordinary integer DDA, with the added expense of extra ray/surface intersection checks along the path of the ray. These extra checks are necessary because of imprecision in the calculation of the ray path with an integer calculation.

The surface of the height field could be represented with a spline, rather than a triangular tessellation. This would require some analysis of the local behavior of the spline, as the altitude of the surface within a cell might pass above that of the highest altitude at the corners of the cell. The behavior of the surface would be less well defined, and therefore ray/surface intersections would be more challenging and costly to compute.

6. CONCLUSIONS

Grid tracing is a very efficient technique for ray tracing a limited class of complex tessellated objects. This algorithm has been used to successfully render approximately 3 million unique triangles in a single scene. Such a scene can easily have all polygons appear at pixel size or below, thereby eliminating the flat polygonal surfaces from the image of a tessellated surface. The major limitation to the use of this algorithm is the real memory capacity of the host computer (e.g., page faults); this can be ameliorated by implementing a hierarchical data structure of smaller grids. The grid tracing technique is general to all regular height fields, and calls for creative applications.

ACKNOWLEDGEMENTS

This work was accomplished through the generous support of Benoit Mandelbrot as a part of our ongoing quest for more beautiful and realistic forgeries of nature, using the latest results in fractal geometry.

The work was funded, in part, through the Office of Naval Research contract N00014-88-K-0217.

Development was carried out with the invaluable assistance of Nick Carriero, Rob Bjornson, and David Gelerenter, using C-Linda[1] on the Encore Multimax. That system is supported, in part, by NSF grants DCR-8601920 and DCR-8657615.

APPENDIX A: Code Segment of DDA Inner Loop

```
/* traverse the 2D grid of surface height data */
/* uses a modified non-integer Bresenham DDA */

        register double     nearZ, farZ, minZ, error, delta, deltaX, deltaY, deltaZ;
        register int        xCell, yCell;
        double      signX, signY;


        /* set initial position */
        nearZ = intersection_point.z;
        farZ = (/* z value of ray at far end of cell */);
        xCell = (/* x index of cell at ray intersection with bounding box */);

        yCell = (/* y index of cell at ray intersection with bounding box */);
        /* set DDA variables */
        error = (/* an octant-specific initialization dependent on
                   exact (floating point) intersection point with cell */);
        delta = (/* delta for error */);
        deltaX = (/* delta for x */);
        deltaY = (/* delta for y */);

        deltaZ = (/* delta for z */);
        /* SIGN() returns +1 for a positive argument; -1 otherwise */
        signX = SIGN(ray->dir.x);
        signY = SIGN(ray->dir.y);

        /* inner loop: while still in the grid, traverse. driving axis of DDA is x axis */
        do      {
                minZ = MIN(nearZ,farZ);
                if ( minZ <= HighestAlt[xCell][yCell] )
                        if ( Intersection(xCell, yCell, ray) )
                                return;
                if ( error > VERY_SMALL ) {
                        /* check the cell that a normal DDA would skip */
                        yCell += signY;
                        if ( (yCell < 0) || (yCell == sideMax) )
                                /* bounding box has been exceeded */
                                break;
                        else
                                if ( minZ <= HighestAlt[xCell][yCell] )
                                        if ( Intersection(xCell, yCell, ray)
                                                return;
                        error--;
                        }
                else if ( error > -VERY_SMALL ) {
                        /* ray crosses at exactly the corner of the cell (unusual) */
                        yCell += signY;
                        error--;
                        }
                xCell += signX;
                error += delta;
                nearZ = farZ;
                farZ += deltaZ;
                } while ( (nearZ >= boxBottom) && (nearZ <= boxTop) &&
                        (xCell >= 0) && (xCell < sideMax) &&
                        (yCell >= 0) && (yCell < sideMax) );
```

References

1. Ahuja, S., N. Carriero, and D. Gelerenter, ''Linda and Friends,'' *IEEE Computer*, August, 1986.
2. Amanatides, J., K. Musgrave, R. Skinner, L. Slipp, and A. Woo, *Optik Users' Manual, Version 0.9.5,* Yale University, March, 1988.
3. Arvo, J. and D. Kirk, ''Fast Ray Tracing by Ray Classification,'' *Computer Graphics*, vol. 21, no. 4, pp. 55-64, July, 1987.
4. Bouville, C., ''Bounding Ellipsoids for Ray-Fractal Intersection,'' *Computer Graphics*, vol. 19, no. 3, pp. 45-52, July 1985.
5. Fournier, A., D. Fussel, and L. Carpenter, ''Computer Rendering of Stochasic Models,'' *Communications of the ACM*, vol. 25, no. 6, pp. 371-384, June, 1982.
6. Fournier, A. and T. Milligan, ''Frame Buffer Algorithms for Stochastic Models,'' *Computer Graphics and Applications*, vol. 5, no. 10, October, 1985.
7. Fujimoto, A., T. Tanaka, and K. Iwata, ''ARTS: Accelerated Ray Tracing System,'' *IEEE Computer Graphics and Applications*, vol. 6, no. 4, pp. 16-26, April, 1986.
8. Glassner, A. S., ''Space Subdivision for Fast Ray Tracing,'' *IEEE Computer Graphics and Applications*, vol. 4, no. 10, pp. 15-22, October, 1984.
9. Kajiya, J. T., ''New Techniques for Ray Tracing Procedurally Defined Objects,'' *Transactions on Graphics*, vol. 2, no. 3, pp. 161-181, July, 1983.
10. Kaplan, M. R., ''Space-Tracing, a Constant Time Ray-Tracer,'' *SIGGRAPH85 Tutorial on the Uses of Spatial Coherence Ray Tracing*, 1985.
11. Kay, T. L. and J. Kajiya, ''Ray Tracing Complex Scenes,'' *Computer Graphics*, vol. 20:4, pp. 269-278, August, 1986.
12. Mandelbrot, B. B., *The Fractal Geometry of Nature,* W. H. Freeman, New York, 1983.
13. Mastin, G. A., P. A. Watterberg, and J. F. Mareda, ''Fourier Synthesis of Ocean Scenes,'' *IEEE Computer Graphics and Applications*, vol. 3, pp. 10-23, March, 1987.
14. Miller, G. S. P., ''The Definition and Rendering of Terrain Maps,'' *Computer Graphics*, vol. 20, no. 4, pp. 39-48, August, 1986.
15. Peitgen, H. O. Ed., *The Science of Fractal Images,* Springer, New York, 1988.
16. Perlin, K., ''An Image Synthesizer,'' *Computer Graphics*, vol. 19, no. 3, pp. 287-296, July, 1985.
17. Rogers, D. F., *Procedural Elements for Computer Graphics,* pp. 34-38, McGraw Hill, New York, 1985.
18. Rubin, S. and T. Whitted, ''A Three Dimensional Representation for Fast Rendering of Complex Scenes,'' *Computer Graphics*, vol. 14, no. 3, pp. 110-116, July, 1980.
19. Snyder, J. M. and A. H. Barr, ''Ray Tracing Complex Models Containing Surface Tessellations,'' *Computer Graphics*, vol. 21:4, pp. 119-128, July, 1987.

20. Sweeny, M. A. J., ‘‘The Waterloo CGL Ray Tracing Package,’’ *Masters Thesis*, University of Waterloo, Waterloo, Ontario, 1984.
21. Voss, R. F., ‘‘Fourier Synthesis of Gaussian Fractals: 1f Noises, Landscapes, and Flakes,’’ *State of the Art in Image Synthesis Tutorial Notes*, vol. 10, SIGGRAPH, 1983.
22. Weghorst, H., G. Hooper, and D. Greenberg, ‘‘Improved Computational Methods for Ray Tracing,’’ *ACM Transactions on Graphics*, vol. 3:1, pp. 52-69, January, 1984.
23. Whitted, T., ‘‘An Improved Illumination Model for Shaded Display,’’ *Communications of the ACM*, vol. 23, no. 6, pp. 343-349, June, 1980.

Table of Contents