The HERO Algorithm for Ray-Tracing Octrees

Mark Agate, Richard L. Grimsdale and Paul F. Lister

VLSI & Graphics Research Group
School of Engineering, University of Sussex
Brighton BN1 9QT, UK

Abstract

An algorithm is presented for rapid traversal of octree data structures, in order to enhance the speed of ray tracing for scenes of high complexity. At each level of the octree, the algorithm generates the addresses of child voxels in the order they are penetrated by the ray. This requires only a few arithmetic operations and simple logical operations. A depth-first search of the tree is used to yield the first terminal voxel hit by the ray, thus hidden objects are not processed. The algorithm is designed specifically for implementation as HERO: A Hardware Enhancer for Ray-tracing Octrees.

1. Introduction

Computation of ray/object intersections is widely recognised as being the major bottleneck in ray tracing. Even when object descriptions are made as simple as possible, ray tracing a scene of even moderate complexity can be an extremely slow operation, due to the nesting of two large loops (one for all rays, the other for all objects). Plunkett and Bailey [1] have applied array processing techniques to enhance the performance of a ray tracing system, but even with the power of a supercomputer, the performance falls short of real-time. Attempts to speed up ray tracing significantly must therefore be based on algorithmic developments, exploiting either image coherence to reduce the complexity of the outer loop, or object space coherence to reduce the number of intersections required in the inner loop.

To exploit image coherence, it is necessary to use a beam tracing approach so that a whole area of the image can be processed at a time. This is done by Heckbert and Hanrahan [2], who cast a single rectangular pyramid for the entire screen, recursively subdividing this into smaller pyramidal beams wherever the image is complex. Whilst this can be expected to yield a considerable speedup over conventional ray tracing if large areas of the image are homogeneous, a characteristic of ray traced images is the lack of image coherence which results from rendering of surface characteristics and illumination effects. Pyramidal beam intersections, though advantageous for antialiasing, are considerably more complex to compute than point sampling with rays, and the extra complexity of recursive beam subdivision makes this approach undesirable for direct implementation in hardware.

Object coherence is partly exploited by Whitted [3], who uses bounding volumes to reduce the complexity of ray/object intersection calculations, though the size of the inner loop is still heavily dependent on the number of objects, since every spherical bounding volume must still be processed for every ray. Kay and Kajiya [4] use hierarchical object bounding, whereby objects made up of several smaller objects are bounded by parallelopipeds which are stored in a tree structure. A ray which fails to penetrate the outermost bounding volume of an object need not be intersected with any volumes or objects inside it, and the complexity of the inner loop is thereby reduced. A similar approach is used by Dippe and Swensen [5], though the shapes of the bounding volumes themselves are varied, resulting in more complex ray/volume intersection calculations.

In a similar way to that in which Bresenham's algorithm generates the locations of 2-D pixels which are intersected by a line, Amanatides and Woo [6] have developed an incremental algorithm which returns the 3-D voxels hit by a ray. As well as eliminating ray intersection calculations with objects which are far from the line of the ray, an incremental voxel tracer can terminate as soon as the first intersection with an object is found, without having to intersect the ray with other objects and subsequently find the nearest. Occluded objects therefore remain unprocessed.

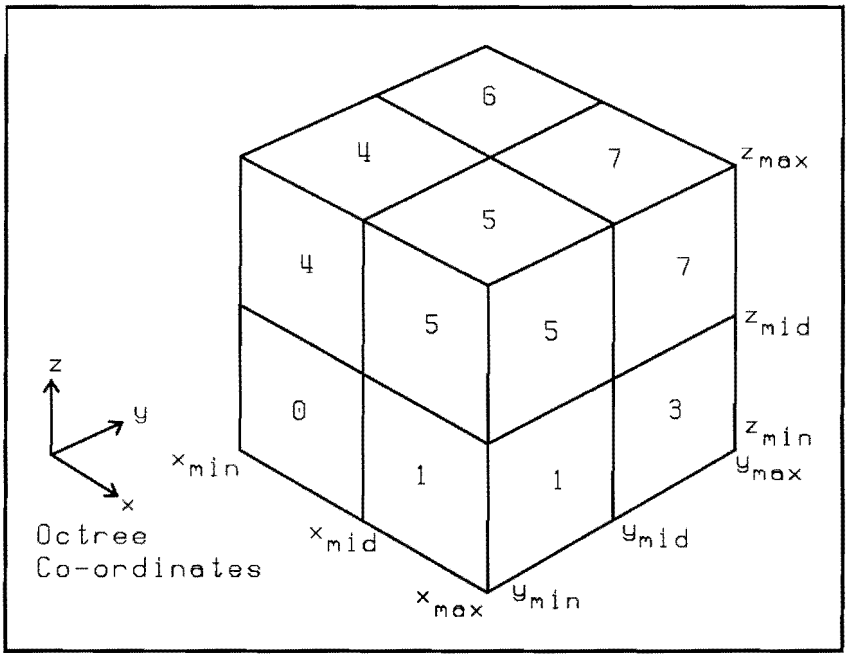

Figure 1 Octree Definition

Further economy in ray/object intersection calculations can be afforded if it is the object space, rather than the object descriptions, which is subdivided hierarchically. The most common technique for space subdivision is the generation of an octree, as illustrated in fig. 1. Voxels are stored in a tree structure, each node having between zero and eight children. A voxel containing only a simple object or surface has no children,

but if its contents are more complex than this, it is subdivided into eight child voxels as shown. Any child voxels which are empty can be eliminated, those which are simple form leaf nodes, and others are subdivided further. Note that the voxels in an octree may be of many different sizes. When such a structure is ray traced, voxel contents can be searched in the order that the voxels are hit by the ray, thus leading straight to the first object hit. An additional bonus with octrees is that large areas of empty space are rapidly traversed by the ray.

Glassner [7] uses a scheme in which the child voxels of a node are numbered from 1 to 8. The ray is traced through the octree by moving along the line of the ray a certain distance, then determining which voxel the resulting point is inside. Although a considerable speedup is obtained over conventional ray tracing, this scheme is not ideal for hardware implementation, due to the voxel numbering system adopted. It is shown in the next section that simple logic operations may be employed for voxel address generation if an octal system is used instead, with nodes labelled from 0 to 7.

Fujimoto et al [8] trace rays through voxels using two incremental line-drawing type processors in a similar way to Amanatides and Woo, but within an octree data structure. It is observed that the image rendering time is "Virtually independent of the number of objects in the scene", and that for very large numbers of objects, this technique of ray tracing actually becomes faster than projective methods. Peng et al [9] use an approach to voxel traversal which appears to be essentially heuristic, in that the ray is assumed to pass to the neighbouring voxel in the direction of the maximum ray direction co-ordinate; if this is not found to be the case, the second most likely neighbour is tested, and if this fails, the least likely neighbour is hit. All these techniques undoubtedly provide considerable speed enhancements. However, the algorithm presented here removes the need to test a ray against a voxel, since the addresses of the child nodes are generated in the order they are hit. This entails only a few arithmetic operations for each parent voxel, and simple logic operations for address generation. These operations are ideally suited to hardware implementation, thus HERO (a Hardware Enhancer for Ray-tracing Octrees) has been conceived.

2. Development of the HERO Algorithm

An efficient way to represent a convex object with planar faces is to define it as the intersection of half-spaces. (A half-space is the volume of space to one side of an infinite plane). The cube in fig. 2 is defined by six bounding planes, each with a unit normal vector $\underline{c}_i$ pointing away from the interior of the object, and position constant d_i, so that all points $\underline{P}$ on a particular bounding plane satisfy the equation:

$$\underline{c}_i \cdot \underline{P} + d_i = 0$$

Thus each face of a solid modelled in this way is geometrically defined by just four numeric values.

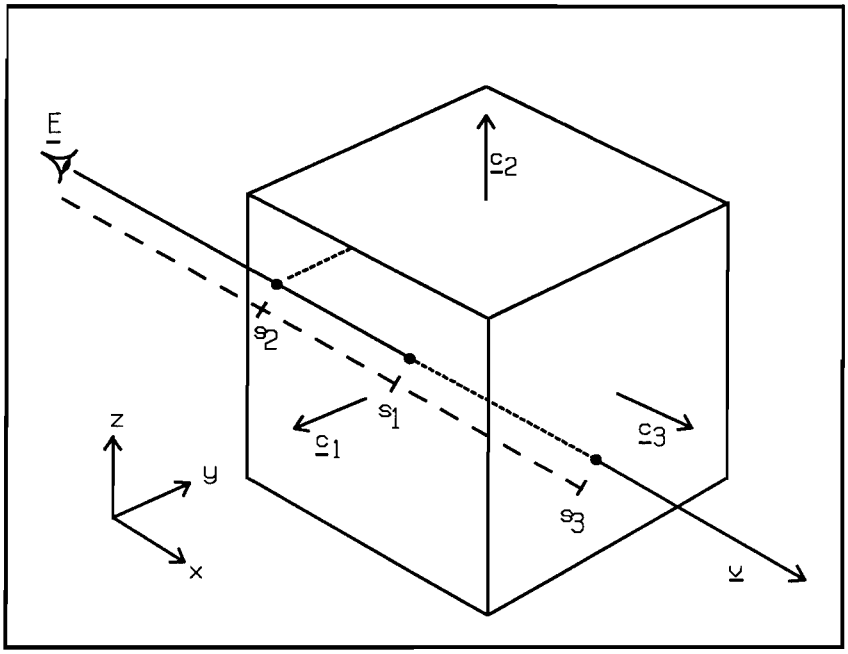

Figure 2 Cube Defined by Half-Spaces

All points $\underline{P}$ along the line of a ray can be defined by the equation:

$$\underline{P} = \underline{E} + s * \underline{v}$$

where $\underline{E}$ is the ray origin position, $\underline{v}$ is the direction vector and s is a positive scalar variable representing the distance along the ray. The ray hits an object defined in the above way only if there exists a range of points on the line of the ray which are simultaneously to the inside of all the bounding planes for that object. For any one bounding plane, the distance parameter s_i to the point at which the ray hits the plane ($\underline{c}_i .\underline{P} + d_i = 0$) can be found from:

$$s_i = -(\underline{c}_i .\underline{E} + d_i)/(\underline{c}_i .\underline{v})$$

If the angle between the ray and the surface normal is obtuse, points on the ray at distances greater than s_i from the ray origin will be inside the half-space defined by the bounding plane. Conversely, if this angle is acute, only points with $s < s_i$ will be inside the half-space. Thus for each bounding plane, a positive value of the dot product $\underline{c}_i .\underline{v}$ indicates that s_i is an upper limit on the required range of distances, while a negative value indicates that s_i is a lower limit. To determine if the ray hits the object, all s_i values are computed and categorised as upper or lower limits. The maximum lower limit ($s_{l\,max}$) and the minimum upper limit (s_{umin}) are then found; if $s_{l\,max} < s_{umin}$, there exists a range of distances for which points on the ray are internal to all the half-spaces. If both limits are positive, then a forward-travelling ray hits the object at the point:

$$\underline{P} = \underline{E} + s_{l\,max} * \underline{v}$$

In the example shown in fig. 2, $s_{l\,max}$ is equivalent to s_1, and s_{umin} is s_3. From $s_1 < s_3$,

it may be deduced that the ray penetrates the cube, and does so at a distance s_1 from the ray origin.

It is important to consider the effects of numeric overflow in the calculation of distance values; this can occur if the ray is parallel or nearly parallel to the bounding plane under consideration, causing the value of $\underline{c}_i \cdot \underline{v}$ to become very small. In such cases, even if the value of s_i should have a magnitude of infinity, it is sufficient to represent this by the maximum numeric magnitude allowed, as long as the sign of the result is correctly evaluated. Fig. 3 demonstrates two such situations. In both cases, $\underline{c}_i \cdot \underline{v}$ is zero, since the rays lie parallel to the bounding plane. Assuming that this zero value is represented as positive, both values of s_i will be interpreted as upper limits on s. In the first case, the value of $\underline{c}_i \cdot \underline{E}_1 + d_i$ is positive (since $\underline{c}_i$ points towards $\underline{E}_1$), hence the computed value for s_i will be minus infinity. Since this is an upper limit on s, it is clear that the ray does not penetrate the half-space defined by the bounding plane along its entire length. In the second case, $\underline{c}_i \cdot \underline{E}_2 + d_i$ is negative (as $\underline{c}_i$ points away from $\underline{E}_2$), thus a value of plus infinity is returned for s_i, indicating that the ray is inside the defined half-space along its entire length. Conversely, if $\underline{c}_i \cdot \underline{v}$ were in fact a small negative value, the sign of s_i would be inverted in each of the above cases, but these would be interpreted as lower limits on s, hence the system is well-behaved.

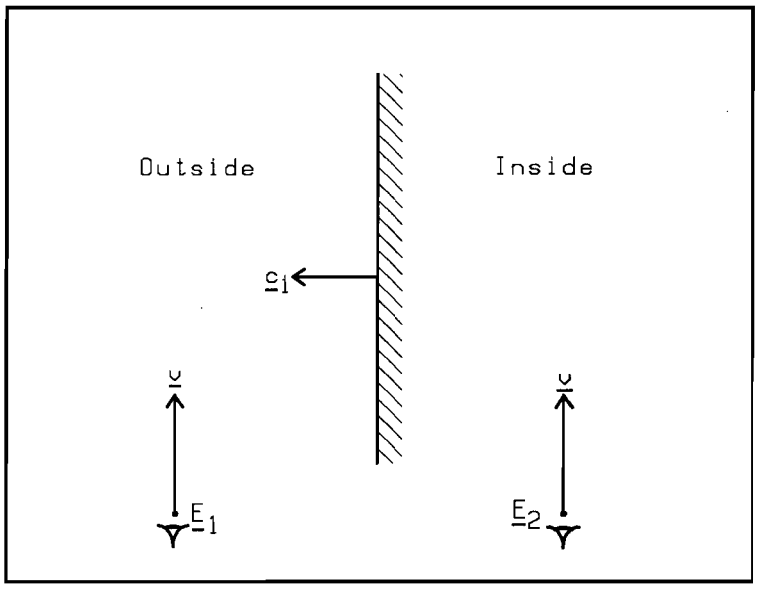

Figure 3 Rays Parallel to Bounding Planes

The root node of an octree may be defined in the same way as the cube of fig. 2. If the octree and ray are both in a co-ordinate system which has axes parallel to the edges of the octree root node, as in fig. 1, then all unit normal vectors $\underline{c}_i$ will have two co-ordinates equal to 0 and one equal to plus or minus unity. The dot product operation in

computation of s distances can therefore be reduced to a simple co-ordinate selection and possible sign inversion.

The bounding planes for each node of the octree can be labelled x_{min}, x_{max}, y_{min}, y_{max}, z_{min} and z_{max}, as in fig. 1. The planes $x = x_{mid}$, $y = y_{mid}$ and $z = z_{mid}$ define the new bounding planes for the child voxels of the parent node. Each child voxel has a numeric identifier between 0 and 7 as shown, which can be represented by a 3-bit binary word. If the three bits are considered to correspond to the co-ordinates z, y, x in order, then a '0' digit in the child's identifier indicates that it shares the parent's min boundary for that co-ordinate, while a '1' indicates that it shares its parent's max. For example, child node 3 (mask = 011) shares its parent's z_{min}, y_{max} and x_{max} boundaries. The child node's z_{max} value will be its parent's z_{mid}, its y_{min} will be its parent's y_{mid}, and its x_{min} is its parent's x_{mid}.

If a ray is found to penetrate the root node of the octree, it is subsequently necessary to determine which child voxels are hit by the ray. A simplistic approach would be to test each of the non-empty child voxels for penetration by the ray using the half-space method, then sort them by their $s_{l\,max}$ values to determine the order in which they are hit. It should be noted that the ray need only be intersected with the parent's x_{mid}, y_{mid} and z_{mid} planes once; the 3-bit child node identifiers can then be used to determine which distance values apply to each child.

A further simplification may be afforded by noting that the order in which child nodes are hit can be found by testing the children in the correct order. The order in which they should be tested depends solely on the octant of space into which the ray points. For example, if the x_v, y_v and z_v components of the ray direction vector $\underline{v}$ are all positive, then the child nodes can be tested in the order 01234567. If (say) nodes 5, 7 and 1 are found to be hit, these must be penetrated in the order 1, 5, 7. As another example, suppose x_v and y_v are positive, but z_v is negative; the order in which child nodes should be tested is then 45670123. A technique for generation of these orderings from the ray direction vector is as follows.

For the ray direction vector $\underline{v}$, generate a 3-bit 'VMASK', with individual bits corresponding to $\text{sign}(z_v)$, $\text{sign}(y_v)$ and $\text{sign}(x_v)$; '0' for positive values and '1' for negative. The order in which child nodes should be tested can be found by running a counter from 0 to 7, and returning the value of COUNTER XOR VMASK. For example, a ray with negative x_v but positive y_v and z_v would have VMASK = 001, giving rise to the child node ordering 10325476.

The next stage in simplifying the algorithm is to notice that, for any parent node intersected by the ray, between 1 and 4 of its child nodes will be intersected. Additionally, only certain combinations of child voxels can be hit. For the example shown in fig. 4, with VMASK = 000, once it is found that child node 1 is intersected, it is known that nodes 0, 2, 4 and 6 cannot be hit. Logical derivation of the order of nodes hit can in fact be achieved in the following way.

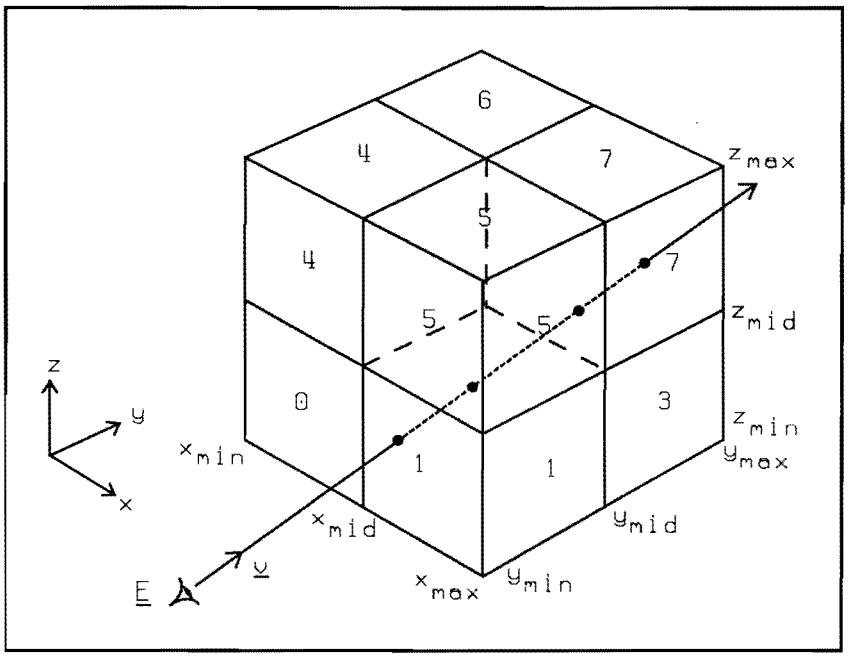

Figure 4 HERO Example 1

For the simple case VMASK = 000, let the identifier of the first child hit be CHILDMASK. The individual bits of CHILDMASK are set according to the following conditions ('0' for false, '1' for true):

$$\text{CHILDMASK} = s_{zmid} < s_{lmax} \;,\; s_{ymid} < s_{lmax} \;,\; s_{xmid} < s_{lmax}$$

where all s values are computed for the parent node. It can be seen that the identifiers of subsequent children hit (if any) can be found by setting the as yet unset bits of CHILDMASK. For example, if node 1 is the first child hit (CHILDMASK = 001), as in fig. 4, then the only other children which can be intersected have CHILDMASK equal to 011, 101 or 111 (i.e. nodes 3, 5 or 7). Similarly if node 5 were the first child hit, only node 7 could subsequently be hit. The order in which these bits should be set is found as follows:

1. For the parent node, compute the distances along the ray to the mid-plane intersections, i.e. s_{xmid}, s_{ymid} and s_{zmid}.

2. Sort these s values into ascending order. Enter the corresponding masks - 001 for s_{xmid}, 010 for s_{ymid} and 100 for s_{zmid} - into a list which indicates the ascending order of s values. In the example shown in fig. 4, where $s_{xmid} < s_{zmid} < s_{ymid}$, this list will be as follows:

MASKLIST [1] = 001
MASKLIST [2] = 100
MASKLIST [3] = 010

3. Take each element of MASKLIST in turn, and if the corresponding bit of CHILDMASK is not already set, that element should be ORed into CHILDMASK to produce the identifier of the next child hit. Thus for fig. 4, the initial value of CHILDMASK is 001, and subsequent identifiers 101 and 111 are produced, by ORing in elements 2 and 3 of MASKLIST, respectively.

Two further considerations are necessary for generalisation of the algorithm. The first is that the generation of new child identifiers should stop when the last child voxel is hit. In a similar way to that in which CHILDMASK is generated, the mask for the last child hit can be generated by:

$$\text{LASTMASK} = s_{zmid} < s_{umin} \,,\; s_{ymid} < s_{umin} \,,\; s_{xmid} < s_{umin}$$

The termination condition can therefore be easily set. Secondly, the analysis so far has concentrated on child voxels hit by a ray in the octant with VMASK = 000. For rays in other octants, each new CHILDMASK generated should be XORed with VMASK to produce the actual identifier of the next child hit. In effect, CHILDMASK replaces the counter which was used to determine the order in which child voxels should be tested. The XOR operation implements a transformation of CHILDMASK into the correct octant of space, according to the direction of the ray.

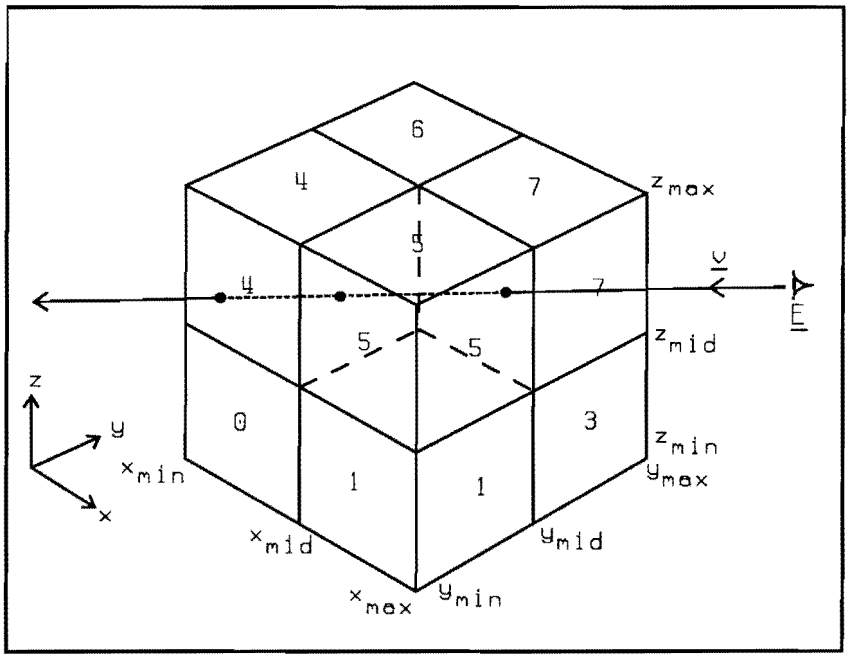

Figure 5 HERO Example 2

A second example, shown in fig. 5, has VMASK equal to 111. The distance s_{1max} is equivalent to s_{xmax}, and s_{umin} is equivalent to s_{ymin}. The value of CHILDMASK is initially 010, and LASTMASK is 011. Since $s_{ymid} < s_{xmid} < s_{zmid}$, MASKLIST is as follows:

MASKLIST[1] = 010
MASKLIST[2] = 001
MASKLIST[3] = 100

The first child hit therefore has an identifier equal to the initial value of CHILDMASK XOR VMASK, i.e. 010 XOR 111, which is 5. A logical OR of MASKLIST[1] into CHILDMASK does not create a new value, but when MASKLIST[2] is ORed in, CHILDMASK becomes 011, which is equal to LASTMASK. Hence the second and final child hit has identifier 011 XOR 111, giving the correct value of 4.

3. Summary of the Algorithm

For the ray $\underline{E} + s * \underline{v}$, the following 3-bit mask is generated:

$$VMASK := Sign(z_v),\ Sign(y_v),\ Sign(x_v)$$

For the root node of the octree, the distance parameter s for each of the six bounding planes is found:

$$s_{xmin} := (x_{min} - x_E)/x_v \ ; \text{ etc.}$$

Each s value is categorised as either an upper limit or lower limit on the range of s for points inside the root node. A '0' in the x-bit position of VMASK implies that s_{xmin} is a lower limit and s_{xmax} an upper limit, while a '1' implies the opposite. The s values for the y and z planes are similarly categorised. The maximum of the lower limits (s_{lmax}) and the minimum of the upper limits (s_{umin}) are found. If $s_{lmax} < s_{umin}$, the root node is penetrated by the ray, otherwise it is not.

For any parent node intersected by the ray, the distance values for intersection of the ray with each of the node's mid-planes are calculated, i.e. s_{xmid}, s_{ymid} and s_{zmid}. These are sorted into ascending order, and the masks 001 for s_{xmid}, 010 for s_{ymid} and 100 for s_{zmid} are entered into MASKLIST[1..3] in ascending order of their corresponding s values. Two further masks are also generated:-

$$CHILDMASK := (s_{zmid} < s_{lmax}),\ (s_{ymid} < s_{lmax}),\ (s_{xmid} < s_{lmax})\ ;$$
$$LASTMASK := (s_{zmid} < s_{umin}),\ (s_{ymid} < s_{umin}),\ (s_{xmid} < s_{umin})\ ;$$

The following operations then generate the identifiers of the child nodes in the order they are hit:

```
i := 1 (* index into MASKLIST *)
LOOP
   SearchChild(CHILDMASK XOR VMASK);
      (* check next intersected child *)
   IF CHILDMASK = LASTMASK THEN EXIT;
      (* all intersected children searched *)
   ELSE
      WHILE (MASKLIST[i] AND CHILDMASK) <> 0 DO
         (* find next valid masklist element *)
         i := i + 1;
      END; (* WHILE *)
      (* now generate new CHILDMASK *)
      CHILDMASK := CHILDMASK OR MASKLIST[i];
   END; (*ELSE *)
END; (*LOOP *)
```

While the above demonstrates how the correct child node identifiers can be generated, it is also necessary to associate the proper values for $s_{l\,max}$ and s_{umin} with each intersected child. For any parent node, the first child hit shares its parent's $s_{l\,max}$, and the last child hit shares its parent's s_{umin}. If more than one child is hit, the s_{umin} value for all but the last is equal to either s_{xmid}, s_{ymid} or s_{zmid}. The correct value is selected according to the entry in MASKLIST used to generate the subsequent child hit. For example, if the CHILDMASK of the next child hit is generated by ORing in the mask 010 form MASKLIST, then the current child's s_{umin} is equal to its parent's s_{ymid}. Each child except the first one hit has $s_{l\,max}$ equal to the previous child's s_{umin}.

4. Software Implementation

The main data structure used by the HERO algorithm is the octree, which defines bounding volumes for objects contained or partially contained within its terminal nodes. Ray intersections with these objects are calculated separately from the HERO algorithm, so HERO can be applied to any object model for which a bounding octree is constructed (e.g. polygons, half-spaces, quadrics etc.). Each octree node contains a flag to indicate whether or not it is terminal, and if this is true, the node simply contains a pointer to a list of the objects it bounds. Parent voxels contain a list of 8 pointers to child nodes, arranged so that the address of the pointer to child voxel n is equal to the base address of the parent node plus n words. A NIL pointer to a child indicates an empty child node. In addition, each parent node has the positions of its mid-planes (x_{mid}, y_{mid} and z_{mid}) stored to facilitate computation of the distance values s_{xmid}, s_{ymid} and s_{zmid}.

A depth-first search of the octree is required, so that the first intersected terminal voxel returned is known to be the first one hit by the forward-travelling ray. Whenever a parent node's intersected children are exhausted, the algorithm must backtrack one level up the tree and then continue in a depth-first manor. A stack is therefore used for implementation of this search.

One approach to using a stack for the search would be to push information regarding all intersected child nodes (between 1 and 4) for any one parent, entering these in descending order of distance. The top node could then be popped and used for the next level of search, and so on. However, a more efficient approach is to store status information for each parent node on the stack, so whenever a parent is popped, either the next child hit is derived and the parent's status is updated, or if all its children are exhausted, the parent is removed from the stack. In order to implement this operation, the index to MASKLIST should also be stored on the stack, together with a pointer to the previous child's s_{umin} (to be used as the next child's s_{lmax}). Each stack node therefore contains the following:

Pointer to octree node,
s_{lmax}, s_{umin}, s_{xmid}, s_{ymid}, s_{zmid},
CHILDMASK, LASTMASK, MASKLIST [1..3],
MASKLIST_index, Previous_s_{umin}_pointer.

For development purposes, the authors' implementation uses the terminal voxels themselves as simple cubic objects. In order to generate secondary rays (for illumination and reflection) it is necessary to know which face of the cube is struck by the ray. This information takes the form of one further 3-bit mask (SLMASK) carried on the stack, to indicate whether the ray strikes an x, y or z-plane. In conjunction with VMASK, this reveals whether the face struck is $x = x_{min}$, $x = x_{max}$ etc.

The only arithmetic operations required for each parent node (except the root) are computations of s_{xmid}, s_{ymid} and s_{zmid}, plus simple comparisons to generate the various masks. Although it would appear that the new s values could be generated by mid-point interpolation (e.g. $s_{xmid} = (s_{xmin} + s_{xmax})/2$), this is not a reliable technique due to the accumulation of errors. A better solution is to compute the reciprocals of the ray direction components ($1/x_v$, $1/y_v$ and $1/z_v$) once for each ray. These are used in the computation of s_{xmid}, s_{ymid} and s_{zmid} for each parent voxel by the normal ray/plane intersection expression, i.e.

$$s_{xmid} = (x_{mid} - x_E) * (1/x_v) \text{ etc.}$$

Each s value can therefore be calculated by a subtraction and multiplication.

5. Hardware Implementation

Application of parallel processing to ray tracing is highly desirable to minimise image generation time. With almost all ray tracing algorithms, it is easy to conceive a machine which allocates a processor for each new ray, as long as the entire world model is duplicated for each ray processor to avoid memory contention. However, with large world models and bounding octrees, this approach becomes extravagent in its use of memory. Fortunately, the structure of the octree can be exploited to allow other strategies.

At a high level, the HERO algorithm can be considered as a stack-based tree searcher. Distribution of processing without memory contention can be achieved in such a scheme by storing each level of the octree in a different block of memory, each with its

own HERO processor. The processor for any one level performs the HERO algorithm, passing its results either up or down to the next tree level processor as appropriate. There is a large amount of message passing in this scheme, but only between adjacent processors. The maximum number of rays which can be processed simultaneously is limited by the depth of the octree, but other strategies for allocation of voxels to processors are also possible.

At a lower level, the close relationship between arithmetic and logical processing in HERO can be exploited in a hardware implementation of the algorithm. A major goal is the minimisation of the amount of floating point processing required, particularly comparisons for generating the various 3-bit masks. If the floating point representation is chosen such that the exponent occupies the most significant bits of the floating point word, and the sign bits of the exponent and mantissa are suitably manipulated, then a floating point number which is larger than another will be represented by a word which has a larger integer magnitude. Simple greater than/less than comparisons of floating point numbers can then be achieved using fixed point comparators.

Sorting of the MASKLIST values into ascending order of referenced s value would appear to be a rather more complex operation, but it can in fact be achieved using only three comparisons of the above type with a few simple logic operations. The boolean results of these comparisons are labelled as follows:-

$$A = (s_{xmid} < s_{ymid}),\ B = (s_{xmid} < s_{zmid}),\ C = (s_{ymid} < s_{zmid}).$$

Using 'X' to represent states which cannot happen, the truth table for the various MASKLIST orderings against A, B and C is as follows:

A B C	MASKLIST[1]	MASKLIST[2]	MASKLIST[3]
0 0 0	1 0 0	0 1 0	0 0 1
0 0 1	0 1 0	1 0 0	0 0 1
0 1 0	X X X	X X X	X X X
0 1 1	0 1 0	0 0 1	1 0 0
1 0 0	1 0 0	0 0 1	0 1 0
1 0 1	X X X	X X X	X X X
1 1 0	0 0 1	1 0 0	0 1 0
1 1 1	0 0 1	0 1 0	1 0 0

By inspection, MASKLIST can be generated by the following logic functions:

Element	Bit 2 (ms)	Bit 1	Bit 0 (ls)
MASKLIST[1]	B NOR C	(NOT A) AND C	A AND B
MASKLIST[2]	B XOR C	NOT(A XOR C)	A XOR B
MASKLIST[3]	B AND C	A AND (NOT C)	A NOR B

Parallel processing can be implemented at a low level within HERO by executing all arithmetic operations concurrently, and all mask generation operations concurrently. Computation of each of the values s_{xmid}, s_{ymid} and s_{zmid} consists of a subtraction and multiplication, thus three parallel arithmetic units are used to generate these values

simultaneously. Generation of CHILDMASK, LASTMASK and MASKLIST involves nine arithmetic comparisons, which are achieved by nine integer comparators in parallel. Although current work is aimed at implementation of HERO using floating point processors and logic circuits, there is considerable scope for implementation of these functions on a VLSI integrated circuit (HEROIC).

6. Conclusion

Octree-based ray-tracing is an efficient technique for realistic rendering of complex scenes. The HERO algorithm provides a fast method for determining the nodes of an octree penetrated by a ray in the order they are hit, and is well-suited to VLSI hardware implementation. HERO could be applied to any world model for which a bounding octree can be generated, and hence could form an accelerator for a variety of graphics systems. Applications on which our hardware-enhanced octree ray tracing is targeted include workstations and high quality visual systems for simulation.

7. References

[1] Plunkett, David J. & Michael J. Bailey, "The Vectorisation of a Ray Tracing Algorithm", IEEE Computer Graphics & Applications Vol. 5, no. 8, August 1985, pp 52-60.

[2] Heckbert, Paul S. & Pat Hanrahan, "Beam Tracing Polygonal Objects", Computer Graphics Vol. 18, no. 3 (SIGGRAPH '84 Conference Proceedings), July 1984, pp 119-127.

[3] Whitted, Turner, "An Improved Illumination Model for Shaded Display", Comm. ACM Vol. 23, no. 6, June 1980, pp 343-349.

[4] Kay, Timothy L. & James T. Kajiya, "Ray Tracing Complex Scenes", Computer Graphics Vol. 20, no. 4 (SIGGRAPH '86 Conference Proceedings), August 1986, pp 269-277.

[5] Dippe, Mark & John Swensen, "An Adaptive Subdivision Algorithm and Parallel Architecture for Realistic Image Synthesis", Computer Graphics Vol. 18, no. 4 (SIGGRAPH '84 Conference Proceedings), July 1984, pp 149-158.

[6] Amanatides, John & Andrew Woo, "A Fast Voxel Traversal Algorithm for Ray Tracing", Eurographics '87, August 1987, pp 3-10.

[7] Glassner, Andrew S., "Space Subdivision for Fast Ray Tracing", IEEE Computer Graphics & Applications Vol. 4, no. 10, October 1984, pp 15-22.

[8] Fujimoto, Akira, Takayuki Tanaka & Kansei Iwata, "ARTS: Accelerated Ray Tracing System", IEEE Computer Graphics & Applications, Vol. 6, no. 4, April 1986, pp 16-26.

[9] Peng, Qunsheng, Yining Zhu & Youdong Liang, "A Fast Ray Tracing Algorithm Using Space Indexing Techniques", Eurographics '87, August 1987, pp 11-23.