2

A Distributed Data Model for Raytracing

Jorma Skyttä and Tapio Takala

Ray tracing is a superior method for producing realistic images. It can take into account all natural phenomena covered by classical ray optics in image formation, and that without any extra modeling effort. The main disadvantage is its high cost in terms of computer time. Production of ray traced images of reasonably complex scenes takes long in real time with a moderate general purpose computer [Whi80].

The basic idea of ray tracing is the brute force algorithm for simulating the path of a ray of light in the whole model space. As no global information of the model is used to anticipate the interactions of the ray with model elements, every ray must be tested against every object and most of the processing time is consumed to ray-object intersection calculation. At each intersection found the ray is divided into reflected and refracted components and into a ray directed to each light source to produce shadows. Higher quality images need more pixels to be calculated and the number of elements in a scene grows linearly with model complexity, leading to steep increase of the computational complexity of the whole problem.

In order to limit the growth of the problem, methodologies for splitting the model space have been used. The idea has been to construct a hierarchy of model elements and thus to be able to discard a large amount of non-intersecting elements with a single test. Several methods have been reported with various geometric division principles [Rub80, Gla84, Weg84, Kap85, Kay86]. Most of the methods presented have concentrated on finding suitable surface types for performing the hierarchal division without splitting the original coherence of the model [Dip84, Kay86]. Even manual interaction is often needed to reach this goal [Gol87].

Up to now the distribution of computation to physically different processors has not been considered much although parallel computation is necessary for efficient use of ray tracing methodology for image generation. The rough classification of approaches is based on two division principles [Sev88]:

1) Image space subdivision, where each ray originating from a different pixel is treated independently and

2) Object space subdivision, where the model space is split into subregions each treating rays independently.

The image space division principle leads to severe memory contention problems as every processor has to have access to the same model database. The only way to avoid this is to copy all the model data to each processor, which we cannot afford with large models.

The object space divisions presented have mostly concentrated on diminishing the amount of intersections calculations, not on distributing the model data to physically separated processors. A simple constant boundary box division [Cle85] is not adequate to achieve uniform loading of a group of processors. It is further assumed that after installing a graphic element to the model it is neither moved nor modified because otherwise the rigid boundary concept would result to formation of additional split elements [Sky87]. The more complicated division schemes, which avoid the splitting effect caused by updating, rely implicitly on the availability of a centralized data base service controlling the total model information. Typically in these methods there may be several references to the same element located at different cells [Tam82], and thus at worst case we calculate the same ray-object intersection several times unless global marking is employed.

The Tree of Processors

Our goal is to introduce a data structure containing each element only once without cross references between cells or to a centralized global store. In order to permit flexible modifying, the cell boundaries must not be rigid. This prevents unnecessary element splitting. We still go a bit further and totally abandon the idea of splitting a natural geometric element by an artificial boundary imposed by hardware constraints. Now the set of constraints seems to be contradictory and impossible to satisfy. In many cases the sets of elements are not linearly separable and cannot be separated with planes or even with more general boundary surfaces.

Our aim is to bring the separation to coordinate level. This can be accomplished with the concept of interval boundaries. The basic schema for utilizing interval coordinate axis boundaries for constructing a hierarchy of 3-D geometric objects relies on a set of rules determining the location of an element in the hierarchy by traversing the interval boundary sets beginning from the root of the tree. As the boundaries are intervals, this traversal contains parallel paths which can be elaborated in parallel provided that necessary hardware is in use.

A ray being processed can be considered a packet relayed around the physical network hierarchy based on the initial geometric coherence of the model. The preservation of coherence and the equal loading of network nodes require an automatic hierarchy construction schema, which distributes the model into the network. Our method relies on local decision making based on interval classification. In order to make local processing feasible, we stick to a simple binary tree hierarchy: elements at this node, elements at left node and elements at right node. As already stated, it is difficult, often even impossible to develop a geometric rule to perform this separation. However, employing intervals provides quite a simple solution to the problem at coordinate axis level. We do not have to construct separating planes, but define one coordinate axis as the dividing axis with a set of intervals dividing encountered elements into left and right halfspaces plus a nonseparable common area.

Formally this is accomplished with a set of three intervals: (Cl, Cr) defining the center area, (Ll, Lr) defining the left side space and (Rl, Rr) defining the right side space. These intervals have to be true intervals (length > 0) and their mutual relationship has to obey the assertion:

$Lr <= Rl$	; left and right spaces do not overlap
$Cl < Rl$	; center and right areas may only partially overlap
$Cr > Lr$	; center and left areas may only partially overlap

These assertions guarantee the existence of a true interval hierarchy where we can specify a meaningful set of search paths for any element with a known search interval. In order to perform meaningful searches in 3-D space, we have to define the three intervals for one coordinate axis at each node, but the axis used is alternated in different levels of the hierarchy. Similar intervals, defining the maximum dimensions of the left and right subtrees as well as limits of the root, are specified in all coordinate directions at each node. However, the assertions are valid only in the dividing direction.

At each node of the binary tree we can always search for an element in all of the three subsets, but using our set of rules many search paths can be terminated already by simple comparisons of the search interval and our assertions. The net result of this operation is the start of multiple search paths simultaneously, but only some of them remain active to the end of the search. The paths created can be processed either sequentially one by one until termination of each, or simultaneously due to branching. However, utilization of this form of parallelism is quite tedious since a control lock point is needed at each branching. The basic ray tracing algorithm contains already enough parallelism to utilize the whole number of processors in use as each ray is treated separately, and by choosing the sequential alternative for treatment of logically branched rays we are able to save memory and code at each node.

Creating an interval tree structure featuring the stated properties is rather simple. In the beginning each element brought into the system is positioned at the center area of the root of the tree. As a preset threshold value for memory space utilization at that node is exceeded, a balancing process is initiated. It picks either the leftmost element (Min Xr) or the rightmost element (Max Xl) and tries to move it to the respective subtree. Typically this leads to an update of the interval definitions, and balancing is prohibited unless for all nodes, all the constraints are satisfied. For an empty node the interval is undefined, and it receives its first numerical values with the first element positioned there. Later on the favoured direction for balancing is determined by the fill ratios of the respective subtrees. The process has great resemblance with ordinary weight balancing of graphs using scalar keys. Now we only add a set of constraints specifying which element can occupy what location, and that keeps our distributed version coherent with the original model. Also the system is similar to B-trees in the sense that it does balancing through sorting the internal keys of a single node, with the exception that no unique sorting can be performed due to interval relations specifying only partial ordering relations between the elements.

Algorithm for Tracing Ray-Objects

Our general idea in applying the processor tree for ray tracing is that each ray is an independent object moving through the space subdivided between processors. Individual rays are either primary rays initiated for each pixel of a raster image, or secondary rays spawned at points where a ray intersects an object surface (i.e. reflected, refracted or shadow rays).

Principally all ray objects can be processed in parallel. However, coordinated sequential operation of processors may be needed for specific purposes. When spawning new parallel reflection/refraction/shadow ray objects, typically implemented with subroutine calls in a single-processor environment, their initiator ray is stored in a processor node and is set to wait until each secondary ray has returned, thus performing recursion. Also while processing one ray, it may be necessary to check for the nearest intersection point in several cells. This could be done in parallel, but it is easier to serialize by providing the ray object with routing information, i.e. a list of nodes telling which processors are to be visited one-by-one, similar to iteration in conventional programming.

The ray tracing is initiated by a camera object, which is placed in the processor tree according to its focal point. Thus the camera is always inside the modelling space. The processor containing the camera will produce for each pixel a primary ray as a data packet, send the rays to other processors when needed, and collect the corresponding return packets containing intensity values for pixels, which are then output from the tree to a display system.

The main task for each ray is to find its first intersection point with all objects in the data base, to spawn secondary rays there, and to calculate the returned light intensity at that point. An outline of the basic ray tracing algorithm is as follows:

```
begin with parameters:
        - type of ray      /* reflection/shadow */
        - direction vector,
        - starting point   /* the ray's initialization point */
        - end point        /* end of search interval;
                                  for shadow ray: end point = light position,
                                  otherwise: end point = undefined) */
```

```
for each object in space do
        check for first intersection within ray's interval
                                /* i.e. between start and end points */;
        if intersection found then
                if shadow_ray then return zero intensity
                                /* terminate immediately */
                else assign: end point = intersection point
                                /* search interval is shortened as only
                                  nearer intersections are searched for */
        endif
endfor;
if end point defined then
        /* nearest intersection is found for a reflection ray */
        if not maximum recursion depth reached then
                spawn new rays for:
                        - reflection
                        - refraction
                        - a shadow ray towards each light source;
        /* then wait for secondary rays: */
        while not all secondary rays returned do
                if return packet obtained then
                         add intensity
        endwhile;
        return total intensity
else
        /* no intersection was found */
        if shadow_ray then
                /* no shadowing obstacle found */
                return light source intensity
        else    /* the ray didn't hit the scene at all */
                return background intensity
terminate.
```

A ray is terminated, if the maximum recursion depth is reached (for a reflected/refracted ray), if the light source is reached or an obstacle found (for a shadow ray), or if the ray is going to the background outside the whole modelling area. In each case, a return packet is formed and sent back to the ray's initial node. When all descendants of a non-terminated secondary ray have returned, its total illumination is calculated and put into the corresponding return packet. For a primary ray, the pixel intensity value is forwarded to a display.

The directions of a ray are calculated in the processor, where the first intersection point is found. For reflected and refracted rays the information needed (the intersecting surface's properties) is localized in that processor only, whereas calculation of shadow rays' directions needs that all light positions are broadcast to all processors.

Pruning the Object Intersection Search

The brute-force algorithm utilizing no structure of the model must visit every object in the whole space for each ray to find the nearest intersection. In a cell-organized modelling space the algorithm can be localized by visiting the cells sequentially (brute force is applied only within each cell), starting from the cell where the ray initiates, and then following the ray direction until first intersection is encountered.

For disjoint cells (eg. in octree or voxel-type organization) the strategy is strictly sequential. If no intersection is encountered in one cell, the search can simply continue in the next one, which is determined according to the point where the ray leaves the cell. The cells have a linear ordering along the ray. If all cells are of equal size, the ordering calculation resembles the DDA line rasterization algorithm [Fuj86].

In our system the areas of all nodes in the tree are not disjoint, and thus cannot be processed independently with the simple sequential procedure outlined above. Typically the area of an intermediate node (C-interval) will overlap with both of its subtree areas, and may overlap with potentially all higher nodes' areas up to the root (see Figure 1). Assuming we have found an intersection in one cell, we cannot be sure it is the nearest one before checking all the overlapping cells, too. Neither can we, without checking the overlapping areas as well, determine that there is no intersection within a cell's boundaries.

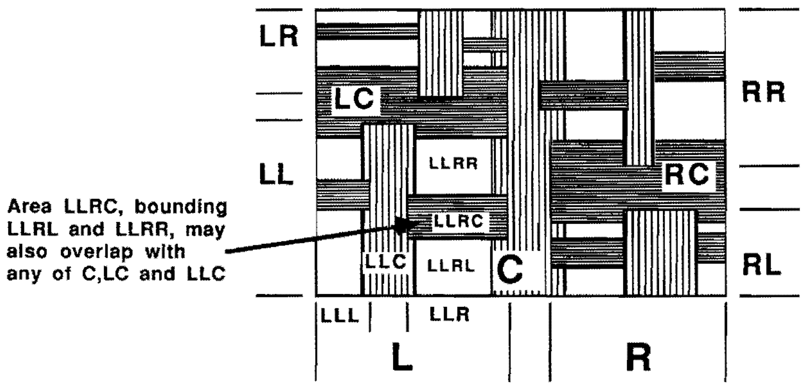
LR
LC
LL
Area LLRC, bounding LLRL and LLRR, may also overlap with any of C,LC and LLC
LLRR
LLRC
LLC
LLRL
C
RR
RC
RL
LLL
LLR
L
R

Figure 1: Subdivision of a 2-D space (the code of a node specifies its position in the tree, e.g. LLRC = left-left-right-center).

Search in several cells and comparison of results is necessary to determine the nearest intersection. However, even overlapping cells can be partially ordered along a ray, and this property can be used to organize the parallel search sequentially. Also the bounding boxes of objects are stored in hierarchical fashion, and can be checked against the dynamic search interval being the range along the ray from its starting point to the nearest intersection found so far. This method will effectively prune the search candidates.

Assuming the ray initiates at an intermediate node (e.g. LLRC in the figure), the search for nearest intersection with objects in the scene is first performed there, is then propagated down and finally up in the processor tree. The following aspects are taken into account when implementing these processes:

1) Downward: Each C-area (LLRC) may overlap with its both left and right descendants (LLRL and LLRR), and thus both subtrees have to be considered. However, if we first check the bounding box of a subtree (which is stored in the intermediate node), typically only one or neither of those intersects the ray within the search interval (e.g. if we go down from LLRC, and the starting point isn't within LLRR, then the whole LLRR can be discarded). As the search - even though pruned - may branch into several subtrees whose results have to be compared, this will be implemented using recursion (the initial ray packet sends descendant packets and waits for results).
2) Upward: In principle a cell (LLRC) may overlap with all its ancestors (C, LC, LLC) in the tree - thus the whole path up to the root node has to be checked before we can determine the first intersection. In this case also the search interval is first checked against bounding boxes of the nodes. Because there now is only one linear path of nodes being visited, the process will be implemented with iteration (the ray packet leaves its initial node, but is provided with return address and a list of upper nodes being visited).
3) If none of the initial node's subtrees contain an intersection, nor one of the upper nodes within the search boundaries, then the process is continued with another subtree, determined according to the ray's exit point from the initial subtree (e.g. if the ray direction is straight up from LLRC, and LLRR doesn't report intersection, then LC will recognize that the search continues in the LR subtree).

Checking the potential intersections for each ray at several processors may cause tedious overhead, and should be reduced to the minimum. The interval borders at each node can prune many unnecessary checks. However, all the upward check queries have to proceed up to the root node, unless some knowledge about the intermediate nodes' areas is stored in the subnodes also. One way to create this checking information is to calculate, during filling and balancing of the tree, extents of the "private" area of each subtree, which is guaranteed not to overlap with any of the center areas of its ancestor nodes.

Keeping the areas of the intermediate nodes as narrow as possible will optimize the performance of upward directed intersection queries in the tree. Whether this is feasible, depends much on the material stored in the tree. In special cases, it may be necessary to subdivide a large object into a set of smaller ones just for efficiency, although our goal is to avoid object splitting.

References

[Cle85] J. Cleary et. al., "Multiprocessor Ray Tracing", *Computer Graphics Forum*, vol. 5, no. 1, pp. 3-12, March 1986.

[Dip84] M. Dippe and J. Swensen, "An Adaptive Subdivision Algorithm and Parallel Architecture for Realistic Image Synthesis", *Computer Graphics*, vol. 18, no. 3, pp. 149-158, July 1984.

[Fuj86] A. Fujimoto et. al., "ARTS: Accelerated Ray-Tracing System", *IEEE Computer Graphics and Applications*, pp. 16-26, April 1986.

[Gla84] A. Glassner, "Space Subdivision for Fast Ray Tracing", *IEEE Computer Graphics and Applications*, pp. 15-22, Oct. 1984.

[Gol87] J. Goldsmith and J. Salmon, "Automatic Creation of Object Hierarchies for Ray Tracing", *IEEE Computer Graphics and Applications*, pp. 14-20, May 1987.

[Kap85] M. Kaplan, "Space Tracing, a Constant Time Ray-Tracer", SIGGRAPH'85 Course Notes, no. 11: *State of the art in Image Synthesis*, San Francisco, CA, 1985.

[Kay86] T. Kay and J. Kajiya, "Ray Tracing Complex Scenes", *Computer Graphics*, vol. 20, no. 4, pp. 269-278, Aug. 1986 (SIGGRAPH'86).

[Rub80] S. Rubin and T. Whitted, "A 3-Dimensional Representation for Fast Rendering of Complex Scenes", *Computer Graphics*, vol. 14, pp. 110-116, July 1980.

[Sev88] S. Gaudet et al., "Multiprocessor Experiments for High-Speed Ray Tracing", *ACM Transactions on Graphics*, vol. 7, no. 3, pp. 151-179, July 1988.

[Sky87] J. Skyttä and T. Takala, "VLSI-based Partially Ordered Tree for Storing 3-D Geometric Information in Data Base", *IEEE CompEuro'87*, Federal Rep. of Germany, 11-15 May 1987, pp. 746-749.

[Tam82] M. Tamminen and R. Sulonen, "The EXCELL Method for Efficient Geometric Access to Data", *19th Design Automation Conference*, Las Vegas, 1982, pp. 345-351.

[Weg84] H. Weghorst, G. Hooper and P. Greenberg, "Improved Computational Methods for Ray Tracing", *ACM Transactions on Graphics*, vol. 3, no. 1, pp. 52-69, Jan. 1984.

[Whi80] T. Whitted, "An Improved Illumination Model for Shaded Display", *Communications of the ACM*, vol. 23, no. 6, pp. 343-349, June 1980.