PARALLEL SPATIAL ENUMERATION OF IMPLICIT SURFACES USING INTERVAL ARITHMETIC FOR OCTREE GENERATION AND ITS DIRECT VISUALIZATION

Nilo Stolte† and Arie Kaufman‡
{stolte|ari}@cs.sunysb.edu

†School of Computer Science & Eletronic Systems
Kingston University
Penrhyn Road, Kingston upon Thames
Surrey KT1 2EE England

‡Computer Science Department
State University of New York at Stony Brook
Stony Brook, NY 11794-4400 U.S.A.

ABSTRACT

This article presents a new parallel method for implicit surface voxelization - the determination of which cells of a regular grid in 3-space intersect the zero-set of an implicit function. The serial version of the method uses interval arithmetic to rapidly prune regions of space where the surface does not lie, and a novel octree generation/storage scheme for recording the voxels the surface meets. In the parallelization, good speedups on up to 7 processors are achieved, although the serial version is also very efficient. The parallelization is accomplished through a master-slave scheme with dynamic load-balancing.

We also describe a method for rendering voxels directly. This method is effective when the voxels are small compared to pixels, hence is appropriate for very high-density voxel grids.

Even though an octree is used in this paper, the algorithm can also be used for 3D grids or other chosen data structures. This flexibility arrives from the fact that the voxels storage is independent of the subdivision. Once a voxel is produced, its storage is accomplished by using the voxel's coordinates only.

Key Words: Implicit Surfaces, Voxel, Voxelization, Spatial Recursive Subdivision, Octree, Spatial Enumeration, Visualization.

1. INTRODUCTION

The transformation of geometric surfaces into voxels is a significant research topic for Volume Visualization. It allows mixing geometric with volumetric data into the same volume. Its use is very popular for accelerating Ray-Tracing [SC95, YCK92]. For parametric surfaces the parametric space can be subdivided recursively to produce polygons, while for implicit surfaces the three-dimensional space can be subdivided to produce voxels. Space recursive subdivision is an elegant way to produce efficient implicit surfaces voxelization. Other advantages are: simplicity to deal with manifold objects, no need for clipping, low algorithm complexity and facility to classify regions inside and outside the surface. Octrees are natural data structures to store volumes whose interior is homogeneous or empty, and to avoid representing the voxels outside a surface.

Although octrees are not natural candidates for parallelization, good algorithms exist addressing this subject. One example that particularly fits our problem of surface voxelization is [BFG94]. This algorithm exhibits fairly good results with up to 4 processors. For more than 4 processors the results are not as satisfying. As in [BFG94], we use a shared-memory machine and get approximately the same behavior, but with better results.

Unfortunately, the greatest limitation of the algorithm in [BFG94] is the assumption that the containment of a surface into an octant can be known at any moment. For certain subdivision algorithms [KB89, Duf92, Tau94, SC97] this assumption is not correct. With these subdivisions we can determine only if the surface is *not* contained in an octant. When the subdivision reaches the leaf level, there is no guarantee that the voxel really contains a part of the surface. Nevertheless, the probability of the voxel belonging to

the surface grows quickly at each further subdivision, and at the last level we assume that this probability is very high. The voxelization obtained is guaranteed to envelop the surface.

These subdivision algorithms require a totally different approach for parallelization. First, the octree must be separated from the subdivision. The subdivision must continue until the last level, and only then, can the voxel be stored into the octree. This implies that the octree storage must be very fast. Thanks to the efficient octree traversal algorithm presented in this article, the time of storing voxels in the octree is negligible in relationship to the rest of the task.

The most time-consuming part of the algorithm, thus the best candidate for the parallelization, is the test to know which octants definitely do not intersect the surface. This part of the algorithm also includes the calculation of the normal vector (only on the last level) for every voxel, for later use during the visualization process. These tasks are very time-consuming and parallelization is desirable. We assign these tasks to several slave processes that run in parallel. The master process creates the slave processes when the voxelization is required, controls the work balance, kill the slave processes when the work is done, and displays the voxelized scene. This approach has very promising results.

2. SERIAL VOXELIZATION ALGORITHM

We use interval arithmetic to voxelize implicit surfaces, that is, surfaces given by the set of points which are solutions to a function of the kind $F(x, y, z) = 0$.

Our voxelization [SC97] is accomplished by subdividing the space in a recursive way. Each subdivided octant is represented by three intervals, one for each variable (X=[x_0,x_1], Y=[y_0,y_1] and Z=[z_0,z_1]), where the lower and upper bounds correspond to the octant bounding coordinates. The implicit function, after its translation to interval arithmetic, is generally called an "inclusion function" [Sny92]. The result of applying the three intervals in the inclusion function is an interval with two real bounds. If the interval lower bound is greater than zero, then the octant is guaranteed to be totally outside the surface. If the interval upper bound is less than zero, then the octant is guaranteed to be completely inside the surface. In both cases the octant is rejected. Otherwise, the octant might intersect the surface and it is further subdivided. The algorithm in Fig. 1 shows how the subdivision works. This algorithm is different from the actual implementation, as it is given for clarity purposes. In the actual implementation: the recursion is implemented with a loop and a external stack; the array S and the function "**Width**" are not used; and the function is passed only the lower octant coordinates as arguments instead of the whole intervals. We have obtained [SC97] better results with interval arithmetic than with other recursive subdivision methods such as the one used by Kalra and Barr in [KB89].

```
Width(I) {
    I is [i0,i1];
    return(i1-i0);
}

Voxelize(X,Y,Z) {
    X is [x0,x1], Y is [y0,y1] and Z is [z0,z1];
    S(8,3) is a matrix containing items of type [s0,s1];
    F(X,Y,Z) is the inclusion function for f(x,y,z);
    if (leaf level) {
      calculate and normalize normal vector;
      /* See alg. Fig. 2 */
      store_in_octree(x0,y0,z0,normal);
      return;
    }
    /* Subdivide X, Y and Z */
    xh←x0+Width(X)/2;
    yh←y0+Width(Y)/2;
    zh←z0+Width(Z)/2;
    S(0,0)←[x0,xh]; S(0,1)←[y0,yh]; S(0,2)←[z0,zh];
    S(1,0)←[xh,x1]; S(1,1)←[y0,yh]; S(1,2)←[z0,zh];
    S(2,0)←[x0,xh]; S(2,1)←[yh,y1]; S(2,2)←[z0,zh];
    S(3,0)←[xh,x1]; S(3,1)←[yh,y1]; S(3,2)←[z0,zh];
    S(4,0)←[x0,xh]; S(4,1)←[y0,yh]; S(4,2)←[zh,z1];
    S(5,0)←[xh,x1]; S(5,1)←[y0,yh]; S(5,2)←[zh,z1];
    S(6,0)←[x0,xh]; S(6,1)←[yh,y1]; S(6,2)←[zh,z1];
    S(7,0)←[xh,x1]; S(7,1)←[yh,y1]; S(7,2)←[zh,z1];

    for(i←0; i<8; i←i+1;) {
       [f0,f1]←F(S(i,0),S(i,1),S(i,2));
       if (0∈[f0,f1]) {
          Voxelize(S(i,0),S(i,1),S(i,2));
       }
    }
}
```

Figure 1: Serial Voxelization Algorithm

In this article, we use different kinds of implicit surfaces chosen for their complexity and high voxelization times. The normal vector calculation (applying the analytical gradient of the function) at the leaf nodes (voxels) and its normalization are time consuming and contribute significantly by the high voxelization times. The normal vectors are used for visualization purposes. In Scene 1 (see Fig. 4) we use 10 spheres blended together using our polynomial blending function. The equation of this surface is:

$$F(x, y, z) = \sum_{i=1}^{10} g(a_i \cdot f_i(x, y, z)) - C$$

Where $f_i(x, y, z)$ are sphere equations and $g(r)$ is our polynomial blending function [Sto96] (see [Bli82, Mur91, Wyv94, PASS96] for blending functions):

$$g(r) = \begin{cases} \frac{1}{R^8} \cdot (r^2 - R^2)^4 & \text{if} \quad r <= R \\ 0 & \text{if} \quad r > R \end{cases}$$

The other two scenes, Scene 2 and 3, are given by implicit functions given in spherical coordinates. Their equations and rendering are given in Fig. 5. A special technique, described in [SK98], is used to convert rectangular intervals to spherical intervals.

```
char *octree;/* pointer to the first free octree byte */
char *free_space; /* pointer to the first free byte in a block */
int free_bytes; /* number of remaining free bytes in a block */
int X_ant, Y_ant, Z_ant, mask1, mask2;
init_octree( ) {
/* Initialize masks as follows (each square is a bit) */
/* n = number of octree levels */
/* nb+1 = number of variable bits */
```

	nb		n+3	n+2	n+1	n	n-1	n-2		5	4	3	2	1	0
mask1←	0	...	0	0	0	1	0	0	...	0	0	0	0	0	0
mask2←	1	...	1	1	1	0	0	0	...	0	0	0	0	0	0

```
octree←free_space←alloc_block( ); /* allocates one block */
free_bytes←Size_of_Block−Bytes_in_Cell;
free_space←free_space+Bytes_in_Cell;
push(octree);
/*variables to find common parent */
X_ant←0; Y_ant←0; Z_ant←0;
}
store_in_octree(X,Y,Z,input)
int X,Y,Z;
any input;
  { char **pcel;
    /* Ascend octree to find a common parent */
    while ( ( (X and mask2) ≠ (X_ant and mask2) ) or
            ( (Y and mask2) ≠ (Y_ant and mask2) ) or
            ( (Z and mask2) ≠ (Z_ant and mask2) ) )
      { pop();
        mask1←mask1<<1; mask2←mask2<<1;
      }
    pcel←pop();
    while (TRUE) /* Descends octree until the voxel*/
      { push(pcel);
        if (Z and mask1) pcel←pcel+4;
        if (Y and mask1) pcel←pcel+2;
        if (X and mask1) pcel←pcel+1;
        if ((mask1 and 1) = 0)
          { mask1←mask1>>1; mask2←mask2>>1;
            if (*pcel = 0) /* if node doesn't exist, creates it */
              { if (free_bytes<Bytes_in_Cell)
                  { free_space←alloc_block( ); /* allocates */
                    free_bytes←Size_of_Block; /* one block */
                  }
                *pcel←free_space; /*creates and descends */
                pcel←free_space;
                free_space←free_space+Bytes_in_Cell;
                free_bytes=free_bytes-Bytes_in_Cell;
              }
            else pcel←*pcel; /* Otherwise descends only*/
          }
        else break; /* Leaf reached. Exit loop */
      }
    X_ant←X; Y_ant←Y; Z_ant←Z;
    *pcel←input;
  }
```

Figure 2: Octree generation algorithm

When the subdivision arrives at the last level, the normal vector is calculated and stored into the octree. The octree is a convenient way to represent huge voxel spaces without consuming too much memory. It also allows us to represent interior empty space very economically. The voxelization method with minor modifications can also detect interior regions.

3. SERIAL OCTREE GENERATION

Our octree is a classical pointer octree, where the root node is defined by a pointer called "octree", as shown in Fig. 2. This pointer points to an array of pointers with eight elements, each one representing one eighth of the original volume. Each of these arrays is called a cell. A null pointer means that the region is empty, while a non-null pointer points to another array of eight pointers, further subdividing the region. This process continues until the leaf node is found, where each non-null pointer points to a voxel.

The efficiency of our octree lies into its simplicity. We keep one integer variable *"mask1"* with a set bit exactly at the bit position *"n"*, where *"n"* is the current octree level, which is the total number of octree levels in the beginning (see Fig. 2). We use this bit to filter the coordinates bits and to control the algorithm as in the octree ray traversal algorithm in [SC95].

The algorithm in Fig. 2 is given in a "C-like" pseudo-code. For the sake of clarity the type castings are omitted; each assignment is given by a ←, the logical commands are written with its names (**and** and **or**) instead of symbolically, and the recursive stack operations are denoted by **push** (to put and element into the stack) and **pop** (to remove an element from the stack).

Once initialized, the octree is dynamically created by calling **store_in_octree**( ) for each new produced voxel. This function receives 4 parameters - the three voxel coordinates (*X*, *Y* and *Z*) and a pointer to the voxel content (*input*). In our case, it is the pointer to the surface normal in the voxel.

A significant feature of this algorithm is that it does not require descending all octree levels from the root. It starts from the *cell* where the last voxel was stored. In most cases the current voxel will lie in the same *cell* or in a nearby relative *cell*. If it does not lie in the same *cell*, the algorithm ascends some levels until the common parent is found. This happens in the first part of the algorithm.

To find the common parent we use the variable *mask2* as shown in the algorithm. This part is considerably efficient because it is translated to very few machine instructions and the variables used are always in the cache memory. The variable *mask2* is used to filter the most significant bits from the coordinate values. While the most significant bits of the current voxel coordinates filtered by *mask2* are not equal to the previous voxel coordinates most significant bits (also filtered by *mask2*), the algorithm goes up one level (**pop** command), and shifts both mask variables to the left. When both most significant bits become equal, the common parent is found and the next part of the algorithm will be executed to descend the octree using the variable *mask1*. The *mask2* variable is shifted left to be able to filter the most significant bits of the coordinates for the octree level immediately upper to the current level. The variable *mask1* is shifted left to be able to filter the correct coordinate bit corresponding to the resulting octree level when the the common parent is finally found. The variable *mask1* is then used to descend the octree.

The next part of the algorithm descends the octree from the common parent *cell*, creating new *cells* when it does not yet exist (when **pcell=0*).

4. PARALLEL RECURSIVE SUBDIVISION

Our parallel implementation is a simple master-slave configuration. The master creates the slaves and controls their activities. The slaves and the master run in different processors. This configuration was implemented into a shared memory SGI Challenge multiprocessor system. The master maintains an internal work stack where all octants that are going to be subdivided are stored. Initially, only the first eight octants are stored into this stack. The master creates the slaves and enters into a loop until the work is completed. In this loop, the master scans for all non-idle slaves queues in search of their results to store them into the stack or, at the leaf level, into the octree. Initially all slaves are idle; thus only the eight original octants remain in the stack. After that, the master distributes the octants from the stack to the idle slaves, if there are any.

Each slave that receives one octant starts to subdivide it and test if the surface is contained in each sub-octant. This test is the most time-consuming task, thus the focus of our parallelization algorithm. If the test is true for a given sub-octant, it is stored into the slave queue. This queue has only eight positions, and can be accessed by two different indices: one for the master and one for the slave. When the master scans a slave queue it uses its own index. When this index is smaller than the slave index, it is incremented and the octant from its corresponding position in the slave queue is transferred to the appropriate data structure. If the slave working octree level is a leaf level, the octants are voxels and are not written into the working stack but directly into the octree. In this way, the quantity of information passing through the work stack is reduced, thus significantly contributing to the optimization of the dynamic load balance, leading to a better performance. Once the slave is finished and its entire queue has been transferred away, it becomes idle waiting for a new octant from the master. A considerable amount of the time is spent in the normal vector calculation and normalization, which take place at the last level (leaf level) only. In this way, the method automatically atributes more priority to the higher levels, since the processing of a higher level is faster than the leaf level. Thus, higher levels are likely to be processed more often by the master. Therefore, it is highly improbable that the work stack would be empty with slaves processors waiting for other processors to fill it. Then, the processing of the higher levels would probably show a constant increase in efficiency for a growing number of processors, but at the leaf level the efficiency would gradually degrade because of the serial copy managed by the master. In those cases, more sophisticated methods for copying the information from the slaves to the stack/octree should be taken into account.

The algorithm is shown in Fig. 3.

```
slave( )
  { while (TRUE);
      { wait for a master job;
        get octant(X,Y,Z,my→level);
        i ← my→i ← -1;
        for (each of eight sub-octants)
          { determine Xs,Ys and Zs for sub-octant;
            if (octant(Xs,Ys,Zs,my→level) may cut surface)
              { i ← i+1;
                if (my→level is leaf);
                  { put voxel(Xs,Ys,Zs,normal) in my→queue[i]
                  }
                else put octant(Xs,Ys,Zs) in my→queue[i];
                my→i=i;
              }
          }
      }
  }
master( )
  { init_octree( );
    initialize data structures;
    push first eight octants into work stack;
    make copies of slave( ) to all processors and execute them;
    while (there is still work)
      { for (each non-idle slave)
          { i ← slave→master_i;
            level ← slave→level;
            if (level is leaf);
              { while (i < slave→i)
                  { i ← i+1;
                    get voxel(X,Y,Z,normal) from slave→queue[i];
                    store_in_octree(X,Y,Z,normal);
                  }
              }
            else
              { while (i < slave→i)
                  { i ← i+1;
                    get X,Y,Z from slave→queue[i];
                    push octant (X,Y,Z,level+1) in work stack;
                  }
              }
            slave→master_i ← i;
            if (slave is done)
              { slave→master_i ← -1;
                make slave idle;
              }
          }
        for (each idle slave)
          { pop octant (X,Y,Z,level) from work stack;
            if pop was successful give octant to slave;
          }
      }
    kill all slaves;
  }
```

Figure 3: Parallel recursive subdivision

5. RESULTS

Table 1 summarizes our results for a resolution of 512^3. We show the times as a function of the number of slaves, and the yield in relationship to the time spent for just one slave. The yield is calculated by dividing the estimated time ($\frac{t_1}{n}$, where t_1 is the time for just one slave and n is the number of slaves) by the real time (t_n), that is:

$$yield = \frac{t_1}{n \cdot t_n}$$

The yield calculated this way gives a clear idea of the slaves' activity. The algorithm was conceived to have a high parallel performance because the work tends to be evenly distributed among the slaves. Nevertheless, the results showed an unexpected outstanding performance in Scene 3 for 2 to 4 slaves and for 7 slaves.

Figure 4: Scene 1, ten spheres blended

(a)

(b)

Figure 5: (a) Scene 2, $sin(4\theta) \cdot sin(8\phi) - R = 0$ and (b) Scene 3, $sin(3\theta) \cdot sin(4\phi) - R = 0$

Thus, Scene 3 can be considered a good test scene for further improvements of the algorithm. The improvements can be obtained by forcing the same conditions for other scenes through the implementation of a smarter task distribution algorithm.

	Scene 1		*Scene 2*		*Scene 3*	
slaves	*time*	*yield*	*time*	*yield*	*time*	*yield*
1	88 s	–	111 s	–	71	–
2	45 s	97%	58 s	95%	35 s	100%
3	34 s	88%	38 s	97%	23 s	100%
4	26 s	84%	31 s	89%	17 s	100%
5	21 s	83%	28 s	79%	16 s	88%
6	20 s	73%	24 s	77%	14 s	83%
7	14 s	89%	21 s	75%	11 s	92%

Table 1: Performance results for a voxelization resolution of 512^3

6. VISUALIZATION METHOD

The visualization method used to generate images for this article is based on high-resolution voxel spaces (resolution is 512^3 in Figures 5 and 4). Voxels are stored in an octree, thus, allowing quite huge discrete spaces without a very high memory consumption. Normal vectors are calculated during the voxelization (times in Fig. 1 include this calculation) by evaluating the gradient in the middle of the voxel and then normalizing it. A voxel, located at the leaf octree level, is just a pointer to a structure containing the three normal vector components, color and other information. Higher octree level nodes contain only octree children pointers, when they exist, or zero otherwise. All the voxels are considered as points and rendered using SGI's GL or OpenGL.

This visualization method can allow close-ups of the surface using levels of details. Levels of details are quite natural to hierarchical voxel models, the models used in this article, because the transition between the original and refined model is indistinguishable.

A major advantage of our method is that no special hardware is necessary to use voxels. In addition, it allows the mixing of polygonal models (for representing polygonal objects) with voxels at graphics engine level, thus, eliminating the need to convert polygons to voxels while profiting from hardware rendering for polygons.

The algorithm describing the visualization technique is given in Fig. 6. The variables *cell* and *root* have initially the address of the root of the octree. Variable i is an index varying from 0 to 7 used to access the current octree element into an eight elements cell. These eight elements identify eight equal sided neighbour cubes, defining a recursive subdivision of a single cube. Each of these elements contains a pointer to a new cell, when this cell contains any part of the

```
cell = root = octree root cell address;
i=0;
push(cell);
push(i);
do {
   /* ascend the octree until i<8 */
   while ((i>7) and (cell≠root)) {
      pop(i);                    /* Ascend one*/
      pop(cell);                 /* octree level.*/
      X=X>>1;  Y=Y>>1;  Z=Z>>1;
   }
   while  (i≤7) { /* Descend or move right */
     aux=cell[i];
     X=(X<<1) or (i and 1);          /* Calculate */
     Y=(Y<<1) or ((i>>1) and 1);     /* the voxel */
     Z=(Z<<1) or ((i>>2) and 1);     /* coord.    */
     if (aux=Leaf Node) {            /* Leaf?     */
        Display voxel (X,Y,Z) as a point with the
        normal vector pointed by "aux";
        i=i+1;                       /* Go right  */
        X=X>>1;  Y=Y>>1;  Z=Z>>1;
     }
     else {                          /* Not Leaf!*/
        if (aux≠0) {                 /* Empty?   */
           push(cell);               /* Descend 1*/
           push(i+1);                /* level    */
           cell=aux;
           i=0;
        }
        else {                       /* Empty !  */
           i=i+1;                    /* Go right */
           X=X>>1;  Y=Y>>1;  Z=Z>>1;
        }
     }
   }
} while (cell≠root)
```

Figure 6: *Visualization Algorithm*

surface, or a null pointer otherwise. The recursion is controlled by a stack denoted by the instructions **push** (to introduce a value in the stack) and a **pop** operator (to extract a value from the stack). The variable i is assigned a zero value denoting a left to right tree traversal. Both, *cell* and i are pushed in the stack to start the recursive traversal. The recursion is implemented by the **do-while** loop as shown in the algorithm. The first part inside the loop ascends the tree if i reaches an index greater than 7. Since i is zero in the beginning of the algorithm, the control passes immediately to the second part which descends the tree. This part is a **while** loop which takes place while $i \leq 7$, indicating that this part also advances to all the elements of the current cell from left to right. The voxel coordinates X, Y and Z are built, bit by bit, from the i values. Notice that the previous coordinates bits are saved by shifting them to the left at each new interaction.

If the current cell is a *leaf* node, then X, Y and Z contain the complete coordinates of the voxel to be displayed and the current element (*cell[i]*) contains a pointer to the normal vector of the voxel. These informations are sent to the graphics card using GL point primitives to display the point with the normal vector. In practice, these informations are first stored in a list and when the list is full all the points are displayed at once to increase efficiency. These details are omitted in the algorithm. Notice that after displaying the voxel, i is incremented to advance to the next element to the right of the current element. Also notice that the coordinate variables must be shifted one bit to the right.

If the cell does not correspond to a *leaf* node, and if the current element (*cell[i]*) is zero, the element does not exist, therefore the algorithm advances to the next element (by incrementing i) and shifts the coordinates one bit to the right. However, if the current element is not zero, the address of *cell* and the next element index (i+1) are saved in the stack, and the algorithm descends the tree by attributing to *cell* the address contained in the current element (*cell[i]*) and making i equal to zero (to restart from the extreme left side again in the new cell).

Once i reaches the value 8, that happens when all the elements of a cell were visited, the control is passed again to the main loop that continues if *cell* $\neq$ *root*. This time $i > 7$, and the first *while* loop takes the control. This loop extracts from the stack: (1) the indexes i of the current elements and (2) the cell addresses corresponding to all those cells that were already completely visited. At each interaction this loop also shifts the coordinates one bit to the right. Notice that the loop either stops when a cell not yet completely visited is found (denoted by i values less or equal to 7) or when the root cell is found. If the root cell is found and i is greater than 7, all cells in the tree were visited and the algorithm finishes.

At the current time, this method allows interactive visualization for easy surface inspection. The images produced in this article are snapshots from the visualization method viewing window. Image quality is comparable to that of ray-casting (see Fig. 5 and Fig. 4).

7. CONCLUSION

Former parallel voxelization algorithms using recursive subdivision and octree storage [BFG94] are not previewed for a whole class of subdivision algorithms, particularly for the one used and explained in this article and the ones in [KB89, Duf92, Tau94, SC97]; these subdivisions only guarantee that a surface is not contained in an octant, not the opposite. Our approach not only solves this problem but also generalizes its application to a broader variety of spatial recursive subdivisions and data structures. Since our algorithm does not store any voxel until the bottom level of the subdivision is reached (the voxel level); it can also be used with 3D arrays or other data structures, besides octrees. The key factor for obtaining

this feature is isolating the octree from the subdivision algorithm, a must for the kind of subdivision algorithm we use. This contributes to a higher degree of abstraction, thus giving more flexibility. We show that this flexibility is not coupled with loss of performance, as it is normally the case, thanks to our efficient octree generation algorithm and our optimized dynamic load balance scheme. However, other data structures might not profit from the memory coherency that the octree provides, which is in certain cases can be very important.

8. AKNOWLEDGEMENTS

The authors would like to thank Frank Dachille for his interest and help in this work, giving examples and initial ideas for the parallelization algorithm.

9. REFERENCES

[BFG94] M. A. Bauer, S. T. Feeney, and I. Gargantini. Parallel 3D Filling with Octrees. *Journal of Parallel and Distributed Computing*, 22:121–128, 1994.

[Bli82] James Blinn. A Generalization of Algebraic Surface Drawing. *ACM Transactions on Graphics*, 1(3):235–256, July 1982.

[Duf92] Tom Duff. Interval Arithmetic and Recursive Subdivision for Implicit Functions and Constructive Solid Geometry. *Computer Graphics*, 26(2):131–138, July 1992.

[KB89] Devendra Kalra and Alan Barr. Guaranteed Ray Intersections with Implicit Surfaces. *Computer Graphics*, 23(3):297–306, July 1989.

[Mur91] Shigeru Muraki. Volumetric Shape Description of Range Data using "Blobby Model". *Computer Graphics*, 25(4):227–235, July 1991.

[PASS96] A. Pasko, V. Adzhiev, A. Sourin, and V. Savchenko. Function Representation in Geometric Modeling: concepts, Implementation and Applications. *The Visual Computer*, 25(4):227–235, July 1996.

[SC95] Nilo Stolte and René Caubet. Discrete Ray-Tracing of Huge Voxel Spaces. In *Eurographics 95*, pages 383–394, Maastricht, August 1995. Blackwell.

[SC97] Nilo Stolte and René Caubet. Comparison between different Rasterization Methods for Implicit Surfaces. In Rae Earnshaw, John A. Vince and How Jones, editor, *Visualization and Modeling*, chapter 10, pages 191–201. Academic Press, April 1997. ISBN: 0122277384.

[SK98] Nilo Stolte and Arie Kaufman. Robust Hierarchical Voxel Models for Representation and Interactive Visualization of Implicit Surfaces in Spherical Coordinates. In *Implicit Surfaces'98*, pages 11–18, Seattle, June 1998.

[Sny92] John M. Snyder. Interval Analysis For Computer Graphics. *Computer Graphics*, 26(2):121–130, July 1992.

[Sto96] Nilo Stolte. *Espaces Discrets de Haute Résolutions: Une Nouvelle Approche pour la Modelisation et le Rendu d'Images Réalistes.* PhD thesis, Université Paul Sabatier - Toulouse - France, April 1996.

[Tau94] Gabriel Taubin. Rasterizing Algebraic Curves and Surfaces. *IEEE - CGA*, pages 14–23, March 1994.

[Wyv94] Brian Wyvill. Explicating Implicit Surfaces. In *Proceedings of Graphics Interface '94*, pages 165–172, Banff, Alberta, May 1994. Canadian Information Processing Society.

[YCK92] Roni Yagel, Daniel Cohen, and Arie Kaufman. Discrete Ray Tracing. *IEEE - CGA*, 12(5):19–28, 1992.