Visualizing Remote Sense Depth Maps using Voxels

Nilo Stolte
stolte@dcs.kinsgton.ac.uk

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

Abstract

This article presents a new method for visualizing remote sense depth maps by using voxels. Voxels have been introduced in the 80's to accelerate Ray Tracing, a very time consuming rendering algorithm. With the advent of new technologies for obtaining volumetric data in medical images, voxels were immediately accepted for their representation. New kinds of algorithms were created for visualizing this kind of data. This new research area opened the door for completely new concepts and the idea of discrete graphics has flourished. However, the huge amounts of memory as well as high processing times generally necessary for accomplishing discrete graphics have practically discarded it as a practical visualization tool.

New techniques to represent high-resolution volumes using hierarchical data structures and visualization techniques able to efficiently skip over empty space significantly changed this panorama. This is the ideal arena for normal Computer Graphics surfaces to be visualized in the voxel format. The visualization of remote sense depth maps can also benefit of these techniques and profit of a number of advantages over traditional graphics.

Key Words: Voxel, Voxelization, Implicit Surfaces, Octree, Visualization.

1 Introduction

Voxels are 3D versions of pixels and have been used fairly commonly by the Computer Graphics community. They have been used mainly for accelerating time consumming visualization algorithms such as ray-tracing and radiosity [Fujimoto *et al.*, 1986, Glassner, 1984, Snyder and Barr, 1987, Jevans and Wyvill, 1989, Endl and Sommer, 1994]. A new application for the use of voxels has arrived when 3D medical imagery such as CT and MRI started to appear. These data can be easily obtained in the voxel format. A whole new field in visualization then started to consolidate itself, namely "Volume Rendering" (also known as "Volume Visualization") [Kaufman *et al.*, 1993]. The main problem in this area are the high rendering times necessary to render volumes, basically based on the ray-casting algorithm. In between standard Computer Graphics and Volume Rendering another technique called "Discrete Ray-Tracing" showed up [Yagel *et al.*, 1992]. It basically works as Ray-tracing but entirely in a voxel volume. Recent techniques [Stolte and Caubet, 1995] have been introduced to accelerate the rendering time of discrete scenes mostly containing empty space. This is the ideal arena for normal Computer Graphics primitives to be visualized in the voxel format. This requires a conversion from analytic surfaces to voxels, process called "voxelization" [Kaufman, 1987a, Cohen and Kaufman, 1990, Kaufman, 1987b]. Several kinds of surfaces can be converted to voxels, including one the most modern modeling paradigm: implicit surfaces [Kalra and Barr, 1989, Duff, 1992, Taubin, 1994, Stolte and Caubet, 1997].

Remote sense depth fields are basically terrains, that can be seen as surfaces. This kind of data can be immediately mapped into voxels and it can be easily visualized using the techniques explained in the former paragraph. The big advantage of using voxels in this context is that each "pixel" from the remote

sense depth maps can be stored directly in each voxel. This eliminates the problem of texture mapping. Also clipping parts of the terrain to add new features is not necessary; the voxels ocupying the area corresponding to new feature can be simply deleted and the voxels of the new feature superposed. New features can be all sort of objects, such as synthetic objects (i.e. objects that have been voxelized).

The use of voxels to represent remote sensing depth maps is much more natural and makes manipulation of the data as simple as drawing objects on the screen. New techniques to visualize these kind of voxels models (surface-based voxel models) interactively, using GL/OpenGL points, enables stereoscopic visualization and manipulation using virtual reality techniques.

2 Voxels storage: Octree

High-resolution 3D grids are essential for good quality representation and rendering in discrete graphics. The same is valid for 2D screens. The image quality and the representation will be better in higher resolutions than in low resolutions. Few years ago, it was very uncommon to find high resolution frame-buffers in an ordinary computer. Nowadays, this situation has changed because the price of the memory has drastically dropped.

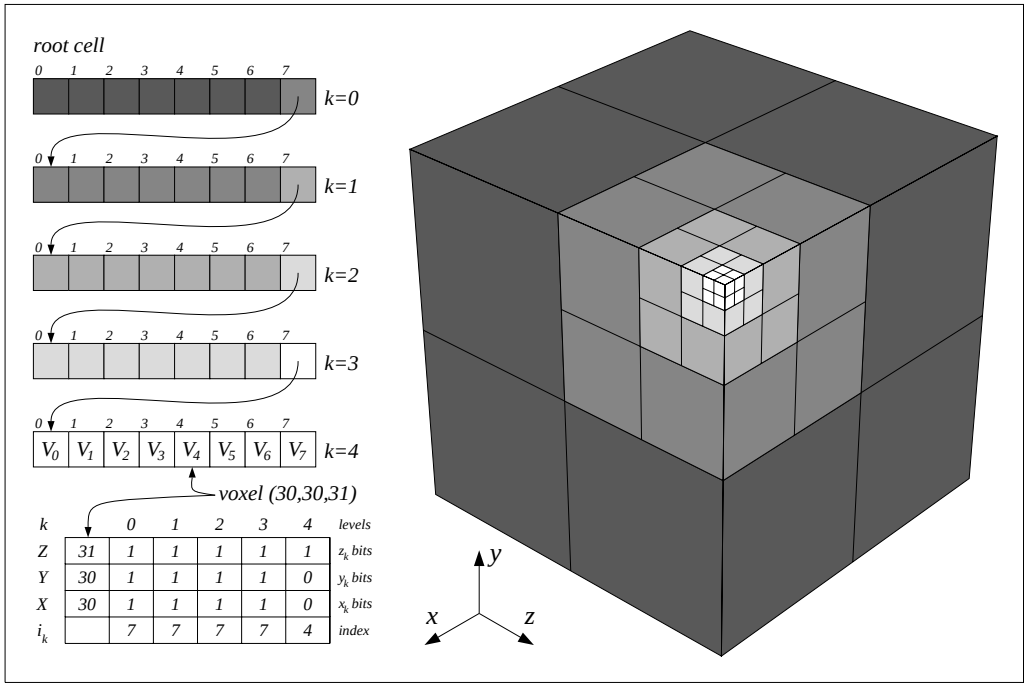

Figure 1: *Left: Octree in the memory; Right: correspondent volume in the space*

However, three-dimensional frame buffers consume much more memory than 2D frame buffers. For example, a 1024^3 3D grid is equivalent to 1024 1024^2 frame buffers. Therefore, the memory consumption for a 3D grid can be thought as 3 orders of magnitude greater than an equivalent 2D buffer. Clearly, ordinary computers can still not support 3D grids of such high resolutions. Thus, it is not unusual to find fully or partially compressed volumes to compensate the enormous memory requirements. This problem resembles that of real time video image sequences, which require some kind of compression. Compression algorithms are very time consuming and for real-time video they are generally implemented in hardware. Unfortunately, there is no such available hardware to enable high-resolution discrete graphics in today's machines.

In 2D images only the color and/or intensity stored in the pixel is enough to precisely describe one image. Three-dimensional data sets also generally store the intensity/color in each voxel. However, this is not enough to completely describe the 3D volume for rendering purposes. In the rendering process the

illumination is calculated using the normal vectors.

Equipments generating medical data sets generally deliver convoluted volumes. In these conditions, it is easy to approximate the normal vectors using a technique called *central difference*. In this technique, the voxel intensities are considered as implicit function values and the normal is calculated by applying the gradient to the function. Since the function is not known, the gradient is calculated by applying the partial derivatives definition. Nevertheless, the derivatives calculated in this way are correct only when the limit of the size of the voxel tends to zero. This would imply an infinite resolution volume. Obviously this is not possible since the voxel size is always greater than zero because of the finite resolution of the volume. Therefore, the normal vectors calculated in that way will be always approximate.

Another difficulty is that the normal vectors must be normalized in order to be useful in the rendering process. This demands an additional amount of time in the rendering process or extra memory to store the normalized normal vector in the voxels. Thus, the already very high memory requirements result completely unpractical if the normal vectors are stored in the volume.

Our solution is to represent in high resolutions only the part of the volume where the surface transitions take place. This can be accomplished by using an octree. In fact, the octree allows in this way to compress the volume, assuming that most of it is empty, which is generally the case in most common scenes. An interesting feature, though, is that the interior part of the volumes in this representation is known as opposed to polygon representations. Polygon representations (polytopes) do no allow the representation of their interior space. This can be seen as a significant disadvantage of polytopes.

The compression obtained with the octree allows the normal vectors storage in the voxels and at the same time very high resolutions in normal worksations.

Figure 1 illustrates the octree implementation representing an octree with 5 levels (diagram on the left side) and its visualization in the space (on the right). In practice, an octree with 9 levels is usually used, thus defining a volume of 512^3. In Fig. 1 five cells, one for each different level, are shown. Each cell has eight elements, all representing the eight equally sized subdivisions of the cell. At the last level (level k, such that $k = 4$), each element corresponds to a voxel of a 3D grid representing a $2^5 \times 2^5 \times 2^5$ volume. These voxels can be noted by their coordinates, (X, Y, Z), as follows: $V_0(30, 30, 30)$ - the only one not visible in the 3D representation-, $V_1(31, 30, 30)$, $V_2(30, 31, 30)$, $V_3(31, 31, 30)$, $V_4(30, 30, 31)$ -indicated in the figure-, $V_5(31, 30, 31)$, $V_6(30, 31, 31)$, $V_7(31, 31, 31)$.

The order of the voxels in a cell is the same as adopted in [Fujimoto *et al.*, 1986]. The definition of this order for an octree of n levels is given as follows. This octree represents a volume of $2^n \times 2^n \times 2^n$ resolution. Each coordinate X, Y, Z of a voxel in this volume is a binary number defined by an accumulation of powers of two. Let k be an index of a bit in the binary coordinate (where $k{=}n{-}1$ is the rightmost bit, and $k{=}0$ is the leftmost bit). Then, the index of an element in a cell is given by the following formula: $i_k{=}x_k{+}2{\cdot}y_k{+}4{\cdot}z_k$, where:

$$X = \sum_{k=0}^{n-1} x_k \cdot 2^{(n-1)-k} \qquad Y = \sum_{k=0}^{n-1} y_k \cdot 2^{(n-1)-k} \qquad Z = \sum_{k=0}^{n-1} z_k \cdot 2^{(n-1)-k}$$

$$0 \leq X \leq 2^n - 1 \qquad 0 \leq Y \leq 2^n - 1 \qquad 0 \leq Z \leq 2^n - 1$$

$$x_k\,, y_k, z_k \in \{0, 1\}$$

It is clear by this definition the relationship between the voxels' binary coordinates and the levels of the octree. Each bit correspond to one octree level and the combination of the three bits of each coordinate in a level k forms the index of the element in each cell of the octree at the level k. This relationship is fully exploited in the implementation of the octree generation presented in the next section.

3 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

```
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

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 [Stolte and Caubet, 1995].

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 Visualizing the octree using Smart Points

The visualization method used to generate images for this article is based on high-resolution voxel spaces. 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 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. 3. 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 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.

```
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);                   /* Descend1 */
           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 3: *Visualization Algorithm*

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.

5 Voxelization

To create the octree, the objects need first to be converted to voxels. This process is called voxelization. There are voxelization algorithms for various kinds of objects: polygons, parametric surfaces and implicit surfaces.

These objects can be integrated with remote sense depth maps in the same voxel volume. The depth maps also have to be converted to voxels. This conversion, however, is straightforward.

Depth maps are normally composed by two files. The two files represent a matrix of pixels of a satellite image. One of the files is the image itself that is normally used as a texture to be applied in the model. The other file is another matrix of pixels of the same resolution as the previous file. However, instead of representing the color/intensity of the pixel it contains the height this pixel has in relationship to the ground. In this way, each pixel is really a voxel, where the X and Y coordinates are given by the image's lines and columns and the Z coordinate is given by the pixel indexed by X and Y.

In this sense, there is no conversion, since depth maps are already voxels, just represented in a different way. However, difficulties exist. The Z coordinates of the depth map can change very quickly from one pixel to another. If just only one voxel would be generated for each map's pixel, holes could show up in the voxel model. For this reason, for each pixel in the map, all the eight neighbour pixels must be analysed. If the neighbours' heights are never more than one unit different than the pixel's height, then only one voxel is generated for that pixel. Otherwise, new voxels have to be piled under and/or over the original pixel to fill the height gaps.

The normal vectors have to be calculated by central difference as explained in section 2. The voxel can also store the texture image given by the first depth map file.

6 Conclusion

This article has shown several techniques to allow visualizing remote sense depth maps using voxels. The visualization technique can be also used for visualizing other kinds of objects. The objects have to be first converted to voxels to be visualized. It is shown that the conversion from depth maps to voxels is straightforward. The texture file from the depth map can be directly stored into the voxels. Therefore, no special care is necessary on how the texture mapping will be handled. The approach presented here is highly appropriated for visualizing and manipulating depth maps, since they are basically surfaces. The underground information can be stored in the octree using no extra memory.

An intrinsic advantage of this approach is that objects can be manipulated or edited as in a drawing program on a 2D screen. The volume is a 3D screen that behaves exactly as the 2D counterpart.

Hence, voxels are a much more natural way to represent depth maps. At the same time high quality visualization is achieved at interactive speeds. LCD glasses can be easily integrated in the system for virtual reality applications.

References

[Cohen and Kaufman, 1990] D. Cohen and A. Kaufman. 3D Scan-Conversion Algorithms for Linear and Quadratic Objects. *Volume Visualization*, pages 280–301, 1990.

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

[Endl and Sommer, 1994] Robert Endl and Manfred Sommer. Classification of Ray-Generators in Uniform Subdivisions and Octrees for Ray Tracing. *Computer Graphics* forum, 13(1):3–19, March 1994.

[Fujimoto *et al.*, 1986] Akira Fujimoto, Takayaki Tanaka, and Kansei Iwata. ARTS: Accelerated Ray Tracing System. *IEEE - CGA*, 6(4):16–26, 1986.

[Glassner, 1984] Andrew S. Glassner. Space Subdivision for Fast Ray Tracing. *IEEE - CGA*, 10(4):15–22, 1984.

[Jevans and Wyvill, 1989] David Jevans and Brian Wyvill. Adaptative Voxel Subdivision for Ray Tracing. In *Proceedings of Graphics Interface '89*, pages 164–172, Toronto, Ontario, June 1989. Canadian Information Processing Society.

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

[Kaufman *et al.*, 1993] Arie Kaufman, Daniel Cohen, and Rony Yagel. Volume Graphics. *IEEE Computer*, 26(7):51–64, July 1993.

[Kaufman, 1987a] A. Kaufman. An Algorithm for 3D Scan-Conversion of Polygons. In *Eurographics'87*, pages 197–208, Amsterdam, August 1987. North Holand.

[Kaufman, 1987b] Arie Kaufman. Efficient Algorithms for 3D Scan-Conversion of Parametric Curves, Surfaces, and Volumes. *Computer Graphics*, 21(4):171–179, July 1987.

[Snyder and Barr, 1987] John Snyder and Alan Barr. Ray Tracing Complex Models containing Surface Tesselations. *Computer Graphics*, 21(4):119–128, 1987.

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

[Stolte and Caubet, 1997] 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.

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

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