7 The Flipping Cube: A Device for Rotating 3D Rasters

Roni Yagel

ABSTRACT Driven by the prospect of three-dimensional rasters as a primary vehicle for future 3D graphics and volumetric imaging, this paper introduces an architecture for real-time rendering of high-resolution volumetric images. The Flipping Cube Architecture utilizes parallel memory organization and a unique data orientation scheme in order to support contention free access to viewing rays.

7.1 Introduction

The swift advances in performance, availability and price of computing power, memory, and disk-space are transforming long thought techniques into reality. One typical example to this trend is the revolution taking place in the field of volume graphics. Grasping the feeling of this revolution requires no more than three decades of perspective. The display of computer graphics in the sixties was based on vector drawing devices and an 'object based' approach to scene representation and rendering. A symbolic representation of the scene objects was stored in a display-list and managed by the computer. Refreshing the screen was accomplished by rendering the vectors comprising the objects in the display-list. An alternative approach, termed *raster graphics* utilizes a 2D frame-buffer of pixels for scene representation and a 'raster based' renderer. This renderer colors the pixels in the frame-buffer that correspond to the discrete representation of the symbolic objects. Screen-refresh is performed by re-displaying the frame-buffer on the screen, thus there is no need to save the symbolic object representation.

The main disadvantage of the raster-based approach is the memory and processing power it requires, which delayed its appearance until the early seventies when the technology was able to provide cheap and fast hardware to support it. The major advantages of vector-graphics are its ability to perform object related operations on the display-list and the fact that the lines it draws are continuous (i.e., without aliasing). On the other hand, the main appeal of raster-graphics is that it decouples rendering from screen-refresh which makes it insensitive to the scene complexity. It is also able to perform block operations, and it is suitable for the display of sampled images, that is, graphics and digitized images can be intermixed. Moreover, the 'object based' approach can still be imitated by maintaining a display-list which is redrawn to the frame-buffer as a result of each change in the scene.

The above discussion describes the state-of-the-art in 2D graphics. In 3D graphics, however, the ruling majority of rendering methods still employ an 'object based' approach called *surface graphics* rather than employing a three dimensional 'raster based' approach. Traditional 3D surface graphics (pay attention to the striking resemblance to 2D vector graphics) represent the scene as a set of surface primitives kept in a display-list. Any change to the scene, viewing parameters, or rendering parameters requires re-rendering this list of surfaces. The 3D equivalent to 2D raster graphics is *volumetric graphics* which utilizes a 3D raster for 3D scene representation by coloring the voxels that correspond to the discrete representation of the scene objects [5].

The same appeal that drove the transfer of the graphics world from vector graphics to raster graphics, once the hardware and processing power became available, is already driving the migration of a variety of applications from 3D surface graphics to volume graphics. Examples of this trend started to appear in the beginning of the seventies in applications involving 3D rasters (volumes) such as in medical and scientific visualization [11]. These fields still provide most of the applications for volume visualization [22, 24]. The obvious advantages are recently attracting also traditional surface-graphics-based applications such as rendering fractals [12, 21], rendering gaseous phenomena [7], modeling and rendering of growth processes [10], rendering complex objects containing surface tessellation [23, 3], constructive solid geometry and CAD [28, 5, 27], and applications that benefit from the intermixing of the two approaches in medicine [16, 19], cell biology [28], and molecular biology [9].

The major drawback of the volumetric approach to computer graphics is the memory and processing power it requires. Viewing of a moderate size volume requires the solution of the hidden voxel removal problem for millions of voxels. While we are certain memory will become available in several years, volume rendering requires the realization of special purpose hardware, a volume engine, that will serve as the volumetric counterpart of polygon engines available today for surface graphics. The design of a system with a real-time interactive capabilities must employ high performance hardware based on multiprocessing and parallel memory organization.

In the late 1980's, we witnessed the appearance of several special-purpose voxel based architectures for volume visualization. The *Cube* architecture [14] is based on simultaneous processing an axial beam of voxels. The octree based *Insight* system [20] performs recursive back-to-front (BTF) projection. The *PARCUM* system [13] is based on interleaved memory that allows parallel fetch of a macro voxel. Hidden voxel removal is performed by fetching macro voxels in a "ray-casting" fashion combined with Z-buffer based hidden-voxel inside each macro voxel. The *Voxel Processor* architecture [8] is based on the partition of voxel space into equal sub-cubes. Each sub-cube is projected by applying recursive BTF and the multiple 2D mini-pictures are merged into a final image.

The spreading recognition of the importance of volume visualization drove numerous manufactures to add volume rendering capabilities to their general purpose graphics engines. The reader is referred to [15] for a broader comparative survey of both special and general purpose architectures for volume rendering. This paper describes a memory organization for volumetric graphics that can support real-time rendering of high-resolution volumes. The next section provides definition of terms that are used in this paper while the following sections describe the principles of the Flipping Cube architecture.

7.2 Nomenclature

Let $G \subset Z^3$ be a finite set of the integral grid points

$$G = \{ (x,\ y,\ z) \mid X_{min} \le x \le X_{max},\ Y_{min} \le y \le Y_{max},\ Z_{min} \le z \le Z_{max} \}$$

for some integers $X_{min},\ X_{max},\ Y_{min},\ Y_{max},\ Z_{min},\ Z_{max}$.

We define a *Y-slice at k* and a *Z-slice at k* in a similar way. We define an *X-beam at* $(k_y,\ k_z)$ as the set of voxels

$$\{ (x,\ y,\ z)\ y \equiv k_y,\ z \equiv k_z \}$$

In a similar way we define a *Y-beam at* $(k_x,\ k_z)$ and a *Z-beam at* $(k_x,\ k_y)$. We define an *X-beam at* $(k_y,\ k_z)$ as the set of voxels

$$\{ ((x,\ y,\ z)\ y \equiv k_y,\ z \equiv k_z \}$$

In a similar way we define a *Y-beam at* $(k_x,\ k_z)$ and a *Z-beam at* $(k_x,\ k_y)$. We define the *volume faces* as the six slices: X slices at X_{min} and X_{max}, Y slices at Y_{min} and Y_{max}, and Z slices at Z_{min} and Z_{max}.

We define three types of neighborhoods of the voxel $\nu = (x, y, z)$ which are denoted by $N^6(\nu)$, $N^{18}(\nu)$, and $N^{26}(\nu)$. They are defined as follows:

$$N^6(\nu) = \{ \ (x,y,z),\ (x \pm 1, y, z),\ (x, y \pm 1, z),\ (x, y, z \pm 1) \ \}$$

$$N^{18}(\nu) = \left\{ \ N^6(\nu) \cup (x \pm 1, y \pm 1, z),\ (x, y \pm 1, z \pm 1),\ (x \pm 1, y, z \pm 1) \ \right\}$$

$$N^{26}(\nu) = \left\{ \ N^{18}(\nu) \cup (x \pm 1, y \pm 1, z \pm 1) \ \right\}$$

in 2D there are only two types of neighborhood $N^4(\nu)$, and $N^8(\nu)$ defined as follows:

$$N^4(\nu) = \{ \ (x,y),\ (x \pm 1, y),\ (x, y \pm 1) \ \}$$

$$N^8(\nu) = \left\{ \ N^4(\nu) \cup (x \pm 1, y \pm 1) \ \right\}$$

We use the letter ξ to stand for the three possible 3D neighborhood types (i.e., 6, 18, 26) and the two 2D neighborhood types (i.e., 4, 8).

An *equality relation* e is defined as $e\ :\ D \times D -> \{0,\ 1\}$. Two voxel values are said to be *equal* if they are "equal" in some sense appropriate to the application. For example, in medical imaging, e may equalize all voxels representing the same tissue type, same organ etc. Usually e is an equivalence relation in which the equivalence classes are called materials, tissue types, densities depending on the application.

We say that a voxel μ is *ξ-adjacent* to the voxel ν iff $\mu \in N^\xi(\nu)$. A set of voxels P is called a *ξ-path* if

$$P = \left\{ v_0, \ldots, v_n \mid e(v_i,\ v_{i+1}) \textbf{ and } v_{i+1} \in N^\xi(v_i),\ i = 0, \ldots, n-1 \right\}$$

Two voxels μ and ν are *ξ-connected* if there exists a ξ-path P such that $\mu = v_0$ and $\nu = v_n$. A *ξ-curve* is an ξ-path P that either contains a single voxel or that each of

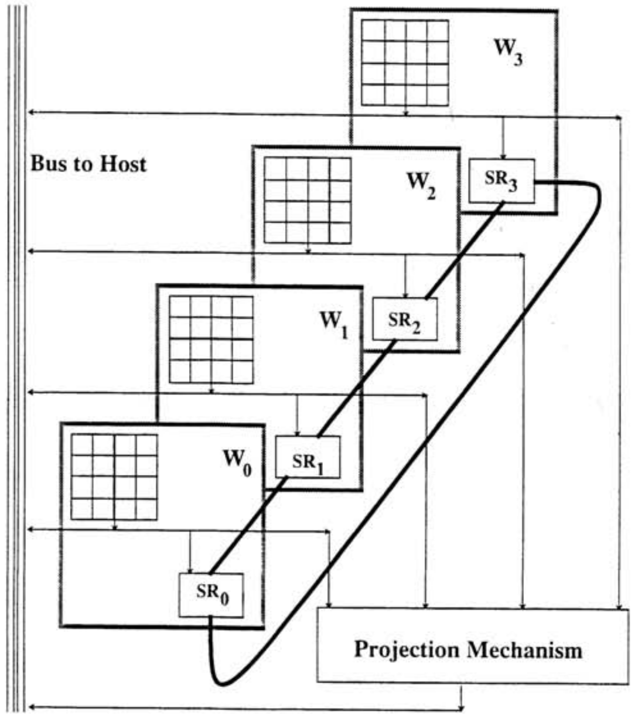

FIGURE 7.1. Block diagram of the Flipping Cube Architecture.

its voxels has exactly two ξ-adjacent voxels in P. If the curve is *open* then it has two exceptions called the *endpoints*, each of which has only one ξ-adjacent voxel in P. A ξ-*line* is an open ξ-curve such that all its voxels are pierced by the straight line that passes through the center of its two endpoints. A ξ-*ray* is a ξ-line such that its two endpoints belong to the volume faces.

We define the X-*distance* (denoted by S_x) between the coordinates (x_0, y_0, z_0) and (x_1, y_1, z_1) to be $|x_1 - x_0|$ (in a similar way we define the Y-*distance* and the Z-*distance*, denoted by S_y, S_z respectively). It can be shown that a 6-line from (x_0, y_0, z_0) to (x_1, y_1, z_1) has exactly $S_x + S_y + S_z$ voxels. This size is also called the *6-distance*, *taxi-driver distance*, or *Manhattan distance* between the two points. It can be shown that the *26-distance* between the two points, that is, the minimal number of voxels traversed by a 26-line between these points, is $max(S_x,\ S_y,\ S_z)$. The *step* between (x_0, y_0, z_0) and (x_1, y_1, z_1) is is the tuple $x_1 - x_0,\ y_1 - y_0,\ z_1 - z_0$. The *form* of a path is the sequence of steps between consecutive voxels in the path.

If L is a ξ-line such that, without loss of generality, $S_x \geq S_y \geq S_z$ we say that X is the *major axis* of L, Y is the *median axis* of L, and Z is the *minor axis* of L. It is easy to see that a 26-line in which, without loss of generality, X is the major axis, does not contain two voxels having the same X coordinate.

7.3 The Flipping Cube Architecture

The Flipping Cube Architecture is composed of four major components (see Figure 7.1): memory modules, bus for host/memory communication, shift mechanism for moving information between the memory modules, and a ray-projection mechanism. The main operation performed by the system is a parallel fetch of all voxels comprising a projection ray and their rapid processing by the projection mechanism in order to determine the final color of the pixel from which the ray was cast.

Memory. The memory is regarded by the user as a three dimensional storage of $N \times N \times N$ voxels. We also assume that $N = 2^n$ for some n which simplifies the hardware implementation, and we use the letter M to stand for the value of $N-1$. The axes of the cubic memory are denoted by U, V, and W. In the following discussion we use left-hand coordinates where positive rotation along an axis is defined to be a *clockwise* rotation when looking from the positive direction to the origin (see Figure 7.2). The data can be read into the memory in 48 different ways[1] but we will be interested only in three of the six orientations in which the origin of the data coincide with the memory origin (see Figure 7.3). When data is read in such a way that the Z dimension (of the data) extends along the W axis (of the memory) we say that the data is *Z-parallel* (and the same for X and Y) (see Figure 7.2).

The cubic memory is partitioned into N modules each of which is capable of storing N^2 voxels. Modules are named $W_0, W_1, \ldots, W_M$ where module W_k stores the *W-slice at k* (see Figures 7.1, 7.3). The addressing of a voxel is straightforward, that is, (i,j,k) is at address (i,j) in module W_k.

Bus. The memory sub-system is accessed by the host via a bus which can hold a 3D memory address and few data items. The bus is also the channel transferring commands from the host computer to the Flipping Cube system. Commonly, the host performs global computations and initializations, and broadcasts the results on the bus.

Shift Mechanism. The only way memory modules communicate with each other is via a shift mechanism (see Figure 7.1). Each memory module can place a voxel data in its own register which is part of the shift mechanism. The shifter, comprised of this set of registers, is capable of rotating the set of N voxels (one from each memory modules) arbitrary number of steps. Then, each memory module can read from its shift-register the (new) voxel data found there after the rotation. Although the shift-mechanism seems as a primitive and restrictive communication paradigm, the fact is that the system does not require a more sophisticated mechanism. Moreover, the restrictiveness of this communication paradigm makes it suitable for our synchronous SIMD approach and allows a simple yet efficient hardware implementation, such as the conveyor [2, 4].

[1] The origin of the data can be placed in one of eight corners of the memory cube. For each such placement we can choose one of the three axes to hold X, then there are two axes left choose from for the Y axis.

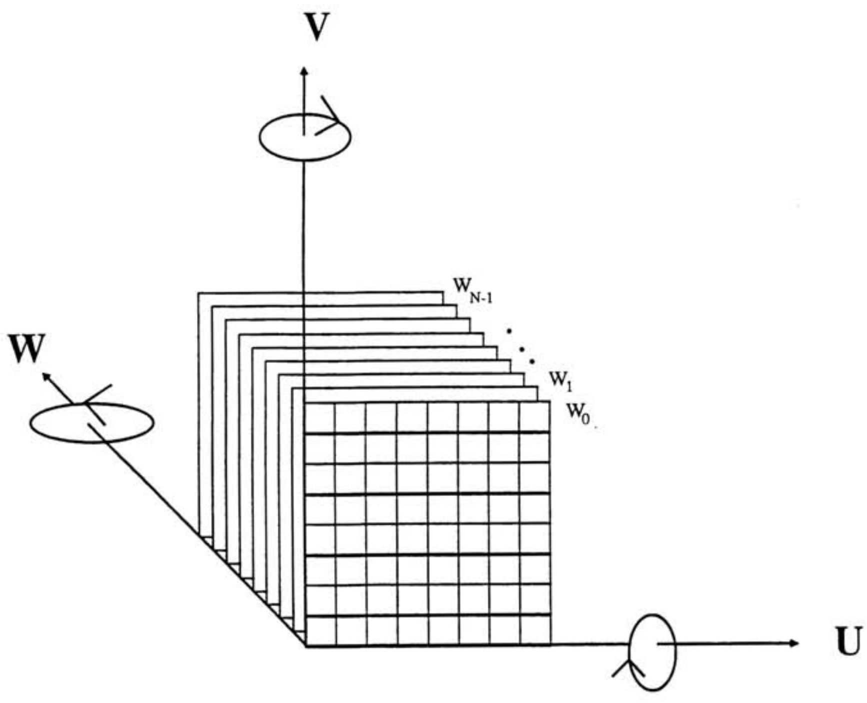

FIGURE 7.2. Memory modulation and axes notation.

Projection Mechanism. The system is equipped with a mechanism to perform fast projection of a ray of voxels. This projection mechanism accepts N voxels, one from each module, and performs hidden voxel removal. The exact method of hidden voxel removal may vary from traditional first/last opaque projection [14], through maximum/minimum projection [25], weighted additive projection [18], and semitransparent composition [6]. An example of a first/last opaque projection mechanism is the voxel multiple-write bus (VMWB) used in the Cube architecture [14].

Each module at each memory cycle can either read a voxel data from its shift register or the bus into the memory, write a voxel from the memory onto the bus or the shift register, or send a voxel from the memory to the projection mechanism. Except for writing to the bus, which is performed by a single module at a time, all other operations are performed by all modules simultaneously. Therefore, multiple access to the bus is undesired since the bus constitutes a major communication bottleneck in the system.

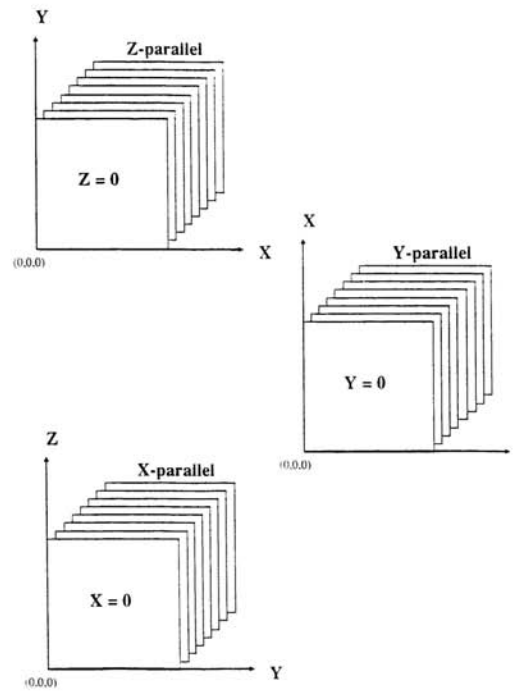

FIGURE 7.3. The three data parallelism schemes.

7.4 Parallel Projection in Flipping Cube

The slice-oriented memory modularization allows conflict free parallel access to N voxels - one from each module. This means that we can extract a ray of voxels in parallel as long as we do not require more than one voxel from each *W-slice*. Thus, providing an orthographic projection parallel to the W axis is trivial; to project a *W-beam at* (u, v), each module places on the projection mechanism the voxel at (u, v). Moreover, as mentioned in Section 7.2, 26-rays have a unique metric in which the length of a line extending form (x, y, z) to $(x + \Delta x, y + \Delta y, z + \Delta z)$ is equal to $max(|\Delta x|, |\Delta y|, |\Delta z|)$ voxels. We also recall that the line's longest dimension is said to be its *major* dimension. In Section 7.2 we concluded from the 26-line metric that along its major dimension a line has at most one voxel in each coordinate. Therefore, we realize that in the slice-wise memory modularization along W, as in the Flipping Cube, we can access a 26-ray in parallel as long as W is its major axis. A parallel projection in which rays are W-major is called *W-major projection* (and the same for U and V). In contrast to parallel projection, in perspective viewing or recursive ray tracing we may traverse rays of all three majority types.

Assuming that data is now in *X-parallel* form, and we need to perform *Y-major projection*, we need to re-orient the data in the memory so that it will become *Y-parallel*. This operation is called *major-switching* and it can be implemented by a mechanism that is based on the *Flipping* operation which is actually a $\pm 90^0$ rotation around an axis.

Denote the $+90^0$ and -90^0 flipping along the W axis by $+W$ *flip* and $-W$ *flip* respectively (and similarly for U and V). Observing all $k = 48$ possible data orientations inside the volumetric memory, it seems that the system has to provide $k(k-1)/2 = 1128$ procedures to perform the switching between each two possible orientations. However, we observe that we may be able to restrict our system and allow only a small subset of all the 48 possibilities in such a way that will provide the basic three parallelism schemes as well as a set of procedures for convenient and efficient switching between them. Carefully considering several options for this restricted subset of orientations (e.g., the six orientations where the origin is fixed, one orientation from each of the basic eight origin placements, all orientations in which a specific data axis is restricted to two specific memory axes) we chose to explore the option consisting of the six orientations in which the origin is fixed and coincides with the memory origin. Another series of experiments with these six orientations made it clear that we can build an efficient system even when restricting the set of allowed orientations (and therefore, number of switching procedures) to three as in Figure 7.3.

A careful examination of these three possible data orientations (see Figure 7.3) shows that in order to switch from any orientation to another, two steps of flipping are sufficient (see Figure 7.4). The first is either a $-V$ flip or a $+U$ flip while the second is a $+W$ flip or a $-W$ flip. It seems that although the flipping mechanism needs to be implemented only in four variations ($-V$, $+U$, $\pm W$), the fact that we restricted ourselves to only three legal orientations implies that major-switching must go through two flipping operations rather than one. However, we observe that a $\pm W$ does not have to be actually performed because of two reasons. First, we observe that in the case $\pm W$ rotation no voxel changes its W coordinate or in other words, it stays in the same memory module. Furthermore, if we examine the position of the voxel at (u, v) after a sequence of $\pm W$ flipping operations we observe that the voxel's new position (denoted by $(u\prime, v\prime)$) is a simple function of its previous position and the desired flipping, as follows:

$$(u\prime, v\prime) = \begin{cases} (u, v) & 0^0 \\ (v, M-u) & 90^0 \\ (M-u, M-v) & 180^0 \\ (M-v, u) & 270^0 \end{cases} , \quad M = N - 1$$

Since we assume that N is a power of two, $M - k$ is equivalent to the bitwise negation of k and therefore:

$$(u\prime, v\prime) = \begin{cases} (u, v) & 0^0 \\ (v, \neg u) & 90^0 \\ (\neg u, \neg v) & 180^0 \\ (\neg v, u) & 270^0 \end{cases}$$

Thus, by adding a mechanism to compute $(u\prime, v\prime)$ as a function of the accumulated W-rotation we can still access any W-ray without contention.

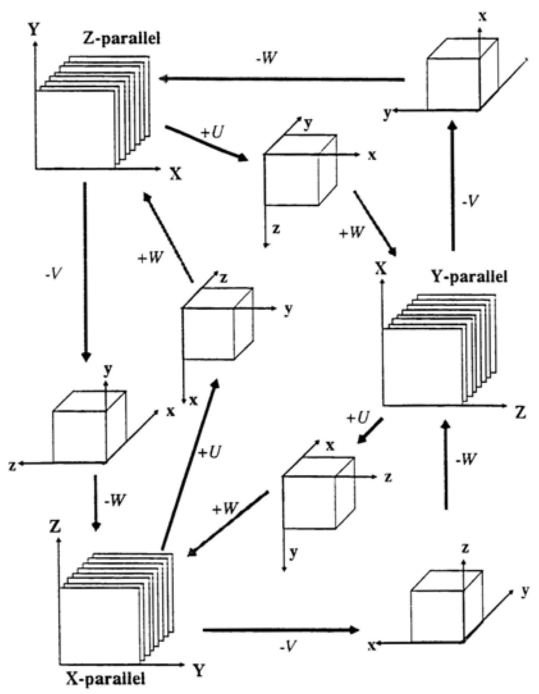

FIGURE 7.4. The set of flipping operations.

In summary, flipping between the three possible data orientations can be achieved by performing only *one* flipping operation (either $+U$ or $-V$) and one $\pm W$ 'virtual flipping' that is implemented by an address mapping mechanism. The following section describes one possible method for the implementation of the $+U$ and $-V$ flipping operations which is based on an additional N^2 voxel memory in a skewed organization. Another method, utilizing a 3D shearing algorithm is described elsewhere [26].

7.5 The Flip-Buffer

The Flip-Buffer is a piece of 2D memory array of N^2 voxels. The Flip-Buffer supports two parallel operations: writing a column of N voxels simultaneously and reading a row of N voxels simultaneously. The flipping algorithm is based on flipping a single $U-$slice or $V-$slice by copying its W-beams into the columns of the flip-buffer and then copying them back from the rows of the Flip-Buffer.

Let us denote by *row_read*$(u,\ v,\ j)$ the operation of reading the j^{th} row of the Flip-Buffer and writing it into the 3D memory as the W-beam at $(u,\ v)$ and denote by

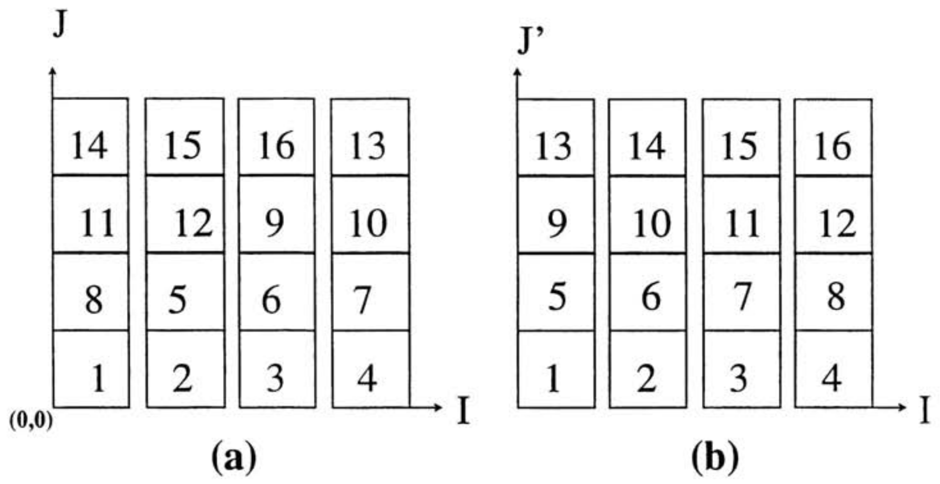

FIGURE 7.5. (a) 2D Skewed memory organization of the values in (b).

column_write$(u,\ v,\ i)$ the operation of reading the W-beam at $(u,\ v)$ and writing it as j^{th} column in the Flip-Buffer (see Section 7.2 for the definition of *beam*). The flipping algorithms can be described as follows (for the $+U$ case on the left, $-V$ on the right) :

```
for $u := 0$ to $M$ do
    for $v := 0$ to $M$ do
        column_write($u,\ v,\ v$);
    end_for;
    for $v := 0$ to $M$ do
        row_read($u, M - v, v$);
    end_for;
end_for;
```

```
for $v := 0$ to $M$ do
    for $u := 0$ to $M$ do
        column_write($u,\ v,\ v$);
    end_for;
    for $u := 0$ to $M$ do
        row_read($M - u,\ v, u$);
    end_for;
end_for;
```

The actual implementation of the Flip-Buffer can be achieved by a special purpose memory such as the Prism [17], however, this requires high linkage complexity – all memory module have to be connected to this auxiliary device. We propose here a solution based on a skewed memory organization that better adapts to the Flipping Cube architecture and requires merely an additional mechanism for a trivial address mapping. The 2D skewed memory is composed of N modules of N voxels each. If each such module holds one column of the 2D Flip-Buffer we can support parallel access to rows (see Figure 7.5(b)). If we rotationaly shift the j^{th} row j steps to the right, we create a skewed organization that supports parallel access to both rows and columns ((see Figure 7.5(a)). The proof of this claim is simple; since only rows where shifted, no voxel changes its row-position which means that we can still access rows in parallel. Since the voxel at $(i,\ j)$ is shifted to $((i+j)\ mod\ N,\ j)$ we observe that the items in the j^{th} column (i.e., $(j,0),(j,1),\ldots,(j,M)$) where mapped to memory modules $(j+0)\ mod\ N, (j+1)\ mod\ N,\ldots,(j+M)\ mod\ N$, that is, to N different modules. In conclusion, we can achieve parallel access to both rows and columns by placing voxel (i,j) of the Flip-Buffer in address $((i+j)\ mod\ N,\ j)$.

If we regard W_k (the k^{th} memory module of the Flipping Cube) as a 2D array of N^2 voxels (see Figures 7.1, 7.3) we can implement the Flip-Buffer by simply adding to W_k another column that holds the k^{th} module of the Flip-Buffer. That is, the item $(i,\ j)$ of the Flip-Buffer can be found in $(N,\ (i+j)\ mod\ N,\ j)$. The fact that the Flip Buffer is part of the memory module is advantageous since there is no need to have N lines running from the N memory module to an auxiliary device. Instead, communication is achieved by means of the shift mechanism as we describe shortly. The implementation of the *column_write*$(u,\ v,\ i)$ operation is achieved in three steps. First, the W-beam at $(u,\ v)$ is placed in the shift mechanism by having W_k copy the voxel $(u,\ v)$ to its shift register. Next, the beam is shifted i steps to the right (assuming W_0 on the left, W_M on the right) and then copied into the memory by having W_k copy the content of its shift register into address $(N,\ (k-i)\ mod\ N)$. Similarly, the implementation of the *row_read*$(u,\ v,\ j)$ operation is achieved in three steps. First, the value at $(N,\ j)$ is placed by W_k in its shift register. Next, the shift mechanism rotates the its values j steps to the left. Finally, W_k copies the content of its shift register into $(u,\ v)$.

In summary, each of the memory modules contains $(N+1)N$ voxels and supports four memory operations: *voxel_read, voxel_write, row_read, and column_write.* Voxel_read and voxel_write are performed on the first N memory columns while and row_read and column_write are performed on the $(N+1)^{th}$ column. This set of operations can be implemented by a simple mapping of the incoming address $(u,\ v)$ into an internal address $(u\prime,\ v\prime)$ computed by:

$$(u\prime,\ v\prime) \quad = \quad \begin{cases} (u,\ v) & voxel\_read \\ (u,\ v) & row\_read \\ (N,\ v) & voxel\_write \\ (N,\ (k-i)\ mod\ N) & column\_write \end{cases}$$

If we denote by t_s the time required for an arbitrary shift and by t_m the time it takes the memory to fetch one voxel, we say that both row_read and column_write require two read/write operations and one shift operation. Since the algorithm performs one row_read and one column_write operation for each beam, we conclude that flipping of the whole volume requires $N^2(2t_s\ +\ 4t_m)$ time units and N^2 additional memory. If we denote by t_p the time it takes for the projection mechanism to project the ray, we can compare the flipping time with the projection time: $N^2(t_m\ +\ t_p)$. This comparison leads us to believe that the flipping operation will not require more than few frame times because $t_s \approx t_p$ and both have complexity of $logN$. We must also keep in mind that this hardly noticeable delay is required only when the viewing direction changes its major axis which itself happens very rarely, for example, when spinning the volume 360^0 we will need only four flipping operations.

Acknowledgements:

This project has been partially supported by the National Science Foundation under grants MIP-8805130 and IRI-9008109, and grants from Hewlett Packard Company and Silicon Graphics. I would like to thank my colleague and friend Dany Cohen for numerous fruitful discussions. Special thanks are due to my advisor Professor Arie Kaufman for his support and advise.

7.6 References

[1] R. Bakalash and A. Kaufman, MediCube: a 3D Medical Imaging Architecture. *Computers & Graphics*, 13(2):151–157, 1989.

[2] R. Bakalash and X. Zhang, Barrel Shift Microsystem for Parallel Processing. In *Proceedings Micro 23, 23rd Symposium and Workshop on Microprogramming and Microarchitecture*, Orlando, FA, November 1990.

[3] J. G. Cleary and G. Wyvill, Analysis of an Algorithm for Fast Ray Tracing using Uniform Space Subdivision. *The Visual Computer*, 4:65–83, 1988.

[4] D. Cohen and R. Bakalash, The Conveyor - an Interconnection Device for Parallel Volumetric Transformations. This volume, Chapter 6.

[5] D. Cohen, A. Kaufman and R. Yagel, Volumetric Graphics. Technical Report TR 91.01.30, Department of Computer Science, SUNY at Stony Brook, January 1991.

[6] R. A. Drebin, L. Carpenter, and P. Hanrahan, Volume Rendering. *Computer Graphics*, 22(4):64–75, August 1988.

[7] D. S. Ebert and R. E. Parent, Rendering and Animation of Gaseous Phenomena by Combining Fast Volume and Scanline A-buffer Techniques. *Computer Graphics*, 24(4):367–376, August 1990.

[8] S. M. Goldwasser, R. A. Reynolds, D. A. Talton, and E. S. Walsh, High Performance Graphics Processors for Medical Imaging Applications. In P. M. Dew, R. A. Earnshaw, and T. R. Heywood, editors, *Parallel Processing for Computer Vision and Display*, pages 461–470. Addison-Wesley, Reading, MA, 1989.

[9] D. S. Goodsell, S. Mian, and A. J. Olson, Rendering of Volumetric Data in Molecular Systems. *Journal of Molecular Graphics*, 7:41–47, March 1989.

[10] N. Greene, Voxel Space Automata: Modeling with Stochastic Growth Processes in Voxel Space. *Computer Graphics*, 23(3):175–184, July 1989.

[11] J. F. Greenleaf, T. S. Tu, and E. H. Wood, Computer-Generated Three-Dimensional Oscilloscopic Images and Associated Techniques for Display and Study of Spatial Distribution of Pulmonary Blood Flow. *IEEE Transaction on Nuclear Science*, NS-17:353–359, 1970.

[12] J. C. Hart, D. J. Sandin, and L. H. Kauffman, Ray Tracing Deterministic 3-D Fractals. *Computer Graphics*, 23(3):289–296, July 1989.

[13] D. Jackel and W. Strasser, Reconstructing Solids from Tomographic Scans - The PARCUM II System. In A. Kaufman, editor, *Volume Visualization*, pages 358–371. IEEE Computer Society Press, 1991.

[14] A. Kaufman and R. Bakalash, Memory and Processing Architecture for 3-D Voxel-Based Imagery. *IEEE Computer Graphics & Applications*, 8(11):10–23, November 1988.

[15] A. Kaufman, R. Bakalash, D. Cohen, and R. Yagel, Architectures for Volume Rendering – Survey. *IEEE Engineering in Medicine and Biology*, 9(4):18–23, December 1990.

[16] A. Kaufman, R. Yagel, and D. Cohen, Intermixing Surface and Volume Rendering. In K. H. Hoehne, H. Fuchs, and S. M. Pizer, editors, *3D Imaging in Medicine, Algorithms, Systems, Applications*, NATO ASI Series, Volume F60, p.217-227, Springer-Verlag, Berlin, 1990.

[17] C. Kornfeld, The Image Prism: A Device for Rotating and Mirroring Bitmap Images. *IEEE Computer Graphics & Applications*, 7(5):21–30, May 1987.

[18] R. Lenz, P. E. Danielsson, S. Cronstrom, and B. Gudmundson, Interactive Display of 3D Medical Objects. In K. H. Hoehne, editor, *Pictorial Information Systems Medicine*, NATO ASI Series, Volume F19, p.459-468. Springer-Verlag, Berlin, 1986.

[19] M. Levoy, A Hybrid Ray Tracer for Rendering Polygon and Volume Data. *IEEE Computer Graphics & Applications*, 10(3):33–40, March 1990.

[20] D. J. Meagher, Applying Solids Processing Methods to Medical Planning. In *Proceedings NCGA'85*, National Computer Graphics Association, Dallas, TX, pages 101–109, April 1985.

[21] A. Norton, Generation and Display of Geometric Fractals in 3-D. *Computer Graphics*, 16(3):61–67, July 1982.

[22] P. Shirley and H. Neeman, Volume Visualization at the Center for Supercomputing Research and Development. In C. Upson, editor, *Proceedings of the Chapel Hill Workshop on Volume Visualization*, pages 17–20, Chapel Hill, NC, May 1989.

[23] J. M. Snyder and A. H. Barr, Ray Tracing Complex Models Containing Surface Tessellations. *Computer Graphics*, 21(4):119–124, July 1987.

[24] C. Upson and M. Keeler, V-BUFFER: Visible Volume Rendering. *Computer Graphics*, 22(4):59–64, August 1988.

[25] J. W. Wallis, T. R. Miller, C. A. Lerner, and E. C. Kleerup, Three-Dimensional Display in Nuclear Medicine. *IEEE Transactions on Medical Imaging*, 8(4):297–303, December 1989.

[26] R. Yagel, *Efficient Methods for Volume Graphics*. PhD thesis, Department of Computer Science, State University of New York at Stony Brook, December 1991.

[27] R. Yagel, D. Cohen, and A. Kaufman, Discrete Ray Tracing, *IEEE Computer Graphics and Aplications*, 12(5):19-28, September 1992.

[28] R. Yagel, A. Kaufman and Q. Zhang, Realistic Volume Imaging, In *Visualization'91 Proceedings*, San Diego, CA, October 1991, pages 226-231.