7

The *voxblt* Engine: A Voxel Frame Buffer Processor *

Arie Kaufman

The *voxblt Engine (vE)* is a 3D frame-buffer processor which manipulates and processes "3D bitmaps" (voxel maps) stored in a *cubic frame buffer* of *voxels*. The *vE* is the 3D counterpart of the 2D frame buffer processor, which is an extended version of the 2D *bitblt* and RasterOps engines.

The primitives of the *vE* are subcubes of the cubic frame buffer and are of three kinds: *rooms* (3D windows), *jacks* (3D cursors), and *figurines* (3D icons). In addition to manipulating these primitives, the *vE* also serves as a monitor for interaction, as an interface for 3D input devices, and as a channel for inputting into the cubic frame buffer 3D voxel images from either 3D scanners or a voxel image database. The *vE* has been developed as part of the CUBE system, in which it operates as an independent processor executing its own commands stored in a 3D frame-buffer display list. A *room manager* , which is the 3D counterpart of the 2D window manager, has been implemented on top of the *vE*.

1. Introduction

In the past five years, some attention has been given to developing architectures and hardware for volumetric processing [2, 3, 6, 7, 9, 10, 12, 13, 15, 19]. This paper focuses on a new breed of volumetric processing, that of a voxel frame buffer processer, i.e., a *voxblt Engine (vE)*. The *vE*, which processes voxel maps, is the 3D counterpart of the 2D frame buffer processor (FBP) [11]. The latter is an extended version of the 2D *bitblt* [4] and RasterOps [18] engines.

Voxel-based systems are based on the concept that the volume of interest is discretised, sampled, and stored in a the *Cubic Frame Buffer (CFB)*, i.e., a full 3D cellular memory of voxels (unit volume cells). Each voxel holds a density value in the broad sense, representing the colour, material, texture, and translucency ratio of a small unit cube of

* This work was supported by the National Science Foundation under grants DCR 8603603, CCR 8743478, CCR 8717016, and MIP 8805130.

the real scene. The stored 3D images are directly manipulated using cellular-map or voxel-map operations, employing a voxel frame buffer processor, such as the *vE*. A typical CFB is a huge memory buffer of the order of 128M bytes for a 512^3 resolution. In addition to the storage requirements, an extremely large throughput has to be supported, and a special architecture and parallel processing is imperative. The voxel representation is especially suitable for empirical data as in medical imaging, biology, geology, and 3D image processing. In addition, this representation is becoming common place in traditional synthesis applications, like CAD, scientific volume visualisation, and 3D simulation and animation.

The *vE* is part of the CUBE architecture [10], a 3D voxel-based graphics system. The CUBE system, depicted schematically in Figure 1, is a multiprocessor system with three processors accessing the CFB, to input, manipulate, and view and render the CFB images.

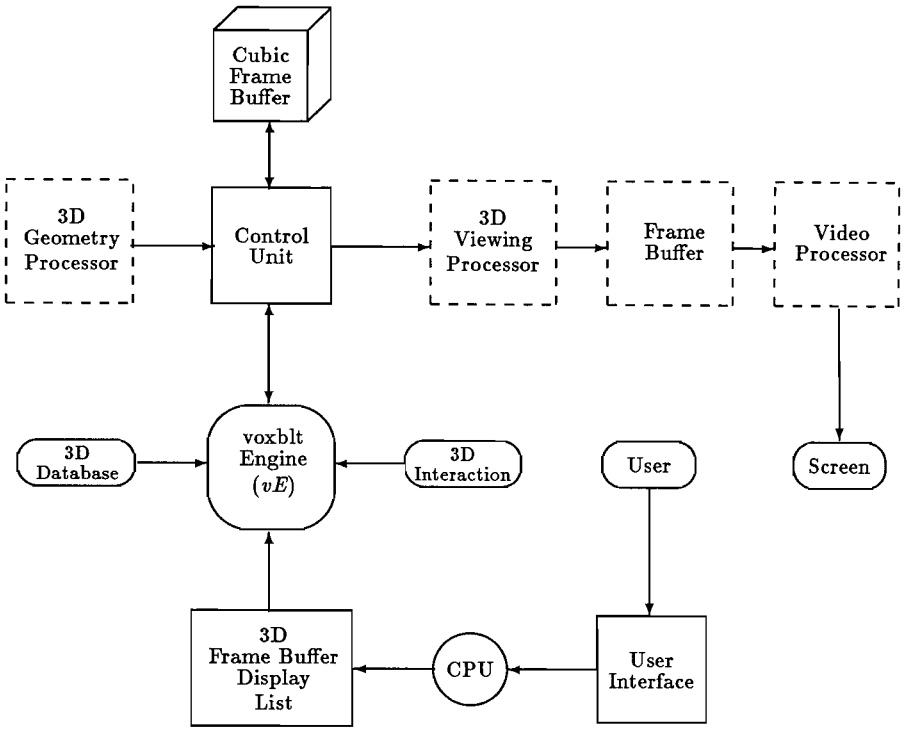

Figure 1: Schematic Diagram of the *vE* as Part of the CUBE Architecture.

The *vE*, which is one of these processors, channels into the CFB empirical/scanned voxel images, which are the primary source of CFB data. A geometric model is another source of CFB data. The 3D Geometry Processor scan-converts synthetic 3D geometric models

into their 3D voxel representation within the CFB, possibly intermixed with the empirical data. Once the images, empirical and/or synthetic, are in the CFB they are manipulated and transformed by the *vE*. The third processor, the 3D Viewing Processor, generates 2D shaded orthographic projections of the CFB image in a conventional 2D frame buffer of pixels. Arbitrary parallel projections can be generated by first transforming the images with the *vE* and then viewing them through a principal orthographic direction.

In the CUBE architecture the *vE* performs part of the operations that conventionally would be performed by a geometry processor, a viewing processor, and a video processor. By doing so it frees the other processors to concentrate on their own tasks. In addition, the *vE* gives hardware support to operations that would have ordinarily been performed in software. This hardware support frees the main CPU from the burden of running in software, for example, the *room manager*, which is the 3D counterpart of the 2D window manager.

2. *vE* Primitives

The *vE* directly manipulates and processes 3D voxel maps, i.e., sub-arrays of the CFB. The primitives of the *vE* are unstructured portions of the CFB of three types: *rooms* (3D windows), *jacks* (3D cursors), and *figurines* (3D icons). Since the CFB is an unstructured 3D array of voxels, these primitives are the only objects with special meaning to the *vE*. The primitives are manipulated according to the *vE* instruction list stored by the main CPU in a 3D Frame Buffer Display List (FBDL3).

The prevailing primitive is the room, which is a 3D rectangular box or a cuboid of voxels. The room is the 3D counterpart of the 2D window. It is defined by two (x, y, z) coordinates defining extreme corners of the room, and a priority value. In case of a space conflict between primitives, the one with the highest priority occupies that common space within the CFB. This feature is useful in supporting three-and-a-half dimensional environment, which is paramount for room management. A room, like its 2D counterpart, defines either a "room" (enclosure) of interest to operate upon, or a 3D extent of an object that can be thresholded or masked, for example, using the *vE* commands and registers.

The jack, which is the 3D counterpart of the 2D cursor, is a small 3D box which provides a visual feedback for 3D graphics interaction. A jack is defined by its edge length and its center coordinates. In addition to the system-defined jacks, the user may define the adequate pattern for each user-defined jack tailored to the application. The 3D jack mechanism enables the user to dynamically tour the exterior or interior of the objects, and improves user orientation in the 3D environment. Traditionally, 2D cursors have been handled by the video processing units. Supporting jacks by the *vE* alleviates the high load already assumed by the video processor and by the 3D viewing processor. While writing a 3D jack into the CFB, the voxel data covered by it is saved within the *vE* for later restoration (when the jack moves from that location). The jack priority is higher than that of the rooms and the figurines, and all jacks have the same priority. Since a 3D jack

might be hidden from the user in one or all views, there is an option to display the projection of the jack on the screen as if the jack floats in the fourth dimension.

The *vE* supports user-defined 3D figurines. This primitive is similar to the room primitive, but it is usually smaller and holds an application-oriented constant image. A figurine is defined by two (x, y, z) coordinates and a priority value. The figurine might be either bounded (anchored) to a room or float independently, and it gets its priority according to this status. A floating figurine behaves like a room and is assigned an independent priority. A bounded figurine, however, is assigned the priority value of the bounding room, and it is affected by all the operations performed on the bounding room.

3. *vE* Tables

The *vE* supports five tables for managing its operations. The tables are designed as integral parts of the processor in order to reduce the access time and to increase the overall speed of the *vE*.

The Room Table (RT) holds information of all the currently defined rooms. Each room has a unique entry in the RT, which is indexed by the room identifier. In addition, the RT holds for each room, its two (x, y, z) coordinates, its priority, and a figurine identifier list indexing all the figurines bounded to this room.

The Jack Table (JT) holds information of all the user-defined and system-defined jacks. Each jack has a unique entry in the JT, which is indexed by the jack identifier. The JT holds the jack center coordinates, its size, its location in the scratch storage area of the JT, and a code specifying whether the original jack pattern or the data that is currently covered by it is in the scratch area.

The Figurine Table (FT) holds information of all the currently defined figurines. Each figurine has a unique entry in the FT, which is indexed by the figurine identifier. The FT holds for each figurine two (x, y, z) coordinates, its status (bound/float) and its priority. For a bounded figurine the FT also holds the room identifier of the bounding room.

The Priority Table (PT) handles room and figurine priorities. Every room and every figurine has an entry in the PT. This entry is defined when a new room or figurine is inserted in the RT or the FT, respectively. The purpose of this table, which is sorted by the priority, is to speed up the overlapping of primitives when performing operations on them, such as clipping. The PT provides the *vE* with a mechanism to rapidly find the priority order between the primitives.

The Device Table (DT) holds information to supports interaction with the 3D input devices. Each input device has a unique entry in the DT which identifies it. The table also holds a reference to the jack, if any, attached to that device.

4. *vE* Registers

Internal registers of the *vE* can be read, set, or reset, and all subsequent operations are consequently affected. The *vE* has three operating states: passive, active, and suspended. The passive state is the initialisation state. The system registers that affect the configuration of the *vE* are alterable only during the initial passive state. All the other registers and register fields thereof are alterable also during the active state, in which the *vE* operations are executed. During the suspended state the CPU handles traps.

The *vE* registers are divided into four main groups: address, mode, system, and input registers. The *address* registers include general purpose address registers, a Stack Pointer (SP) which points to the top of a conventional stack, and a Program Counter (PC) which points to the next executable instruction in the FBDL3.

The *mode* registers are divided into two subgroups. The first subgroup specifies the environment of the system, including the Cfb_Size, Max_Rooms, Max_Jacks, Max_Figurines, colour/translucency mode (Colour_Mode), number of bits comprising each component of a three colour system (Colour_Field), and which colour planes in the CFB are enabled (Depth_Plane). Each mode register is enabled by a specific bit in the Status register.

The second subgroup includes registers which affect the voxel writing into the CFB: zooming control (Zoom), boolean operations on voxels and jacks (Voxel_Ops and Jack_Ops), masks for texture and filtering (Texture_Mask and Filter), Background and Foreground colours, and CFB orientation (Orient), i.e., orientation by 0°, 90°, 180°, or 270° about the primary axes. Any operation performed on the primitives that write voxel information into the CFB, e.g., **copy**, is directly affected by these registers. For example, if the X field of the Zoom register, ZoomX, indicates a zoom of factor 2, any voxel written to the CFB is automatically replicated along the *X* direction. If the Origin register specifies that the CFB X origin is at its X_{max} axis (instead of X_{min}), all subsequent operations into the CFB will be reflected (rotated by 180°) with respect to the X CFB origin.

The *system* registers define the *vE* configuration. They include the State register, which defines the operating state of the *vE* (passive, active, or suspended), and the Initial register which determines the configuration of the system (three fields for CFB *X*, *Y* and *Z* dimensions, and three fields for the maximum number of rooms, figurines, and jacks), and is alterable only during the initial passive state.

Three additional system registers are used for handling errors and traps. The Trap_Enable register has a one bit field for each type of error. When an error occurs, the Trap_Enable register is used to determine if this error should cause a trap, i.e., change the operating state to suspended. The Error register also contains one bit field for each type of error. It is used by the **jumperr** instruction to jump to an appropriate error handling routine based upon the type of error. The Stop register is used to pass the error category back to the CPU. It is set only if an error is detected and trapping is enabled.

The *input* registers define the input environment, assisting in the user-*vE* interaction. In the current implementation two 3D input devices are supported. One is a true 3D device, a Polhemus 3SPACE Isotrak specifying 3D position and 3D orientation, called by us the *kite*. The other is a common 2D mouse used for 3D positioning, by exploiting horizontal, vertical and diagonal moves (see [14]). The Input_Enable and Device_Status registers of each device indicate whether the device is currently enabled, ready, or "done". In addition, there is a Data register for each device which contains the position and orientation information accepted from that device.

5. *vE* Repertoire

The *vE* instruction set contains commands for manipulating the three types of primitives (i.e. rooms, figurines, and jacks) as well as system instructions which affect the overall operation of the system. The large sets of operations for handling rooms, figurines and jacks, and the underlying *vE* registers provide an extremely flexible environment for manipulating the CFB voxel images. All primitives are created and introduced into their tables by **define**, removed by **delete**, "painted" uniformly by **erase**, read from the CFB by **read**, written into the CFB by **write**, moved by **translate**, copied by **copy**, swapped with another primitive by **swap**, and rotated about a major axis by **rotate**. Both rooms and figurines are scalable (**scale**) with maximum size restriction applied only to figurine. The scaling factor can be any real number (see algorithm details in Section 8.2). Rooms can be rotated about any major axis through any angle (see algorithm details in Section 8.3), but the rotation angles for figurines and jacks are restricted to multiples of 90°.

Rooms and figurines, which are assigned priorities, are subject to a change of priority command which may affect future placements of primitives in the CFB. Figurines have an exclusive instruction for bounding/unbounding figurine to/from rooms. An exclusive instruction for jacks is for loading a jack pattern (e.g., from the FBDL3, from the stack) to the JT storage area.

The system instructions of the *vE* include register manipulation, (**set, setf** - set field, **get, getf, reset**), program flow control (**jump, jumperr, call, return**), stack control (**push, pop**), state control (**active, halt**) and input device control (**attach** device, **disattach**).

6. *vE* Internal Architecture

The *vE* is an independent co-processor, which fetches and executes its instructions listed in the FBDL3 stored in the host computer memory by the CPU. The basic hardware organisation of the *vE* consists of four components: The Fetch Unit, for fetching the next instruction to be executed from the FBDL3, decoding it, and fetching data if necessary; the Execution Unit for executing the instructions; the CFB Management Unit for managing the access to the CFB; and the Control Unit for controlling and synchronising the other units.

The *vE* internal organisation can be classified as a pipeline architecture. While fetching an instruction, the previous one is decoded and the one before it is being executed. Since large amounts of voxels require complex and lengthy computations, the execution stage in the pipeline is the bottleneck of the system. Therefore, an emphasis has been placed on efficient and diligent algorithms.

The Execution unit manipulates the CFB images, controls the interaction with 3D input devices and handles the processor tables. Care has been taken to provide simple and fast operations, especially for the large rooms. The Execution unit consists of five sub-units: the Input sub-unit manages the 3D interaction with 3D devices, by polling them at specified intervals. The Geometric sub-unit performs the geometrical operations: translation, scaling, and rotation. These operations employ incremental transformation technique, in which each voxel is transformed with a maximum of 3 additions (see Section 8), and exploit the parallel memory and the barrel shifter (see Section 4). The Clipping and Priority sub-unit handles the clipping of a primitive to other primitives based on their priorities (see Section 9). The Voxblt sub-unit performs voxelwise operations such as filtering, boolean operations and texture masking (see Section 10). The Table sub-unit manages the *vE* tables as an aid to the other Execution sub-units.

The CFB Management unit is responsible for writing and reading single voxels, beams of voxels, or intervals of voxels, to and from the CFB memory. All CFB accesses are channelled through a global control unit monitored by the viewing processor which is the heaviest user of the CFB, arbitrating the access of the other processors to the CFB. The next section describes briefly the memory organisation and its accessing avenues.

7. Memory Parallelism and Barrel Shifting

A unique skewed memory organisation enables all the CUBE processors, including the *vE*, to store and retrieve a full beam of voxels simultaneously [8]. The n^3 voxel memory is divided into n distinct memory modules each having n^2 voxels. A voxel (x, y, z) is mapped onto the k-th module by:

$$k = (x + y + z) \bmod n \qquad 0 \leqslant k, x, y, z \leqslant n-1, \tag{1}$$

where the internal mapping (i, j) within the module is:

$$i = x, \qquad j = y \tag{2}$$

Since two coordinates are constant along any beam, regardless of the view direction, the third coordinate guarantees that only one voxel from the beam resides in any one of the modules.

The physical memory is thus divided into *n* independently accessible memory modules, each with its own local addressing unit, and its own local processing unit (see Figure 2). Each voxel of a beam is fetched from a separate memory module and is processed by the associated processing unit. The processing units are used primarily for selecting the

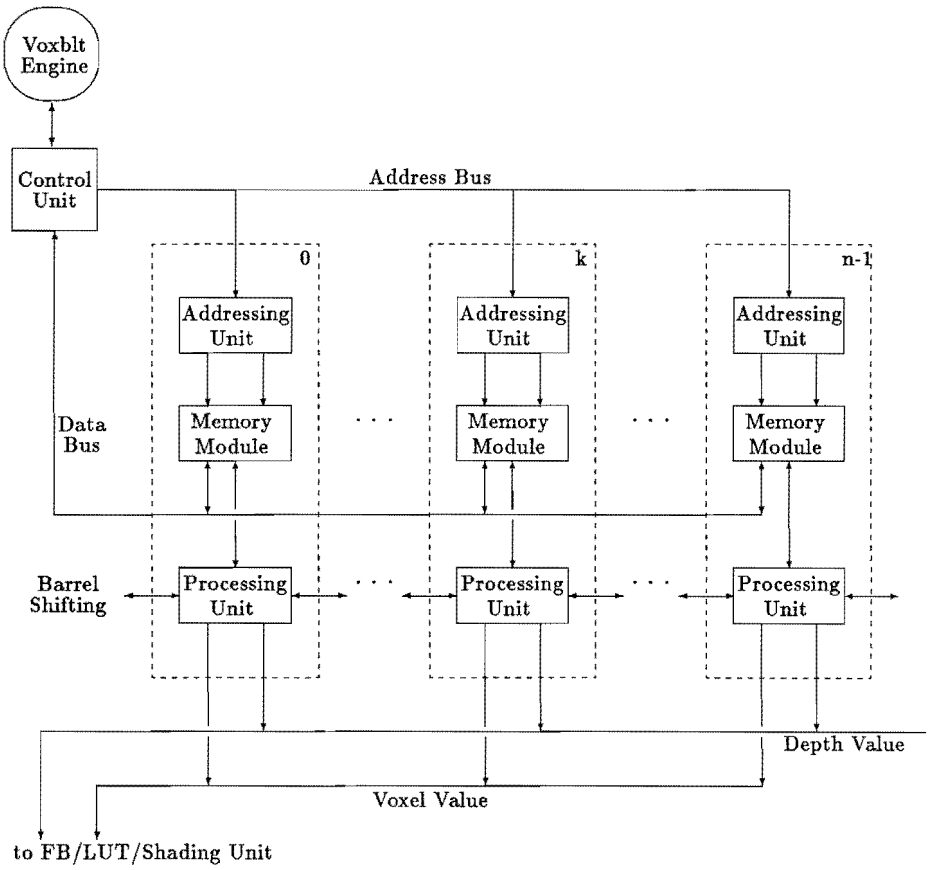

Figure 2: Parallel Memory Organisation with a Barrel Shifter.

opaque voxel closest to an observer in order to generate a projection. In addition, the n processing units are connected through bi-directional links which enable the barrel shifting of the voxel beam across the processing units by a given offset, and then writing it back into the CFB memory modules at a different location modulo n. This is actually a fast memory to memory beam move mechanism. The barrel shifter provides immediate neighbours links as well as "hard-wired" power-of-2 neighbour skips. The number and the extent of the skips depend upon the length of a beam, n, with an attempt to minimise the average shifting time.

The parallel memory access and the barrel shifter are exploited by the *vE* to speed up many of its operations on rooms, on beams, or parts thereof. Specifically, geometric transformations, like translation or 3D rotation about an axis, mirroring, erasing with a

constant colour, etc., are all using these mechanisms. For example, the 3D rotation about the *x*-axis through an arbitrary angle involves taking beams parallel to the *x*-axis and relocating them to the new position. The time to rotate a primitive of n^3 voxels is proportional to $n^2 \log n$, rather than the conventional n^3, where n^2 is proportional to the number of beams and $\log n$ is proportional to the barrel shifting cost. In our implementation, rotating a room of 256^3 voxels about a primary axis using these mechanisms is estimated to take only 45 milliseconds employing immediate neighbour and 8-skip neighbour links.

8. Geometric Algorithms

Special care has been taken to provide simple and fast geometric transformations especially for the large rooms. All transformations, carried out by the Geometric sub-unit of the *vE*, including translation, scaling (anamorphic with non-integral factors), and rotation (through any angle), are performed using incremental transformation techniques, in which each voxel or a voxel beam is transformed, based on the new position assumed by its immediate previous neighbour, with a maximum of 3 additions/increments. The algorithms are based on inter- and intra-scan-line coherency. In other words, after calculating the new location for one voxel/beam, its neighbours relate to it as they did in the original location.

The geometric algorithms also resolve possible overlapping between the source and the destination primitives, by performing the operations in a predefined sequence according to the orientation algorithm (see Section 11). This is necessary since there is no scratch CFB memory available to temporarily hold old voxel information while performing the transformation.

8.1. Translation Algorithm

The **translate** operation moves a primitive from its source position to its destination position. The relative translation is characterised by a vector (T_x, T_y, T_z), and the equation:

$$(X_{dest}, Y_{dest}, Z_{dest}) = (X_{src}, Y_{src}, Z_{src}) + (T_x, T_y, T_z) \quad (3)$$

An incremental algorithm specifies the position of the next voxel along the X-axis to be translated $(X_{src} \pm 1, Y_{src}, Z_{src})$ in terms of its predecessor's destination position $(X_{dest}, Y_{dest}, Z_{dest})$:

$$(X_{src} \pm 1, Y_{src}, Z_{src}) + (T_x, T_y, T_z) = (X_{dest} \pm 1, Y_{dest}, Z_{dest}) \quad (4)$$

Thus, the next voxel position in the destination primitive can be determined by only *incrementing* the appropriate coordinate of the last voxel without performing an addition for every voxel. Similar results apply to Y and Z.

Care must be taken to avoid overlapping, that is, if the destination primitive is partially overlapping with the source primitive, then some of the source voxels might be overwritten

by the copied voxels before the source voxels are translated, which could distort the destination primitive. The current algorithm sets the Orient register so that the translation starts from the source corner that is also situated in the destination primitive range (see Section 11). If no overlap occurs, then the order of translation is not important.

8.2. Scaling Algorithm

The equations for scaling a room about its origin are given in Equation 5. The shifting to and from the origin are accounted for in the CFB addressing of the source and the destination.

$$X_{dest} = X_{src}\, S_x \qquad Y_{dest} = Y_{src}\, S_y \qquad Z_{dest} = Z_{src}\, S_z \tag{5}$$

Like rooms, figurines are also scalable, but with maximum size restriction. Jacks are not scalable.

The scaling algorithm for rooms simultaneously scales a room along the three axes, where the three scaling factors, S_x, S_y and S_z, can all be different and not necessarily integers. Consequently, the scaling algorithm requires, therefore, three non-integer multiplications and three roundings for each voxel, which can result in heavy computations for large rooms. The complexity of the computations can be significantly reduced by using an incremental algorithm which is based upon:

$$(X_{src}+1)\, S_x = X_{src}\, S_x + S_x = X_{dest} + S_x \tag{6}$$

$$(Y_{src}+1)\, S_y = Y_{src}\, S_y + S_y = Y_{dest} + S_y \tag{7}$$

$$(Z_{src}+1)\, S_z = Z_{src}\, S_z + S_z = Z_{dest} + S_z \tag{8}$$

The coordinates of the next transformed voxel can thus be obtained by simply adding the respective scaling factor to the *X*, *Y*, or *Z* coordinate of the predecessor voxel and rounding the results to integer coordinates.

Precautions must be taken if the scaling factors are greater than 1. In this case, applying the above transformation to a primitive may produce holes in the scaled primitive, due to an increase in the room volume with no increase in voxels. To avert this problem *back scaling* is introduced, i.e., the incremental transformation process is reversed. By employing this technique, several voxels in the destination primitive can be mapped to the same voxel in the source primitive. To do this we transform two opposite corners of the source room, which yields two opposite corners in the destination room. We then step through the destination room. At each point in the destination room, we reverse-transform the coordinates to determine the voxel value for that point. This can be done by multiplying the destination coordinates by $(1/S_x, 1/S_y, 1/S_z)$. A more efficient way to do this is to step through the source room in parallel with the stepping through the destination room. The increments to be used in stepping through the source room are $(Xincr/S_x, Yincr/S_y, Zincr/S_z)$, where Xincr, Yincr and Zincr are the X, Y, and Z increments (1 or -1) used in stepping through the destination room.

As with the translation algorithm, precautions must be taken to avoid overwriting of voxels in the source room before they have been scaled. This is accomplished by properly choosing the corner of the room to be used as a starting point for the transformation based on the values of S_x, S_y and S_z.

Below is a pseudo-C code for scaling, assuming that (X_S, Y_S, Z_S) and (X_E, Y_E, Z_E) are the starting and ending corner of the destination primitive, respectively, and (X_{src}, Y_{src}, Z_{src}) is the starting corner of the source primitive (the reverse-transformation of (X_S, Y_S, Z_S)).

```
X_dest = X_S;   Y_dest = Y_S;   Z_dest = Z_S;
SXincr = Xincr/S_x;   SYincr = Yincr/S_y;   SZincr = Zincr/S_z;
While (Z_src <= Z_E)
{
        Round Z_src
        While (Y_src <= Y_E)
        {
                Round Y_src
                While (X_src <= X_E)
                {
                        Round X_src
                        if ( (X_dest, Y_dest, Z_dest) is not clipped )
                                Copy from (X_src, Y_src, Z_src) to (X_dest, Y_dest, Z_dest)
                        Increment X_src by SXincr
                        Increment X_dest by Xincr
                }
                Increment Y_src by SYincr
                Increment Y_dest by Yincr
        }
        Increment Z_src by SZincr
        Increment Z_dest by Zincr
}
```

8.3. Rotation Algorithm

The rotation algorithm of the *vE* rotates a room about its origin through any angle θ relative to a major axis. Positive angles produce clockwise rotations when looking down an axis toward the origin. A rotation is performed about only one axis at a time. The following discussion is limited to a rotation about the Z axis. Similar equations hold for rotating about the X and Y axes. The equations for rotating about the Z axis are:

$$X_{dest} = X_{src} \cos\theta - Y_{src} \sin\theta \tag{9}$$

$$Y_{dest} = X_{src} \cos\theta + Y_{src} \sin\theta \tag{10}$$

Like for scaling, the shifting to and from the origin are accounted for in the CFB addressing of the source and destination rooms. Since all the voxels along beams parallel to the Z-axis are identically rotated, entire beams are moved for each new coordinate calculation employing the parallel memory and the barrel shifter.

Once again an incremental algorithm is employed to make it possible to determine a voxels position without the need to perform heavy computations. In addition, *back rotation*, from destination to source [5], is employed to eradicate holes which could be produced from the forward rotation of a room. To accomplish this task the eight corners of the source room are first rotated to locate the destination room. Once calculated, the minimum coordinates are back rotated by the rotation equation, Equations 9 and 10, by replacing θ by $-\theta$. From this point on, the incremental procedure is used to back scan the entire room. The constant increments for all voxels/beams are merely $\sin\theta$ and $\cos\theta$. Equations 11 and 12 give the coordinates of the point $(X_{x+1}, Y_{x+1}, Z_{dest})$ back-rotated from $(X_{dest}+1, Y_{dest}, Z_{dest})$, while Equations 13 and 14 give the point $(X_{y+1}, Y_{y+1}, Z_{dest})$ back-rotated from $(X_{dest}, Y_{dest}+1, Z_{dest})$:

$$X_{x+1} = (X_{dest} + 1)\cos\theta + Y_{dest}\sin\theta = X_{src} + \cos\theta \quad (11)$$

$$Y_{x+1} = -(X_{dest} + 1)\sin\theta + Y_{dest}\cos\theta = Y_{src} - \sin\theta \quad (12)$$

$$X_{y+1} = X_{dest}\cos\theta + (Y_{dest} + 1)\sin\theta = X_{src} + \sin\theta \quad (13)$$

$$Y_{y+1} = -X_{dest}\sin\theta + (Y_{dest} + 1)\cos\theta = Y_{src} + \cos\theta \quad (14)$$

Note that either set employs only two fixed-point additions that can be performed in parallel. Note also that the destination room can be clipped first, then only the visible parts need to be back-rotated.

As in the scaling and translation algorithms, the overlap problem is overcome by starting the scan at the source vertex which is also located within the destination room.

9. Scan-line Clipping Algorithm

The *vE* Clipping sub-unit supports several simultaneous rooms and figurines, each of which is assigned a unique priority, which is specified at time of definition, and may be changed by the user later on. If collision occurs, temporal priority governs. When a room or an figurine, *r*, with two opposite corners, $(X_{min}, Y_{min}, Z_{min})$, and $(X_{max}, Y_{max}, Z_{max})$, is written into the CFB, by copying it from outside or by moving information within the CFB, the clipping algorithm is employed in order to determine which portions of *r* are not obscured by any of the other rooms or figurines. Any room with a higher priority will not be overwritten by the incoming object. However, when reading a room, overlapping is ignored and priorities play no role in that process.

The clipping algorithm is a 3D generalisation of a scan-line algorithm [1] adjusted for low-level processor implementation. For each scan-line (row of voxels), only the visible segments are written into the CFB. This is intrinsicly different from the algorithm proposed by Pike [16] at the window manager level, in which the window in question is recursively subdivided into subrectangles and all the subrectangles have to be accounted for. Here, on the other hand, only the visible segments have to be handled by the vE, the invisible sections are handled by the application, e.g., the room manager.

A scan-line is obtained differently in five different cases: constant colouring, writing, copying, zooming, and when orienting a room. The scan-line is clipped by matching it against the active table. This table holds the identifiers of those rooms which intersect with the current scan-line. To make the process of creating the active table more efficient, a temporary table, InterTable, is created as the algorithm is invoked. This table lists the identifiers of all rooms that intersect with r, having priorities higher than that of r. It is sorted by their Z_{min}, Y_{min} and X_{min}. As each scan-line is processed, the active table is updated from the InterTable. The InterTable is created in Z_{min} order so that scanning through the table can be stopped as soon as the first intersecting room with a Z_{min} greater than the current Z-plane in encountered, thereby making the search less costly. For similar reasons, the InterTable is kept also in Y_{min} and X_{min} order. The algorithm in pseudo-C code follows:

```
Save all jacks
For (Zscan = Z_min ; Zscan ≤ Z_max ; Zscan++)
{
        Remove from InterTab primitives below Zscan
        Sort by Y then X all primitives in InterTable intersecting the Zscan line
        For (Yscan = Y_min ; Yscan ≤ Y_max ; Yscan++)
        {
                While (LeftBound ≤ X_max)
                {
                        RightBound = Right X of leftmost primitive ("left primitive")
                        LeftBound = Left X of next leftmost primitive ("right")
                        While (LeftBound ≤ RightBound and InterTab not empty)
                        {
                                Get next "right" primitives and update LeftBound
                                If ("left" and "right" are adjacent in sorted list)
                                {
                                        Set run from RightBound to LeftBound along
                                                Yscan line to the voxel value of r;
                                        Get next "left" and "right" from list
                                }
                                Else    Get the next "left" primitive and update
                                        RightBound if the right edge of the new "left"
                                        primitive is to right of RightBound
                        }
                }
        }
}
Restore saved jacks
```

Note that the algorithm has been described assuming that the ZoomX, ZoomY, ZoomZ and Orient registers are set at their default values, 1, 1, 1 and 0°, respectively. However, other values assigned to these registers may drastically affect the way it executes, especially since the sequence by which r is clipped is not the order of the input stream. In addition, the voxels or intervals of voxels have to be replicated according to the Zoom registers before writing runs of voxels between RightBound and LeftBound.

Note also that when writing a voxel into the CFB, the mode registers, like Colour_Mode, Texture_Mask and Voxel_Ops, are consulted (if they are enabled) to determine the correct colour. If the Texture_Mask is enabled, the data will be written selectively according to the binary mask. If the Voxel_Ops is enabled, a boolean operation defined by the register will be performed between the source primitive and the destination one before the data is placed in the CFB (see the next section).

Boolean Function	Index	Dest = 0 Src = 0	Dest = 0 Src = 1	Dest = 1 Src = 0	Dest = 1 Src = 1
background	0	0	0	0	0
Dest and Src	1	0	0	0	1
Dest and (not Src)	2	0	0	1	0
Dest	3	0	0	1	1
(not Dest) and Src	4	0	1	0	0
Src	5	0	1	0	1
Dest xor Src	6	0	1	1	0
Dest or Src	7	0	1	1	1
not (Dest or Src)	8	1	0	0	0
not (Dest xor Src)	9	1	0	0	1
not Src	10	1	0	1	0
Dest or (not Src)	11	1	0	1	1
not Dest	12	1	1	0	0
(not Dest) or Src	13	1	1	0	1
not (Dest and Src)	14	1	1	1	0
foreground	15	1	1	1	1

Table 1: Bitwise Voxel-Ops and Jack-Ops.

10. Voxel-Ops and Jack-Ops Algorithm

The voxel-ops and jack-ops algorithm is performed before writing any data into the CFB. It is governed by two registers: the Voxel_Ops register which applies to rooms and figurines, and the Jack_Ops register which applies to jacks. There is a total of 16 voxel-wise boolean operations which operate on a source voxel (Src) and a destination voxel (Dest) and place the result in Dest. The boolean operations are summarised in Table 1. The four bits of the index of the boolean function given in either register are exactly the code utilised by the *vE* to perform the function. These operations are legal only when the processor is working either in the Gray_Scale or in the Binary mode. In the Binary mode, the Background colour represents a 0 and a Foreground colour represents a 1.

Angles		y = 0				90				180				270			
		z=0	90	180	270	0	90	180	270	0	90	180	270	0	90	180	270
x=0	x↓	x↓	y↓	x↑	y↑	z↑	z↑	z↑	z↑	x↑	y↑	x↓	y↓	z↓	z↓	z↓	z↓
	y↓	y↓	x↑	y↑	x↓	y↓	x↑	y↑	x↓	y↓	x↑	y↑	x↓	y↓	x↑	y↑	x↓
	z↓	z↓	z↓	z↓	z↓	x↓	y↓	x↑	y↑	z↑	z↑	z↑	z↑	x↑	y↑	x↓	y↓
	x↑	x↑	y↑	x↓	y↓	z↓	z↓	z↓	z↓	x↓	y↓	x↑	y↑	z↑	z↑	z↑	z↑
	y↑	y↑	x↓	y↓	x↑	y↑	x↓	y↓	x↑	y↑	x↓	y↓	x↑	y↑	x↓	y↓	x↑
	z↑	z↑	z↑	z↑	z↑	x↑	y↑	x↓	y↓	z↓	z↓	z↓	z↓	x↓	y↓	x↑	y↑
Orient:		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
x=90	x↓	x↓	y↓	x↑	y↑	z↑	z↑	z↑	z↑	x↑	y↑	x↓	y↓	z↓	z↓	z↓	z↓
	y↓	z↓	z↓	z↓	z↓	x↓	y↓	x↑	y↑	z↑	z↑	z↑	z↑	x↑	y↑	x↓	y↓
	z↓	y↑	x↓	y↓	x↑	y↑	x↓	y↓	x↑	y↑	x↓	y↓	x↑	y↑	x↓	y↓	x↑
	x↑	x↑	y↑	x↓	y↓	z↓	z↓	z↓	z↓	x↓	y↓	x↑	y↑	z↑	z↑	z↑	z↑
	y↑	z↑	z↑	z↑	z↑	x↑	y↑	x↓	y↓	z↓	z↓	z↓	z↓	x↓	y↓	x↑	y↑
	z↑	y↓	x↑	y↑	x↓	y↓	x↑	y↑	x↓	y↓	x↑	y↑	x↓	y↓	x↑	y↑	x↓
Orient:		16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
x=180	x↓	x↓	y↓	x↑	y↑	z↑	z↑	z↑	z↑	x↑	y↑	x↓	y↓	z↓	z↓	z↓	z↓
	y↓	y↑	x↓	y↓	x↑	y↑	x↓	y↓	x↑	y↑	x↓	y↓	x↑	y↑	x↓	y↓	x↑
	z↓	z↑	z↑	z↑	z↑	x↑	y↑	x↓	y↓	z↓	z↓	z↓	z↓	x↓	y↓	x↑	y↑
	x↑	x↑	y↑	x↓	y↓	z↓	z↓	z↓	z↓	x↓	y↓	x↑	y↑	z↑	z↑	z↑	z↑
	y↑	y↓	x↑	y↑	x↓	y↓	x↑	y↑	x↓	y↓	x↑	y↑	x↓	y↓	x↑	y↑	x↓
	z↑	z↓	z↓	z↓	z↓	x↓	y↓	x↑	y↑	z↑	z↑	z↑	z↑	x↑	y↑	x↓	y↓
x=270	x↓	x↓	y↓	x↑	y↑	z↑	z↑	z↑	z↑	x↑	y↑	x↓	y↓	z↓	z↓	z↓	z↓
	y↓	z↑	z↑	z↑	z↑	x↑	y↑	x↓	y↓	z↓	z↓	z↓	z↓	x↓	y↓	x↑	y↑
	z↓	y↓	x↑	y↑	x↓	y↓	x↑	y↑	x↓	y↓	x↑	y↑	x↓	y↓	x↑	y↑	x↓
	x↑	x↑	y↑	x↓	y↓	z↓	z↓	z↓	z↓	x↓	y↓	x↑	y↑	z↑	z↑	z↑	z↑
	y↑	z↓	z↓	z↓	z↓	x↓	y↓	x↑	y↑	z↑	z↑	z↑	z↑	x↑	y↑	x↓	y↓
	z↑	y↑	x↓	y↓	x↑	y↑	x↓	y↓	x↑	y↑	x↓	y↓	x↑	y↑	x↓	y↓	x↑
Orient:		48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63

Table 2: Orientation Table Used by the Orientation Algorithm. Letters within the table refer to the coordinates of the primitive:

↓ - refers to the minimum coordinate value,

↑ - refers to the maximum coordinate value.

11. Orientation Algorithm

The Orient register of the vE automatically affects the final orientation of all primitives being written into the CFB. When writing into the CFB, the clipping algorithm produces scan-lines to be placed in the CFB according to the Orient register. The clipping algorithm is unaware of the actual orientation or exact location of the scan-line (or primitive for that matter) on which it is currently working. To accomplish this six values, based upon the Orient register, are employed: X_{min}, Y_{min}, Z_{min}, X_{max}, Y_{max} and Z_{max}, marked

as x↓, y↓, z↓, x↑, y↑, and z↑, respectively. If, for example, the Orient register is set such that a 180° rotation is performed about the Z-axis, (orient = 2), then the top and the bottom have been switched, so have been the left and the right, however the front and the back have remained unchanged. To reflect this action, x↓ is the value which corresponds to x↑, y↓ to y↑, z↓ to z↓, and so on (see Table 2). When several rotations are to be performed (orient about more than one axis), turning is done about the X, Y and Z axes in that order. Angles refer to clockwise turning when looking down the specified axis toward the origin.

12. Implementations

A reduced-resolution hardware prototype system of 16^3 voxels, 8-bit each, has been realised in hardware and has been integrated under an IBM-AT with a mouse. This prototype has been running successfully in real time under the CUBE software environment. The hardware realisation consists of 16 modules, each of which is implemented as a custom-built printed-circuit board, containing one CFB module, its addressing and processing units, with barrel shifting links between boards. Measured time for a small 16^3 room rotation about a primary axis, for example, is only tenth of a millisecond, while the rotation time for a large 512^3 room is estimated to be 170 milliseconds.

Many components of the CUBE prototype are ideal for VLSI implementation. Currently, a larger and more compact implementation in VLSI using CMOS 2.0μ technology is being developed, where a single VLSI chip implements one board including its memory module. An alternative implementation calls for one chip to hold several processing and addressing units excluding their memory modules which will be realised with off-the-shelf components.

The vE has been designed, developed and simulated in software at the Ben-Gurion University of the Negev and at the State University of New York at Stony Brook. The simulations have been written in C on a VAX-780 and SUN workstations running Unix. At the Ben-Gurion University the simulation has implemented most of the vE instructions using a RAMTEK-9400 as the output device [17].

A more complete simulation has been implemented at Stony Brook as part of the CUBE system [10, 12] on SUN workstations using SunView and the CUBE environment. A 3D input device supported in this implementation is a Polhemus 3SPACE Isotrak device, the *kite*. It is used for absolute and relative 3D positioning and 3D orientation, as both a 3D locator and a 3D pick device. The mouse is another input device employed as a 3D locator, a "triad" mouse (a la [14]). The third degree of freedom is achieved when the mouse is moved diagonally as visually shown in the projection of the jack on the screen.

The *vE* user interface relies on the CUBE system user interface. Currently, three interface modes are supported: batch mode for reading a FBDL3 from a file or memory, keyboard mode for providing vE commands one at a time from the keyboard, and "cubespace"

mode for a fully interactive dialogue, in which several 3D interactive views of the 3D space are provided [12].

One of the direct and most important applications of the vE is the 3D voxel-map *room manager*, i.e., the 3D counterpart of the 2D window manager. In our SUN implementation a room manager has been programmed in software on top of the vE simulation. It employs major parts of the vE repertoire. The room manager allows the application to freely create, delete, move, resize, scale, rotate, swap, and copy overlapping voxel-map rooms or figurines. The room manager will take care of all the operations and will maintain records of the obscured portions of the primitives, so that the application can focus on the contents of the primitives. The basic idea of the voxel-map room manager is based on Rob Pike's window manager and his layered approach to graphics in overlapping bitmap layers [16]. However, the operations listed above are supported directly by the *vE*. Specifically, clipping of primitives is supported at a lower level by the *vE*.

13. Acknowledgement

This work was supported by the National Science Foundation under grants DCR 8603603, CCR 8743478, CCR 8717016, and MIP 8805130. The author wishes to thank all the dedicated individuals who have worked on the *vE* project: R. Bakalash, C. Lin, T. Rosenwaks, M. Waldman, and X. Zhong.

References

1. J. D. Foley and A. van Dam, *Fundamentals of Interactive Computer Graphics*, Addison-Wesley, Reading, MA, 1982.
2. S. Goldwasser and R. A. Reynolds, "An Architecture for the Real-Time Display of Three Dimensional Objects," *Proceedings International Conference on Parallel Processing*, 1983.
3. S. M. Goldwasser, "A Generalized Object Display Processor Architecture," *IEEE Computer Graphics & Applications*, vol. 4, no. 10, pp. 43-55, October 1984.
4. L. J. Guibas and J. Stolfi, "A Language for Bitmap Manipulation," *ACM Transactions on Graphics*, vol. 1, no. 3, pp. 191-214, July 1982.
5. R. D. Hersch, "Raster Rotation of Bilevel Bitmap Images," *Proceedings EUROGRAPHICS '85*, pp. 295-308, Nice, France, September 1985.
6. D. Jackel, "The Graphics PARCUM System: A 3D Memory Based Computer Architecture for Processing and Display of Solid Models," *Computer Graphics Forum*, vol. 4, pp. 21-32, 1985.
7. G.J. Jense and D.P. Huijsmans, "Hardware Support for the Display and Manipulation of Binary Voxel Models," in this book, 1989.

8. A. Kaufman, "Memory Organization for a Cubic Frame Buffer," *Proceedings EUROGRAPHICS '86*, pp. 93-100, Lisbon, Portugal, August 1986.

9. A. Kaufman, "Towards a 3-D Graphics Workstation," in *Advances in Graphics Hardware I*, ed. W. Strasser, pp. 17-26, Springer-Verlag, 1987.

10. A. Kaufman and R. Bakalash, "Memory and Processing Architecture for 3-D Voxel-Based Imagery," *IEEE Computer Graphics & Applications*, pp. 10-23, November 1988.

11. A. Kaufman, "A Two-Dimensional Frame Buffer Processor," in *Advances in Graphics Hardware II*, ed. A.A.M. Kuijk, and W. Strasser, Springer-Verlag, 1988.

12. A. Kaufman, "The CUBE Workstation - a 3-D Voxel-Based Graphics Environment," *The Visual Computer*, vol. 4, no. 4, 1988.

13. D.J. Meagher, "Interactive Solids Processing for Medical Analysis and Planning," *Proceedings NCGA'84*, vol. 2, pp. 96-106, Anaheim, CA, May 1984.

14. G.M. Nielson and D.R. Jr. Olsen, "Direct Manipulation Techniques for 3D Objects Using 2D Locator Devices," *Proceedings ACM Workshop on Interactive 3D Graphics*, pp. 175-182, Chapel Hill, NC, October 1986.

15. T. Ohashi, T. Uchiki, and M. Tokoro, "A Three-Dimensional Shaded Display Method for Voxel-Based Representation," *Proceedings EUROGRAPHICS '85*, pp. 221-232, Nice, France, September 1985.

16. Rob Pike, "Graphics in Overlapping Bitmap Layers," *ACM Transactions on Graphics*, vol. 2, no. 2, pp. 135-160, April 1983.

17. T. Rosenwaks, "A 3-D Frame Buffer Processor," M.S. Thesis, Ben-Gurion University, Beer-Sheva, 1985.

18. R. Sproull, I. E. Sutherland, A. Thompson, S. Gupta, and C. Minter, "The 8 by 8 Display," *ACM Transactions on Graphics*, vol. 2, no. 1, pp. 32-56, January 1983.

19. T. Uchiki and M. Tokoro, "SCOPE: Solid and Colored Object Projection Environment," *Transaction of the Institute of Electronics and Communication Engineers of Japan*, vol. 68-D, no. 4, pp. 741-748, April 1985.