# M-Buffer: A Flexible MISD Architecture for Advanced Graphics

*Bengt-Olaf Schneider*
*Jarek Rossignac* [1]

ABSTRACT

Contemporary graphics architectures are based on a hardware-supported geometric pipeline, a rasterizer, a z-buffer and two frame buffers. Additional pixel memory is used for alpha blending and for storing logical information. Although their functionality is growing it is still limited because of the fixed use of pixel memory and the restricted set of operations provided by these architectures. A new class of graphics algorithms that considerably extends the current technology is based on a more flexible use of pixel memory, not supported by current architectures.

The M-Buffer architecture described here divides pixel memory into general-purpose buffers, each associated with one processor. Pixel data is broadcast to all buffers simultaneously. Logical and numeric tests are performed by each processor and the results are broadcast and used by all buffers in parallel to evaluate logical expressions for the pixel update condition.

The architecture is scalable by addition of buffer-processors, suitable for pixel parallelization, and permits the use of buffers for different purposes. The architecture, its functional description, and a powerful programming interface are described.

*CR Categories and Subject Descriptors:*

**B.4.2** [Input/Output and Data Communications]: Input/Output Devices – Image Display
**C.1.1** [Processor Architectures]: Single Data Stream Architectures – MISD
**I.3.1** [Computer Graphics]: Hardware Architecture – Raster Display Devices

*Additional Keywords:*

Graphics buffer, frame buffer, depth buffer,buffer processor, complex update conditions, pixel-level rendering algorithms

[1] Authors' address: IBM Thomas J. Watson Research Center, P.O. Box 704, Yorktown Heights NY 10598
Schneider: bosch@watson.ibm.com, (914) 784-6002
Rossignac: jarek@watson.ibm.com, (914) 784-7630

## 1.1 Introduction

### 1.1.1 Advanced Pixel Rendering

An increasing number of graphics algorithms require deep frame buffers, i. e. a large number of bits per pixel [5]. For example, a technique that uses two z-buffers (Z1, Z2), two frame buffers (F1, F2), one alpha buffer (A) and a bit plane (V) for correctly displaying scenes with several layers of transparent objects is described in [7]. The technique uses several passes to sort transparent objects in order to determine the correct aggregate transparency of all visible transparent objects. The two z-buffers are used to sort incoming surface points by depth in order to find the furthest visible transparent object. The following statements describe the operations that are performed at each pixel during the scan-conversion in order to determine the next surface.

ALGORITHM 1

```
if (Z1 < z < Z2)
{ V = 1 ; Z2 = z ; F2 = c ; A = a ; }
```

where $c$, $a$, and $z$ are the color, the alpha value, and the depth for the scan-converted object at a given pixel.

Other examples of the power of multiple depth buffers are found in [6, 8]. The first requires, for each pixel, two z-buffers, two frame buffers, two bit-planes and a counter. The other requires three z-buffers, one frame buffer and one bit-plane. The following two code fragments are extracted from the inner loops of the latter algorithm. They compute in Z3 the front-most surface points behind points in Z2. Later, points in Z3 belonging to front-facing surfaces are copied to Z2.

ALGORITHM 2

```
if (Z2 < z-ε < Z3) Z3 = z ;
if (Z2 < z-ε) pixel_flag = !pixel_flag ;
```

For each pixel, the algorithm finds the nearest surface points behind those already in Z2. The parity of the number of surfaces found behind Z2, computed in the pixel_flag, determines whether the surface in Z3 is front-facing or back-facing.

ALGORITHM 3

```
if (pixel_flag && (Z2 != Z3)) Z2 = Z3 ;
```

This statement copies new pixel information from Z3 to Z2 if the point in Z3 belongs to a front-facing surface (see ALGORITHM 2).

These techniques require the following capabilities:

- Simultaneous update of several pixel buffers.
- Conditional buffer update based on tests performed on the contents of several buffers.
- Different update conditions per buffer involving either values from the scan-conversion or from other buffers.
- Programmable update functions and conditions.

This functionality is a special case of what can be formulated by the following instructions executed at each pixel processed by the rasterizer.

$$b_i = \mathcal{F}_i(b_1, b_2, \ldots, s) \tag{1.1}$$

where $b_1 \ldots b_n$ are the values stored in the respective buffers at that pixel, and $\mathcal{F}_1 \ldots \mathcal{F}_n$ are buffer specific update functions. The surface data $s$ are generated by the rasterizer for that pixel and include several data fields, e. g. color $s_c$, depth $s_z$, or alpha value $s_\alpha$.

For example, the function $\mathcal{F}_3$ of ALGORITHM 2 could be written as:

$$\mathcal{F}_3 = \begin{cases} s_z & \text{if } b_2 < s_z - \varepsilon < b_3 \\ b_3 & \text{otherwise} \end{cases}$$

One can argue that the anticipated applications of this technology do not require the full generality of the functions $\mathcal{F}_i$. Indeed, in this paper we restrict $\mathcal{F}_i$ to a specific subset, which subsumes existing and emerging techniques and is amenable to an efficient hardware implementation.

### 1.1.2 Existing Buffer Architectures

In a typical graphics system the application generates graphics primitives (for example triangles) in world coordinates and sends them to the geometry sub-system, which transforms them to screen coordinates and clips them against the boundaries of the viewing volume. The rasterizer discretizes the primitives into pixels values, such as color and depth, which it sends *sequentially* to a buffer sub-system from which the final image is displayed on the monitor.

The buffer sub-system contains buffers for storing and processing pixel information. It supports a set of pixel-level algorithms, such as z-buffering or alpha-blending. This paper focuses on practical extensions of the buffer sub-system to support advanced rendering algorithms. Full hardware support for these algorithms is currently not available in commercial systems.. The architectures of the Silicon Graphics GTX and VGX [2, 1] provide some flexibility in how the buffers can be used. For example, the color of rendered pixels may be compared with the color of the pixels already in the frame buffer. In addition to the z-test, the *stencil* bit-planes in the VGX architecture provide a second test. However, efficient use of the stencil planes is hampered by a non-general programming interface that seems to result from hardware limitations.

In [3], pixel maps are stored in main memory, allowing applications to allocate the needed graphics buffer with virtually arbitrary size and depth. The main CPU accesses and manipulates the contents of these *virtual pixel maps*. Since the CPU performs those operations sequentially, the performance of pixel operations drops when numerous buffers have to be managed and/or when the operations involve many steps.

## 1.2 Parallel Buffer Operations

Image space partitioning is currently the prevalent technique used to achieve high buffer update rates. In these SIMD architectures each processor handles a subset of the pixels on the screen and manages all buffers for those pixels (figure 1.1a). Image space partitioning architectures are balanced for simple buffer update operations, i. e. the memory bandwidth is matched by the pixel generation rate of the rasterizer. Since rasterizers generate pixels much faster than a single memory chip can store them, several banks of memory are interleaved to obtained the necessary memory bandwidth. The following pseudo-code characterizes this solution:

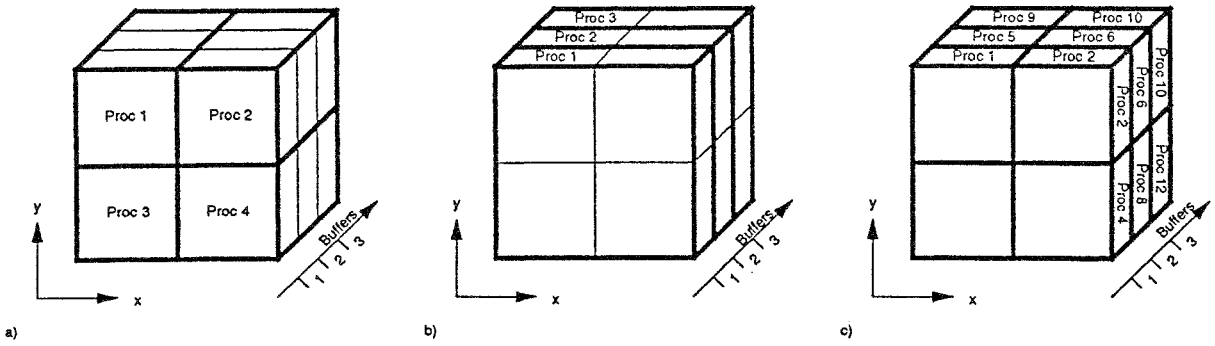


FIGURE 1.1. a) Image Space Partitioning. b) Buffer Space Partitioning. c) Mixed Partitioning.

ALGORITHM 4

1. *Rasterizer:*
   Generate pixel value at $(x, y)$
   and send to appropriate buffer.

2. *Buffer Sub-System:*
   **forall** buffers $i = 1 \ldots n$ **do_sequential**
   $b_i = \mathcal{F}_i(b_1 \ldots b_n, s)$ ;

The number of steps performed by this algorithm is linear in $n$, the number of buffers to be updated for each pixel. If $n$ increases the buffer sub-system saturates, causing the rasterizer to stop and the memory access rate to become smaller than the memory bandwidth. A $n$-fold speed-up is achieved if the buffer operations (step 2 in ALGORITHM 4) are performed in parallel, because the update time for each pixel is then independent of the number of buffers to be updated. Figure 1.1b depicts the corresponding partitioning. Each processor manages only one buffer and computes the respective update functions and update conditions. However, at a given time all processor are working on the same pixel, thus forming a MISD parallel processor, described by the following pseudo-code:

ALGORITHM 5

1. *Rasterizer:*
   Generate pixel value at $(x, y)$.

2. *Buffer Sub-System:*
   **forall** buffers $i = 1 \ldots n$ **do_parallel**
   $b_i = \mathcal{F}_i(b_1 \ldots b_n, s)$ ;

The two strategies for parallelizing the operation of the buffer sub-system are orthogonal. They can be combined to form a mixed multiprocessor system (figure 1.1c). This arrangement has two degrees of freedom: the number of partitions in screen space and in buffer space. These to variables can be exploited to optimize performance across a family of applications. Since image space partitioning has been addressed elsewhere [4], we focus here on the buffer space MISD aspect.

## 1.3 Advanced Update Functions

Formulation (1.1) defines a class of algorithms requiring that, for updating a given buffer at a given pixel, the values of all buffers at that pixel are accessible. Providing all buffers with simultaneous access to the values stored in all other buffers, requires $n$ pixel busses or $n^2$ point-to-point connections. Since the cost of the processing elements (each takes $n$ inputs) and the communication requirements are prohibitive, we will restrict the class of update functions as follows:

In the first phase of the buffer update operations, each processor $i$ has access to the contents of its buffer and to the surface data broadcast by the rasterizer. The processor performs a test $\mathcal{T}_i$ that usually compares the surface data to data previously stored in the associated buffer. The tests $\mathcal{T}_i$ may differ for different processors. The test results $r_i$ are broadcast to every processor. All processors perform these steps in parallel, receiving the same surface data on a data bus and writing the value $r_i$ on a subset of the result bus. (Each processor has exclusive write access to one or more bits of the result bus.)

In the second phase, each processor $i$ evaluates its update condition $\mathcal{C}_i$ based on the value on the entire result bus. If $\mathcal{C}_i$ is True, the content of buffer $i$ is updated according to an update function $\mathcal{U}_i$ associated with that buffer.

These restrictions yield algorithms that include all algorithms mentioned earlier. For example, for a standard z-buffer algorithm $\mathcal{T}_i$, $\mathcal{C}_i$, $\mathcal{U}_i$ are chosen as follows:

$$\begin{aligned} \mathcal{T}_1 &: s_z < b_1 \\ \mathcal{C}_1, \mathcal{C}_2 &: r_1 \\ \mathcal{U}_1 &= s_z \\ \mathcal{U}_2 &= s_c \end{aligned}$$

where buffer 1 is the depth buffer and buffer 2 stores the pixel colors, and $s_c$ and $s_z$ are the color and the depth of the scan-converted surface.

For ALGORITHM 2 the buffer processors may be programmed as follows:

$$\begin{aligned} \mathcal{T}_2 &: b_2 < s_z - \varepsilon \\ \mathcal{T}_3 &: b_3 > s_z - \varepsilon \\ \mathcal{C}_1 &: r_2 \\ \mathcal{C}_3 &: r_2 \;\&\&\; r_3 \\ \mathcal{U}_1 &= !\, b_1 \\ \mathcal{U}_3 &= s_z \end{aligned}$$

where buffer 1 stores the pixel flag and the buffers 2 and 3 are the depth buffers Z2 and Z3 respectively.

## 1.4 The M-Buffer Architecture

The M-Buffer Architecture developed here to support the operations defined above is a regular multi-processor buffer sub-system that uses buffer space partitioning. It includes several buffer modules each of which contains a buffer memory and a buffer processor (figure 1.2). The number of buffers may vary, providing the possibility of scalable systems.

A host computer programs the buffer processors by setting the processors' internal registers for operations and data, thus selecting each buffer's $\mathcal{U}_i$, $\mathcal{T}_i$ and $\mathcal{C}_i$. All buffer modules are connected to a common pixel bus that broadcasts the pixel values generated

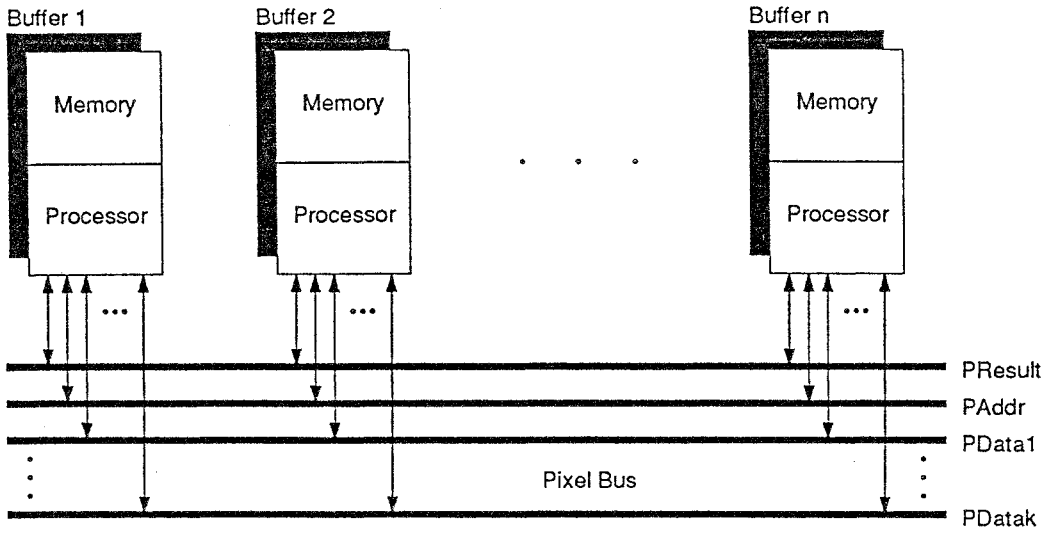


FIGURE 1.2. M-Buffer System Block Diagram.

by the rasterizer. The pixel bus has several sub-busses: The pixel address bus, PAddr, broadcasts the pixel address $(x, y)$. There are $k$ pixel data busses, PData, for the surface data $s$. (For example, PData may carry 32 bits for the intensity and 32 bits for the depth.) Each buffer processor broadcasts test results $r_i$ to the other buffers using the pixel result bus, PResult. This unique feature provides each buffer with access to the test results of all other buffers simultaneously, thus providing the arguments for the complex update conditions $\mathcal{C}_i$.

For every pixel, each buffer module cycles through the following sequence of steps:

GA: Generate a pixel address and write it to the pixel address bus.

MA: Address the buffer memory. Each active buffer module applies this address to its buffer memory.

RD: Read the buffer memory. The active buffers simultaneously read a pixel value from the buffer memory. The rasterizer places data on the pixel bus.

TU: Compute the tests and update functions. The active buffers compute the functions $\mathcal{T}$ and $\mathcal{U}$. The test result $r$ is placed on the pixel result bus.

CD: Evaluate the condition function. The active buffers compute the condition $\mathcal{C}$ using the information on the pixel result bus.

WR: Write the buffer memory. Depending on the result of $\mathcal{C}$, the buffer memory may be updated with the result of $\mathcal{U}$.

Pipelining the pixel processors allows to process up to three pixels at the same time and reduces the effective processing time for each pixel from six to two steps. The following table shows how the steps of consecutive pixels overlap while processed in a pixel processor.

| Step | Sequences | | |
|---|---|---|---|
| 1 | GA | | |
| 2 | MA | | |
| 3 | RD | GA | |
| 4 | TU | MA | |
| 5 | CD | RD | GA |
| 6 | WR | TU | MA |
| 7 | | CD | RD |
| 8 | | WR | TU |
| 9 | | | CD |
| 10 | | | WR |

For clarity, the rasterizer was described as the only pixel source in the M-Buffer. In fact, every buffer module can act as a pixel source by taking the role of the rasterizer. The buffer processors can generate pixel addresses for a screen region and place these addresses on the pixel address bus (step GA). In step RD up to $k$ buffers can write pixel values to the pixel data busses. ($k$ is the number of pixel data busses.) The host processor, i. e. the application, controls which buffer is providing the pixel address and which buffers communicate their pixel value across the pixel data busses. The ability to use buffer modules as pixel sources is very useful for inter-buffer pixel transfer, e. g. conditional BitBlts (see ALGORITHM 3).

In step RD, the value read from buffer memory is written to a pixel data bus. Instead, this value can be written to the pixel address bus, thus interpreting pixel data as addresses. This capability gives the architecture the flexibility of indirectly addressing pixels, as required for texture mapping and other look-up tables. Indirect addressing schemes are implemented by adding to the basic sequence, shown above, extra RD-MA steps for each level of indirection. The M-Buffer architecture provides the possibility to implement up to 16 levels of indirection.

## 1.5 The Buffer Modules

There are three types of buffer modules. *Surface Buffers* store pixel data associated with surfaces to be displayed, e. g. depth values, colors, texture maps, normals etc. *Control Buffers* maintain information used to further differentiate the operation of other buffers, e. g. pixel counters, pixel flags, screen masks. *Virtual Buffers* form a unified and general mechanism for interfacing the M-Buffer to other components of a graphics system, such as the rasterizer. The following sections describe in detail the internal structure of the buffer modules.

### 1.5.1 Surface Buffers

Figure 1.3 shows a block diagram of a surface buffer module. Each surface buffer module contains five major blocks all of which are controlled by a central controller. These blocks are connected to the pixel bus via the bus interface. Each of the blocks can be programmed by loading internal registers. A host processor has access to these registers via the host interface, which also provides read access to some internal registers of the buffer controller, thus allowing the host to query the current status of a buffer module.

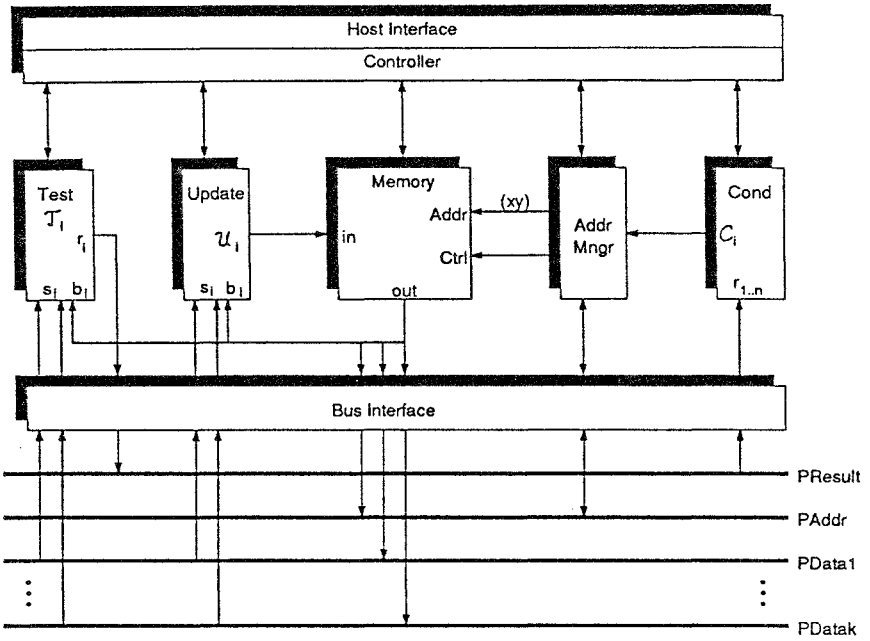


FIGURE 1.3. Surface Buffer.

Address Manager

BitBlts and window motions require that pixel address of the source pixel and the destination pixel differ. A programmable offset (displacement) can be added to each pixel address read from the pixel bus. For pixel transfers within one buffer, pixels must be generated in the right order to properly process overlapping source and destination regions. The host defines this order by specifying the starting pixel of the rectangular screen region being transferred.

The pixel address is then applied to the buffer memory together with the necessary signals for controlling DRAM memory. For write cycles (step WR), the state of the update signal provided by the condition block determines whether the write operation is actually performed.

For many applications it is advantageous to process only those pixels further that are contained within a screen region including pixels that were processed during previous steps. To support this facility, the address manager can automatically update a bounding box around all pixels that were either visited or actually changed.

Since usually only parts of the screen need to be transferred, buffers can be programmed to broadcast pixel data only within a rectangular sub-screen. The extent of that sub-screen can either be set by the host or be the bounding box containing the visited or changed pixels.

Buffer Memory

The surface data is stored in the central buffer memory, which provides 32 bits for each pixel. Standard DRAM memory is used in most of the buffer modules. Only buffers whose contents must be displayed are equipped with more expensive VRAMs.

Datapaths

The Test and Update blocks provide a large set of arithmetic and logical functions for implementing the $\mathcal{T}_i$ and $\mathcal{U}_i$. They take their arguments from the output of the pixel

memory, the values read from the pixel data busses, and an internal operand register. The operand register holds a reference value or a constant used in the functions $\mathcal{T}_i$ and $\mathcal{U}_i$. For example, the operand register in the datapath may store a value for initializing the buffer memory. The host can program these blocks by writing an op-code into their internal command registers and by setting the operand registers.

Each buffer controls one or more lines of the pixel result bus. The output of the test block is connected to these lines.

Condition

The entirety of the test results of all buffer modules forms the content of the pixel result bus. This information is used by the condition block to evaluate the update conditions, $\mathcal{C}_i$. This block is implemented as a look-up table built with static RAM. The address of this table is formed by the pixel result bus. Its output is a single bit, the result of the update condition, that signals whether the buffer memory should be updated. For example, if buffer $i$ must be updated only if the test performed by buffer 0 has passed, i. e. if $r_0 = 1$, the look-up table contains a 1 in memory locations with an odd address and a 0 in those with an even address.

### 1.5.2 Control Buffers

Typically, advanced pixel-level rendering algorithms require some control information per pixel (flags [7], counters [6] or masks [8]) which does not require the number of bits typically needed for surface data, is not compared to surface data, and is manipulated through bit-oriented operations. Also, some algorithms must be able to select the update function $\mathcal{U}_i$ depending on the results of all tests $\mathcal{T}_i$,

$$\mathcal{U}_i = \text{op-code}(r_1, r_2, \ldots, r_k) \tag{1.2}$$

Control buffers are specialized to support these operations by differing from the basic architecture of surface buffers (figure 1.3) in the following aspects: The buffer memory is only 8 bits deep. There is no connection to the pixel data busses. The test and update blocks support a different set of operations. For control buffers, the function table internal to the condition block has additional bits which store an op-code in each word according to equation (1.2). These extra bits are connected to the update block and convey an op-code that is selected dynamically depending on the value of the pixel result bus. For example, a counter in control buffer $i$ should be incremented whenever the test performed by buffer 0 passes, i. e. $r_0 = 1$, and should be decremented if that test fails. Then the look-up table in the condition block contains an increment command in all words with an odd address and a decrement command in the even addresses.

### 1.5.3 Virtual Buffers

Various components need to be interfaced to the buffer sub-system, e. g. rasterizer, display sub-system, video input, procedural texture generators, input devices etc. The virtual buffer concept provides a unified view of these components for the M-Buffer. Although their internal structure may be very different, virtual buffers exhibit the same behavior towards the pixel bus as actual buffers by mimicking some or all of the following aspects:

- Placing pixel addresses on the pixel address bus.
- Reading addresses from the pixel address bus.

- Writing pixel data to the pixel data busses.
- Controlling line(s) of the pixel result bus.

For example, the rasterizer is implemented as a virtual buffer. It places pixel addresses and pixel values onto the pixel address bus. For scan-conversion, one line of the pixel result bus may signal whether the scan-converted pixel belongs to a front-facing or a back-facing surface.

The advantage of the virtual buffer concept is that very different devices can be connected to the M-Buffer as long as they show the required behavior on the pixel bus. Since the buffer modules are working synchronously, the speed of a buffer module does not affect the functionality of the entire M-Buffer. However, it does affect its performance.

### 1.5.4 Software Buffers

The M-Buffer architecture does not require the presence of all buffer modules. As in other buffer systems, some buffers will always be implemented in hardware. However, other buffers can also be implemented in software and be interfaced to the hardware as virtual buffers. Although software buffers will slow down the buffer system, they open an avenue for a wide range of cost-performance tradeoffs. Entry-level systems are still able to support rendering algorithms using many buffers without physically providing these buffers. Instead, the host CPU and system memory take the role of the extra buffer modules. Software buffers resemble the virtual pixel maps described in [3].

For example, a basic M-Buffer system may be constructed from only three surface buffers, one depth buffer and two color buffers for double buffering. In order to use that system for (ALGORITHM 1), two software buffers are allocated to accommodate the extra depth buffer Z2 and the pixel mask V.

## 1.6 Programming Interface

We have developed a programming interface for configuring, programming, managing, and interacting with the M-Buffer system.

*Configuration Commands*

determine the overall behavior of the M-Buffer system by defining which buffers are reading and which buffers are writing the pixel bus.

*Programming Commands*

provide access to the internal registers of each buffer module's functional blocks. They let the application define the test function $\mathcal{T}$ and the update operation $\mathcal{U}$ by storing op-codes into the update and test datapaths. The programming interface also allows the application to download tables constituting the update condition $\mathcal{C}$. The operation of the address manager is defined by specifying the offset and the operation mode of the address generator.

*Management Commands*

save and restore the entire status of a selected buffer module on a stack maintained by the programming interface. The state of a buffer module can also be read and written directly by the application. These commands serve two purposes: They give system software a fast and convenient means to change the buffer configuration if a context switch occurs. These commands allow the application to define the necessary buffer configurations outside the

main loop and quickly invoke the active configuration by, for instance, popping it from the stack.

*Interaction Commands*

enable the application to query the status of a buffer module, e. g. whether it has completed its operation and whether it has updated any pixels since the last query for update.

Programming the M-Buffer is done in two steps. First, the overall system is configured and the individual buffers are programmed. Then, a start command is issued to the rasterizer or to the buffer controlling the pixel address bus, thus initiating the actual M-Buffer operation. At any time, the host can query the status of the running operation. After the operation finished, the host may query status information, e. g. whether any pixels were changed, and/or reconfigure the M-Buffer for the next operation.

## 1.7 Conclusion

The M-Buffer graphics architecture supports a new class of graphics algorithms. Its organization has several conceptual and practical benefits.

The buffer modules are universal. The buffer processors are not tailored to specific tasks, but provide the elementary operations necessary for implementing a large family of advanced pixel rendering algorithms.

M-Buffer systems are scalable: They can be easily extended by adding more buffer modules, thus facilitating the task of configuring graphics systems to meet different cost-performance requirements.

Buffer modules can be implemented in software, thus enabling low-cost, high-functionality graphics systems.

The M-Buffer architecture is orthogonal to and thus complements existing graphics buffer architectures that subdivide the screen into sets of pixels that are assigned to one processing element.

The fact that the performance of a M-Buffer system is independent of the complexity of the update condition and the number of buffers to be updated makes this architecture especially suited for advanced pixel-level rendering algorithms.

## 1.8 References

[1] POWER Series, Technical Report. Technical report, Silicon Graphics, Inc., —.

[2] Kurt Akeley. The Silicon Graphics 4D/240GTX Superworkstation. *IEEE Computer Graphics & Applications*, 9(4):71–83, July 1989.

[3] Brian Apgar, Bret Bersack, and Abraham Mammen. A Display System for the Stellar Graphics Supercomputer Model GS1000. *Computer Graphics*, 22(4):255–262, August 1988.

[4] John D. Foley and Andries van Dam. *Fundamentals of Interactive Computer Graphics.* Addison-Wesley Publishing Company, 1982.

[5] Alain Fournier and Donald Fussell. On the Power of the Frame Buffer. *ACM Transactions on Graphics*, 7(2):103–128, April 1988.

[6] Jack Goldfeather, Steven Molnar, Greg Turk, and Henry Fuchs. Near Real-Time CSG Rendering Using Tree Normalization and Geometric Pruning. *IEEE Computer Graphics & Applications*, 9(3):20–28, May 1989.

[7] Abraham Mammen. Transparency and Antialiasing Algorithms Implemented with the Virtual Pixel Maps Technique. *IEEE Computer Graphics & Applications*, 9:43–55, July 1989.

[8] Jarek Rossignac and Jeffrey Wu. Correct shading of regularized CSG solids using a Depth-Interval Buffer. In *Proceedings of the Eurographics Workshop on Graphics Hardware, 1990.* Eurographics, September 1990.

## Appendix: A Working Example

The following example will show the practical use of the concepts presented in the main part of the paper. It implements the multipass transparency algorithm presented in [7]. The algorithm uses two z-buffers (Z1, Z2), two frame buffers (F1, F2), and one bit-plane for a visit flag (V). The algorithm proceeds in two steps: (1) All opaque objects are scan-converted into the first z-buffer and the first frame buffer. (2) The transparent objects are scan-converted into the second z-buffer/frame-buffer pair. The pixel information is only written if the depth of the scan-converted pixel is between the depths of the first and the second z-buffer. Every pixel updated in this way is marked by setting the visit flag. After all transparent objects have been processed, the colors of the two frame buffers are merged using alpha-blending and the depth in z-buffer 2 is copied into z-buffer 1. Step 2 is repeated until no pixel is changed, i. e. all pixel flags are Zero.

The following pseudo-code describes how this algorithm is implemented using pre-defined configurations of the M-Buffer:

MULTIPASS TRANSPARENCY

```
/*
 * Step 1: Scan-convert opaque objects
 */
use_config (depth_buffer) ;
init (Z1, ∞) ; init (F1, background) ;
for (all opaque objects o)
    scan_convert (o) ;

/*
 * Step 2: Iterate scan-converting the transparent
 *         objects until no pixels are changed
 */
repeat
{ use_config (depth_interval_buffer) ;
  init (Z2, 0) ; init (V, 0) ; reset_bbx (V) ;
  for (all transparent objects t)
      scan_convert (t) ;
  box = get_bbx (V) ;
  done = (box.l > box.r) && (box.t < box.b) ;
  if (!done)
  { use_config (buffer_transfer) ;
    scan_bbx (V) ;
  }
} until (done) ;
```

After the required buffer configuration has been retrieved by using the *use_config* commands, one buffer is started to generate pixels with the commands. The command *scan_convert* generates all pixels covered by the specified object. The command *scan_bbx* produces pixel addresses for all pixels inside a screen region. The extent of a buffer's bounding box is obtained by *get_bbx*. The *init* command saves the contents of the specified buffer onto the stack, configures it to write the specified value into all its pixel locations, starts the buffer, and finally restores the original buffer contents.

Below, we show (simplified) pseudo-code segments defining various buffer configurations. Typically, they would be available as a library of pre-defined buffer configurations. The *enable* and *disable* commands do the obvious for the specified buffers. Commands starting with *load_* program the respective block in the buffer module defined by the first argument. The *read_* and *write_* commands define which sub-bus of the pixel bus is read or written by the specified buffer. The contents of a M-Buffer configuration is defined using a *make_config* ... *close_config* pair and is activated with the *use_config* command. The command *track_changes* programs the address generator of the specified buffer to construct a bounding box around updated pixels.

The following pseudo-code uses several global symbols, e. g. Z1 has been defined as the address of a particular surface buffer and R is the address of the rasterizer's virtual buffer. The symbols PDATA1 and PDATA2 stand for the values on the respective pixel data busses. MEM stands for the value read from the internal memory. The values $r[\,]$ are the result bits returned by the indicated buffers. The scan-conversion routine places

the depth-value $s_z$ on the first and the color value $s_c$ on the second pixel data bus.

DEPTH BUFFER

```
make_config (depth_buffer) ;
disable (Z2) ; disable (F2) ;
enable (Z1) ; enable (F1) ;
disable (V) ; enable (R) ;
read_addr (Z1) ; read_data1 (Z1) ;
read_addr (F1) ; read_data2 (F1) ;
write_addr (R) ; write_data1 (R) ; write_data2 (R) ;
load_test (Z1, PDATA1 < MEM) ;
load_update (Z1, PDATA1) ;
load_update (Z1, PDATA1) ;
load_condition (Z1, r[Z1]) ;
close_config() ;
```

DEPTH-INTERVAL BUFFER

```
make_config (depth_interval_buffer) ;
enable (Z2) ; enable (F2) ;
enable (Z1) ; disable (F1) ;
enable (V) ; enable (R) ;
read_addr (Z1) ; read_data1 (Z1) ;
read_addr (Z2) ; read_data1 (Z2) ;
read_addr (F2) ; read_data2 (F2) ;
read_addr (V) ;
load_test (Z1, PDATA1 < MEM) ;
load_test (Z2, PDATA1 > MEM) ;
load_update (Z2, PDATA1) ;
load_update (F2, PDATA2) ;
load_update (V, 1) ;
load_condition (Z1, never) ;
load_condition (Z2, r[Z1] && r[Z2]) ;
load_condition (F2, r[Z1] && r[Z2]) ;
load_condition (V, r[Z1] && r[Z2]) ;
track_changes (V) ;
write_addr (R) ; write_data1 (R) ; write_data2 (R) ;
close_config() ;
```

BUFFER TRANSFER

```
/* Configuration for conditional buffer transfer:
   if (V) then Z1,I1 = Z2,I2 */
make_config (buffer_transfer) ;
enable (Z2) ; enable (F2) ;
enable (Z1) ; enable (F1) ;
enable (V) ;  disable (R) ;
read_addr (Z1) ; read_data1 (Z1) ;
read_addr (F1) ; read_data2 (F1) ;
read_addr (Z2) ; write_data1 (Z2) ;
read_addr (F2) ; write_data2 (F2) ;
write_addr (V) ;
load_test (V, MEM == 1) ;
load_update (Z1, PDATA1) ;
load_update (F1, blend(MEM,PDATA2) ) ;
load_condition (Z1, r[V]) ;
load_condition (F1, r[V]) ;
load_condition (Z2, never) ;
load_condition (F2, never) ;
load_condition (V, never) ;
close_config() ;
```