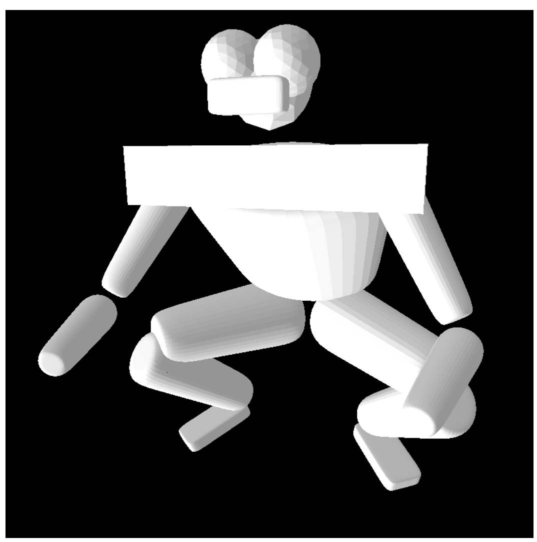
Efficient Rendering on Massively Parallel Architectures

Matthew Nicholas Pappas, Oren J. Tversky and D. Brookshire Conner

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-92-46
October 1992

Efficient Rendering On Massively Parallel Architectures



Matthew Nicholas Pappas
Oren J. Tversky
D. Brookshire Conner

1.0 Introduction

In this paper, we discuss the design and implementation of some three-dimensional rendering algorithms on massively parallel computers. These algorithms employ SIMD-style parallelism. That is, each processor runs the same instructions on different data. We have implemented some of these algorithms on the Connection Machine 2, a computer manufactured by the Thinking Machines Corporation.

This paper should be accessible to people interested in computer graphics or massively parallel algorithms. We have taken great care to introduce both fields sufficiently that this paper should be quite self-contained. As a result, this paper should be readable by anyone familiar with some elementary theoretical computer science.

This paper is intended both for the graphics community and for the parallel processing community. In view of this, we have felt it necessary to review certain aspects of both areas, resulting in a lengthy report. The reader who finds our explanations of either area too terse should refer to the references for further reading. [1] [2]

1. James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes, *Computer Graphics: Principles And Practice*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1990.

2. F. Thomson Leighton, *Introduction To Parallel Algorithms And Architectures: Arrays, Trees, Hypercubes*, Morgan Kaufmann Publishers, San Mateo, California, 1992.

2.0 Terminology

- The symbol p is used exclusively to denote the number of physical processors in a massively parallel computer. This symbol is introduced in section 2.2.
- The symbols O, o, Ω, ω, and Θ are used exclusively to denote asymptotic relationships between functions. These symbols are introduced in section 2.1.
- The symbol f is used to denote the number of *candidate pixels* or *fragments* generated by the serial z-buffer algorithm. This symbol is introduced in section 3.1.
- The symbol s is used exclusively to denote the number of *spans* generated by a given triangle. A span is a horizontal group of connected fragments, and is introduced in section 7.1.

2.1 Asymptotic Orders Of Growth

In the following discussion, we employ some standard notation from computer science literature.

- $f(n) = O(g(n))$ indicates that the function $f(n)$ has an asymptotically smaller or equivalent order of growth than the function $g(n)$.
- $f(n) = \Omega(g(n))$ indicates that the function $f(n)$ has an asymptotically larger or equivalent order of growth than the function $g(n)$.
- $f(n) = o(g(n))$ indicates that the function $f(n)$ has an asymptotically strictly smaller order of growth than the function $g(n)$.
- $f(n) = \omega(g(n))$ indicates that the function $f(n)$ has an asymptotically strictly larger order of growth that the function $g(n)$.
- $f(n) = \Theta(g(n))$ indicates that the function $f(n)$ satisfies both $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

The recent book by Cormen, Lieserson, and Rivest contains an excellent introduction to this notation.[3]

2.2 Parallel Performance Metrics

Consider an algorithm which completes a computation in t steps on machine with p processors. The *work* of the parallel algorithm is defined as $t \times p$, and is denoted by w.[4]

Let τ denote the time required to solve the same problem using the best known sequential algorithm. It is obvious that $\tau \le w$ since the problem can be solved on the serial machine by simulating the parallel machine on the serial processor. The simulation will take time

3. Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest, *Introduction To Algorithms*, M.I.T. Press, Cambridge, Massachusetts, 1991.

4. F. Thomson Leighton, *Introduction To Parallel Algorithms And Architectures: Arrays, Trees, Hypercubes*, Morgan Kaufmann Publishers, San Mateo, California, 1992.

proportional to the time required by the parallel algorithm multiplied by the number of parallel processors that must be simulated serially. Serial simulation can be done in time proportional to the work of the simulated parallel algorithm.

On the other hand, the best serial algorithm to solve a given problem will usually do better than simulating the best known parallel algorithm. For many interesting problems, $w > \tau$. *Efficiency*, denoted by e, is defined as τ / w and is at most one. Restated, $e = (\tau \times p) / t$. Intuitively, parallel algorithms with efficiency equal to one are able to achieve a linear speed-up as processors are added to the system. For many problems it is impossible to achieve linear speed-up, but optimal parallel efficiency remains an important goal.

3.0 The Serial Algorithms

3.1 The Serial Z-Buffer Algorithm

We have crafted an algorithm in the style of the normal serial z-buffer algorithm. The z-buffer algorithm has been used very successfully to date in the graphics community for both real-time applications and high-quality rendering. For a broad range of effects at a limited cost, the z-buffer is an excellent algorithm. We will not attempt to examine other rendering methods in this paper.[5]

The framebuffer contains two locations per discrete screen position (or *pixel*). The first storage location contains a color value, and the second contains a z-value. The z-value locations of the framebuffer are referred to collectively as the *z-buffer*.

The z-buffer algorithm operates by considering each primitive in sequence. For each primitive, the z-buffer algorithm generates all pixels which that primitive may influence, sequentially. These candidate pixels or *fragments* are then sent to a framebuffer. When a fragment has been generated it is sent to the framebuffer. The fragment color and fragment z-value are both sent. The incoming fragment z-value is compared with the z-value already in the z-buffer, and new fragment color value and z-value are stored if and only if the fragments's z-value is smaller than the z-value already in the z-buffer.

The theoretical running time of the serial z-buffer algorithm is $\Theta(f)$ where f is the number of fragments in the scene. The lower-bound on any parallel z-buffer is therefore $\Omega(f/p)$. We will get surprisingly close to this asymptotic goal.

3.2 Spline Tesselation

Most z-buffers employ simple primitives, such as planar polygons, often only triangles. Such primitives can be used to approximate curved surfaces, but require a great deal of space to produce an adequate approximation. Spline curves are one method to describe smooth shapes using less space than required by polygons.[6]

5. James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes, *Computer Graphics: Principles And Practice*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1990.

6. James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes, *Computer Graphics: Principles And Practice*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1990.

3.2.1 Computing The Rational Polynomial Coefficients

The spline surfaces we used are called *rational tensor-product bicubic* surfaces. They are rational bicubics because they represent a function from $\Re^2 \to \Re^3$, with each domain direction being a rational cubic function. Precisely,

$$F(u, v) = \begin{bmatrix} f_1(u, v) \\ f_2(u, v) \\ f_3(u, v) \end{bmatrix} / f_4(u, v)$$

where

$$f_i(u, v) = \sum_{k=0}^{3} \sum_{j=0}^{3} c_{ijk} u^j v^k$$

and the c_{ijk} are the coefficients derived below. Cubic functions are employed because they represent a good compromise between flexibility (having just enough degrees of freedom to produce an s-curve) and controllability (since higher-order functions have more degrees of freedom). Rational functions are employed because they allow the representation of conic sections.

The geometry of a tensor-product surface can be represented by a 4×4 array of four-dimensional points, referred to as the *geometry array* or *control hull*, g_{ijk}. This produces a three-dimensional array of real numbers, termed a tensor. This tensor can be multiplied by basis matrices to produce coefficients of cubic polynomials. These coefficients are the c_{ijk} used in the cubic functions defined above. The c_{ijk} are defined by

$$\begin{bmatrix} c_i \end{bmatrix}_{jk} = \begin{bmatrix} B \end{bmatrix}_{jk} \times \begin{bmatrix} g_i \end{bmatrix}_{jk}$$

where $\begin{bmatrix} B \end{bmatrix}_{jk}$ denotes the basis matrix, i.e. a two-dimensional matrix, subscripted by j and k.

3.2.2 Covering The Spline Patch With Triangles

There are a variety of techniques for rendering the spline surface. The one that we will employ in this paper is to cover the spline surface with triangles. While this is not necessary, and other methods are in common use, this technique is easy to implement, and has been proven to be quite fast for a number of production software systems and special-purpose hardware.

The spline coefficients can be used to generate a grid of points which lie on the surface (often referred to as a *tesselation grid*). The coefficients can be expressed as four 4×4 arrays, one per dimension of the original points. By multiplying one of these arrays by a vector of a parametric value expressed as a cubic (i.e., $\begin{bmatrix} u^3 & u^2 & u & 1 \end{bmatrix}$), the array of coefficients becomes a single vector of four coefficients. This vector can be multiplied by a second parametric value expressed as a cubic (i.e., $\begin{bmatrix} v^3 & v^2 & v & 1 \end{bmatrix}$), producing a single coordinate value. Finally, the fourth coordinate dimension is used as a normalization factor to account for the rational bicubic denominator.

Once the algorithm has generated a rectilinear grid of surface positions, this surface tesselation can be easily transformed into a list of triangles or triangle strips. This is performed by effectively stitching the grid positions together into triangles that cover the surface.

3.2.3 Adaptive Subdivision

Determining an appropriate number of grid points is difficult. If the patch is under-tesselated, then its curvature will not be represented realistically. If the patch is over-tesselated, then the triangle renderer will receive voluminous numbers of triangles which redundantly represent flat portions of the surface. A simple solution is to simply chose some constant number of tesselation grid points for every patch. This is in general a poor solution, but it is straightforward for a first implementation. More elegant solutions that consider surface curvature are referred to as *adaptive subdivision* techniques. Complex adaptive subdivision techniques need not necessarily even use a uniform rectilinear grid, but these methods are beyond the scope of this paper.

4.0 Fundamental Parallel Algorithmic Techniques[7]

4.1 The Virtual Processor Paradigm

The fundamental algorithms described in this section all employ a notion of virtual processors.[8] Simply put, each physical processor simulates some larger number of virtual processors. On a given processor, all virtual processors behave entirely independently. This simple paradigm operates by simulating the action of virtual processors through looping. At no time is information exchanged between the virtual layers on a given physical processor during normal operation. (Communication steps can induce intra-processor communication, but this is a more general and expensive mechanism.) In effect, all communication between data-elements requires a communication step, even if that information is local on a given physical processor. This paradigm can be constraining, especially if adjacent data-points on the same physical processor wish to propagate information, for example, in forward differencing. We will address extensions to this paradigm later in section 8.0.

One interesting metric for processor virtualization is the virtualization ration, defined as:

$$r = \frac{v}{p}$$

where r is the virtualization ratio, v is the number of virtual processors, and p is the number of physical processors.

4.2 Semigroup Operators

We will discuss some applications of semigroup operators in the following discussion of reductions and prefix scans. Technically, semigroups are composed of a set with an operator which maps two elements of the set to a new element of the set. The operator must be associative, and there must be an *identity* element in the set.

Add, *multiply*, *logical or*, and *logical and* perform the obvious integral, floating-point, or boolean calculations. Note that division and subtraction are not semigroup operators since they are not associative.

The *copy* binary semigroup operation simply returns its left operand, unless that operand is the identity, in which case it returns its other operand. This is useful in conjunction with segmentation and monotonic communication. The copy is clearly associative.

4.3 Global Reduction

One of the most useful algorithms on a parallel machine is the *global reduction algorithm*. The reduction problem is to determine the "sum" of some list of values under a semigroup

7. F. Thomson Leighton, *Introduction To Parallel Algorithms And Architectures: Arrays, Trees, Hypercubes*, Morgan Kaufmann Publishers, San Mateo, California, 1992.

8. *The Connection Machine: CM-200 Series Technical Summary*, Thinking Machines Corporation, Cambridge, Massachusetts, 1991.

operation. One example of the application of this algorithm is to compute the integer sum of a list of integral values (see figure 1). Another example is to compute the boolean logical-or of a list of boolean values. The implementation of this algorithm exploits the properties of semigroups to solve this problem in time $\Theta(n/p + \log p)$. This algorithm is very fast in practice.

FIGURE 1 Global Add Reduction

2	1	0	0	3	5	1	0	7	4	2	3	0	1	10	0	1	5

$\downarrow$

45

4.4 Segmented Prefix Scan

The *prefix scan problem* is similar to the global reduction problem. The algorithm also operates on a list of values with a semigroup operation. The algorithm requires a numbering of the processors, i.e. each processor mush have a unique index. The prefix scan generates for each datum in the list (i.e. processor), one output value. The output value on each processor is the sum of the data residing on all processors before this processor (induced by the processor ordering). That is, each processor computes the prefix sum of all elements before its own element. One example of an application of this algorithm is to compute the integer prefix sums of a list of integral values (see figure 2).

FIGURE 2 Prefix Add Scan

2	1	0	0	3	5	1	0	7	4	2	3	0	1	10	0	1	5

$\downarrow$

0	2	3	3	3	6	11	12	12	19	23	25	28	28	29	39	39	40

Note that the last processor's value is not used in the computation of any of the partial sums. This is sometimes called an exclusive prefix scan for this reason.

The simple prefix scan algorithm operates deterministically in time $O(n \log n/p)$ where n is the number of operands. If $n = \omega(p)$ (i.e., $n \gg p$), a somewhat more sophisticated prefix scan algorithm (with pipelining) operates deterministically in time $O(n/p)$. In practice, the prefix scan algorithm is extremely fast.

Rather surprisingly, the prefix scan algorithm can also solve the *segmented prefix scan* problem in the same (asymptotic) amount of time. Segment boundaries are marked by a Boolean field per processor which is non-zero at the beginning of each segment. The segmented problem is to perform the prefix scan, but accumulate partial sums only up to explicitly marked segment boundaries. In effect, the segmented prefix scan allows the user to produce independent prefix scans on data consisting of independent sets with varying numbers of elements. An example of the application of the segmented prefix scan is in figure 3.

FIGURE 3 Segmented Prefix Add Scan

Boolean field	1	0	0	0	1	0	1	1	0	0	1	0	0	0	1	0	0	0
scan operand	2	1	0	0	3	5	1	0	7	4	2	3	0	1	10	0	1	5
	0	2	3	3	0	3	0	0	0	7	0	2	5	5	0	10	0	11

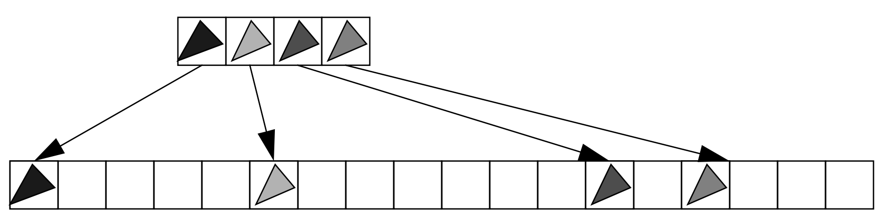
Note that the segment bits denote the beginning of a new segment, so the last value in each segment is not used in any of the partial prefix sums. Again, this is an exclusive segmented prefix scan.

4.5 Monotonic Communication

Given a problem where each processor needs to send a piece of data to some other processor, routing this data to the target processor addresses can be quite computationally costly. In some special cases, this communication is simple and has good worst-case asymptotic behavior. One such case is that of monotonic communication.

Suppose each processor has at most one datum and one destination for that datum. If the reordered data after the communication step has its order preserved, then the communication is monotonic. The class of monotonic communication problems includes the special cases of packing and spreading of data. An example of spreading data is in figure 4.

FIGURE 4 Spreading Data, A Form Of Monotonic Communication



Monotonic communication operates in $O((n/p)\log p)$ on a hypercube communication network.

4.6 Data Distribution And Spreading

One useful application of both the segmented prefix scan and monotonic communication is the *data distribution and spreading problem*. We will use this technique extensively in the algorithms described later in this paper, so it is worthwhile to examine it in a general setting now.

Consider that a certain segment of an algorithm may require a different number of data elements than previous sections of the algorithm. We will use triangles and spans as example data elements in this section, although this algorithm will be applied to other datatypes throughout this paper. For example, it is natural to perspective-transform triangles on a *processor-per-triangle* basis; whereas later in the algorithm it is natural to scan-convert the triangles on a *processor-per-span* basis. (If triangles and spans are unfamiliar to the reader, refer to section 7.1.1, although the technical details are really unimportant for the sake of this discussion.)

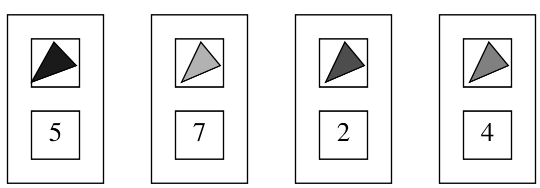
The problem that we will tackle is to transform a virtualization into a new virtualization in which each original data element in the first virtualization is allocated some number of virtual processors in the second virtualization. Then, the source processors must send their data to all destination processors which have been allocated for its use. This is the *remapping* problem.

Consider the problem from the perspective of the data layout convenient for the processor-per-triangle data virtualization. Each triangle processor must determine an *expansion factor* which quantifies the number of span processors that the given triangle processor will require. The data must then be sent from the processor-per-triangle layout to the new processor-per-scan layout.

The *revirtualization* or *remapping* problem is performed in the following steps.

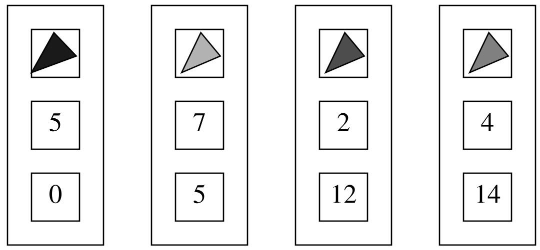
1. Each (triangle) processor determines its *expansion factor*, the number of processors that it will require in the new (span) virtualization. See figure 5 for an illustration of the process.

FIGURE 5 Computing The Expansion Factors



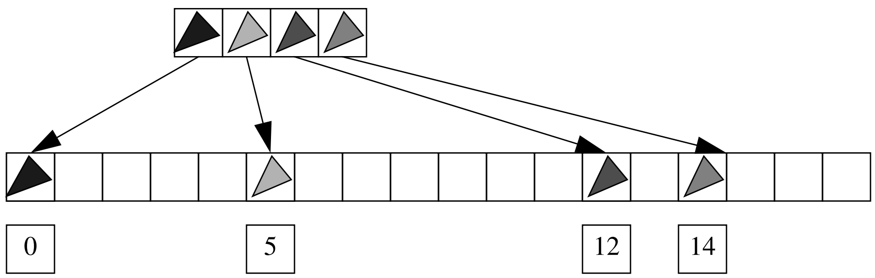
2. Each (triangle) processor determines its destination address, leaving room for the (span) expansion. This can be performed with a (non-segmented) prefix scan operation, described in section 4.4, using the *add semigroup operator* described in section 4.2. See figure 6 for an illustration of this process.

FIGURE 6 Computing The Destination Addresses



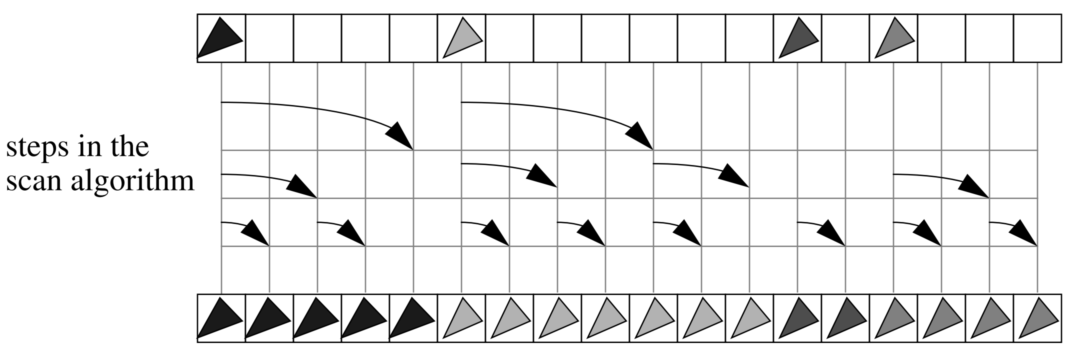
3. Each (triangle) processor sends its (triangle) data element to the destination address computed in step 2. This can be performed with a *spreading* operation, described in section 4.5. See figure 7 for an illustration of this process.

FIGURE 7 Distributing Triangles To The Span Processors



4. Each (span) processor spreads its received (triangle) data element (from step 3) to all processors following it, up until the next (span) processor which received data (in step 3). This can be performed with a *segmented prefix scan* operation, described in section 4.4, using the *copy semigroup operator* described in section 4.2. Segment boundaries begin on (span) processors which received data in step 3. See figure 8 for an illustration of this process.

FIGURE 8 Spreading Triangles Along The Span Segments



Thus, each span processor will now contain the primitive from which it will now be computed. The best part about this algorithm is that it runs in time $O((n/p)\log p)$, where n is the larger of the number of input data elements (primitives) and the number of output data elements (spans). It is extremely fast in practice.

4.7 Local Memory Indirection

Each processor on recent massively parallel machines have had hardware support for *local memory indirection*.[9] This means that each processor can employ a local pointer into its own memory. When such a local pointer is dereferenced, each processor retrieves a value from a position in its local memory. The positions can be different on different processors. This capability can be very useful, and is a slight deviation from the strict SIMD paradigm of earlier machines. This capability is quite common now, although we will see later that it can be quite an expensive operation. This will in fact be problematic in the triangle strip renderer of section 8.0, and will again be addressed in section 11.0.

4.8 Two-Dimensional Grid Communication

We have seen that the hypercube communication network is quite useful in section 4.3, section 4.4, and section 4.5. While this is true, the hypercube network is expensive to manufacture, and it is often rather slow in practice. There is a lot of overhead in using the hypercube.

Another common communication network on massively parallel machines is the two-dimensional grid network. In a grid, each processor is connected to its nearest neighbors (either four or eight) in a two-dimensional grid layout. When processors communicate on the grid, each processor must send data in the same direction and for the same distance at the same time.

This is much different than the hypercube, where each processor can attempt to send data to any other processor at any time. The hypercube can unfortunately suffer from serious collision or contention problems if different data collides at any stage in the multistage hypercube communication network. We have been careful to avoid or minimize the effects of this collision in the hypercube algorithms that we employ, but in general this is a difficult task.

The grid has no collision or contention problems whatsoever. When grid communication is appropriate, it is very fast indeed. And the hardware is easy to manufacture, so the overhead and latency are much lower than the hypercube. Grid communication will be employed in section 8.0.

4.9 Many-To-One Packet Routing With Combining

Consider the problem of sending fragments to the z-buffer, preserving for each pixel only the fragment closest to the camera. This problem is technically known as a *many-to-one packet routing* problem in the parallel algorithms literature. Any algorithm which performs this communication must trivially be $\Omega(n/p)$ (this is a lower bound). Three algo-

9. Sherryl Tomboulian and Matthew Pappas, "Indirect Addressing And Load Balancing For Faster Solution To Mandelbrot Set On SIMD Architectures," *The Third Symposium On The Frontiers Of Massively Parallel Computation*, I.E.E.E., 1990.

rithms are presented below, with varying performance in theory and practice. In theory, the lower bound is within reach. We have implemented the first two of the following algorithms.

4.9.1 Z-Buffer Sending With Collisions

The simplest means of moving the fragments to the framebuffer is to simply send them to the software z-buffer using the hypercube. The Connection Machine's libraries[10] provide routines which send data in a virtual processor dataset to another virtual processor dataset with arbitrary destinations. If there are collisions in the target dataset, a combiner function can be provided which resolves the conflicts. This sending algorithm operates in the obvious naive way: the function loops until all data has reached its destination, sending non-colliding source data items to the target dataset, combining those source items at the destination when there are collisions. Since each processor can only receive one datum per machine cycle, looping is required to handle the collisions. For the purposes of z-buffering, the combiner simply chooses the fragment with a minimal z-value. The running time of the algorithm is $O(n)$ in the worst case, where n is the number of data elements sent. This is theoretically abysmal, but this simple algorithm actually performs well in preliminary benchmarks on a simple scene. It is expected to perform hideously in certain cases which may arise in practice, for example when the complexity of the scene is concentrated in a small area of the screen.

4.9.2 Sorting With Prefix Scan

One algorithm which performs the many-to-one packet routing problem moderately well in theory is the *sorting/prefix scan algorithm*. The algorithm first uses the screen-space z-buffer destinations, which are (x,y) pairs, to compute a sorting key. The key is defined as $k = y \times w + x$ where w is the width of the screen in pixels. This key obviously induces a row-major ordering of the screen pixels. The fragments are then sorted using this key. Sorting can be easily be made to run in time $O((n \log^2 n) / p)$ using one of the deterministic algorithms like the Batcher sort.[11] For the case when $n = \omega(p)$, sorting runs in time $O((n \log n) / p)$.

After the sort, row-contiguous pixels on the screen are located in adjacent virtual processors in the Connection Machine. Now, the processors containing the sorted fragments are segmented in preparation for a segmented prefix scan. Segment boundaries are raised at boundaries between fragments located at different screen pixels, i.e. with different sorting keys. A segmented prefix scan can now be used to find a closest pixel at each screen pixel by performing a segmented prefix scan, using the a combiner which selects the fragment with a minimal z-value. Again, the segmented prefix scan operates in time $O((n \log n) / p)$.

10. *C* Programming Guide*, Thinking Machines Corporation, Cambridge, Massachusetts, 1991.

11. F. Thomson Leighton, *Introduction To Parallel Algorithms And Architectures: Arrays, Trees, Hypercubes*, Morgan Kaufmann Publishers, San Mateo, California, 1992.

Combining the sort with the segmented prefix scan, the sorting/prefix scan algorithm thus runs in time $O((n\log^2 n)/p)$. When $n = \omega(p)$, the algorithm performs in time $O((n\log n)/p)$. The sorting/prefix scan algorithm does not perform as well as the simpler z-buffer sending with linear collision combining described above in a simple test case. It is unclear whether it will outperform the z-buffer sending on average real scenes. The overhead for this algorithm is very high, since the optimal sorting algorithms are complex.

4.9.3 Ranande Combining

An algorithm which performs the many-to-one packet routing problem extremely well in theory is *randomized Ranande packet routing*.[12] The development of the algorithm is rather complicated. Ranande's basic algorithm implements randomized sorting on a butterfly network or hypercube. The fundamental technique employed is that the algorithm attempts to merge packets going to the same destination after each communication step on the butterfly. It employs pseudo-random (i.e., reproducible) hash-functions to guarantee that at each stage of sorting packets, at most two packets collide at any processor. This guarantee cannot be met deterministically, but it can be met with high probability. Since this algorithm is randomized, the worst-case routing performance will not be elicited by any particular input. For any given input, the algorithm will need to combine only two packets per processor per butterfly communication step with high probability. This conveniently low-collision communication avoids the abominable worst-case performance of the z-buffer sending algorithm described above.

The performance of the Ranande many-to-one packet routing algorithm is $O(n/p + \log p)$ with high probability, assuming that $n = \omega(p)$. This algorithm involves explicit processor virtualization and some tricky hash functions, so it will probably take substantial effort to implement. It is unclear how this algorithm will perform in practice. It performs extremely well in theory, considering that the theoretical lower bound on running time is $\Omega(n/p)$.

Unfortunately, the Connection Machine's high-level languages prohibit implementing an algorithm such as this. In order to perform computations between intermediate hypercube communication steps, the CMIS assembly language must be used. This is an arduous task which we have not yet undertaken.

12. F. Thomson Leighton, *Introduction To Parallel Algorithms And Architectures: Arrays, Trees, Hypercubes*, Morgan Kaufmann Publishers, San Mateo, California, 1992.

5.0 Common Rendering Techniques

Some details of the following implementations are shared. In particular, primitive loading, viewing transformation, shading, clipping, and z-buffer image composition are similar in most of the implementations. Those common components are described here.

5.1 Parallel Primitives And Parallel Pixels

There are several reasons why we are forced to incorporate parallelism into every aspect of parallel rendering. We want to handle extremely large scenes, so processor per primitive approaches are necessary. On a parallel machine like the Connection Machine, data for I/O must be uniformly distributed across the processors for optimal transfer rates. Since a rendering system should have good I/O rates, and the Connection Machine processors each have a very limited amount of memory, the software framebuffer (and z-buffer) must be distributed across the processor array. Thus, processor per pixel approaches are also necessary.

Our algorithms all operate on data-sets whose number of rendering primitives may exceed the number of physical processors on the machine. The most massively parallel systems in use at the present time have on the order of 64k processors. Many interesting rendering scenes have many more primitives than this limit. Also, high resolution images require at least 1280 by 1024 pixels, which totals over one million pixels. We are compelled to consider the issue of data virtualization.

5.2 Loading Primitives Into The Parallel Machine

We tried two methods of loading primitives into the machine. In the first, we wrote a LEX scanner and YACC parser for an ascii file format. Thus, the brunt of input parsing fell entirely on the SUN workstation which acts as the front-end to the Connection Machine. Predictable, the performance for large input files was abysmal. The second technique that we employed was to generate the input data on the Connection Machine itself by, for example, evaluating regularly-spaced points on spheres. This was much more successful, since we could bring the power of the parallel machine to bear on the generation of the input data. We simply ignored the effects of input and output on the programs, since these I/O problems require hardware solutions which remain unavailable to us at this time.

5.3 Viewing Transformation And Shading

Standard viewing, including perspective, transformations are applied at the early stages of each algorithm.[13] These do not differ from their serial counterparts, so we will gloss over their mundane details. Similarly, flat shading calculations are performed before the viewing transformations, and the shaded color values are propagated to later stages of the algorithm. Ultimately these shaded colors are merged into the z-buffer during image composition.

13. James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes, *Computer Graphics: Principles And Practice*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1990.

5.4 Clipping

All of the algorithms involve a z-buffer image composition step during which fragments are merged into the z-buffer. Clipping of the fragments to the window boundaries is performed immediately prior to this step. We mention in the discussions of the algorithms below when we were able to perform larger-grained clipping earlier in the algorithms.

5.5 Z-Buffer Image Composition

At the end of each algorithm, when all the fragments have been generated, image composition is performed. In this step, we use the techniques of section 4.9 to combine fragments so that, for any given pixel, only the fragment closest to the camera is visible. Precisely, we use the technique of section 4.9.1. We also tried the sort of section 4.9.2, with far inferior performance results.

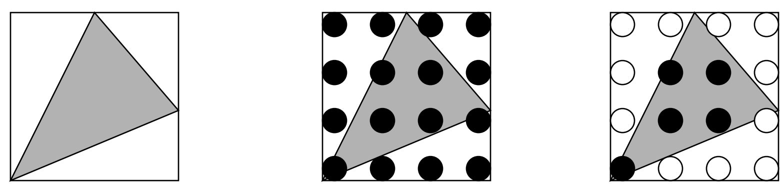
6.0 Footprint Triangle Renderer

The footprint renderer employs two data virtualizations, the triangle virtualization and the fragment virtualization. The renderer first loads triangles into the triangle virtualization, applies the viewing transformation, shades the triangles, and clips triangles.

6.1 Fragment Expansion Factors

For each triangle, the renderer now computes a screen-axis-aligned two-dimensional bounding box. An example of such a bounding box is illustrated in figure 9.

FIGURE 9 A Screen-Axis-Aligned Two-Dimensional Bounding Box For A Triangle



The renderer now computes the area of this bounding box. This area is used as the expansion factor for a data revirtualization, as described in section 4.6. The new fragment data virtualization has one virtual processor for each integer lattice-point contained within the bounding box for any primitive. This data revirtualization propagates the primitives to those fragment processors which are now responsible for that (potential) fragment of that primitive. This is often a large number of virtual processors.

6.2 Fragment Generation

Now each fragment processor determines whether the integer lattice-point that it represents (in its primitive's bounding box) intersects its primitive. This intersection test is performed by solving three half-plane equations to determine whether the integer lattice-point is inside all three sides of the triangle. Each fragment processor also determines the z-value for each fragment by solving the triangle's plane equation at the integer lattice-point. These operations are akin to traditional ray-tracing, and the details of their implementations are similar.[14]

6.3 Finalization

The renderer now discards clipped fragments and performs the z-buffer image composition.

14. *Graphics Gems*, editor Andrew S. Glassner, Harcourt Brace Jovanovich, Boston, 1990.

6.4 Analysis

The time required by this algorithm is $\Theta((f_b/p)\log p)$ where f_b is the number of potential fragments generated, i.e. the sum of the areas of the bounding boxes. Empirically, the ratio of the number of fragments in the bounding boxes to the number of fragments actually present in the triangles, i.e. f_b/f is usually between six and eight. This is unfortunate, and limits the performance of this algorithm severely. Strangely, this is the algorithm employed by the Pixel-Planes Five architecture developed at the University Of North Carolina at Chapel Hill.[15] Although this is only a constant factor in runtime and does not affect its asymptotic performance, it is a large constant factor. This is addressed and solved in the two-pass renderer in section 7.0.

Another deficiency of this algorithm is that it does not perform forward differencing to generate span-triangle horizontal intercepts and fragment z-values. This means that the fundamental computations at the heart of this algorithm involve multiplies and divides, whereas the techniques of forward differencing could reduce these to additions. This is a significant source of optimization which is addressed in section 8.0.

15. Henry Fuchs *et al*, “Pixel Planes 5: A Heterogeneous Multiprocessor Graphics System Using Processor-Enhanced Memories,” *Proceedings of SIGGRAPH ’89, Computer Graphics*, vol. 23, no. 3, 1989, ACM SIGGRAPH, New York, pp. 79-88.

7.0 Two-Pass Triangle Renderer

The two-pass renderer employs three data virtualizations, the triangle virtualization, the span virtualization, and the fragment virtualization. The renderer first loads triangles into the triangle virtualization, applies the viewing transformation, shades the triangles, and clips triangles.

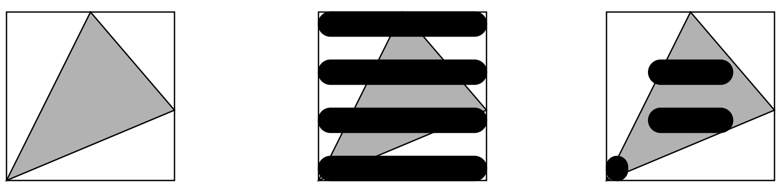
7.1 Triangles

First triangles are transformed into spans. A *span* is a horizontally contiguous group of fragments within one triangle. The span term is commonly used in serial triangle renderers.[16]

7.1.1 Span Expansion Factors

For each triangle, the renderer now computes a screen-axis-aligned two-dimensional bounding box. An example of such a bounding box is illustrated in figure 10.

FIGURE 10 A Screen-Axis-Aligned Two-Dimensional Bounding Box For A Triangle



The renderer now computes the height of this bounding box. This height is used as the expansion factor for a data revirtualization, as described in section 4.6. The new span data virtualization has one virtual processor for each integer lattice-span contained within the bounding box for any primitive. This data revirtualization propagates the primitives to those span processors which are now responsible for that span of that primitive. This number of spans will be smaller or equal to the number of fragments, f, required for the scene.

7.1.2 Span Generation

Now each span processor determines where the integer lattice-span that it represents (in its primitive's bounding box) intersects its primitive. Ignoring degenerate cases for the sake of this discussion, the triangle will intersect in two places on the span. This calculation involves solving two (or three in practice) line-intercept equations to determine the intercept of the left and right edges of the triangle with this span.

16. James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes, *Computer Graphics: Principles And Practice*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1990.

7.1.3 Finalization

The renderer now discards clipped spans.

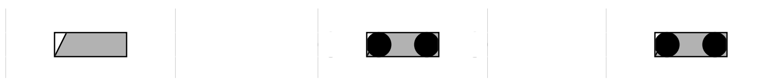
7.2 Spans

Next spans are transformed into fragments.

7.2.1 Fragment Expansion Factors

For each triangle, the renderer now computes a screen-axis-aligned two-dimensional bounding box. An example of such a bounding box is illustrated in figure 10.

FIGURE 11 A Screen-Axis-Aligned One-Dimensional Bounding Box For A Span



The renderer now computes the width of this span. This width is used as the expansion factor for a data revirtualization, as described in section 4.6. The new fragment data virtualization has one virtual processor for each integer lattice-point contained within the span for any primitive. This data revirtualization propagates the spans to those fragment processors which are now responsible for that fragment of that span. This number of spans will be smaller or equal to the number of fragments, f, required for the scene.

7.2.2 Fragment Generation

Each fragment processor determines the z-value for each fragment by solving the triangle's plane equation at the integer lattice-point.

7.2.3 Finalization

The renderer now discards clipped fragments and performs the z-buffer image composition.

7.3 Analysis

The time required by this algorithm is $\Theta((f_t/p)\log p)$ where f_t is the number of fragments generated. This value of f_t should be the same as the number of fragments f required for the serial z-buffer. This technique is the one employed by the *Render renderer produced by the Thinking Machines Corporation.[17]

As was the case with the footprint renderer in section 6.0, one deficiency of this algorithm is that it does not perform forward differencing to generate span-triangle horizontal inter-

17. James B. Salem, **Render: A Data Parallel Approach To Polygon Rendering*, technical report, Thinking Machines Corporation, Cambridge, Massachusetts, 1988.

cepts and fragment z-values. This means that the fundamental computations at the heart of this algorithm involve multiplies and divides, whereas the techniques of forward differencing could reduce these to additions. This is a significant source of optimization which is addressed in section 8.0.

8.0 Triangle Strip Renderer

The triangle-strip renderer employs two data virtualizations, the triangle strip virtualization (which matches the physical machine size) and the fragment virtualization.

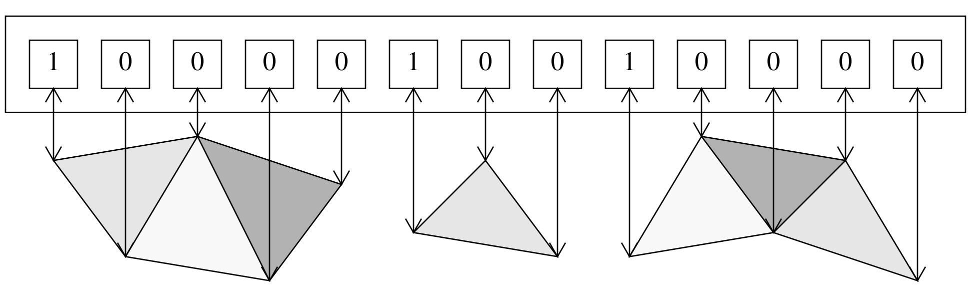
The triangle strip physical virtualization does *not* follow the virtual processor paradigm. Instead, the individual physical processors each hold arrays of triangle data to be processed. Processors advance independently to their next triangle when their current triangle is completed. Processors are allowed to accumulate temporary results for use in computing subsequent data elements. This allows us to use forward differencing, which is a great boon.

The fragment virtualization is used only to accumulate fragments for subsequent image composition. The fragments are accumulated in large batches before composition in order to reduce the impact of the fragment image composition algorithm's high overhead. Recall that the image-composition algorithm relies on high virtualization ratios in order to achieve good asymptotic and pragmatic performance. See section 5.5 and section 4.9. Virtualization ratios are defined in section 4.1.

8.1 Triangle Strip Array Format

Each processor holds some number of triangle strips packed tightly into an array. Each array element holds one three-dimensional point, a color, and a boolean start-marker. The start-marker is used to indicate the beginning of a new triangle strip. Otherwise, the new data point is used to construct a new triangle by discarding the data point three elements back in the array from the current element. This is a standard format for triangle strips, taken from serial implementations, and it has the further property of being wonderfully compact. Throughout this discussion, we refer to data elements in this triangle strip array as triangles, although a triangle is actually defined by an element in the array with the previous two elements in the array. See figure 12 for an illustration of this array format.

FIGURE 12 Triangle Strip Array Format



The renderer first loads data points, colors, and start-markers into the triangle strip virtualization, applies the viewing transformation to each triangle, shades the triangles, and clips triangles.

8.2 Forward Differencing

Since the serial z-buffer algorithm can generate the fragments in any convenient order, forward differencing is often used in its implementation. This optimization speeds up the fragment generation greatly. Though forward differencing makes no impact on the asymptotic running time, it makes an incredible difference in practice.

The triangle strip renderer operates in almost exactly the same manner that a serial z-buffer operates. There are some necessary and unusual techniques applied (in section 8.5 and section 8.6) to ensure inter-processor load balance, but otherwise, the algorithm is straightforward.

8.3 The Data Types

There are a few similar data types on which this algorithm operates:

- Batches

  A Batch is a triangle strip array. There is always one batch per physical processor.

- Triangle strips

  There is always one current triangle strip per physical processor.

- Triangles

  There is always one current triangle per physical processor.

- Spans

  There is always one current span per physical processor.

- Fragments

  There is always one current fragment per physical processor. Keep in mind that there is also a processor virtualization of fragments for the sake of later image composition. Nevertheless, there is only one *current* fragment per processor.

There are some fundamental access routines which operate similarly for each of these data types. They are listed below, for the example, for the span data-type:

- `boolean is_triangle_expired();`

  This routine returns a boolean value which is true if the current triangle has no more spans, i.e. the current span is invalid. Otherwise, it returns false.

- `void iterate_to_next_span();`

  This routine updates the current span to contain the next span in the current triangle. Note that this new current span must still be checked for validity with the is_triangle_-expired() routine.

8.4 The Algorithm

The simple version of the algorithm works as follows. It continually loops while any processor has triangle strips left to render. During each iteration of the loop, the algorithm iterates any data types which may have finished. The pseudocode follows:

• `void slow_render_batch();`

```
while (some batch not finished) {
          where (not is_frag_expired())
                    store_frag();
          reload_fragment();
          reload_span();
          reload_triangle();
          reload_strip();
}
```

The procedure `store_frag()` puts the current fragment into the accumulated fragments processor virtualization for later image composition into the z-buffer, as described in section 5.5.

One of the low-level data type reload routines is shown below. The others are similar enough that they have been omitted.

• `void reload_span();`

```
if ((not is_triangle_expired()) and
     is_span_expired())
          iterate_to_next_span();
```

Note that the above algorithm requires local memory indirection. Local memory indirection was described in section 4.7. Indirection is required since processors will surely finish triangles at different times for certain common input scenes. Thus, various processors will need to load different triangles out of their triangle strip arrays at the same time. Local memory indirection is the simplest way to satisfy this requirement. We will see in section 10.6 that this is a very expensive operation on the Connection Machine. We will address this issue in section 11.2.

8.5 Global Static Load Balancing

The true challenge that this algorithm presents is to ensure that each processor has a roughly equivalent amount of work to do. On the broadest level, each processor should have approximately the same number of fragments at the end of the rendering process. If this condition is not met, then certain processors may have disproportionately large numbers of fragments to render.

One solution to this problem is to shuffle the triangles before any rendering is performed. This does not guarantee good load-balance, but for large numbers of triangles we can expect this to be true by the Law Of Large Numbers.[18]

The simplest way to perform the shuffling is to perform the following steps:

18. Sherryl Tomboulian and Matthew Pappas, "Indirect Addressing And Load Balancing For Faster Solution To Mandelbrot Set On SIMD Architectures," *The Third Symposium On The Frontiers Of Massively Parallel Computation*, I.E.E.E., 1990.

1. On each processor, dice the triangle strips into small triangle-strip buckets. Care must be taken to copy some triangle data points at the beginning of each triangle-strip bucket to insure that triangles on triangle-strip bucket boundaries are not skipped. This step involves no communication.
2. Chose one random number and calculate a one-dimensional grid distance that this represents.
3. Each processor loads one triangle-strip bucket, and then sends this along the one-dimensional grid for the distance proscribed by the random distance selected in step 2. This performs a cyclic shift of the triangle-strip bucket across the processor grid.

An interesting parameter for this algorithm is the triangle-strip bucket size to be used. If the buckets are too small (for example, one triangle each), then we will lose all triangle-strip coherency. If the buckets are too large, then we cannot safely invoke the Law Of Large Numbers. Effectively, randomization will not balance the load unless we shuffle a large number of memory processor layers across the processors. We discuss our choices for the implementation in section 10.3.

8.6 Local Dynamic Load Balancing

Another load-balancing problem to address is that the various reload routines have very different costs. There are good reasons that `reload_span()` requires more time than `reload_fragment()`. Since we are employing forward differencing, `reload_frag()` only has to perform one integer addition for the x coordinate and one for the z coordinate. One the other hand, `reload_span()` has to perform at least four additions, requiring roughly double the time. Thus, we can intuitively expect that `reload_frag()` will be twice as fast as `reload_span()`. While the exact ratio is not precisely two, we will see in section 10.3 that our assumptions are approximately correct.

8.7 Reload Factors

One way to handle the cost-imbalance amongst the reload routines is to determine reload_factors, I, and then only reload a given data-type when a timer has expired.[19] For example, perhaps it is advantageous to only reload a span after reloading two fragments. The reasoning that suggests this approach is the intuition that most spans will contain several fragments. So rendering several fragments per span before reloading the spans will probably save wasted effort reloading spans which have not yet expired.

Consider the following code:

- `void medium_render_batch();`

```
given      If, Is, It, Ir;

while (some batch not finished) {
```

19. Sherryl Tomboulian and Matthew Pappas, “Indirect Addressing And Load Balancing For Faster Solution To Mandelbrot Set On SIMD Architectures,” *The Third Symposium On The Frontiers Of Massively Parallel Computation*, I.E.E.E., 1990.

```
                where (not is_frag_expired())
                        store_frag();
                if (not (i % I_f))
                        reload_fragment();
                if (not (i % I_s))
                        reload_span();
                if (not (i % I_t))
                        reload_triangle();
                if (not (i % I_r))
                        reload_strip();
        }
```

In order to calculate the I reload factors, the program must be timed on a variety of sample scenes, and the factors varied until the fastest run-time is obtained. These factors will probably by scene-dependent, so the factors must be recalibrated for good speed. It is quite unclear how to analytically determine good reload factors.

8.8 On-line Profit Calculation

The better version of the algorithm works as follows. During each iteration of the loop, the algorithm collects some statistics and then decides which of several operations to perform. The algorithm collects statistics for each class of items: fragments, spans, triangles, and strips. For each item type, it calculates a number R which denotes the number of such, for example, fragments which are ready to be computed, i.e. whose spans have not expired. The algorithm requires some compiled constants L which denote the number of seconds required to reload a set of items of a given type. Since we are using a SIMD machine, each and every processor participates in every reload. So the algorithm then calculates *profit factors*, P, which are computed by $P = R/L$. The factor P_f, for example, is defined in units of $(\text{computed fragments})/\text{second}$. So the algorithm then proceeds to determine the item class with the highest profit factor, and then reloads one set of such items. For example, if the algorithm can locally sustain a high rate of $(\text{computed fragments})/\text{second}$, then it will reload one fragment per processor in the next step. If, on the other hand, many spans have expired, and the number of computable fragments is presently low, then the algorithm may choose instead to reload a set of spans. The pseudocode follows:

- `void fast_render_batch();`

```
        given      L_f,  L_s,  L_t,  L_r;

        while (some batch not finished) {
                bool:current   E_s =
                     is_span_expired(&span);
                bool:current   E_t =
                     is_triangle_expired(&tri);
                bool:current   E_r =
                     is_strip_expired(batch, &strip);
                bool:current   E_b =
                     is_batch_expired(batch);
```

```
int:current     C_f =
     ((! E_b) & (! E_r) & (! E_t) & (! E_s));
int:current     C_s =
     ((! E_b) & (! E_r) & (! E_t) & (E_s));
int:current     C_t =
     ((! E_b) & (! E_r) & (E_t));
int:current     C_r =       ((! E_b) & (E_r));
int             R_f =       (+= (C_f));
int             R_s =       (+= (C_s));
int             R_t =       (+= (C_t));
int             R_r =       (+= (C_r));
float           P_f =       R_f / L_f;
float           P_s =       R_s / L_s;
float           P_t =       R_t / L_t;
float           P_r =       R_r / L_r;

if (P_f>P_s and P_f>P_t and P_f>P_r) {
          where (not is_frag_expired())
                   store_frag();
          reload_fragment();
} else if (P_s>P_f and P_s>P_t and P_s>P_r)
          reload_span();
else if (P_t>P_f and P_t>P_s and P_t>P_r)
          reload_triangle();
else
          reload_strip();
```

```
}
```

We will see in section 10.6 that the cost of computing these profit factors on-line is negligible.

9.0 Spline Patches

The spline-patch renderer employs three data virtualizations, the patch virtualization, the tesselation virtualization, and the triangle virtualization. This renderer ultimately generates triangles, which are then passed to one of the various triangle or triangle-strip renderers described elsewhere in this paper. The renderer first loads (the control hulls of) the spline patches into the patch virtualization and then applies the viewing transformation to the control hulls.

9.1 Rational Bicubic Coefficient Calculation

For each patch, each processor now transforms its control hull into the polynomial coefficient array. This is performed purely locally and requires no communication. This involves local matrix multiplications.

9.2 Tesselation Grid Expansion And Evaluation

Each patch processor now determines how many tesselation grid points should be evaluated to triangulate the patch. Again, this is the problem of adaptive subdivision (or actually a simplified version, adaptive subdivision techniques need not be so uniform), described previously in section 3.2.3.

Each processor has now computed an expansion factor for a data revirtualization, as described in section 4.6. The new tesselation data virtualization posses one virtual processor for each tesselation grid-point for any patch. The data remapping copies the patch polynomial coefficient arrays to those tesselation processors which are now responsible for evaluating that tesselation point of that patch. This can be a large number of virtual processors if the tesselation expansion factors were large. This large size is compounded by the fact that the patch data-structures hold large arrays of floating-point values which consume large amounts of memory. This was problematic in the implementation of this algorithm and is address in section 10.4.

Now each tesselation grid-point processor evaluates its spline patch at the proper values of u and v which the given domain grid-point represents.

9.3 Triangulation Expansion And Stitching

The data is now remapped so that each tesselation processor is allocated two triangle processors. The evaluated tesselation points are then sent to the new triangle processors for all triangles which that grid-point abuts. Each triangle processor of course receives three tesselation points at the end of this process, from which it can manufacture a triangle.

9.4 Analysis

The time required by this algorithm is $\Theta((t_s/p)\log p)$, where t_s is the total number of tesselation grid-points evaluated. Again, this algorithm could be improved by applying forward differencing to evaluate the spline tesselation points. Forward differencing for spline patches is conceptually no more difficult than forward differencing for triangles. In the reality of an implementation however, forward differencing for spline patches will probably involve considerable programmer effort.

10.0 Implementations

10.1 Environment And Tools

We implemented these algorithms on the Connection Machine Two system. The machine used was a 4096 processor system with 8k bytes of RAM per processor.[20] The front-end was a SUN SparcStation Two workstation running UNIX. All code was written in C and C*, the TMC parallel variant of the C language with parallel language extensions.[21] However, we were less than satisfied with C*, its documentation, and its performance. The poor development environment is shared with most massively parallel machines in production today. This technology is still quite new, and the compilers, debuggers, programming languages, and tools are still primitive.

10.2 Tri

We implemented the footprint triangle renderer described in section 6.0, and we refer to this program as *tri*. The image in figure 13 illustrates a frame produced by the triangle renderer.

FIGURE 13 The Triangle Renderer

With the wisdom that only hindsight can impart, we now feel that the two-pass triangle-span renderer of section 7.0 would be faster than the footprint renderer due to the large constant factors in the runtime of the footprint renderer (elucidated in section 6.4).

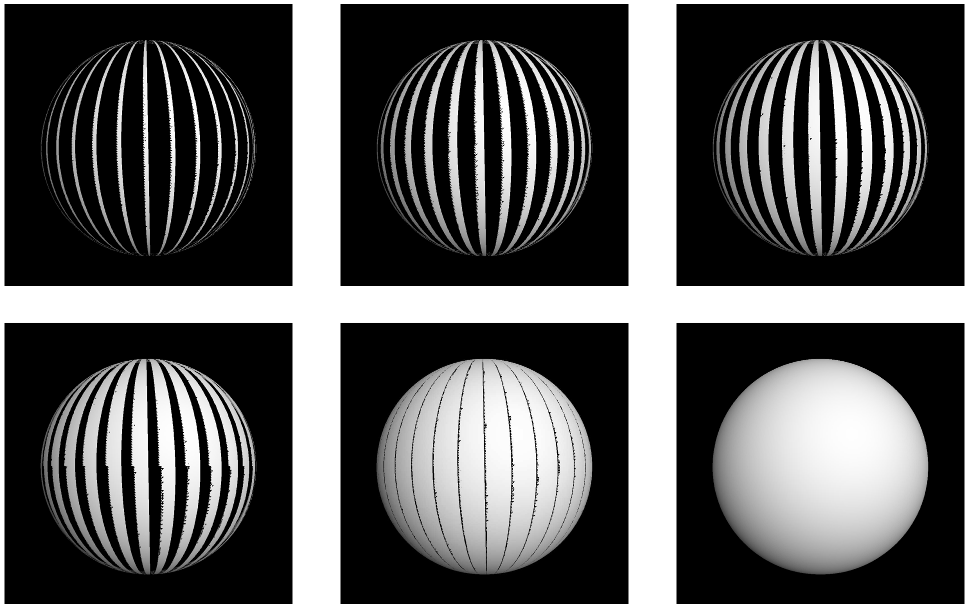
10.3 Tstrip

We implemented the triangle strip renderer of section 8.0. We chose the triangle-strip bucket sizes (for the global static load balancing described in section 8.5) to be very small to insure good load balance. The images in figure 14 illustrate the operation of the triangle strip algorithm. The spheres pictured are in various stages of completion, the final frame being the final output image.

20. *The Connection Machine: CM-200 Series Technical Summary*, Thinking Machines Corporation, Cambridge, Massachusetts, 1991.

21. *C* Programming Guide*, Thinking Machines Corporation, Cambridge, Massachusetts, 1991.

FIGURE 14 The Triangle Strip Renderer In Action



10.4 Spline

We implemented the spline patch triangulator of section 9.0. Due to severe memory restrictions on the Connection Machine, we were unable to actually send the patch polynomial coefficients to each of their tesselation processors. Instead, we chose tesselation expansion factors which are the same on each processor. Then, each patch processor can actually evaluate the tesselation points locally, merely sending the tesselation point results to the tesselation processors. Thus, we did not actually need to send the large patch structures to any tesselation processor. Unfortunately, this prevents us from performing adaptive subdivision since each patch processor must have the same number of tesselation points.

10.5 Performance

Triangle and triangle strip renderer performance results follow.

TABLE 1. Tri and Tstrip Rendering Performance Results for 1024 by 1024 images

program	frames	total triangles	seconds (without any I/O)	triangles/ second
tri	2	77644	237.74	182.33
tstrip	1	131072	119.00	1101.4

Spline triangulation performance results follow.

TABLE 2. Spline Triangulation Performance Results For 18-Triangle-Per-Patch Triangulation

program	patches	triangles generated	seconds (without any I/O or triangle rendering)	triangulated patches/ second	generated triangles/ second
spline	4096	73728	13.36	306.6	5519

10.6 Profiling

The detailed profiling results for all three programs are shown below in three tables. The rows in the tables indicate expensive routines. The times shown are hierarchical, that is they include the times of all routines below them in the call tree.

In some cases, it is impossible for the profiler to determine the call tree, especially when function pointers are called indirectly. For this reason, each table has an “unknown spontaneous” row at the end which indicates the accumulated information for all such routines whose callers are unknown. For some important cases (like prefix scans and hypercube sending), we factored the indirectly called functions run-times back into their callers. So the times for hypercube communication in our routines should be accurate. Nonetheless, these spontaneously-called routines are up to fifty percent of the time for some of the programs. This is disappointing, but it is a deficiency of the UNIX profiler.

Each table also has a row entry for “other” routines. This row includes the timing information for routines not covered by the other entries. We actually have profiling information for these routines, but we have omitted it, since it lends little new information.

10.6.1 Tri Profiling

Triangle renderer profiling results follow.

TABLE 3. Triangle Rendering Profiling Results

routine	seconds	times called	milliseconds per call	percentage of total time
composite fragments	22.66	28	810	9.531%
spread triangles	19.61	28	700	8.249%
generate fragments	13.91	28	500	5.851%

TABLE 3. Triangle Rendering Profiling Results

routine	seconds	times called	milliseconds per call	percentage of total time
clip triangles	1.19	28	43	5.01%
distribute triangles	10.94	48	230	4.602%
area-bound fragments	9.71	48	200	4.084%
shade triangles	4.12	20	210	1.73%
view triangles	3.52	20	180	1.48%
other	22.87			9.620%
unknown spontaneous	129.21			54.349%

The largest factor in the triangle renderer was the image composition.

10.6.2 Tstrip Profiling

Triangle strip renderer profiling results follow.

TABLE 4. Triangle Strip Rendering Profiling Results

routine	seconds	times called	milliseconds per call	percentage of total time
render triangle	48.15	108	446	40.46%
render fragment	18.83	810	23.2	15.82%
render span	6.27	302	20.8	5.27%
view triangle strips	4.03	2	2000	3.39%
static load balance	3.74	2	2000	3.14%
shade	3.64	2	2000	3.06%

TABLE 4. Triangle Strip Rendering Profiling Results

routine	seconds	times called	milliseconds per call	percentage of total time
is_batch_expired	2.10	4489	0.468	1.76%
send fragments	1.73	13	130	1.45%
other	15.89			13.35%
unknown spontaneous	14.62			12.29%

The largest factor in the triangle strip renderer was the phenomenal time incurred by local memory indirection on the Connection Machine Two. We discuss this further in section 11.2.

10.6.3 Spline Profiling

Spline triangulation profiling results follow.

TABLE 5. Spline Triangulation Profiling Results

routine	seconds	times called	milliseconds per call	percentage of total time
eval points	1.27	32	40	9.51%
distrubute points	1.25	64	20	9.36%
tessellate	0.70	2	400	5.2%
generate cubics	0.55	2	300	4.1%
other	0.49			3.7%
unknown spontaneous	9.1			68%

The largest factor in the spline triangulator was the evaluation of the tesselation points, which is appropriate. We were somewhat surprised by how fast this algorithm performed.

11.0 Conclusions

These rendering algorithms and the implementations were quite exciting to develop and implement. While the results are not entirely positive, the area is ripe with new areas for further research.

11.1 Communications Networks

We have used the hypercube network extensively for z-buffer image composition in all of these algorithms. We also used the hypercube for data remapping in all algorithms except for the triangle strip renderer. Thus, we are rather dependent on this feature of the Connection Machine. Some of these algorithms are easily extensible to non-hypercube architectures like two-dimensional arrays. These are much cheaper to manufacture, so this avenue is worth pursuing in the future. Image composition (many-to-one packet routing) is the fundamental algorithm for which non-hypercube architectures perform rather poorly. This is a fertile area for future research. The work of the Pixel Flow team at UNC suggests that pipelined one-dimensional grid architectures may be sufficient, although they require large amounts of memory for buffering.[22] This is intriguing since some attractive new massively parallel architectures do not employ the expensive hypercube network.

11.2 Local Memory Indirection

The largest performance bottlekneck for the triangle strip renderer was the prohibitive cost of local memory indirection. Local indirection is two orders of magnitude slower than uniform memory access on the Connection Machine. Lower costs for local indirection would alleviate this tremendous problem. Note that many algorithms which perform forward differencing, especially those with non-uniform convergence rates, would benefit quite greatly from fast local indirection hardware. Both the MasPar MP-1 and TMC CM-5 architectures have better local memory indirection support, and these machines might be able to thus achieve speedups close to one hundred times over the current triangle strip renderer on the TMC CM-2.

11.3 Special Purpose Hardware

Many authors before us have speculated that perhaps special-purpose graphics hardware will always be significantly faster than general-purpose parallel computers. This is probably the case, and special-purpose hardware will always be at least somewhat faster than general hardware. But dedicated graphics hardware suffers from a number of disadvantages. It is not generally useful for solving different tasks, and, even in the domain of rendering, there are differing techniques and precisions which all have their place in the rendering community. No one piece of graphics hardware will ever be able to generate every effect that is needed in the computer graphics community. In order to work around the deficiencies in their hardware, programmers now and in the future will always work around hardware shortcomings using general-purpose computing software. By implementing the entire rendering process in a general-purpose computer, changes to the ren-

22. Steven Edward Molnar, *Image Composition Architectures*, doctoral dissertation, University Of North Carolina At Chapel Hill, 1991.

dering process should be easier to implement, test, and incorporate into a stable software rendering platform. While it is certainly possible to solve certain useful problems in hardware, there are many rendering tasks, particularly innovative new techniques, that require fundamental changes to the rendering process. Changes to the core of the rendering process are fundamentally impossible in dedicated hardware. They are invitingly simple in general-purpose rendering software.

12.0 Acknowledgments

Innumerable thanks are due to Philip Hubbard and John “Spike” Hughes for their consultation and feedback. Spike’s spur-of-the-moment intuition remains, as always, often more accurate than some of our best-developed ideas. Philip’s diligent direction of our research allowed this project to flower.

Rick Sabourin allowed our work to progress by installing CM software and finding documentation with only a moment’s notice. Thanks to Franco Preparata for occasional discussions about the fine points of fast prefix scans and various architectural restrictions on the CM. Thanks also to Anand Bodapati for discussions about low-level CM languages like CMIS.

Thanks also to Cassidy Curtis, who provided us with the Mummy triangle-strip datasets used as input to the triangle strip renderer. And extremely importantly, thanks to Jeff Achter and Joe Edmonds, fellow students who read early drafts and helped us to stop ourselves from using too much unexplained jargon.

13.0 Bibliography

C Programming Guide*, Thinking Machines Corporation, Cambridge, Massachusetts, 1991.

The Connection Machine: CM-200 Series Technical Summary, Thinking Machines Corporation, Cambridge, Massachusetts, 1991.

Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest, *Introduction To Algorithms*, M.I.T. Press, Cambridge, Massachusetts, 1991.

James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes, *Computer Graphics: Principles And Practice*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1990.

Henry Fuchs *et al*, "Pixel Planes 5: A Heterogeneous Multiprocessor Graphics System Using Processor-Enhanced Memories," *Proceedings of SIGGRAPH '89, Computer Graphics*, vol. 23, no. 3, 1989, ACM SIGGRAPH, New York, pp. 79-88.

Graphics Gems, editor Andrew S. Glassner, Harcourt Brace Jovanovich, Boston, 1990.

Graphics Gems II, editor James Arvo, Harcourt Brace Jovanovich, Boston, 1991.

F. Thomson Leighton, *Introduction To Parallel Algorithms And Architectures: Arrays, Trees, Hypercubes*, Morgan Kaufmann Publishers, San Mateo, California, 1992.

Steven Edward Molnar, *Image Composition Architectures*, doctoral dissertation, University Of North Carolina At Chapel Hill, 1991.

James B. Salem, **Render: A Data Parallel Approach To Polygon Rendering*, technical report, Thinking Machines Corporation, Cambridge, Massachusetts, 1988.

Sherryl Tomboulian and Matthew Pappas, "Indirect Addressing And Load Balancing For Faster Solution To Mandelbrot Set On SIMD Architectures," *The Third Symposium On The Frontiers Of Massively Parallel Computation*, I.E.E.E., 1990.