Parallel Tessellation of Bézier Surface Patches Using a SIMD Processor Array

TR95-043
December 1995

Paul Keller

Department of Computer Science
CB #3175, Sitterson Hall
UNC-Chapel Hill
Chapel Hill, NC 27599-3175

Parallel Tessellation of Bézier Surface Patches Using a SIMD Processor Array

ABSTRACT

Bézier surface patches are widely used in applications ranging from virtual reality to CAD systems. These applications typically display Bézier patches by tessellating them into triangles, which are then rapidly rendered by conventional hardware graphics accelerators. However, the re-tessellation process is computationally expensive and with large databases can become a bottleneck in the typical approach of tessellating using the graphics computer's host processor. In the case of PixelFlow, the rasterization engine consists of an 8K element SIMD processor array. These processors are capable of general-purpose computation with an aggregate peak performance of 3.2 billion floating point multiplies per second and 2.1 billion floating point additions per second. This thesis explores how to exploit the processing power of the SIMD array to rapidly tessellate and render Bézier surface patches. Simulation results from the first implementation show that, with some restrictions, the rasterization hardware can tessellate surface patches with throughput exceeding 2 GigaFlops.

Table of Contents

1.0 Introduction

Bézier surface patches are widely used in applications ranging from virtual reality to CAD systems. Applications that demand high performance typically display Bézier patches by tessellating them into triangles, which are then rapidly rendered by conventional hardware graphics accelerators. The smoothest images result from dynamically tessellating at run time, since the optimal sampling interval depends on the viewing parameters. However, the re-tessellation process is computationally expensive and with large databases can become a bottleneck in the typical approach of tessellating using the graphics computer's host processor. Each board of PixelFlow's rendering engine contains over 8000 general purpose 8-bit rasterization processors. The purpose of this thesis is to explore ways to exploit the processing power of the rendering engine to rapidly tessellate Bézier surface patches and, in so doing, off-load the host and geometry processors.

Since tessellating Bézier patches is computationally expensive, many techniques have been devised to reduce the cost of this operation. For example, it is possible to pre-tessellate a patch so as to prevent re-tessellation at every frame; however, if the sampling is too coarse, the surface will not appear smooth, whereas if the sampling is too fine, rendering performance becomes a bottleneck. Pre-tessellation at multiple resolutions is not feasible since large databases would require hundreds of megabytes for storage. Adaptive tessellation [Kumar 94] re-samples surface patches based on viewing parameters to reduce storage and rendering requirements while maintaining a smooth image. This method works well, but increases the load on the CPU that must perform these computations.

A conventional serial processor quickly becomes overloaded by the tessellation computation. PixelPlanes 5 [Fuchs 89] uses Intel i860 processors for floating point computation. These processors deliver 80 MFlops peak performance. PixelFlow will use 2 HP PA-RISC 7200 processors as part of the geometry engine. Each of these delivers approximately 240 MFlops peak performance, provided the pipeline remains filled with multiply-accumulate

operations. If the peak performance is obtained, this translates to nearly 860K samples/second per processor, or 54K patches/second per processor.

In addition to the geometry processors, each PixelFlow board contains an 8K processor array. In the rendering process, each processor performs the color and intensity computation specific to a pixel and is therefore referred to as a *pixel processor.* Each pixel processor contains 384 bytes of local memory. In addition to the local memory, the processors have access to *texture memory,* so called since it is optimized for storing and retrieving image textures. Clearly, the ability to tessellate patches on this hardware is not by design. The important realization, though, is that the pixel processors can be viewed as an 8K element SIMD[1] (single instruction multiple data) processor array capable of general-purpose calculation. Aggregate peak performance of the 8K processors is 3.2 billion floating point multiplies per second and 2.1 billion floating point additions per second (both in 32-bit single-precision). These staggering numbers pose the question: can the ability to perform general-purpose computation using the rasterization hardware be harnessed and applied to the problem of surface patch tessellation? This, in turn, leads to the thesis which will be supported in the remainder of this document:

> "The rasterization hardware found in the final pipeline stages of the PixelFlow graphics computer provides high performance and is sufficiently programmable to accelerate the process of surface patch tessellation relative to the traditional method of using the host or geometry processors."

Several issues must be confronted to substantiate this assertion. The first is to develop a paradigm that divides the tessellation problem over the 8K processors such that it is suitable for a parallel solution. Since the high performance is obtained through massive parallelism, any reasonable paradigm must try to maintain high processor utilization or else suffer from degraded performance. Once the problem has been mapped to the processors, the next issue that must be confronted is to devise a method of loading data into and transferring data out of the processors rapidly enough so as not to erode the potential performance gain. The solutions to these issues determine the form of the algorithm and are

1. In a SIMD computer, each processing element executes the same instructions but with different data.

developed in Chapter 3. Chapter 4 augments these ideas with implementation details and cycle counts. Chapter 2 highlights the details of the machine which are necessary to understand the parallel tessellation algorithm developed later.

The recurring theme throughout the following pages is the trade-off between simplifying the tessellation process versus sacrificing performance and usability. Generally, by imposing more restrictions on the problem, the solution becomes more feasible, yet, simultaneously, the solution is less usable by many applications. General solutions pay a price in decreased overall performance. For example, limiting the application to generating two surface patches with 64×64 samples simplifies the problem and yields great theoretical performance; however, there probably are no applications that would benefit by using this capability.

Rather than trying to produce a solution that would work within the constraints of all applications (an impossible task), the emphasis within this document is to produce the core of a workable solution without sacrificing performance. Success at this goal implies that there is indeed a workable solution; particular applications can then customize the solution, paying as much of a performance penalty as is necessary and can be afforded.

It should be noted that at the present time PixelFlow does not exist as a physical machine, but only as schematics and simulators. All of the software that was generated for this project has been tested on a low (register transfer) level simulator. However, there are some portions of the machine which presently are not simulated. The parts of the algorithm which are affected have been verified through theoretical calculations only.

The simulation results are encouraging. It takes about 97,000 cycles (plus communication time which can be overlapped with other computation) to generate 4×4 samples for 512 bicubic patches. This translates to 530K patches/second or 8.5M samples/second, giving a total of 2.4 GFlops overall performance. The performance figures are achieved by restricting the application to producing 4×4, 8×8 or 16×16 samples per patch and only tessellating patches of the same degree in the same pass. The former restriction may increase the number of triangles generated. For example, tessellating a patch with 8×8 samples instead of 6×6 samples produces 98 triangles instead of 50 triangles. The latter

restriction may reduce processor utilization for small databases but is not a problem if the data set is sufficiently large since for that case it is likely that there are enough surface patches to yield full utilization with multiple passes.

The remaining portions of this document explore how this performance is achieved and why these limitations exist. The results are summarized in Chapter 5.

2.0 Background

To understand how to utilize PixelFlow's hardware to tessellate Bézier Surface patches, the reader must first become familiar both with the steps involved in tessellating a surface patch and with the intricacies of PixelFlow. The following two sections provide an overview of each of these topics. There are several textbooks which discuss surface patch tessellation in detail so the treatment in this document is for completeness only. On the other hand, there are no complete references for PixelFlow, therefore Section 2.2 covers all of the details which are needed to understand the discussion in this document.

2.1 Tessellation Overview

The one dimensional equivalent of a Bézier patch in 2-space is a Bézier curve, shown in Figure 1. The curve is represented by a set of control points, $\boldsymbol{b}_i$, which do not lie on the curve. Tessellation in the 2D case involves sampling the curve at a series of points, $P(t)$, given by:

$$P(t) = \sum_{i=0}^{n} \boldsymbol{b}_i \binom{n}{i} t^i (1-t)^{n-i}$$

As the sampling interval approaches zero the number of points approaches infinity; the sampled curves becomes precisely the Bézier curve. For more practical sampling intervals the curve is represented by interpolating between the samples, shown as a dotted line in the figure.

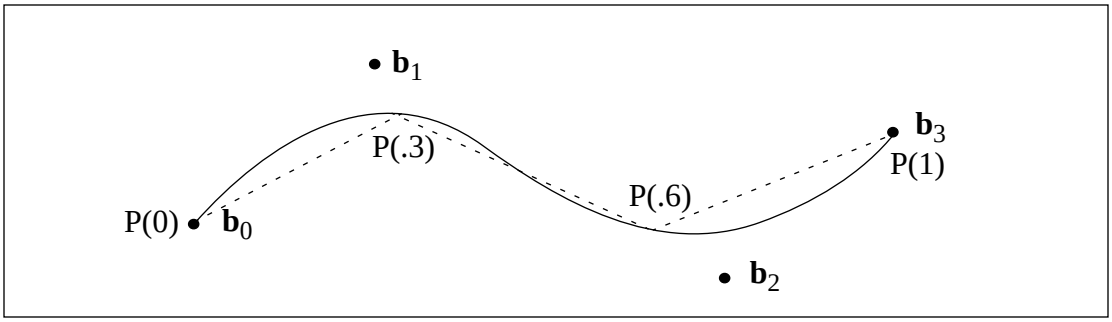


Figure 1 Bézier Curve

The 3D case follows by analogy from the 2D case: a Bézier patch is represented by a 2D mesh of control points which (generally) does not lie on the patch's surface. The goal of the tessellation is to convert this mesh of control points into a patch of points on the surface, as depicted in Figure 2. The dimensions of the control patch, in this document referred to as $m \times n$, determine the degree of the patch. The $p \times q$ points on the generated surface patch need not (and generally do not) correspond to the control patch dimensions. In the limit where $p, q \to \infty$, the generated surface patch, as the curve in the 2D case, becomes precisely the curved surface represented by the control patch. With practical values of p and q, the area between the surface points is interpolated by piecewise planar surfaces. These surfaces are usually triangles since triangles can be rendered rapidly.

There has been and still is active research in the area of determining how best to tessellate a surface. The key questions are: what are the best values for p, q and where on the surface should the $p \times q$ points be generated? The problem of determining where to generate the points is often harder than actually generating the points [Kumar 94], so typically the patch is sampled in a uniform grid. The grid size must still be determined; too fine a grid generates too many triangles which hinders rendering performance, whereas too coarse a grid results in jagged curved surfaces due to the planar interpolation between the samples. The best dimensions depend on the viewing parameters since a patch that fills the screen must be sampled more finely than one that spans just a few pixels [Kumar 94].

The surface points on a Bézier patch can be obtained in several ways: by repeated linear interpolation between the control points (the de Casteljau Algorithm [Farin 93]) or by

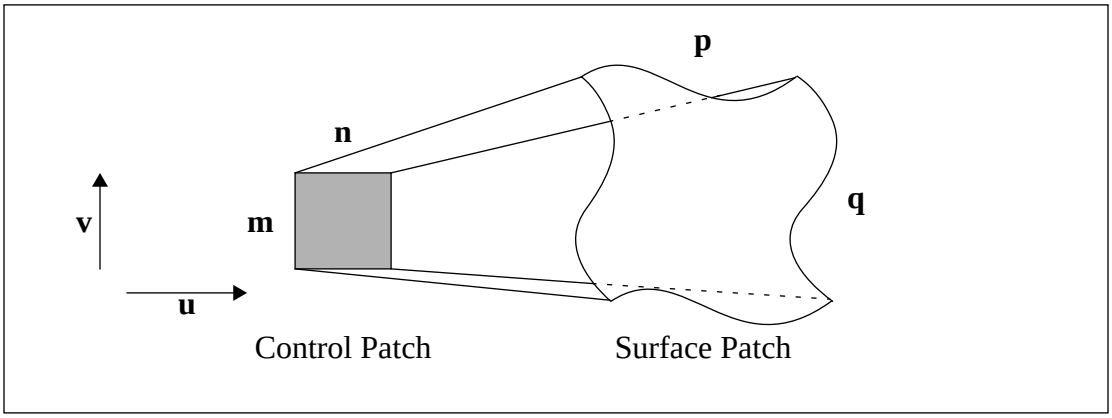


Figure 2 Surface Patch Tessellation

algebraic calculation using Bernstein polynomials [Farin 93]. The two methods achieve the same result; the latter is suited to a parallel implementation and so is discussed below.

2.1.1 Mathematical Background

Given an $m \times n$ control patch, the equation for generating the $p \times q$ points sampled at discrete values of u, v on the Bézier surface is:

$$P(u, v) = \sum_{i=0}^{(n-1)} \sum_{j=0}^{(m-1)} \boldsymbol{b}_{i,j} B_i^n(u) B_j^m(v)$$

where $\boldsymbol{b}_{i,j}$ is one of the $m \times n$ control points and $B_k^r(t)$ are the Bernstein polynomials of the form:

$$B_k^r(t) = \binom{r}{k} t^k (1-t)^{r-k}$$

The points $P(u, v)$ can be generated using the matrix formulation:

$$P(u, v) = \begin{bmatrix} B_0^{m-1}(v) & \dots & B_{m-1}^{m-1}(v) \end{bmatrix} \begin{bmatrix} \boldsymbol{b}_{0,0} & \cdots & \boldsymbol{b}_{0,n-1} \\ \cdots & \cdots & \cdots \\ \boldsymbol{b}_{m-1,0} & \cdots & \boldsymbol{b}_{m-1,n-1} \end{bmatrix} \begin{bmatrix} B_0^{n-1}(u) \\ \cdots \\ B_{n-1}^{n-1}(u) \end{bmatrix}$$

In addition to generating points on the surface of a patch, the normal vectors to the surface at those points must be calculated. This can be done by taking the cross product of the derivatives in the u and v directions. The derivatives are found by calculating the first differences (*e.g.* $\boldsymbol{b}_{1,0} - \boldsymbol{b}_{0,0}$ in u and $\boldsymbol{b}_{0,1} - \boldsymbol{b}_{0,0}$ in v) of the $m \times n$ control patch and then evaluating these as Bézier patches of order $(m-1) \times n$ and $m \times (n-1)$ respectively.

2.1.2 Steps for Tessellation

The math from the preceding section combined with steps listed below generates a set of samples on the surface of a Bézier patch:

1. Evaluate the Bernstein polynomials at u, v for each sample. When sampling with a uniform grid, each column will have a constant value for u and each row will have a constant value for v. So, evaluating the polynomials for each sample reduces to evaluating in u for each column and evaluating in v for each row.
2. Evaluate the Bernstein polynomials at u, v for each sample for the first differences.
3. Perform the matrix multiplications for the control patch in (X, Y, Z) for each sample.
4. Perform matrix multiplications for the first differences in u for (X, Y, Z) for each sample.
5. Perform matrix multiplications for the first differences in v for (X, Y, Z) for each sample.
6. Calculate the normal by taking the cross product of the results of steps 4 and 5 and normalizing the result.

2.2 Hardware Overview

An in-depth description of the PixelFlow hardware is beyond the scope of this document (see [Molnar 92] for an overview and [Eyles 95] for details). The following description is intended to provide sufficient background so that the reader can understand the strengths and weaknesses of PixelFlow's design as applied to the problem of parallel tessellation of surface patches. The first subsection describes the component parts of PixelFlow, and how they are used to render an image. The following sections explore some hardware details that are particularly important for the tessellation process.

2.2.1 PixelFlow Components

PixelFlow consists of a host processor (*e.g.* a UNIX workstation) connected to one or more (up to thirty or forty) special purpose graphics boards, or "nodes". The nodes communicate with the host processor and each other over a high-speed network called the *geometry network*. Each node can be configured to perform one or more of the operations required for rendering an image. For example, a board can scan convert primitives, or shade and texture rendered primitives, or even act as a frame buffer.

There are two principle computational engines on a PixelFlow node: the Geometry Processors (GPs) and an 8K element SIMD processor array. Figure 3 shows how these processors are connected to the remaining components in one PixelFlow node. The key features are described below:

- Two Geometry Processors (GPs): The two GPs on each node are HP PA-RISC microprocessors. These chips perform matrix multiplications, transform vertices, generate rendering instructions for the PEs and are capable of any other general purpose computation, such as image warping and surface patch tessellation.
- Enhanced Memory Chips (EMCs): Each EMC is an integrated circuit containing 8 panels of 32 processing elements (PEs). There are 4 groups of 8 EMCs, called modules. Taken together, these modules of EMCs of panels of PEs form the 8K array of processors. Each processor contains an 8-bit ALU and 384 bytes of memory. The processors can be assigned an X and Y coordinate (though not arbitrarily), as described in detail in Section 2.2.9. In the typical rendering paradigm, the PEs are arranged so that their (X, Y) coordinates form a 128×64 grid. This grid represents a small portion of the screen and each PE represents one pixel. For this reason, the PEs are sometimes referred to as pixel processors. Since the machine is SIMD, all of the processors receive the same set of instructions. It is possible, however, to enable and disable individual or groups of PEs so that some instructions are ignored by some processors.
- Linear Expression Evaluator (LEE): The LEE is physically a part of the EMCs. It evaluates expressions of the form $Ax + By + C$ in parallel for all PEs, where X and Y are each PE's logical screen location. The result can either be loaded into PE memory or

used to enable/disable certain processors. The LEE provides an extremely fast method of loading data into the EMCs, provided that the data can be expressed as a linear function of the PE location.

- Texture memory: A PixelFlow node contains a total of 64 MBytes of texture memory divided into 16 MBytes per module. The 16 MBytes consists of eight 2 MByte synchronous DRAMs. The SDRAMs are addressed in parallel; that is, one read or write accesses each SDRAM simultaneously. For maximum performance, data are stored redundantly in the SDRAMs of all modules. This quadruples the bandwidth possible with a single module at the cost of reducing available storage space by a factor of four.
- Texture ASICs (TASICs): The TASICs provide an interface between the PEs and texture memory. These chips buffer and sequence addresses and data heading from the EMCs to the SDRAMs and data flowing the other direction. The TASICs can also automatically generate sequences of addresses, reducing the data flow from the EMCs. There is a communication path between the TASICs, making it possible to route data from one module's EMCs to another module's SDRAMs, or to the geometry network (discussed below).

  The TASICs execute an instruction stream in parallel with the EMCs. These "texture instructions" are limited to accessing two 32 byte blocks of each PE's memory, called the input local port and output local port, respectively. The instructions read the output port for addresses and data to be written to the SDRAMs and write to the input port with data read from the SDRAMs. This communication path, together with texture memory, provides an alternate access route into and out of PE memory which is not limited to linear expressions and can operate in parallel with computation within the EMCs.
- Image Generation Controllers (IGCs): The IGCs are the instruction sequencers. There are two: one controls the EMCs (called the EIGC) while the other controls the TASICs (called the TIGC). The IGCs execute microcoded instruction streams. The two streams execute in parallel, but can be synchronized through the use of semaphores. Figure 4 contains some sample EIGC and TIGC instructions.

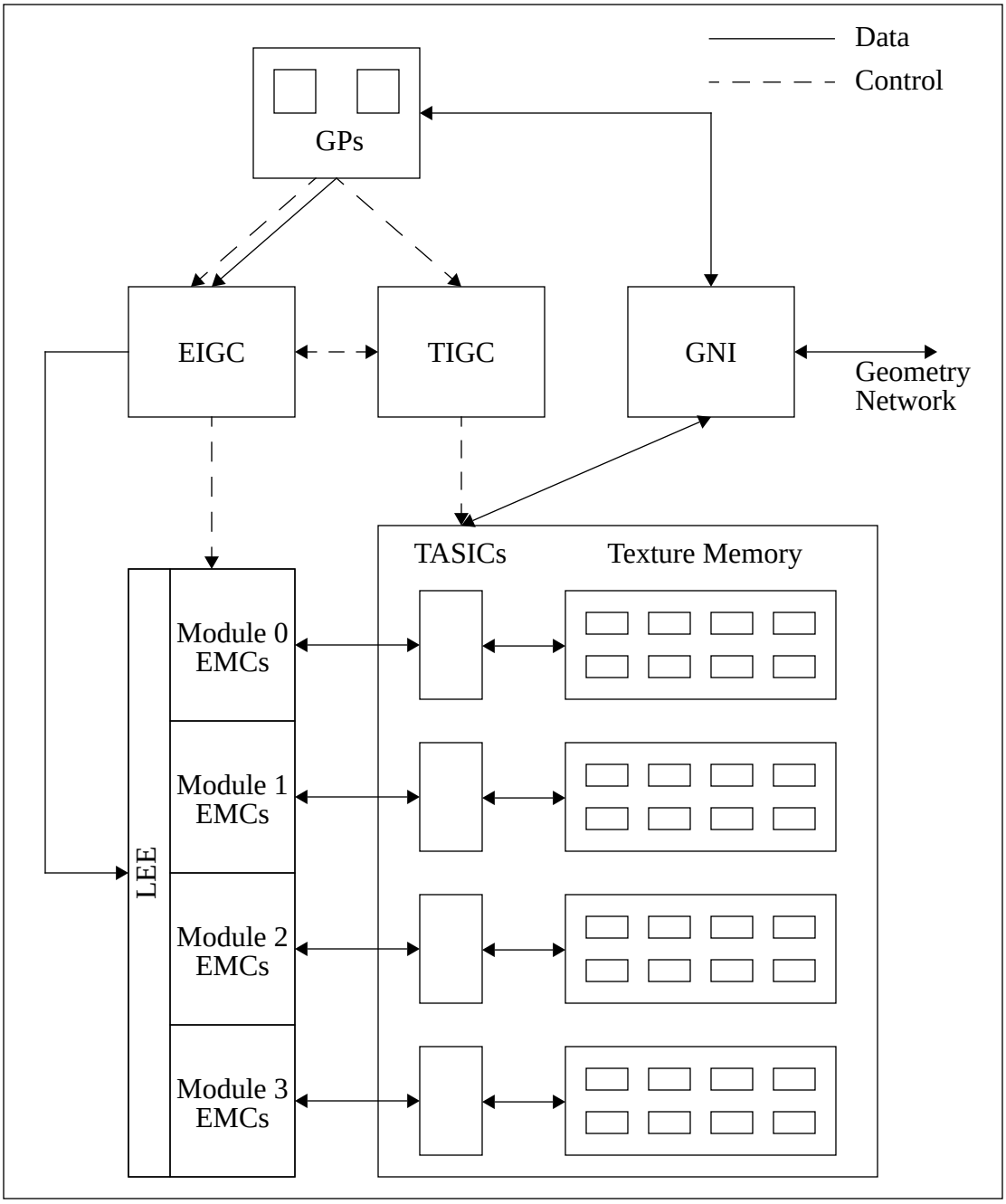


Figure 3 PixelFlow Hardware Overview

- Geometry Network Interface (GNI, pronounced Genie): The GNI is responsible for transferring data between the GPs, texture memory and the EMCs as well as between separate nodes.

Instruction	Description
EMC_SetEnab()	Enables all PEs
EMC_TreeIntoMem_Li(0, 1, A, B, C)	Loads the 1 byte result of *Ax+By+C* into memory location 0 of all enabled PEs
EMC_TreeLTZero_Li(p, 1, A, B, C)	Evaluates *Ax+By+C* and disables processors whose result is greater than or equal to zero
TAS_MemRdBlock4()	Reads 4 byte words from texture memory

EMC_* instructions are sequenced by the EIGC and sent to the EMCs
TAS_* instructions are sequenced by the TIGC and sent to the TASICs

Figure 4 Sample EIGC and TIGC Instructions

2.2.2 Rendering an Image

This subsection provides an example of how PixelFlow's component parts work together to accomplish a task: rendering an image. It is not necessary to understand this process to understand the rest of the document, therefore the remainder of this section may be skimmed or skipped without loss of continuity. It may, however, be beneficial to understand this conventional scenario before embarking upon the details of parallel tessellation. For the sake of understandability, the rendering sequence described below is greatly simplified. In particular, the process described renders flat-shaded triangles without supersampling.

PixelFlow creates an image by dividing the triangles to be rendered over a series of nodes. The GPs transform the vertices of the triangles into screen space and then into instructions for rendering. The instructions are sent to the EIGC which sequences the EMCs, where a 128×64 pixel region of the image is generated. At this point each PE on a node contains the color and Z (depth) value of one pixel in the image. Once each node has rendered its portion of the primitives for the current region, the results from all of the nodes are merged, or *composited*, based on each pixel's Z value. The process repeats for the remaining 128×64 portions of the screen until an entire image is generated. The result is sent to a frame buffer for display.

The (simplified) sequence of instructions for rendering a triangle is as follows:

1. Enable all pixel processors.
2. Disable all pixel processors “on the outside” of the first (arbitrarily chosen) edge of the triangle.
3. Disable all pixel processors “on the outside” of the second edge of the triangle.
4. Disable all pixel processors “on the outside” of the third edge of the triangle. At this point, only those processors corresponding to the interior of a triangle are left enabled.
5. Color the remaining enabled pixel processors according to the color of the triangle.
6. Calculate the Z value at all enabled pixel processors.

Steps 2 through 4 are executed by expressing the edges of the triangle as linear expressions and disabling all pixel processors for which $Ax + By + C < 0$ holds. This is done rapidly since the LEE can evaluate this expression in parallel for all processors. Once all of the pixels lying outside of the triangle are disabled, the color and Z value are calculated in parallel for each processor using the LEE. The process repeats for all triangles, the final result is a partial image stored in the array of pixel processors.

The key points in the rendering sequence are summarized below:

1. The GPs are responsible for the geometry calculations and for generating the IGC instructions for rendering the primitives.
2. Each PE represents one pixel within the current region.
3. Each node iterates over distinct regions of 128×64 pixels.
4. Each node contains a subset of the primitives. When all of the primitives have been rendered into a 128×64 region, the node is finished with the current region. The pixel data is then composited with the corresponding region for the other nodes.

5. An image is complete when all 128×64 regions have been composited.

2.2.3 Floating Point Performance

Floating point calculations on the EMCs are computationally expensive since all floating point calculations must be broken down into a series of 8-bit integer operations. However, since computation can proceed in parallel across 8K processors, the potential aggregate performance is quite high. Table 1 summarizes the peak floating point performance (32-bit single-precision) of one board running at 100 MHz. To realize this performance, all PEs must be enabled and must perform non-redundant computations. This table does not reflect optimizations which can be obtained by eliminating unnecessary calculations in consecutive floating point operations or by writing custom microcode instructions tailored to the operations needed for floating point computation.

Operation	Cycles / Op	Ops / PE-Sec	Ops / Sec
Integer->Float	157	636 K	5.2 G
Add	390	256 K	2.1 G
Multiply	253	395 K	3.2 G
Divide	704	142 K	1.2 G
Square Root	698	143 K	1.2 G

Table 1 Peak Floating Point Performance on the PEs

2.2.4 Data Transfer Bandwidths

Without efficient data movement into and out of the EMCs, floating point bandwidth becomes meaningless. There are three ways to transfer data to and from the EMCs:

1. From the LEE
2. To/From texture memory
3. To/From the GP through the GNI

The bandwidth for loading 8 byte linear expressions into PE memory through the LEE is about 700 GBytes/second. The fastest option for loading general (*i.e.*, non-structured) data is from texture memory, with a peak transfer rate of about 3.2 GBytes/second. This is roughly 16 times the bandwidth between the GPs and the EMCs. Other than sending the

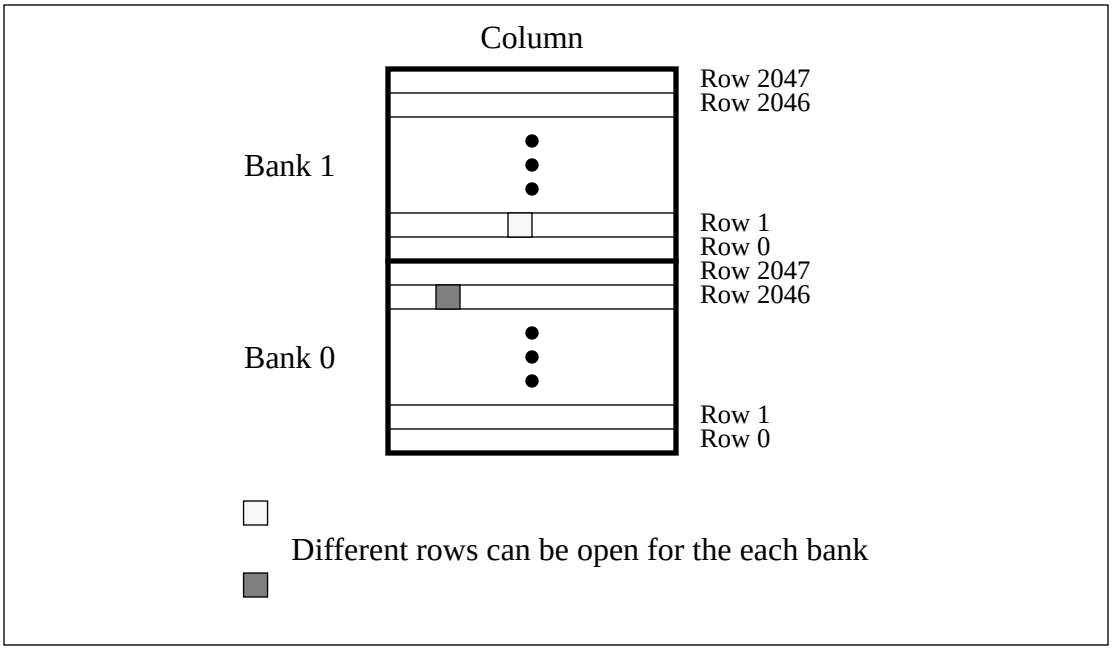


Figure 5 Structure of an SDRAM

final results from the EMCs to the GPs, the focus will be on transferring data through the LEE or from texture memory.

2.2.5 Instruction Streams

As was mentioned above, the EIGC and TIGC execute separate instructions streams in parallel. This concept is very important. The performance of the tessellation algorithm relies heavily on the ability to load data into and transfer data out of the PEs while they are performing computations. A significant portion of Chapter 3 is devoted to posing the tessellation problem so that the TIGC can execute an instruction stream which performs the data transfer while the PEs execute instructions which perform the necessary computation.

2.2.6 Structure of an SDRAM

Each SDRAM supplies 2 megabytes of memory divided into 2 banks of 2048 rows of 512 byte columns, as shown in Figure 5. Data are read or written by first selecting a bank and row (“opening” the row), then selecting one or more columns, at random, within that row. Each bank may simultaneously have one open row. Whenever a new row is required, the bank must first be precharged, then the new row must be selected and finally bytes within the column can be accessed.

This process has several implications. First, once a row has been accessed, one byte of data can be transferred to or from the SDRAM per clock cycle. Also, it is possible to "hide" the row access and/or precharge time by transferring data from one bank while pre-charging the other bank. This is useful when there is a guarantee that new row accesses occur in alternate banks (*e.g.* when the data are duplicated in both banks of the SDRAM). Table 2 shows the SDRAM data bandwidths for various accessing schemes. Note that guaranteeing alternate bank accesses increases throughput, and that at least 8 bytes must be transferred per access to completely hide the bank precharge and row access overhead.

Access Method	Description	Cycles	Bytes Transferred
4 Byte Random Access	Transfer 4 bytes, next transfer can occur in either bank, any row	11	4
4 Byte Alternate Banks	Transfer 4 bytes, next transfer must occur in the opposite bank, any row	6	4
8 Byte Random Access	Transfer 8 bytes, next transfer can occur in either bank, any row	15	8
8 Byte Alternate Banks	Transfer 8 bytes, next transfer must occur in the opposite bank, any row	8	8

Table 2 Steady State SDRAM Data Bandwidth

2.2.7 Reading/Writing Texture Memory

Each module contains its own 8 MBytes of texture memory. As mentioned before, this increases memory bandwidth for some applications (since each module can read/write to its memory in parallel), but implies that loading common data from texture memory into all of the PEs requires the data to be replicated across all of the modules.

Figure 6 depicts the steps necessary for reading from texture memory. Each EMC must supply eight 4-byte addresses. The addresses are sent, byte serially, simultaneously from all EMCs to the TASICs where they are stored in the Address Corner Turner (ACT). The TASICs then iterate over each of the EMCs' addresses, reading one word from each of the 8 SDRAMs in parallel. The TASICs store the data in the Data Corner Turner (DCT). Once one set of addresses is processed, the data are sent back to the EMCs. As with the addresses, this is done by sending each EMC's data byte serially, in parallel for all EMCs.

Data are written to texture memory in a similar fashion, except now data and addresses must flow in the same direction (from the EMCs to the TASICs). First, data and addresses are sent to the TASICs in parallel for each EMC. Once the ACT and DCT are full, a series of writes store the data in the SDRAMs.

There are several possible variations to the general read/write process described above. First, the addresses do not need to be explicitly generated in the EMCs and sent to the TASICs, but can be constructed using the TASICs' address generation ability. The TASICs contain a counter and a crossbar (a MUX which, for each address bit, selects any of the counter's outputs) which can generate sequences of addresses. This is useful for stepping through a predetermined series of memory locations. The two methods can be combined so that an explicit base address is provided by the EMCs and the address generation mechanism then sequences through a set of offsets from the base address. Note that providing an explicit address when reading is free in terms of bandwidth since address and data flow in opposite directions, whereas during writing it halves the bandwidth since both data and address information must be sent from the EMCs to the TASICs.

Another variation reuses either the addresses or the data. That is, once the ACT and DCT are filled, the data can be accessed multiple times. This is useful for writing the same data to multiple addresses in the SDRAMs, or more importantly, for sending the same data to multiple PEs within an EMC.

There is one final question concerning texture memory access: in what order are the PEs within an EMC processed? Before executing the texture instruction, each PE which is to participate must first be marked for reading and/or writing data. The texture instruction then processes all EMCs in parallel, accessing the PEs within an EMC sequentially. The PEs within an EMC may be processed in either panel minor or PE minor order. Under PE minor order, all of the marked PEs in panel 0 are processed, then the marked PEs in panel 1 are processed, etc. For panel minor order, one marked PE is processed from panel 0, then one from panel 1, round robin until all PEs are processed.

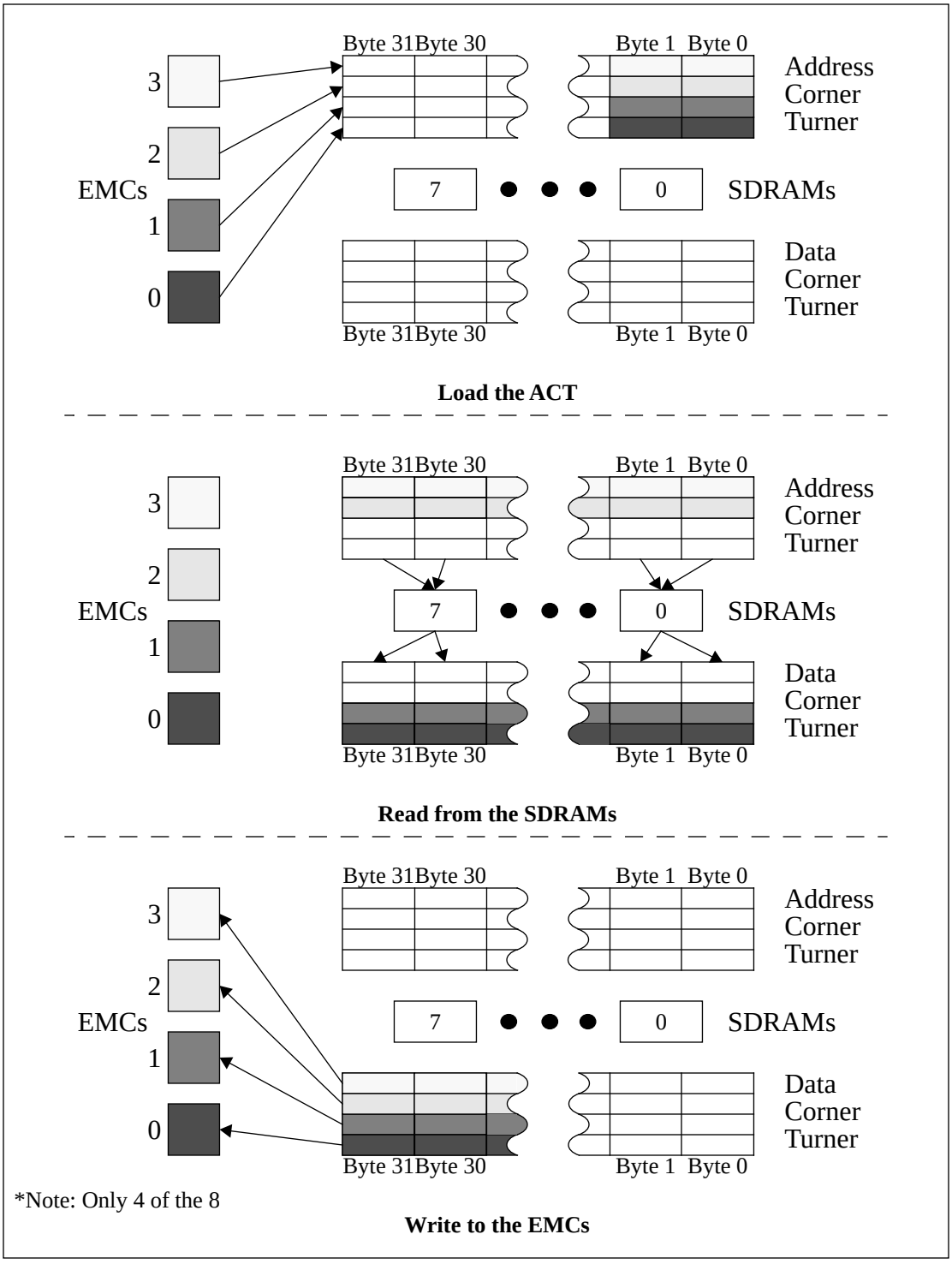


Figure 6 Read from Texture Memory

2.2.8 Inter-PE Communication

There are several possible ways for PEs to communicate with each other. Listed from highest to lowest bandwidth, these are:

1. PixCopy*N* and PixSwap*N* instructions. These instructions only allow communication between PEs on the same panel; however, the operation occurs across all panels in parallel. With these instructions, the 32 PEs on a panel can send or swap data *N* PEs away.
2. Loop-back mode. It is possible to use certain TASIC instructions that read data from PEs marked for writing and then, instead of writing the data to texture memory, loop the data back to other PEs which have been marked for reading. This allows for communication between PEs on a given EMC; which PEs communicate depends on which PEs were marked for reading/writing and the processing order as described above.
3. Writing to texture memory. Finally, it is feasible, though generally not practical, to write data from the PEs to texture memory, then read it back into other PEs. This process is complicated and incurs too much overhead for this application.

2.2.9 Configuring the LEE

As discussed above, the LEE evaluates the expression $Ax + By + C$ in parallel for each PE. The PEs can be viewed as sitting in a two dimensional array; the EMCs contain registers which control the positioning of their PEs [Eyles 95] within the array. Modifying these registers reconfigures the LEE which changes the (X, Y) coordinates of the PEs. The registers are used in the following equations to control the (X, Y) coordinates of the processors:

$$y(P, I) = 2^{ye} \times yp[P] + ys[I \bmod 8]$$

$$x(P, I) = 2^{xe} \times xp\left[(P \bmod 2) \times 8 + \frac{I}{4}\right] + xs[I \bmod 8]$$

where P is the panel number and I is the PE number within a panel. The register sizes are as follows: xe and ye are 2 bits, xp is 8 bits, yp is 7 bits, and xs and ys are 4 bits.

Different configurations lead to different PEs being "adjacent" in both X and Y. That is, one configuration may make PEs 0 and 1 on panel 0 adjacent in Y while another configuration may interleave the panels. Note that it is not possible to position the PEs arbitrarily since the configuration registers apply to sets of PEs (notably whole EMCs and whole panels).

2.2.10 Summary

PixelFlow is a complicated machine with many (sometimes subtle) details. Each of the pieces described in the previous subsections is utilized in the tessellation algorithm to make the parallel implementation feasible. The text in the succeeding chapters contains many references to help the reader review the salient portions of this chapter as they are used to address each subproblem.

3.0 Approaches to Tessellation

The problem of how to tessellate surface patches using PixelFlow pixel processors is probably not solvable in the general sense; there are too many variables to produce one solution which will be efficient for all possible operating paradigms. Therefore it is important to analyze the variables that affect performance to determine how to resolve those variables so a particular application can benefit from a parallel implementation.

This chapter highlights the key issues that arise when using PixelFlow rasterizers to tessellate Bézier surface patches. The goal is to concentrate on the feasibility of the problem, outline the rationale for the major design decisions and to anticipate any implementation problems that may be present without explaining them in detail. The implementation of the ideas discussed in this chapter will be described in depth in the following chapter.

The first section below discusses the tessellation paradigm; that is, how to break up the tessellation problem so that it is solvable with PixelFlow hardware. Section 3.2 goes into more detail and describes mapping the surface patches onto the EMCs. The next five sections proceed step by step through the tessellation process. Section 3.9 mentions a few points that affect the application program and, finally, Section 3.10 summarizes this chapter's results.

3.1 Parallel Tessellation Paradigm

It is clear that the 8K PixelFlow PEs offer a lot of computational power; however, it is not immediately apparent how to use this power to tessellate Bézier surface patches. For example, the SIMD nature of the machine precludes pipelined approaches as all processing elements receive the same instructions and so must work on the same part of the problem.

A related problem concerns how far to proceed with the computation on the EMCs. Some problems can clearly benefit from a parallel implementation while others are fundamentally serial; however, the dividing line between the two is not necessarily obvious. Also, if all computation is performed on the EMCs, the GPs must sit idle, wasting a precious

resource. These issues, combined with communication bandwidth concerns raise the question of how to divide the computation over the GPs and the EMCs. The following sections discuss the scenarios which benefit from performing tessellation on the EMCs, how much of the tessellation computation should be performed on the EMCs and how the problem should be divided over the EMCs' processors.

3.1.1 Load Balancing

The PixelFlow rendering pipeline for a typical application is shown in Figure 7. While the EMCs are rendering frame N, the GPs transform and project the primitives for frame N+1, interrupted only to supply additional (buffered) instructions to the IGC for frame N. If the pipeline is geometry bound, then the EMCs will sit idle. This generates the ideal scenario for using the EMCs for tessellation. On the other hand, if the pipeline is render bound, then there is no payoff in using the EMCs for tessellation.

Note that even in the case where it is advantageous to off-load the tessellation process onto the EMCs, there is still the question of how to allocate the GPs' processing power. The answer to this question is application dependent; it may be possible to divide primitives which need to be tessellated between the GPs and EMCs or perhaps the GPs can perform coordinate transformations for other primitives while the EMCs tessellate.

3.1.2 Division of Labor

Rendering a Bézier patch requires tessellation, projection of the vertices and finally generation of rendering instructions from those vertices. Given this process, the question that naturally arises is: how many stages of this computation should be performed on the EMCs and how many on the GPs? There are three possibilities, in each case the EMCs are supplied with control points but each succeeding case carries the computation on the EMCs farther. The general ideas are presented below; Section 3.7 discusses the problems in detail. The three possibilities are:

1. Generate points and normals in object space. The GPs then proceed with these data as they do with other primitives, as described in Section 2.2.2.

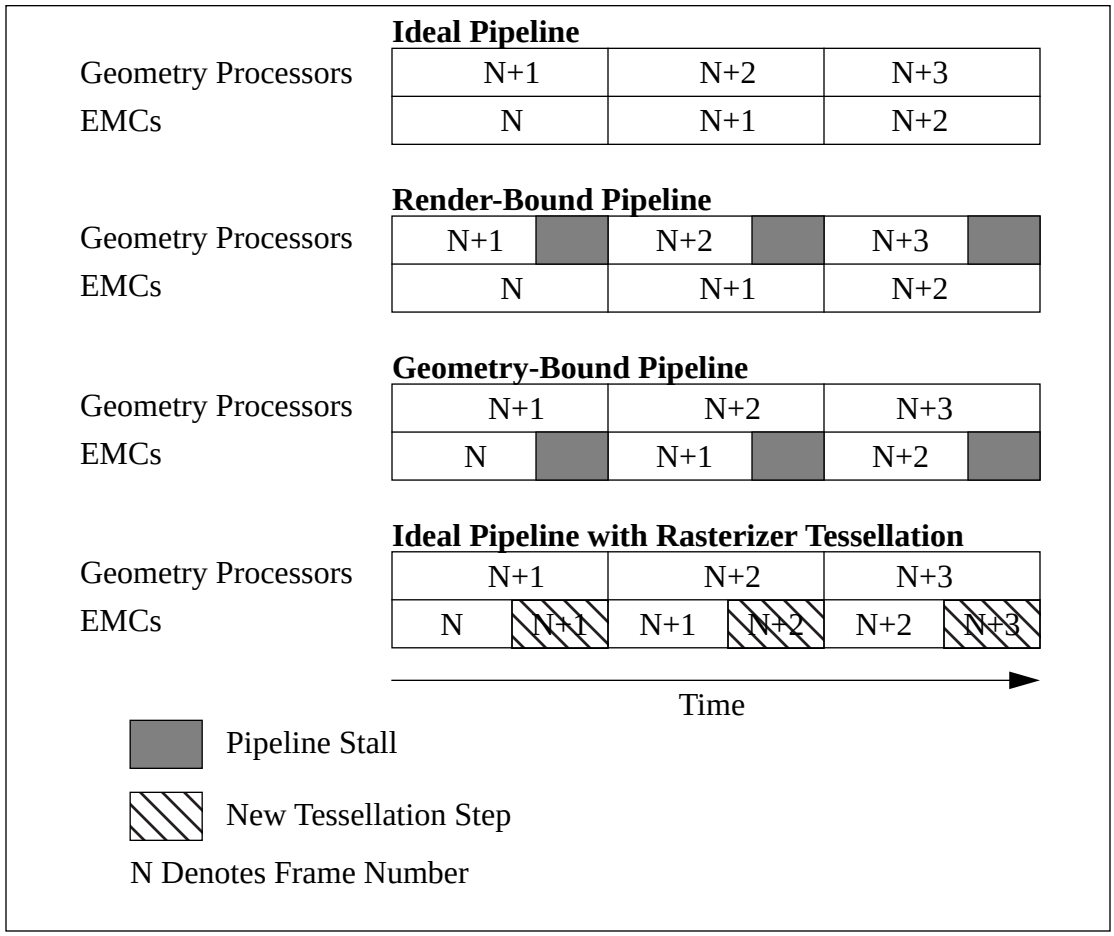


Figure 7 Rendering Pipeline

2. Generate projected points. The EMCs perform the projection and return transformed vertices and normals. This requires that all of the surface patches be stored in the same coordinate system. The points could either be transformed to the canonical view volume, or even projected into screen space and marked for acceptance or trivial rejection.

3. Generate IGC parameters. The EMCs carry the computation through to the generation of the parameters used by the IGC instructions fed to the EMCs to render triangles (again refer to Section 2.2.2 for details).

The first method generates points in object space which lets the application decide when to re-tessellate a particular patch, perhaps based on viewing parameters. The last two methods carry the computation through the viewing transformation. Therefore, either both

intermediate object space data and final transformed points must be sent to the GPs, or all patches must be re-tessellated whenever the viewing parameters change.

The third option introduces other difficulties: bucketization and clipping. Bucketization refers to the process of determining which regions (see Section 2.2.2 for review) a given primitive overlaps; rendering instructions are sent to those regions only. However, if the EMCs generate IGC instructions directly, this stage is skipped. Thus it becomes necessary to calculate the bucketization information on the EMCs and send these data back with the IGC parameters. Finally, since the clipping process may change the amount of data per PE (by introducing new triangles), writing the results becomes more complicated.

3.1.3 PE Allocation

The previous sections addressed how much computation to perform on the EMCs but left open the question of how to divide the problem over the EMCs. Pipelined approaches have already been ruled out. An alternate idea, which utilizes PixelFlow's SIMD nature, is for each PE to generate all of the samples for one patch. There are two limitations inherent in this idea. First, to ensure 100% utilization of the PEs requires tessellation of 8K surface patches (1 patch for each of 8K PEs, a very large data set). Second, since all of the PEs execute the same instructions, all of the patches must be tessellated to the same resolution.

These problems are lessened by instead having each PE determine one sample of a surface patch. This meets the SIMD requirement as long as all patches are of the same degree. The number of patches that are required for full PE utilization is reduced by 1/16 for 4×4 patches and by 1/256 for 16×16 patches. Also, it is conceivable that some patches could be tessellated at one resolution while others are tessellated at a different resolution.

If it is beneficial to reduce the problem from tessellating a whole patch per PE to one sample per PE, why not reduce the problem to generating part of a sample (say, the X coordinate) per PE? While this is feasible, the latter parts of the computation, namely calculating the surface normal and projecting the generated points, require that all of the data for a sample be present in one PE. Even assuming that it is physically possible to do this data

shuffling and that it can be accomplished inexpensively, it still adds complexity and reduces the processor utilization.

The remainder of this document provides an overview to the problem of tessellating surface patches on PixelFlow EMCs given that each PE calculates one sample for one surface patch and that all patches tessellated simultaneously are of the same degree. The question of how much computation to perform on the EMCs will arise again; the goal will be to make decisions which do not preclude performing all stages on the EMCs. However, the focus will be on generating the surface points and normals leaving many of the remaining details for future work.

3.2 Mapping Surface Patches onto the PEs

The next question to be addressed is the mapping of surface points from multiple surface patches onto the PEs. Put another way, which PE generates which surface point for which surface patch? There are several issues which determine the best mapping; some of them are necessary while others are merely convenient.

The primary concern is accessing texture memory. If the control patch is stored in texture memory, then, unless the data are duplicated across modules, all of the samples for a surface patch must lie within one module. Furthermore, the design of the texture memory interface allows each EMC to specify a base address and multiple data items for writing, (see Section 2.2.7 for details). This corresponds to sending a base address for a patch, followed by several samples on the patch. Write bandwidth is improved only if multiple surface points for a patch lie within one EMC; it provides optimal improvement if a whole surface patch maps to within one EMC. More importantly (as described in detail in Section 3.5), the interface allows data to be read once from texture memory and then sent multiple times, provided it is sent to the same EMC. All of these requirements point towards allocating each patch to consecutive PEs within one EMC.

Of secondary concern are the inter-PE communication limitations. In order to generate the IGC instructions needed to render a surface patch on the EMCs, it is necessary for each PE to operate on all of the vertices of a triangle. This requires sending adjacent surface points

to the other PEs operating on the same surface patch. Also, the control points for a patch must be sent to all PEs representing that patch. Inter-PE communication can aid in this process by first loading data into one (or a few) PEs and then using the local communication paths to distribute the data to the remaining PEs for the patch. This implies that a patch should either map to within a panel, since PEs within a panel can communicate rapidly, or should at least map to neighboring panels within an EMC. There are only 32 PEs to a panel, so clearly limiting a surface patch to lying within one panel is too strict.

Finally, it may be more efficient for some operations if the LEE and the patch to PE mapping is coherent. For instance, there should exist a correlation between the sample indices (I, J) for a patch and the (X, Y) coordinates of the PE. Also, a correlation between the same (I, J) coordinates of a sample on different patches would be beneficial, *e.g.*, sample (0, 0) occurs every 16 PEs in X and Y. Last of all, given that patches are contained wholly within modules and EMCs, it would again be helpful if the PEs within these entities were contiguous in (X, Y) space.

As discussed in Section 2.2.9, the LEE is configurable, but not arbitrarily so. For example, it is not possible for two panels on one EMC to be adjacent in X.

All of these considerations give rise to the patch-to-PE mapping and LEE configuration given in Figures 8 and 9. The salient features are:

1. A given surface patch must always fall within one EMC
2. 4×4 sampled patches lie within half of one panel (call this a block)
3. There are two blocks, on consecutive PEs, within one panel
4. Larger (8×8 and 16×16 sampled) patches are formed out of adjacent blocks
5. Corresponding samples on separate patches occur periodically in X and Y
6. Modules, EMCs and panels are contiguous in X and Y

The astute reader may have observed that all sampling resolutions mentioned so far are powers of two. This restriction is imposed for reasons of simplicity and efficiency. For example, allowing non-2^n resolutions complicates mapping multiple patches to the PEs

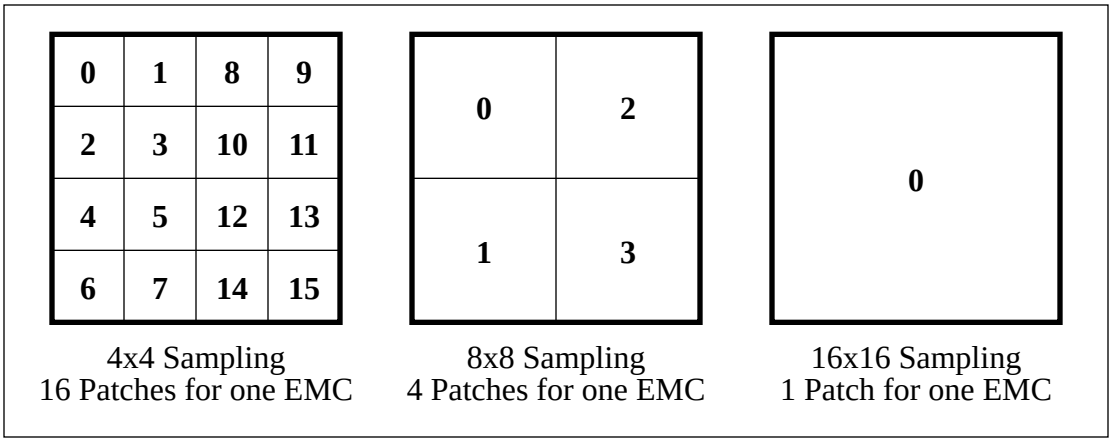


Figure 8 Surface Patch to EMC Mapping

since PE fragmentation becomes an issue, as discussed in Section 4.1. More importantly, as discussed above and below in Section 3.5, a coherent mapping between surface patches

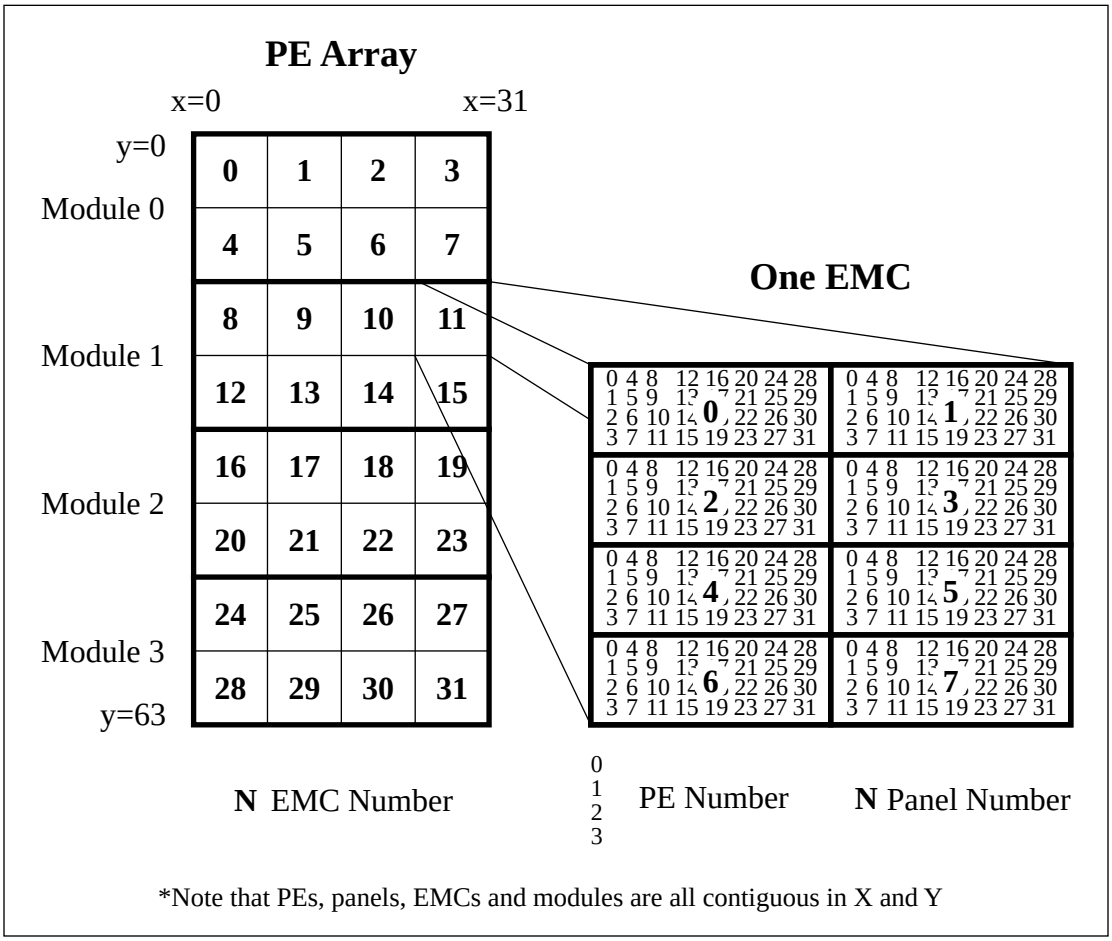


Figure 9 LEE Configuration

and panels of processors decreases the time required to load control patch data into the PEs. Allowing non-2^n sampling will break the mapping by forcing patches to straddle panel boundaries. However, other patch sizes may be used by sacrificing processor utilization. For example, a 5×5 patch could be tessellated using the same number of PEs that an 8×8 patch would use, reducing utilization by 60%.

One final note: the size of a tessellated patch can have a profound impact on the tessellation algorithm's performance. For example, using the LEE to load the control points into each PE requires iterating over each patch, enabling the PEs corresponding to the current patch while disabling the others. For 64×64 samples per patch this takes about 20 cycles. On the other hand, the same procedure requires 2000 cycles for an 8×8 patch since in the iteration process the majority of the PEs sit idle while data are loaded into the enabled PEs. Restricting all patches to no more than 16×16 samples reduces the number of design decisions and makes a single solution tractable. This decision is justified by noting that, in general use, patches are rarely tessellated with more than 16×16 samples.

3.3 Loading the Control Patch Base Addresses

All of the PEs that generate samples for a particular patch must have access to the corresponding control patch. This implies that each processor must be loaded with the appropriate texture memory address. Loading these base addresses into each PE is the first step in the tessellation process; details are provided in Section 4.1.1.

3.4 Loading the Bernstein Polynomials

The next step in the tessellation process is to load the Bernstein polynomials, $B(u)$ and $B(v)$ (see Section 2.1.1 for review), into each PE. The final result of loading these parameters is that, for the same size patch, all PEs in corresponding (I, J) positions contain the same Bernstein polynomials. This suggests the following approach: precompute and cache the parameters on the GPs then broadcast them to the EMCs through the LEE.

For a given patch size, the Bernstein polynomials for one direction (I or J) are common to the PEs perpendicular to that direction. For example, in a bicubic patch the PEs repre-

senting the surface points $P_{i,1} = \{P_{0,1}, P_{1,1}, P_{2,1}, P_{3,1}\}$ vary in $B(v)$ but all receive the same $B(u)$ values. Using this information, it is possible to iterate over the PEs in I loading the polynomials for u, then iterate over the PEs in J, loading the polynomials for v. Although this must be done for each surface patch size (4×4, 8×8, and 16×16), the time consumed is still a small portion of the total tessellation time.

3.5 Loading the Control Points

Loading the control points into the EMCs is perhaps the most important step in the parallel tessellation process. If it is not possible to rapidly load the data, then the EMCs must sit idle, wasting the performance that we are trying to exploit. To obtain peak performance, this section relies heavily on the PixelFlow hardware description, particularly Sections 2.2.4 through 2.2.7.

Ideally, the control patch data base would reside in GP memory to allow easy access for the application. However, this would require loading the data into the PEs through the LEE, which is not only prohibitively expensive in time, but also prevents the EMCs from performing useful computation while the data are being loaded. Instead, the data base is stored in texture memory (and perhaps duplicated in GP memory), which allows the transfer of control points in parallel with computation in the EMCs.

The data fetches from texture memory SDRAMs occur in round robin EMC order. This implies that each read from the SDRAMs is for a different control patch because patches do not span EMC boundaries. Since control patches may be accessed arbitrarily from texture memory, there is no guarantee that two consecutive accesses to texture memory will either lie in the same row or in alternate banks of the SDRAMs. Unless the control patch data are duplicated in both banks, the SDRAM data bandwidth will be one 4-byte word every 11 cycles.

One important realization compensates for the slow SDRAM bandwidth: all samples for the same patch need access to the same control points. This makes it possible to read data into the TASIC and then repeatedly send the same data to the EMCs. More precisely, because of the surface patch to PE mapping, there is a guarantee that 16 consecutive PEs

lying within one EMC will always be working on the same surface patch, so it is only necessary to perform one read for each block a patch overlaps. That is, a 4×4 patch requires one read, an 8×8 patch requires four reads and sixteen reads are necessary for a 16×16 patch. It is tempting to think that a 16×16 patch requires just one read since all processors dedicated to that patch lie within an EMC. However, it is important to realize that another EMC may be divided into 16 4×4 patches. Since the EMCs are processed in parallel, the number of times the data are read must be the maximum required over all of the EMCs, namely, the number of blocks per PE, which is 16.

There is another method which can reduce the cost of loading the control points. Instead of repeatedly sending the same data to consecutive PEs *within a panel*, the data can be sent once per panel and then distributed by utilizing the inter-PE communication capabilities. This has the negative effect of increasing the number of EMC cycles. The trade-off lies in the number of TASIC and EMC cycles consumed: more sends require more TASIC cycles but fewer EMC cycles. The appropriate balance depends on whether or not the EMCs can perform useful computation while the TASICs send the data. The duration of a transfer from texture memory is inconsequential so long as useful computations are performed on the EMCs while the data are being transferred. Section 4.1.3 provides the details for achieving a reasonable balance.

The italics used above highlight the impact of the surface patch to PE mapping devised in Section 3.2. In particular, mapping a surface patch to within an EMC lets the TASICs read data from the SDRAMs once then send the data to the EMCs multiple times, increasing the effective texture memory bandwidth. Next, providing a coherent mapping of patches to panels allows the inter-PE communication paths to distribute the data to the other PEs working on a patch, effectively decreasing the quantity of data that must be sent from texture memory to the EMCs. These two methods combine to dramatically reduce the amount of time required to load a control patch into the PEs.

The process described above trades the problem of loading the control points into the PEs for the problem of loading texture memory addresses into the PEs. Now instead of having to broadcast the control patch to the PEs, the base addresses must be broadcast. At first

glance, these problems seem identical in that both require iterating over each patch. However, only one 4-byte address needs to be loaded for each block, whereas a bicubic control patch, for both points and normals in X, Y and Z, requires loading 480 bytes.

3.6 Tessellation

The tessellation process consists of three major steps:

1. Generating the points $P(u, v)$ on the surface (consists of X, Y, Z)
2. Generating the derivatives in u and v (consists of X, Y, Z for both u and v)
3. Generating the surface normals

The first step is accomplished by performing the tessellation matrix multiplications on the control patches for X, Y and Z, as described in Section 2.1. Similarly, the second step consists of two sets of matrix multiplications per coordinate, one operating on the first differences in u, the other on the first differences in v. The surface normals are generated by taking the cross product of the resulting vectors in u and v.

The operation would be straightforward if all of the data for each control patch was completely resident in PE memory when needed. As can be deduced from the discussion in the previous section, this is not the case. In particular, the data becomes available to the PEs through a series of 32 byte transfers. Ideally, to avoid computation stalls, the number of cycles required to operate on one set of 32 bytes should match the number of cycles required to transfer the next set of 32 bytes. Section 4.1.3 in the implementation chapter provides the cycle counts for processing one “batch” of data for different size control patches.

Another issue to contend with while performing the tessellation computation is PE memory management. Each PE contains 320 bytes[1] for general storage. The procedure described above requires its maximum amount of memory when calculating the derivative value for Z in v. At this time, the surface point, the derivative vector in u, and the X and

1. 256 bytes of main memory plus 64 bytes for the image composition network registers.

Y components for v have been calculated and must all be stored while the Z component for v is generated. This requires a total of 72 bytes (details are found in Section 4.2) in addition to the Bernstein polynomials and any other miscellaneous variables that may need to be stored. Clearly PE memory usage is not a primary concern.

3.7 Projection and IGC Instructions

At the completion of the tessellation process, each PE contains one point on a surface patch in modelling (or world) coordinates. The use of the EMCs could terminate at this point by sending the generated data back to the GPs where processing would proceed as with any other polygonal database. However, provided that all of the surface patches are in the same coordinate system, it is also possible to apply the viewing transformation and even generate the IGC instructions in parallel on the EMCs.

Proceeding beyond generating points in viewing or modelling coordinates introduces several complexities to the tessellation problem. First of all, the generated triangles may be rendered with any of several different lighting and shading models. Since the data required for rendering varies with the model used (from just the point in image space to surface point and normal in world space plus point and normal in image space), each model will require a customized implementation to send the necessary set of data back to the GPs. Second, as will be discussed in detail in Section 3.8, data transfer to the GPs is expensive. By applying the viewing transformation on the EMCs, sending data to the GPs is delayed until the last stage of the computation rather than sending data back as it is generated. Combining these two problems make matters even worse: the data may not be available to be sent to the GPs until the end of the computation, and the lighting model may require more data to be transferred than if the procedure terminated after generating points and normals in world/object space.

Despite these complications, the high performance of the EMCs may still yield a performance gain for at least some of these additional calculations. Since the problem becomes more intricate and application specific, the details are left for future work. However,

Appendix B discusses some implementation details that will aid in performing these additional steps.

3.8 Storing Results

Regardless of how far the computation on the EMCs is carried (world space, image space or IGC parameters), eventually it becomes necessary to send the results to the GPs. From the perspective of the EMCs, communication with the GPs is identical to writing to texture memory. Initially, data are loaded into the local port then a TASIC instruction is issued to read the data. The difference lies in the bandwidth: texture memory transfers occur at 3.2 GBytes/second, the geometry network at 800 MBytes/second and GP memory at 200 MBytes/second.

The most aggressive approach to writing to GP memory utilizes the full 800 MBytes/second of the geometry network (and EMC to GNI path) by making one board a “tessellation server” with four client nodes each receiving one quarter of the results. Since the bandwidth from the GNI to GP memory is one quarter that of the geometry network, the combination fully utilizes the geometry network and GP memory bandwidths. In this scenario, the surface patch database is stored on one node which is dedicated to performing the tessellation computation. The four adjacent boards in the network receive the results of the tessellation and are responsible for rendering the generated patches. This approach introduces many synchronization issues and relies on portions of the hardware which are not yet implemented, and so, while promising, was not pursued. Even the single board approach is laden with details, so the discussion is deferred to Section 4.1.4.

3.9 Application Program

The preceding sections addressed how to use PixelFlow’s rasterization engine to tessellate surface patches, but did not mention the impact on the application program. Some highlights are explained here, the details are left for Section 4.4.

The paradigm described above relies on storing the control patch database in texture memory to obtain peak bandwidth into the EMCs. This retained-mode operation requires the

application to load the patches into texture memory at initialization time. Further modification of the database must be performed by API calls using handles to the control patches. These modifications require writing to texture memory, and so must be synchronized with the tessellation process.

Tessellation is performed by marking the patches which need to be tessellated through API function calls. After all patches for a particular frame have been marked (including the resolution at which to tessellate), the library decides which patches to tessellate and how to allocate them to the PEs. Once the tessellation process finishes, the handles for each patch are updated to point to the results sitting in local memory.

3.10 Conclusion

This chapter addressed the major issues involved in tessellating Bézier surface patches on PixelFlow hardware. The highlights are listed below:

1. Each PE generates one surface point for a patch.
2. PEs are grouped into blocks of 16 consecutive PEs in one panel on an EMC.
3. Each block generates 4×4 samples for one surface patch (4 blocks are used for 8×8 samples and 16 blocks for 16×16 samples).
4. The step by step tessellation process is as follows:
   - Load Base Addresses
   - Load Bernstein Coefficients
   - Calculate X Control Patch
   - Calculate Y Control Patch
   - Calculate Z Control Patch
   - Calculate X' Control Patch
   - Calculate Y' Control Patch
   - Calculate Z' Control Patch
   - Cross Product for Normals

- Write Results

5. The benefits of performing additional computation on the EMCs must be evaluated on an application by application basis.
6. If not done carefully, writing the results back to the GPs can be as expensive as the tessellation process.

The next chapter explores the details of implementing these ideas on PixelFlow.

4.0 Tessellation Implementation

The previous chapter provided an overview of the problem of tessellating surface patches on PixelFlow hardware. The goal of this chapter is to supply the details which were omitted from that introduction. Once again, it is important to realize that this work was completed before PixelFlow was physically constructed. Most of the results described below were implemented and verified in simulation; however, since the simulator is incomplete, some work was done in theory only. In particular, none of the work involving the communication path from the TASICs through the GNI to the GPs has been implemented or simulated.

The first section below contains the step by step details of implementing the tessellation operation on PixelFlow, which can be broken down into a sequence of four operations:

1. Mapping the patches to be tessellated to blocks of PEs
2. Loading the texture memory addresses for each patch
3. Loading the Bernstein polynomials
4. Performing the actual computation

Section 4.2 provides a detailed map of PE memory during the tessellation operation. Section 4.3 discusses the issues of storing the control patch and generated samples in texture memory. Section 4.4 summarizes the application interface to the algorithm and finally, Section 4.5 summarizes the chapter's results.

4.1 Tessellation

At the start of the tessellation process, the application has already marked all of the surface patches which must be tessellated, (described in Section 4.4 below). Each modules' patches are then assigned to the blocks of PEs corresponding to that module. First patches tessellated to 16×16 samples are allocated, in the remaining space 8×8 patches are allocated, and the remaining PEs tessellate 4×4 patches. If any module needs to tessellate more patches than is possible in one pass, the extra patches are either tessellated on a succeeding pass or are marked for tessellation on the GPs.

The patches are mapped onto the blocks of PEs as depicted in Figure 8. They are assigned to blocks based on the sampling resolution: patches with more samples are allocated before smaller patches. This pattern prevents block fragmentation when assigning patches to blocks since as each patch is mapped, the remaining clusters of blocks are able to accommodate any patch of the remaining sizes. The fragmentation problem would be exacerbated by allowing patch sizes other than 2^n. Finally, the allocation pattern facilitates operations that occur only within a particular block size, such as loading the Bernstein polynomials.

As mentioned before, at the cost of quadrupling memory usage, the application can have the flexibility of tessellating any patch on any module. Alternatively, the patches can be stored on just one module, which saves memory but introduces load balancing issues. A possible compromise is to store patches on two modules, providing two options for load balancing but saving some storage space over full replication. The optimal solution is application dependent. For this reason, the simplest method, non-replication, was chosen for the first implementation.

The ultimate goal of the tessellation implementation is to constantly perform floating point computations on all of the PEs, that is, to obtain 100% processor utilization. To achieve this, the time to transfer data from texture memory should match the time required to process that data. This will be the benchmark for the success of the tessellation algorithm.

4.1.1 Loading Addresses

The first step in the tessellation process is to load the control patch into PE memory. However, before this can be done, the texture memory addresses for each patch must be loaded. Section 4.3 discusses in detail how a control patch is stored in texture memory. For now it is sufficient to know that the data can be accessed by first transferring a base address for each patch then generating offsets from that base address, as discussed in Section 2.2.7. Therefore, the first step is to load the base address for each patch.

The address is loaded by iterating over all of the patches and enabling all blocks of PEs corresponding to the current patch while disabling the remaining blocks. Once only the PEs corresponding to the current patch are enabled, the base address is loaded. Also, the

PEs are marked as belonging to a 4×4, 8×8 or 16×16 patch. This step facilitates loading the Bernstein polynomials, as described in the next section.

The number of cycles required for this stage depends on how many surface patches there are to be tessellated, which, in turn, is determined by the number of patches of each sampling resolution. The worst case occurs when all patches are tessellated with 4×4 samples. In this case, addresses for 512 patches must be loaded, each of which takes 12 cycles. The costs for each patch size are summarized in Table 3. Note that in general the number of cycles required will fall somewhere between the values shown since in general there will be a mix of patch sizes. Finally, note that these cycles represent an inefficiency in the tessellation process since this computation cannot be overlapped with any other useful operations.

Patch Size	Cycles Per Patch	Number of Patches	Total EMC Cycles
4x4	12	512	6144
8x8	12	128	1536
16x16	12	32	384

Table 3 Cycles Required to Load Base Addresses

4.1.2 Loading the Bernstein Polynomials

As soon as the patch base addresses are loaded into the PEs, the first data transfer is initiated. While this transfer is in progress, the EMCs load the Bernstein polynomials used in the tessellation process, as discussed in Section 3.4. The worst case performance occurs when at least one patch is tessellated for each possible resolution; that is, there is at least one 4×4, one 8×8, and one 16×16 patch. This is the worst case because, in general, the location of the samples depends on the sampling resolution Therefore different sampling resolutions require different polynomials. Multiple patches for a given resolution are free since the polynomials can be loaded for all patches of the same resolution in parallel.

It takes 5 cycles to load each of the $n + n - 1$ values in u and v for the control patch and first differences. Iterating over each resolution gives a total of:

$$2\,(5\,(n + n - 1)\; + 6)\;\;(4 + 8 + 16)$$

cycles, including 6 cycles of processing overhead. The results are summarized in Table 4. It should be noted that allowing additional non-2^n sampling resolutions would adversely impact the cycle counts since the load operation would have to iterate over each additional sampling resolution.

Control Patch Size	Cycles for 4x4	Cycles for 8x8	Cycles for 16x16	Cycles for All Sizes
4x4	328	656	1312	2296
5x5	408	816	1632	2856
6x6	488	976	1952	3416
7x7	568	1136	2272	3976
8x8	648	1296	2592	4536

Table 4 Cycles Required to Load the Bernstein Polynomials

4.1.3 Calculating Points and Normals

The tessellation computation can begin once the Bernstein polynomials are loaded and the first transfer of control points from texture memory has completed. This stage consumes the largest portion of the total execution time therefore it is important that it proceed efficiently. The groundwork for this phase was laid in Section 3.6, where it was concluded that the tessellation computation cannot proceed efficiently unless the cycles consumed performing calculations on one batch of data is less than the number of cycles required to transfer the next batch of data. Also, reducing the number of cycles required by the TASICs to transfer a batch of data necessarily increases (slightly) the number of cycles required by the EMCs to distribute that data. The rest of this section explains the details of these trade-offs and a possible balance between them. The results presented assume the use of 4-byte floating point values; the analysis will differ with other representations.

The first step is to analyze how many EMC cycles it takes to perform the calculations for one transfer of data from texture memory. Each transfer consists of 32 bytes, 8 floating point values. These 8 numbers constitute the first two columns of control points for a bicu-

bic patch but only the first column of an 8×8 control patch. For this reason, the analysis varies and a custom implementation is necessary for each size control patch.

The goal, regardless of the size of the control patch, is to maximize the number of computations performed given the amount of data transferred. This is best demonstrated by a specific example; the general results are summarized in Table 5. Generating a surface point for a 5×5 control patch requires two matrix multiplications, given by:

$$\text{Result} = \begin{bmatrix} B_0(v) & B_1(v) & B_2(v) & B_3(v) & B_4(v) \end{bmatrix} \begin{bmatrix} P_{0,0} & P_{0,1} & P_{0,2} & P_{0,3} & P_{0,4} \\ P_{1,0} & P_{1,1} & P_{1,2} & P_{1,3} & P_{1,4} \\ P_{2,0} & P_{2,1} & P_{2,2} & P_{2,3} & P_{2,4} \\ P_{3,0} & P_{3,1} & P_{3,2} & P_{3,3} & P_{3,4} \\ P_{4,0} & P_{4,1} & P_{4,2} & P_{4,3} & P_{4,4} \end{bmatrix} \begin{bmatrix} B_0(u) \\ B_1(u) \\ B_2(u) \\ B_3(u) \\ B_4(u) \end{bmatrix}$$

However, one transfer from texture memory sends just 8 of the 25 control points. At this point, the matrices look like:

$$\begin{bmatrix} B_0(v) & B_1(v) & B_2(v) & B_3(v) & B_4(v) \end{bmatrix} \begin{bmatrix} P_{0,0} & P_{0,1} & . & . & . \\ P_{1,0} & P_{1,1} & . & . & . \\ P_{2,0} & P_{2,1} & . & . & . \\ P_{3,0} & . & . & . & . \\ P_{4,0} & . & . & . & . \end{bmatrix} \begin{bmatrix} B_0(u) \\ B_1(u) \\ B_2(u) \\ B_3(u) \\ B_4(u) \end{bmatrix}$$

Even though most of the control patch matrix is not yet available, it is still possible to carry out part of the matrix multiplications. In particular, the left multiplication for the first column and the right multiplication for the first element can be done as well as part of the left multiplication for the second column of control points. The first partial multiplication looks like:

$$\text{Partial} = \begin{bmatrix} B_0(v) & B_1(v) & B_2(v) & B_3(v) & B_4(v) \end{bmatrix} \begin{bmatrix} P_{0,0} \\ P_{1,0} \\ P_{2,0} \\ P_{3,0} \\ P_{4,0} \end{bmatrix} \begin{bmatrix} B_0(u) \end{bmatrix}$$

while the second looks like:

$$\text{Partial} = \begin{bmatrix} B_0(v) & B_1(v) & B_2(v) \end{bmatrix} \begin{bmatrix} P_{0,1} \\ P_{1,1} \\ P_{2,1} \end{bmatrix}$$

Now that we have determined the number of EMC cycles required to process one batch of data, it is possible to try to find a balance between EMC and TASIC cycles for loading the data. Remember from Section 3.5 that it is possible for the TASICs to read data from the SDRAMs, buffer the results, and send them to multiple PEs in an EMC. The data is then distributed to the remaining PEs within a block by the inter-PE communication paths. The total number of cycles required for one transfer to the EMCs depends on how many times the data for one block are sent from the TASICs to the EMCs and how many EMC cycles are spent distributing the data to the remaining PEs within a block. Table 6 summarizes the results.

The values in Tables 5 and 6 agree well if the TASICs send the control points four times per block of PEs. With this convention, the EMCs never wait for the TASICs and only spend about 3% of their time distributing data to the PEs which were not accessed directly by the TASICs.

Size of Column	Multiplies	Adds	Cycles
n = 3	10	6	4870
n = 4	10	7	5260
4 < n < 8	9	6	4617
n = 8	9	7	5144
8 < n	8	7	4754

Table 5 Minimum Number of Operations to Process One Data Transfer
This table shows the minimum number of cycles required to process one data transfer given *n*, the size of the control patch.

Sends Per Block	EMC Cycles	TASIC Cycles
16	0	9,728
8	66	5,632
4	164	3,584
2	326	2,560
1	616	2,048

Table 6 Cycles Required for One Transfer of Data to the EMCs

I implemented a custom TASIC instruction, called `TAS_MemRdRepeat4`, to meet the specification given above and in Section 3.5. The instruction works by reading a set of texture memory addresses from the EMCs. The addresses are used to read a 4-byte data item from each SDRAM, which are buffered within the TASICs. The data are sent to EMCs four times (for four PEs working on the same patch) before the process begins again. The operation continues until all of the requests have been satisfied.

The discussion above establishes that the tessellation process can proceed without significant stalls in computation. The next step is to generate estimated running times based on the control patch size. The matrix operations to find one point on a Bézier surface from the control patch can be done in $mn + n$ multiplies and $n(m-1) + n - 1$ additions. The derivatives take an additional $2nm - m + n - 1$ multiplies and $2nm - n - m - 2$ additions. Finally, the cross product to generate the normals from the derivatives takes 9 multiplies, 5 additions, 1 square root and 3 divides. The results for various patch sizes are shown in Table 7.

Control Patch Size	Total Cycle Count
4x4	89,036
5x5	139,538
6x6	201,614
7x7	275,264
8x8	360,488
9x9	457,286
10x10	565,658
11x11	685,604
12x12	817,124

Table 7 Total EMC Cycles Used in the Tessellation Computation

4.1.4 Storing Results

Once the points and normals have been calculated, they are ready to be written to the GPs. For single precision values, each PE writes 24 bytes which makes the total cycle count approximately:

$$\frac{24 \text{ bytes}}{\text{PE}} \times 8192 \text{ PEs} \div \frac{200 \text{ Mbytes}}{\text{sec}} \times 100\text{MHz} = 98\text{K cycles}$$

This equals the number of cycles required to tessellate bicubic patches, representing an additional 100% performance cost.

There are two ways to avoid or at least reduce this cost. The first is by utilizing coarse-grain pipelining: the data are transferred while the EMCs scan convert a batch of primitives already resident in GP memory. The straightforward implementation only yields an improvement for the last of a series of tessellation operations on the EMCs. However, a more aggressive approach could interleave part of the rendering for the current frame with tessellation for the next frame. The process would proceed as follows: perform one tessellation step then transfer the results back while performing a rendering step, and so on.

The second method utilizes fine-grain pipelining: the data are sent back to the GPs as they become available. That is, the X coordinate is sent back while generating the Y coordinate, the Y coordinate is sent back while generating the Z coordinate, *etc*. This is possible because the method of transferring the control points from texture memory to the EMCs, discussed above, underutilizes the communication bandwidth from the EMCs to the TASICs. Remember, the TASICs send data to the EMCs on every clock cycle whereas the EMCs only send addresses to the TASICs once for (up to) every 16 data items sent the other way. In the particular case of the custom `TAS_MemRdRepeat4` instruction where data are sent for every four addresses, this leaves $(1/4) \times 800$ MBytes/second, a good match for the 200 MBytes/second path back to the GPs. Of course, a new instruction would have to replace the `TAS_MemRdRepeat4` instruction to route the data to the appropriate destinations.

Note that overlapping the data write-back with computation becomes infeasible if the data to be sent back are not available until the end of the computation (*i.e.* in the projection/

IGC parameter case). Under these circumstances, an application could benefit by using the coarse-grain pipeline approach.

4.2 PE Memory Usage

The amount of PE memory needed to perform the tessellation computation, based on the control patch size, is shown in Table 8; a detailed memory map for the bicubic case is shown in Table 9. The 4×4 and 8×8 case are based on actual implementations, the others show a worst-case analysis based on the results of the implemented cases. There are 320 bytes of PE memory available. If an actual implementation of the 12×12 case exceeds 320 bytes, the sequence of the computation could be altered to load the Bernstein polynomials for the surface point, perform the computation, then load the polynomials for the derivatives. With this minor modification, there should be no memory problems through 12×12 control patches using a 4-byte floating point representation.

Control Patch Size	Bytes of Memory Needed
4x4	166
5x5	198
6x6	218
7x7	238
8x8	258
9x9	278
10x10	298
11x11	318
12x12	338

Table 8 PE Memory Usage During Tessellation

Address	Usage
0	X: Coordinate, U Deriv, V Deriv, Temp
16	Y: Coordinate, U Deriv, V Deriv, Temp
32	Z: Coordinate, U Deriv, V Deriv, Temp
48	Bernstein coefficients for point in U
64	Bernstein coefficients for derivative in U
76	Bernstein coefficients for point in V
92	Bernstein coefficients for derivative in V
104	Control patch (after local port)
140	Administrative (patch number, etc.)
144	Temporaries (partial results, *etc.*)
152	Scratch space
166	Free

Table 9 Detailed PE Memory Usage for a Bicubic Control Patch

4.3 Texture Memory Allocation

As discussed in Section 2.2, each module's, physical texture memory consists of 8 SDRAMs, each with 2 banks consisting of 2048 rows with 512 columns. It is conceptually difficult to use this physical address space for allocating, storing and retrieving textures. Therefore, the applications which deal with textures access them with *logical* texture addresses which are translated by low level routines into actual physical texture addresses, as illustrated in Figure 10. In *logical texture space*, a texture row and column[1] index are used to specify a texture element; the logical to physical mapping translates this address through a one-to-one mapping into a corresponding physical address (for details see [Molnar 95]). Although this logical space works well for storing textures, it is inconvenient for accessing control patches. For this reason another level, referred to as *logical linear space*, is layered on top of the logical texture mapping.

There are two driving factors which determine the mapping from logical linear space to the physical address space. The first is a hard requirement: the addresses must support the types of memory accesses that are required by the tessellation operation. The second requirement is softer: it is advantageous if the linear address space works "well" with log-

1. A texture row and column does not necessarily correspond to an SDRAM row and column.

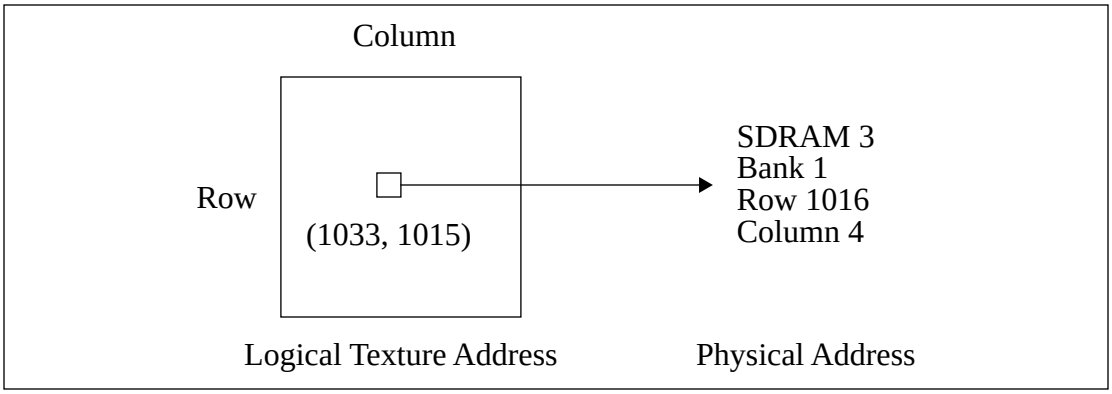


Figure 10 Logical Texture Space

ical texture space. In this case, “well” means that the footprint of linear space in texture memory is square. For example, if four consecutive addresses in linear space corresponded to the texture addresses (512, 512), (1536, 512), (512, 1536) and (1536, 1536), the largest remaining contiguous block of logical texture memory in one bank would be just 512×512 ! A square footprint prevents this fragmentation and leaves greater flexibility for the unused texture memory to be allocated for storing textures - perhaps for radiosity algorithms or even as a frame buffer.

Figure 11 shows how the three spaces are related. The bottom row of numbers shows the bits of the physical address. The line above this shows how the row and column bits for a logical texture correspond to a physical address. Finally, the logical linear space is depicted on top.

Not all of the bits in the logical address spaces map to physical addresses. In particular, notice that logical r0, c0 and the corresponding bits a2 through a4 do not appear in the mapping. These bits decode SDRAMs 0 through 7. For example, the word corresponding to logical address 0 is in SDRAM 0 at physical row 0 and column 0. The word corresponding to logical address 4 is at the same memory location in SDRAM 1, the word at logical address 8 is in SDRAM 2, *etc*. Also, physical c0 and c1 are not represented in logical texture space since the smallest division in logical texture space is implicitly a 4-byte texel.

Finally, note that logical linear address space grows equally fast in the rows and columns of logical texture space which keeps the foot print square, as desired.

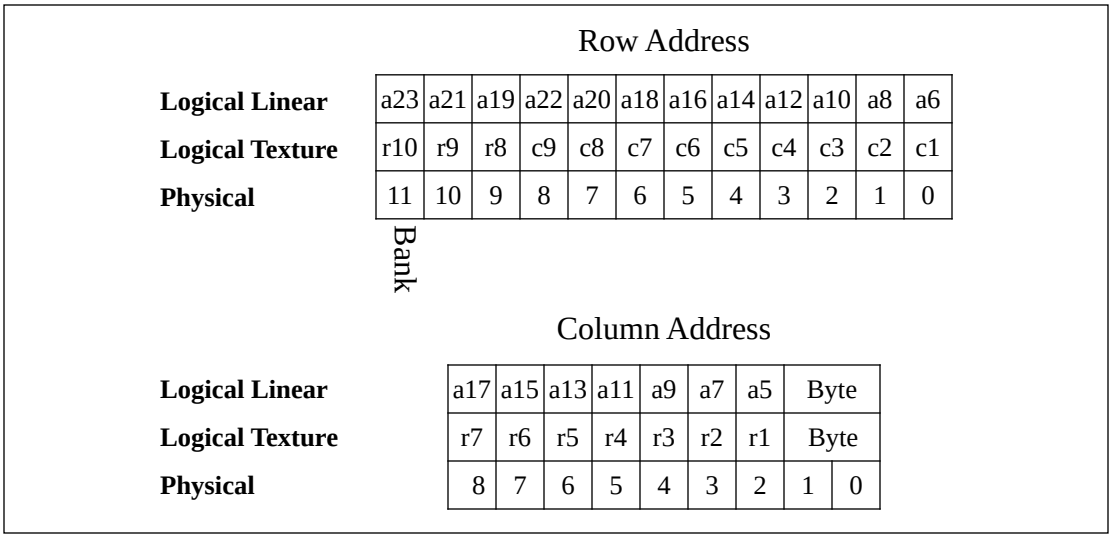


Figure 11 Logical to Physical Address Mapping

The mapping in Figure 11 is sufficient for the standard tessellation approach. However, as mentioned in Appendix B, if the points and normals in world space are to be cached in texture memory, then consecutive writes must access alternate banks of the SDRAMs. This requires a different mapping, where linear address bit 6 must map to the bank select bit. The modified mapping is shown in Figure 11.

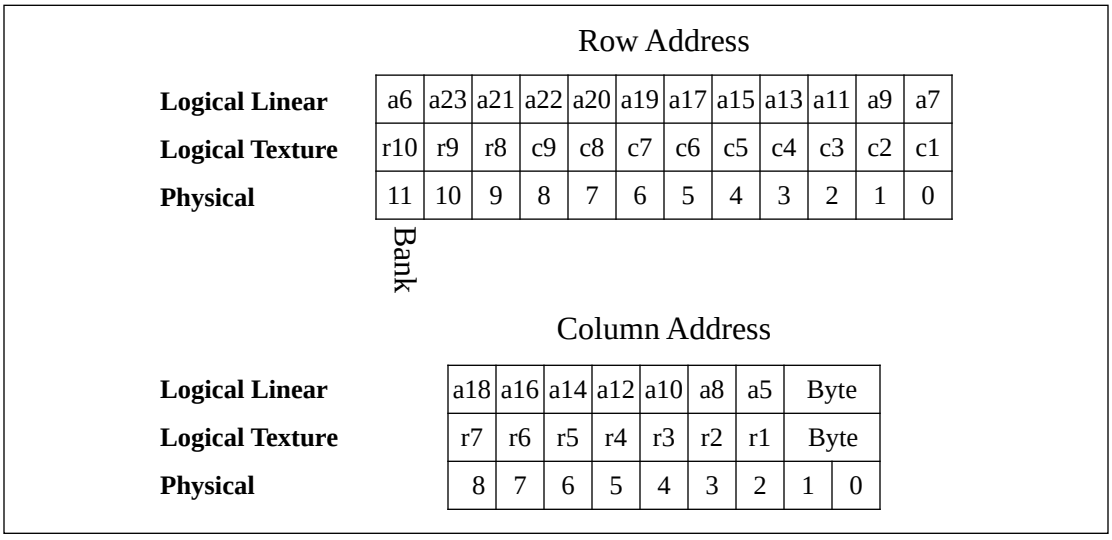


Figure 12 Logical to Physical Address Mapping With Alternating Bank Access

4.4 Application Interface

This section is included to provide an idea of how an application uses the tessellation algorithm described above. The application interacts with the library through two *C++* classes whose member functions are highlighted in Table 10. The naming conventions prefix the class names with `BSP_` for "Bézier Surface Patch" and append `_C` for "class". `BSP_MGR_C` is created by passing a pointer to an array which contains the maximum number of patches that will be stored for each degree. This routine allocates storage and initializes the surface patch manager's internal state.

Surface patches are created by instantiating objects of type `BSP_C`. The tessellation manager, the number of points in the control polygon, and a pointer to the control polygon are passed as arguments to the constructor. The constructor assigns a module and a logical texture address to the patch and then loads the control patch into texture memory (the loading step is not currently implemented). Patches are marked for tessellation by calling the member function `mark` with the desired sample resolution.

Once all patches which need to be tessellated have been marked, the tessellation process begins by calling the `tessellate` member function of the tessellation manager.

Class	Method	Parameters	Description
`BSP_MGR_C`	`Constructor`	Array of the maximum number of patches of each degree	Creates storage for all control patch sizes
	`Tessellate`	None	Tessellates all patches associated with this manager
`BSP_C`	`Constructor`	The manager, size of control patch and array of control points	Assigns a logical texture memory address and storage for a new patch
	`Mark`	Resolution at which to tessellate	Marks the current patch for tessellation

Table 10 Tessellation API

4.5 Conclusion

This chapter augmented the ideas in Chapter 3 with the concrete details provided by an actual implementation. The tessellation steps described in the conclusion of the previous chapter has been updated with synchronization information and cycle counts and is shown in Table 11. Fine-grain pipelining can be used to write the surface point to the GPs while

generating the surface normal. However, since the normal depends on the derivative in all three dimensions, the normal is not ready to be sent back until the end of the computation. Therefore, the remaining 45K cycles must be absorbed through some other coarse pipelining approach, perhaps by overlapping the write with the start of the rendering phase.

Finally, note that the cycle counts given in the table are approximate. In an actual implementation, the number of cycles required to process each batch of data varies. For example, the number of operations which can be performed on the first two columns of control points differs from the number of computations which can be performed for the last two columns of control points, as described in Section 4.1.3.

EMC Operations	From TASICs to EMCs	From EMCs to TASICs	Cycles
Load Base Addresses			6K
Load Bernstein Coefficients	1st Half of X Control Patch		2K
1st Half of X Control Patch	2nd Half of X Control Patch		5K
2nd Half of X Control Patch	1st Half of Y Control Patch		5K
1st Half of Y Control Patch	2nd Half of Y Control Patch	Write Module 0's X	5K
2nd Half of Y Control Patch	1st Half of Z Control Patch	Write Module 1's X	5K
1st Half of Z Control Patch	2nd Half of Z Control Patch	Write Module 2's X	5K
2nd Half of Z Control Patch	1st Part of X' Control Patch	Write Module 3's X	5K
1st Part of X' Control Patch	2nd Part of X' Control Patch	Write Module 0's Y	5K
2nd Part of X' Control Patch	3rd Part of X' Control Patch	Write Module 1's Y	5K
3rd Part of X' Control Patch	1st Part of Y' Control Patch	Write Module 2's Y	5K
1st Part of Y' Control Patch	2nd Part of Y' Control Patch	Write Module 3's Y	5K
2nd Part of Y' Control Patch	3rd Part of Y' Control Patch	Write Module 0's Z	5K
3rd Part of Y' Control Patch	1st Part of Z' Control Patch	Write Module 1's Z	5K
1st Part of Z' Control Patch	2nd Part of Z' Control Patch	Write Module 2's Z	5K
2nd Part of Z' Control Patch	3rd Part of Z' Control Patch	Write Module 3's Z	5K
3rd Part of Z' Control Patch			5K
Cross Product for Normals			7K
		Write Normals	45K

Table 11 Tessellation Steps for a Bicubic Bézier Patch

5.0 Results

This thesis presents the results of an analysis of the problem of how to tessellate Bézier surface patches using the rasterization engine present in each node of a PixelFlow graphics computer. The major issues of how to solve this problem were highlighted in Chapter 3 and the implementation details were explored in Chapter 4. The contributions of my work are summarized below:

1. I devised a paradigm for mapping the surface patches to the processing elements. Each PE generates one surface point for one surface patch.
2. I created a custom LEE configuration to support the surface patch to PE mapping by allowing the Linear Expression Evaluator to efficiently compute the values needed by the tessellation operation.
3. I microcoded a TASIC instruction customized to provide high bandwidth for the task of reading Bézier control patches from texture memory and sending them to the EMCs.
4. I created a logical to physical texture memory map for storing Bézier patches in texture memory. Rapid access from texture memory and the coexistence of control patches with texture images requires that surface patch data be distributed through physical texture memory in a particular fashion. This requires a coherent mapping from a linear address space to physical texture memory which works well with stored texture images.
5. I implemented several *C++* classes for PixelFlow surface patch tessellation. The classes contain function calls to register surface patches, mark them for tessellation at a particular resolution, and to perform the actual tessellation. The implementation supports 4×4 and 8×8 control patches (others may be added as future work). 4×4, 8×8 and 16×16 samples per patch may be generated.

There are two omissions in the above list, which were necessitated by the lack of physical hardware and adequate simulation support. These are: loading the control patches into texture memory from the GPs and storing the results in the EMCs back to GP memory. The implementation details of these pieces are left for future work.

Table 7 provides a summary of the results of tessellating control patches of various degrees. The values for 4×4 and 8×8 control patches have been verified in simulation, the others have been extrapolated from these results. The procedure always produces 8K samples (1 per PE), the duration is not affected by the number of patches which are sampled. Note that the cycle counts listed are EMC cycles only and do not include the time required to write the results back to the GPs. This is a legitimate omission since returning the results may be overlapped with other useful computation on the EMCs.

Figure 13 shows the time required for tessellating a bicubic patch. Note that the tessellation process is computation bound; that is, all pipeline stalls occur in the transfer of data from texture memory to the EMCs. This means that the PEs are constantly busy doing useful work, one of the design goals. Again, ignoring the time required to write the results back to the GPs gives the following expected performance statistics:

1. 97,000 cycles (.97 milliseconds) from initialization to completion of the computation
2. 282 floating point operations per sample
3. 2.3 million floating point operations total
4. 2.4 billion floating point operations per second
5. 530K 4×4 patches per second, 330K 16×16 patches per second
6. ~93% efficiency, compared to peak floating point performance

On PixelFlow, the alternative to tessellating the surface patches in parallel on the EMCs is to use a conventional serial approach on the GPs. Assuming 100% efficiency, the peak theoretical performance is 240 MFlops per processor. The parallel implementation provides a performance gain of a factor of 10 over the theoretical peak performance of conventional approaches.

While this thesis answered most of the questions of how to efficiently tessellate surface patches using PixelFlow hardware, some questions remain unanswered. Of these questions, only one (writing data from the EMCs to the GPs), needs to be addressed for a first

complete implementation. The remaining topics extend and enhance the work that has been done so far. These open issues are highlighted below.

Control Patch Size	Total Cycle Count	Execution Time (mSecs)
4x4	97,000	.97
5x5	148,000	1.5
6x6	209,000	2.1
7x7	283,000	2.8
8x8	367,000	3.7
9x9	465,000	4.7
10x10	574,000	5.7
11x11	694,000	6.9
12x12	825,000	8.3

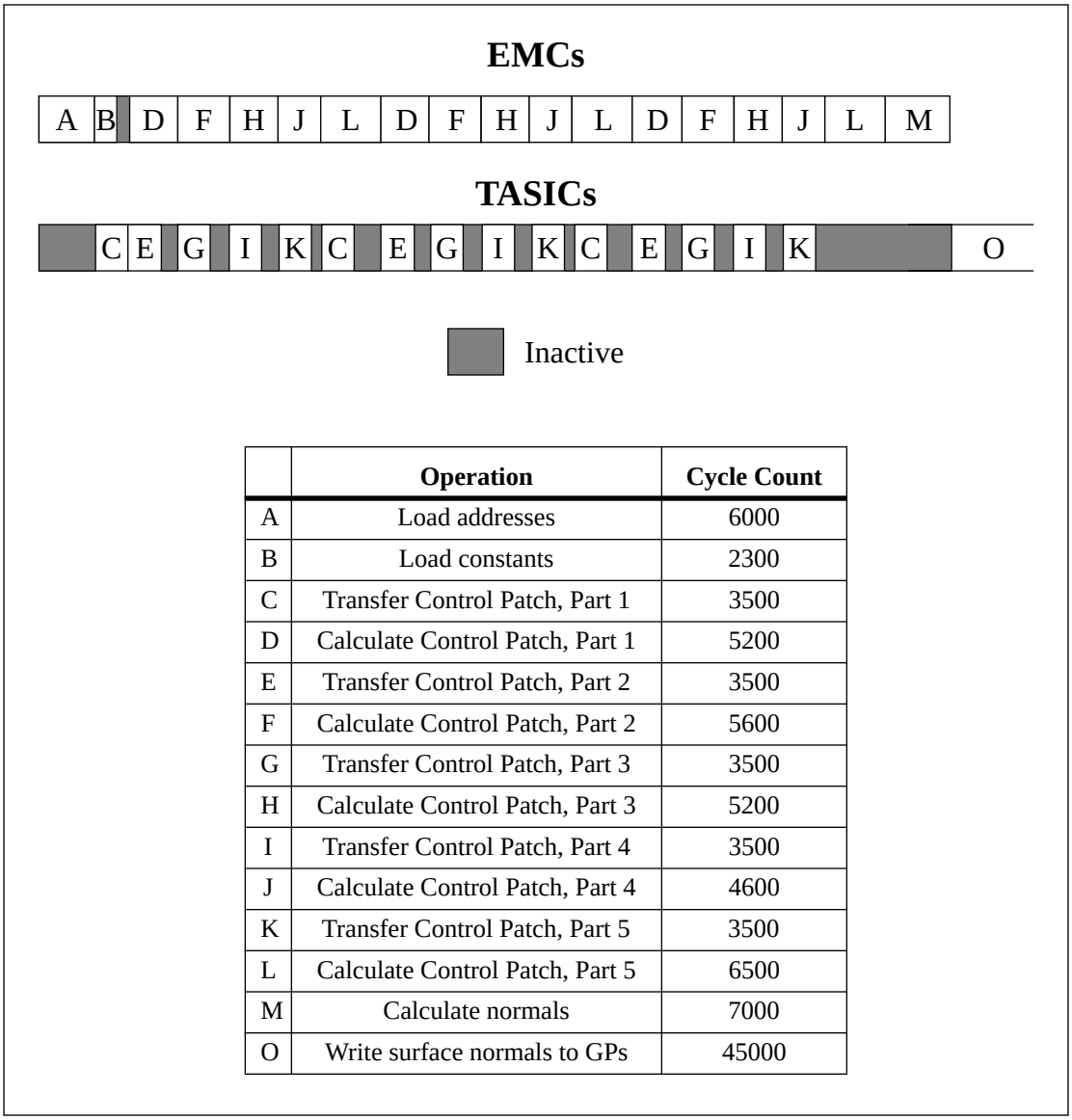
Table 12 Tessellation Performance

5.1 Future Work

Most of the future work involves extending or augmenting the concepts presented in this thesis. However, I do not believe that the first step should be to embark upon new territory but instead should be to incorporate the current implementation into an actual system. The benefits provided by tessellating surface patches using the EMCs are clear: 2 billion floating point operations per second. However, the impact of the limitations is not as immediately obvious. For this reason, before the work is extended, it must be ascertained as to whether or not the benefits provided by tessellating surface patches on the EMCs can be utilized by a real world application.

As has been mentioned before, the data path between the GPs and the TASICs is not implemented in the PixelFlow simulators. Therefore, Section 4.1.4 could only address the issue in a theoretical sense. The algorithm presented in this document will not be usable on an actual machine without implementing this stage of the process.

The discussion throughout this document has relied on the use of 4-byte single-precision floating point values. This is not a requirement; the PEs can operate on other floating point representations. A theoretical discussion of the impact of other representations on the



	Operation	Cycle Count
A	Load addresses	6000
B	Load constants	2300
C	Transfer Control Patch, Part 1	3500
D	Calculate Control Patch, Part 1	5200
E	Transfer Control Patch, Part 2	3500
F	Calculate Control Patch, Part 2	5600
G	Transfer Control Patch, Part 3	3500
H	Calculate Control Patch, Part 3	5200
I	Transfer Control Patch, Part 4	3500
J	Calculate Control Patch, Part 4	4600
K	Transfer Control Patch, Part 5	3500
L	Calculate Control Patch, Part 5	6500
M	Calculate normals	7000
O	Write surface normals to GPs	45000

Figure 13 Stages of Tessellation for a Bicubic Patch

algorithm is provided in Appendix A. Details and implementation of these ideas are left as future work.

Section 3.7 briefly described some of the benefits and pitfalls of carrying the tessellation computation on the EMCs through the succeeding steps in the rendering process: projection and generation of rendering instructions. It was mentioned that the details are highly application dependent. Appendix B provides some of the groundwork and supporting concepts necessary for implementing these stages, leaving the actual implementation as future work.

Many data sets contain trimmed Bézier surface patches. The use of these primitives introduces many complications which may or may not be solvable in a parallel implementation on this machine. While this thesis laid the groundwork for how to approach the task, there are significant hurdles that must still be resolved.

Finally, it is possible to generate surface patches that are sampled with other than 2^n samples. The solution with the greatest probability of success uses the same patch to PE mapping given in Section 3.2, but lets the unused processors in a block sit idle. For example, a 3×3 tessellation would not use 4 of the processors in the 16 processor block. A more complicated approach would be to microcode additional texture instructions to handle the boundary conditions that arise with non-2^n patch dimensions. This may sacrifice performance and is undoubtedly more complicated; the details are left as future work.

5.2 Generalization to Other Problems

This document focused on the tessellation of Bézier surface patches using rasterization hardware. The solution to this particular problem provides insight into the general problem of utilizing the rasterization engine's processing power. In particular, problems with the following attributes are most likely to benefit from an implementation using PixelFlow's rendering engine:

1. The number of cycles required for computing the results for one data transfer matches or is less than the duration of a data transfer.
2. Multiple processors share the same (or partially the same) data set.
3. All processors perform a (nearly) identical sequence of calculations.
4. There is a limited amount of data dependent conditional computation.
5. There is a limited need to share partial results with other processors.
6. Each processor generates a "small" quantity of data.

The first requirement ensures that computation does not stall for data to become available; the second criterion provides some flexibility in meeting this requirement. The next two criteria state the SIMD requirement: for high processor utilization, all processors must

perform the same computation. Item 5 addresses the issue of inter-PE communication, that is, only some processors can communicate with each other rapidly. Finally, the last point states that the algorithm cannot generate a plethora of data since there is only limited communication bandwidth from the processor array to the geometry processors. Problems which meet these criteria are likely to benefit from an implementation using PixelFlow's rasterization hardware.

5.3 Conclusion

The PixelFlow rendering engine contains an 8K element SIMD processor array. Although the array is optimized for scan converting polygons, it is capable of performing general purpose computation. The results described above showed that, with some restrictions, the hardware can tessellate Bézier patches with realized performance of nearly 2.5 billion floating point operations per second. Current serial microprocessors provide peak floating point performance of about 250 million operations per second. This represents a performance gain of about an order of magnitude over conventional approaches. Thus, as postulated in Chapter 1 of this document, the rasterization hardware present in each node of the PixelFlow graphics computer can accelerate the tessellation of surface patches by approximately an order of magnitude over conventional approaches. Also, it is possible that the results of the future work could extend the performance advantage to other parts of the rendering process. Integrating these ideas into applications which render Bézier surface patches in real time should vastly improve the quality and frame rate of those applications.

Appendix A Floating Point Representation

Throughout this document the use of 4-byte single-precision floating point values was assumed; however, the ideas developed in the preceding chapters may be extended to work with higher-precision representations. Since PixelFlow emulates floating point operations with sequences of integer instructions, the floating point representations need not be restricted to IEEE floating point standards. In particular, intermediate representations with, perhaps, 4-byte mantissas can be used.

Performing the computation in higher precision impacts two parts of the algorithm. First, higher precision requires more calculations which increases the number of EMC cycles required. Second, the number of TASIC cycles increases, since more data must be transferred from texture memory. Table 13 gives the expected EMC performance with these representations. Note that the entry for a 3-byte mantissa is based on an actual implementation while the others are extrapolated from that implementation. Also, the impact of extra bytes of precision is more pronounced for multiplication operations than for addition operations.

A compromise which does not impact the number of TASIC cycles required utilizes separate representations for different parts of the computation. For example, the database would be stored as single-precision floating point values as in the current implementation. The left part of the matrix multiplication would then be performed by multiplying the single-precision values and generating a double-precision result. Since the penalty for increased precision is small for floating point additions, these operations are carried out in the higher-precision, as are the comparatively few multiplies for the right hand matrix multiplication. Finally, the result is rounded down to single precision, which limits the amount of data which must be sent to the GPs.

A complete evaluation of the trade-offs between a higher-precision representation or a hybrid approach as well as the modifications to the algorithm to support these changes is

left as future work.

Mantissa Size	Multiply Cycles	Addition Cycles
3	250	400
4	325	440
5	400	480
6	525	520

Table 13 Approximate Cycle Counts for Higher-Precision Floating Point Operations

Appendix B Caching Intermediate Results

The tessellation process produces surface points and normals in world (or possibly model) coordinates. The next step in the rendering process is to project these points into image space. Since the viewing transform generally varies from frame to frame, it is desirable to cache the intermediate values in world space to prevent unnecessary re-tessellation at each frame [Kumar 94]. Texture memory is the best choice of the possible locations to cache the results because of its high data bandwidth.

Each PE contains six 4-byte data items, assuming floating point representation of the point and normal. The data can be transferred between PEs within a block so that 12 PEs in each block contain eight data items, one for each SDRAM. As described in Section 2.2.7, the EMCs will be accessed in round robin order, so there are no guarantees regarding the access order of banks and rows within an SDRAM. Also, for a write, both data and addresses must flow from the EMCs to texture memory. This exceeds the amount of data that can be sent in one pass, so two writes must be performed bringing the total to:

$$\frac{11\ \text{Cycles}}{1\text{Write for 8 SDRAMs}} \times \frac{8\ \text{Writes}}{\text{PE}} \times \frac{192\ \text{PEs}}{\text{EMC}} \times 8\ \text{EMCs} \times 2\ \text{Passes} = 33792\ \text{Cycles}$$

There are several possible improvements to the straightforward method described above. First, it is not necessary for each PE to supply its own texture memory address since consecutive PEs on the same EMC belong to the same surface patch. Instead, it is possible to utilize the TASICs' ability to store a base address for each EMC and use the address generator to specify an offset from this base address. The write described above sent data from 3 out of 4 PEs. The remaining PE can supply the base address for the 3 PEs containing data; the TASICs take care of incrementing the address pointer after servicing one PE from each EMC. The requirement for two passes is eliminated, halving the time to just under 17,000 cycles.

The cost is now primarily in the low utilization of the SDRAM bandwidth, one write every eleven cycles. The only way to improve this is to guarantee that consecutive writes either lie in the same row or in alternate banks. Figure 14 depicts a plan which ensures the latter condition. The writing is again broken into two passes: during the first pass, even

EMCs write data stored in the even columns of PEs while odd EMCs write data stored in the odd columns of PEs. On the second pass the order is reversed. At completion, all of the data for even and odd EMCs is stored in the SDRAMs in the same fashion; however, by writing to alternate banks the bandwidth is improved significantly. The total number of cycles is now given by:

$$\frac{12 \text{ Cycles}}{2 \text{ Writes for 8 SDRAMs}} \times \frac{8 \text{ Writes}}{\text{PE}} \times \frac{96 \text{ PEs}}{\text{EMC}} \times 8 \text{ EMCs} \times 2 \text{ Passes} = 9216 \text{ Cycles}$$

This approach has several ramifications. First, writing alternately to either bank requires that the surface points for a patch straddle both banks of each SDRAM and also that they occupy the same row and column addresses in each bank. These requirements were handled in the logical to physical texture memory address mapping given in Section 4.3.

Second, the method described above makes the footprint of the surface patch in texture memory larger than necessary. For the address generators in the TASICs to automatically generate n addresses, the base address must lie on a $2^{\text{ceiling}[\log n]}$ boundary. The write, however, supplies one address for 6 PEs worth of data. To generate the necessary 3 addresses per bank, the blocks must lie on a 2^3, or 8 word boundary, which wastes 25% of the storage capacity.

The waste can be eliminated by breaking the 6 items of data into two sets, one of 4 items the other of 2 items. Now a different base address can specified for each set; however, data must be written with an extra two passes, as depicted in Figure 14. There is no bandwidth penalty for the extra passes since the bottleneck is still in accessing the SDRAMs, not sending data to the TASICs, therefore sending the extra addresses is free.

In summary, the intermediate points and normals generated by the tessellation process can be efficiently cached to texture memory. The operation requires about 9000 TASIC cycles, which can be overlapped with computation on the EMCs.

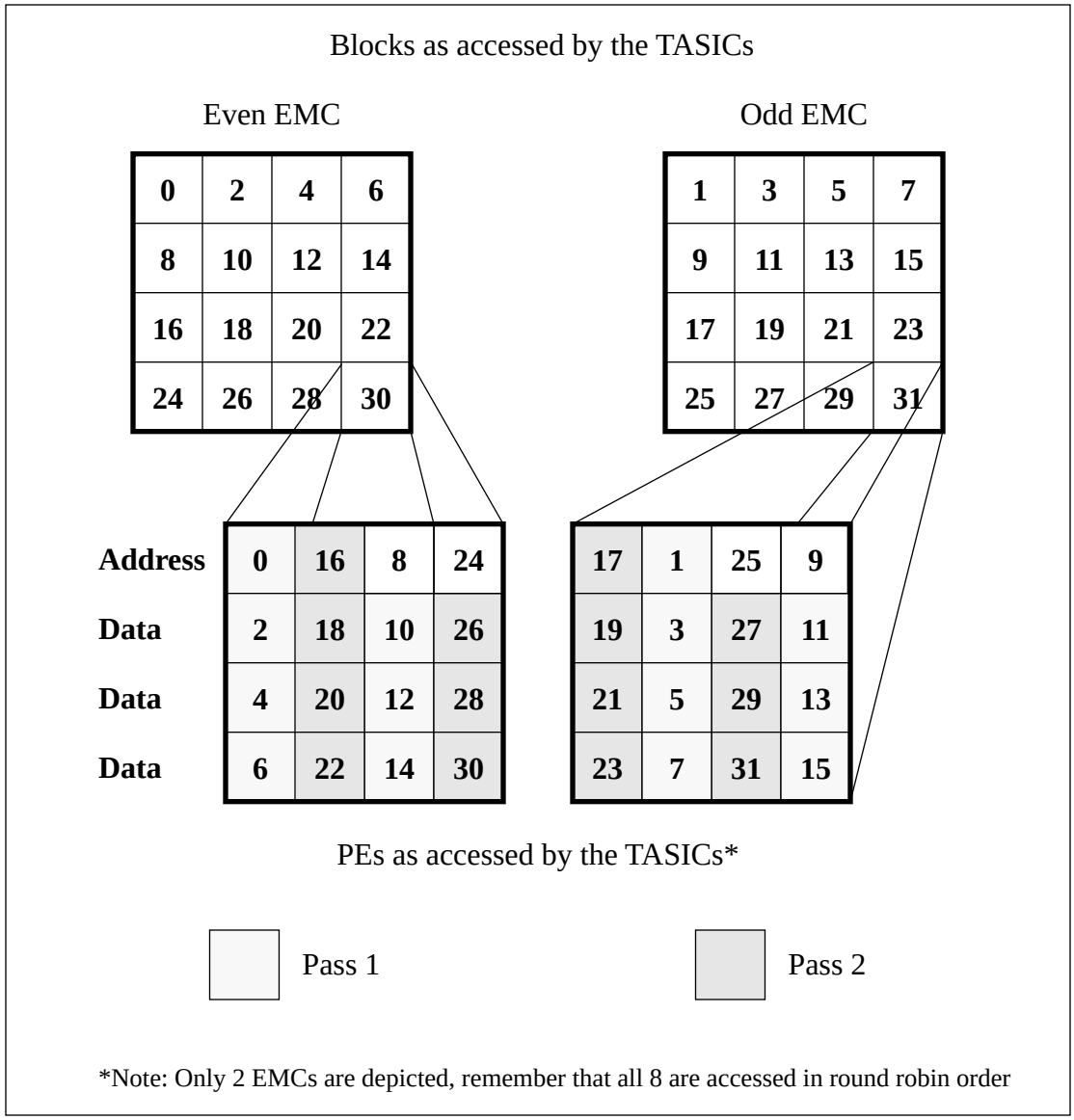


Figure 14 Writing Points and Normals to Texture Memory

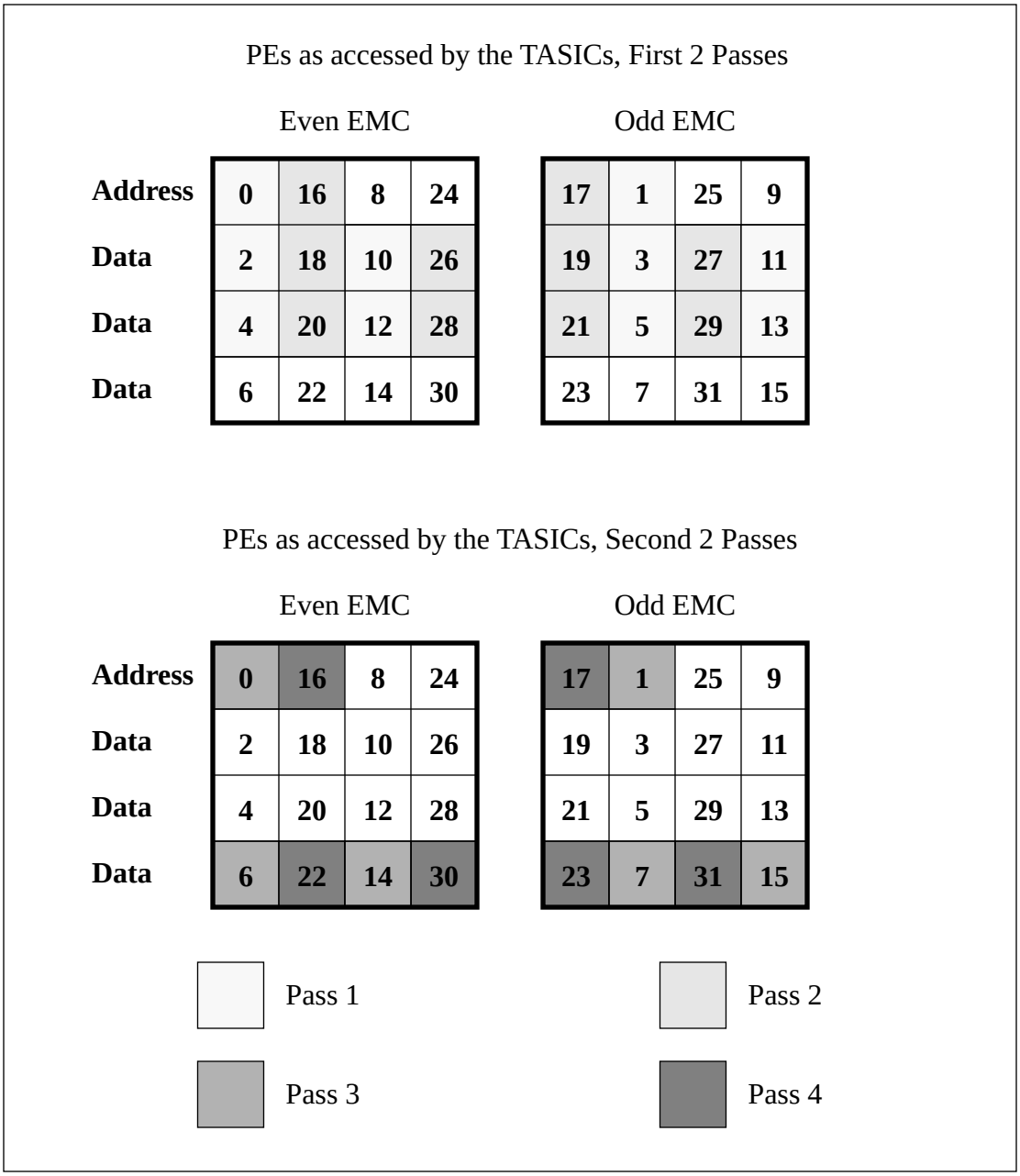


Figure 15 Writing Points and Normals with Efficient Storage

Bibliography

[Eyles 95] John Eyles and Steve Molnar. “PixelFlow Rasterizer Functional Description”, internal University of North Carolina Computer Science Department document, 1995.

[Farin 93] Gerald Farin. Curves and Surfaces for Computer Aided Geometric Design: A Practical Guide, Academic Press Inc., 1990.

[Kumar 94] Subodh Kumar and Dinesh Manocha. “Interactive Display of Large Scale Trimmed NURBS Models”, University of North Carolina Computer Science Department Technical Report #TR94-008, 1994.

[Molnar 92] Steve Molnar, John Eyles, and John Poulton. “PixelFlow: High-Speed Rendering Using Image Composition”, Siggraph '92, pp. 231-240.

[Molnar 95] Steve Molnar, Greg Welch, and Paul Keller. “PixelFlow Texture Organization”, internal University of North Carolina Computer Science document, 1995.