# The Triangle: A Multiprocessor Architecture for Fast Curve and Surface Generation

*Tony D. DeRose*
*Thomas J. Holman*
Department of Computer Science, FR-35
University of Washington
Seattle, WA. 98195

Technical Report 87-08-07
August 1987

*ABSTRACT*

In this paper, we describe the architecture and operation of the Triangle, a pipelined, parallel multiprocessor architecture for the computation and rendering of line segments, conic sections, spline curves, triangular patch surfaces, and tensor product surfaces. Based on a generalization of de Casteljau's algorithm for computing points on Bézier curves, the Triangle can be used as an accelerator to dramatically enhance the performance of standard graphics workstations.

Although there is a wide spectrum of possible implementations, we estimate that a custom VLSI implementation that is currently being designed will be capable of peak rates well in excess of one million points of evaluation per second. Assuming a Motorola 68020 host, such a processor should be capable of completely recomputing 65 bicubic patches in real-time, at a density of 1,000 points per patch.

*KEYWORDS:* Bézier curves, B-splines, computer-aided geometric design, curves and surfaces, graphics hardware.

---

This work was supported in part by the National Science Foundation under grant number DCI-8602141, the Office of Naval Research under grant number N00014-86-K-0264, and the Digital Equipment Corporation.

# 1. Introduction

It is now becoming quite common to design geometric objects with the aid of an interactive computer modeling program. In this type of design, the designer may begin with a mental image of the desired shape, the goal being communication of that shape to the system. The system in turn stores some internal representation of the shape.

For "free-form" shapes such as the outline of a character in a typography system, spline representations have become popular. The design of a spline typically begins by having the designer specify a sequence of controlling points, collectively called a *control polygon*. It is the responsibility of the system to transform the control points into a smoothly varying spline curve. Of course, there are many ways the system could construct such a mapping, two possibilities of which are Bézier and Lagrange curves [5, 6] (see Figure 1).

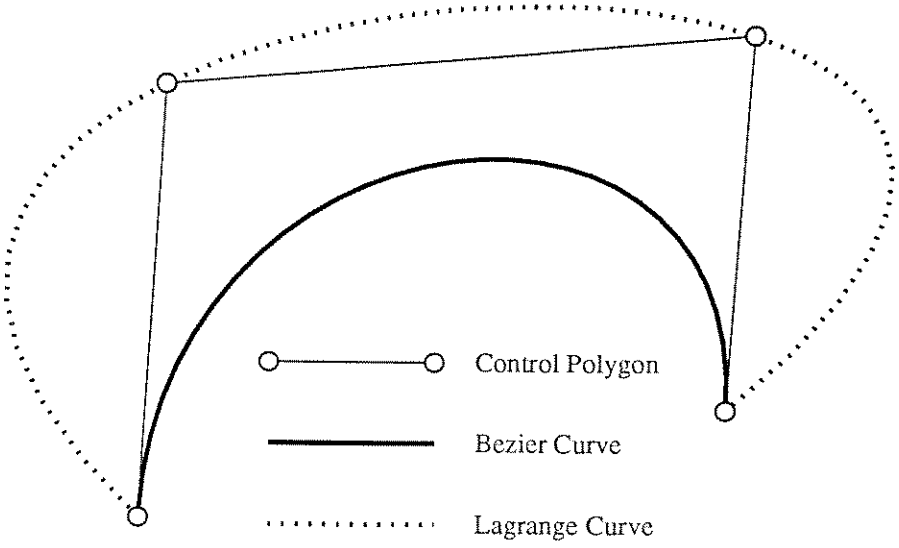



Figure 1. Bézier and Lagrange Curves.

Based primarily on the visual appearance of the curve, the designer may wish to move a control point, thereby changing the shape of the curve. An example of this type of modification is shown in Figure 2. Ideally, the system would dynamically track the pointing device and redraw the curve in real-time, *i.e.*, at least thirty times per second. Even more desirable is the ability to dynamically deform spline surfaces. Unfortunately, dynamic recomputation at these speeds is simply not possible on serial processors.

In this paper, we address the problem of dynamically recomputing points on line segments, conic sections, and spline curves and surfaces at real-time rates. We argue that it is possible to build a simple, practical, single-board multiprocessor — using either off-the-shelf parts or custom VLSI — that meets or exceeds the stated requirements. More specifically,

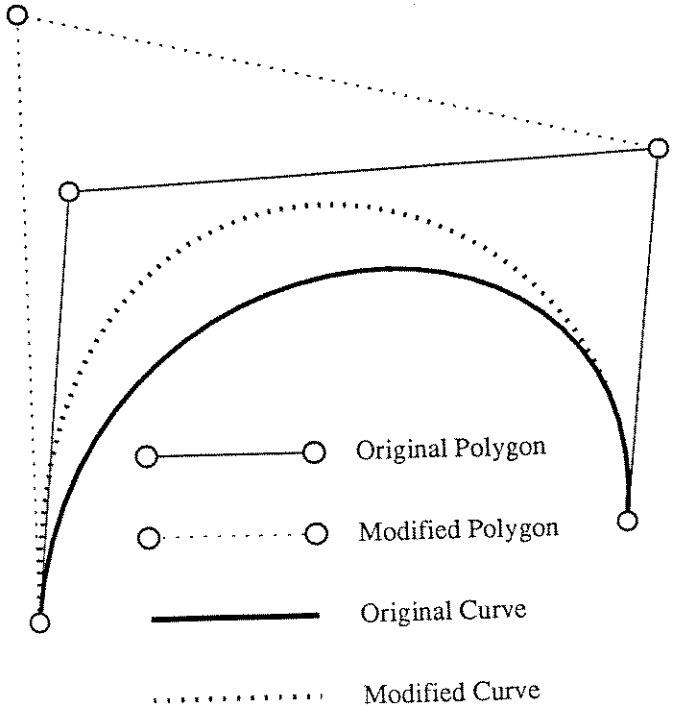



Figure 2. Movement of a Control Point

we show that if the spline representation is in a certain class (a class that contains virtually all the representations in wide-spread use), then it is possible to build a multiprocessor using current technology whose performance is in excess of one million points of evaluation per second.

The architecture we propose, called the Triangle, was motivated by recent work on urn model descriptions of curves [16, 17], which in turn was partially motivated by de Casteljau's algorithm [5, 6] for computing points on Bézier curves and surfaces. By generalizing the algorithm of de Casteljau, points on other types of splines including B-spline and Lagrange curves and surfaces [5, 6] can also be computed at peak throughput. If the spline representation is not in the class directly implemented by the Triangle, conversion to a representation that is implemented can be performed as a preprocessing step. For instance, Beta-spline curves and surfaces [3, 4] cannot be directly implemented, but there exists a relatively simple algorithm for converting the segments of a Beta-spline into cubic Bézier form [7].

Although a prototype has not yet been completed (a fully custom VLSI implementation is currently underway), we envision using a Triangle as a single-board peripheral device in a graphics workstation.* The host initializes the device by configuring it for the curve or surface scheme of interest (Bézier, B-spline, Lagrange, etc.), then supplies the control points for the particular curve or surface to be generated. A short time later points lying on the

* Nonetheless, when we speak of the operation of a Triangle it will be in the present tense.

curve or surface begin to emerge from the Triangle.

We imagine the Triangle being used in surface design applications as follows. Initially, the designer is presented with a smooth-shaded representation of the surface being designed. He is then is allowed to modify the surface by "grabbing" a control point, typically by having the position of the point attached to a pointing device. In response to movement of the pointing device (and hence, movement of the control point), the host uses the Triangle to compute a stream of points that lie on the surface. Visually, the affected portion of the surface changes from a smooth-shaded appearance to a dense arrangement of dots that pepper the surface. The Triangle is used to dynamically update the "dot-representation" while the designer is moving the control point. Since the cycle time of the Triangle is very fast, a dense dot-representation can be generated in a short period of time, fast enough in fact to present the designer with the impression of a smoothly deformable surface. When the designer releases the control point, the host is free to invoke more complicated surface rendering algorithms to redisplay the modified surface in smooth-shaded form.

One potential problem with the above design recipe is that two-dimensional projections of dot-representations of surfaces can be confusing to the viewer. We plan to minimize the confusion by using stereo-graphic imagining of the surface, giving the user the illusion of a true three-dimensional display. This type of three-dimensional stereo-graphic display of dot-representations of complex surfaces has been used effectively for some time in molecular modeling applications, so it is reasonable to expect that it will also work well in geometric design. The only complication introduced by stereo-graphic viewing is that points of evaluation must be passed through two perspective transformations — one for the left eye image and one for the right eye image — before being written into display memory. These transformations can be implemented in hardware using several LSI chips or a small amount of custom VLSI (see Section 8).

The presentation is structured as follows: in Section 2, we briefly introduce the theory upon which the architecture is based; in Section 3, we develop the basic architecture, provide performance estimates for several possible implementations, and show how the Triangle can be used as a powerful accelerator to enhance standard graphics workstations; in Sections 4, and 5, we show how the basic architecture can be used as a building block in the construction of multiprocessors for generating conic sections, rational polynomial curves, and tensor product surfaces; in Section 6, we show that a variant of the Triangle can be used to generate triangular Bézier surface patches; in Section 7, the Triangle is compared to two other architectures for generating polynomial curves, one based on forward differencing and one based on the subdivision algorithm for Bézier curves; finally, in Section 8, implementation details in various technologies are compared and contrasted.

## 2. Theoretical Foundations

The Triangle is based on the theory of certain discrete probability distributions known as *urn models*. The application of urn models to computer-aided geometric design is currently under development, primarily by Goldman and Barry [1, 2, 15, 16, 17]. Rather than

introducing the theory in the context of urn models,† in this section we present an equivalent characterization based on ideas that more closely embrace the issues involved in defining and implementing the Triangle.

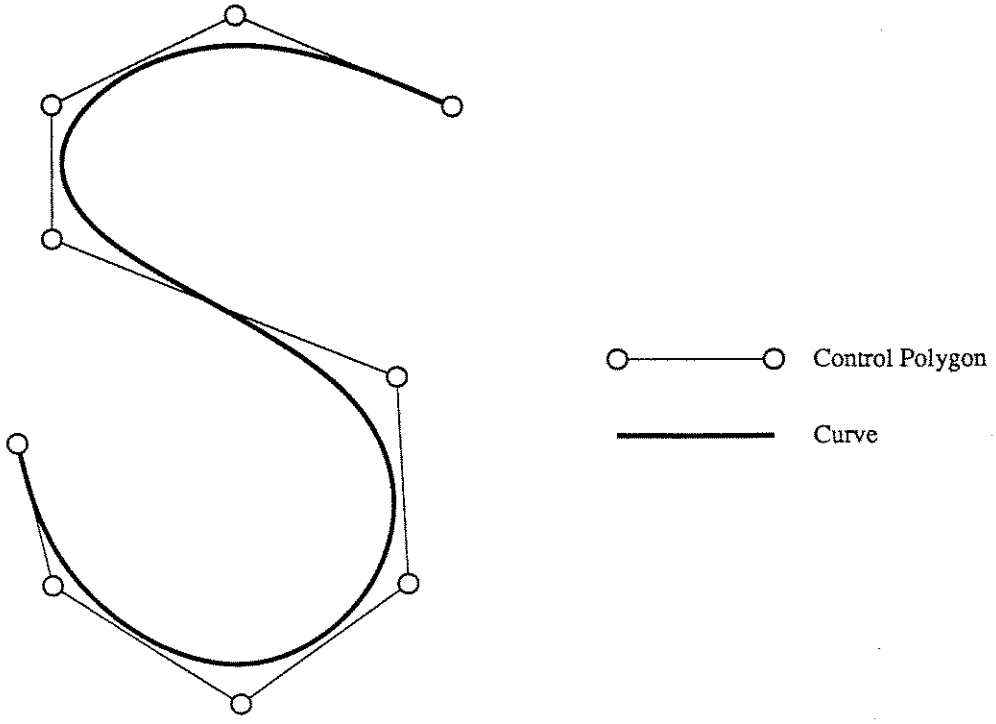



Figure 3. A Stylized Script S

To motivate the underlying theory, consider the design of a stylized script "S" shape as shown in Figure 3. The designer can communicate the basic shape by specifying a control polygon that in some sense approximates the desired shape. The system typically maps the controlling points $\mathbf{V}_i$ into a parametric curve $\mathbf{Q}(t)$ according to the formula

$$\mathbf{Q}(t) = \sum_i \mathbf{V}_i B_i(t), \qquad t \in [t_0, t_1] \tag{2.1}$$

where the functions $B_i(t)$ are "weighting", "blending", or "basis" functions, usually polynomials or piecewise polynomials, chosen to endow the curve with a given set of properties.‡ For instance, if the blending functions are chosen to be the Bernstein polynomials

$$B_i^d(t) = \binom{d}{i} t^i (1-t)^{d-i}$$

† The interested reader is referred to [15] for an introduction to the theory of urn models.

‡ For a good discussion of how blending functions influence the resulting curve, the reader is encouraged to consult Bartels *et al* [5].

then the curve

$$\mathbf{Q}(t) = \sum_{i=0}^{d} \mathbf{V}_i B_i^d(t), \qquad t \in [0,1]$$

is a Bézier curve of degree $d$.

Such a curve can be displayed on a graphics screen by computing points on the curve for various values of the parameter $t$, connecting adjacent points of evaluation with straight lines to obtain a piecewise linear approximation to the true curve (see Figure 4). Naturally, the larger the number of points of evaluation, the closer the approximation will be to the true curve.

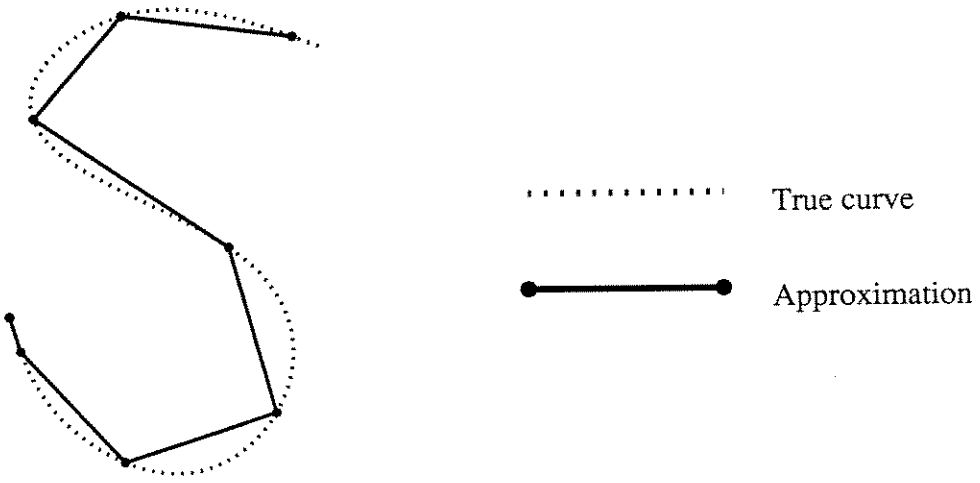



Figure 4. A Piecewise Linear Approximation

In the case of Bézier curves, the "obvious" method for computing the point on the curve corresponding to a fixed but arbitrary parameter value $t$ can be stated as:

*Bézier*( $\mathbf{V}_0, ..., \mathbf{V}_d$, $t$)
    /* *Return* $\mathbf{Q}(t)$ */
    $\mathbf{Q} \leftarrow (0,0)$
    **for** $i \leftarrow 0$ **to** $d$ **do**
        $\mathbf{Q} \leftarrow \mathbf{Q} + \mathbf{V}_i \binom{d}{i} t^i (1-t)^{d-i}$
    **endfor**
    **return Q**

Algorithm 1

This straight-forward algorithm is improved upon by the following elegant method due to de Casteljau (see Boehm *et al*[6]):

*Bézier*( $\mathbf{V}_0, ..., \mathbf{V}_d$, $t$)
    */* Return* $\mathbf{Q}(t)$ *using de Casteljau's Algorithm*/*
    **for** $i \leftarrow 0$ **to** $d$ **do**
        $\mathbf{V}_i^0 \leftarrow \mathbf{V}_i$
    **endfor**
    **for** $j \leftarrow 1$ **to** $d$ **do**
        **for** $i \leftarrow 0$ **to** $d - j$ **do**
            $\mathbf{V}_i^j \leftarrow (1-t)\mathbf{V}_i^{j-1} + t\mathbf{V}_{i+1}^{j-1}$
        **endfor**
    **endfor**
    **return** $\mathbf{V}_0^d$

Algorithm 2

Geometrically, each of the computed points $\mathbf{V}_i^j$, $j > 0$, lies on the line segment $\mathbf{V}_i^{j-1}\mathbf{V}_{i+1}^{j-1}$ so as to break the segment into two parts of relative lengths $t$ and $1-t$, yielding a geometric interpretation of the algorithm as shown in Figure 5 for a cubic curve. The algorithm of de Casteljau can also be viewed as a data-flow type computational graph, as shown in Figure 6 for the case of a cubic curve. Circles denote addition nodes, and the labels on the arcs of the graph indicate that data flowing along the arc should be multiplied by the value of the label before being input to the incident addition node.

de Casteljau's algorithm for a cubic Bézier curve can therefore be viewed as a three-level labeled digraph with a regular triangular interconnection, henceforth called a *triangular graph* or a *triangular computation*. In general, computation of a point on a Bézier curve of degree $d$ can be viewed as a triangular graph with $d$ levels.

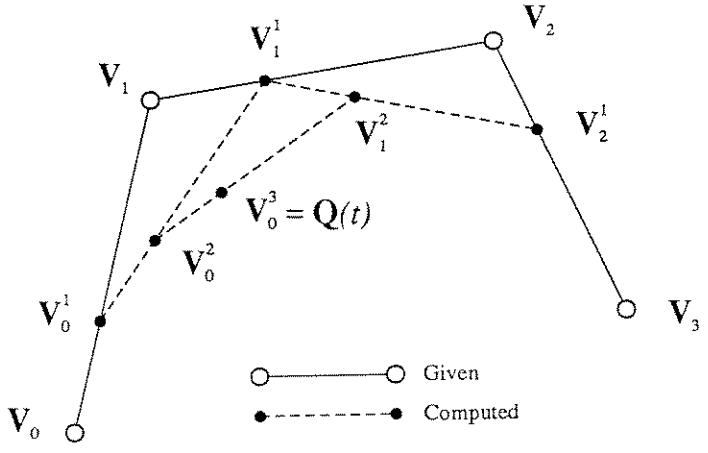



Figure 5. Cubic de Casteljau Diagram

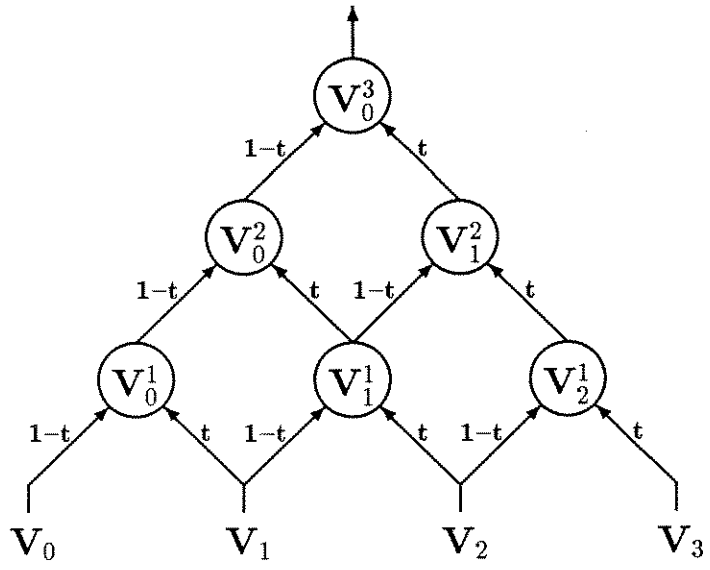



Figure 6. Cubic Computation Graph

Each node $\mathbf{V}_i^j$ in a triangular computation graph has two incident arcs, one from the left and one from the right. For the de Casteljau algorithm, all left arcs are labeled with the function $L(t) = 1 - t$ and all right arcs are labeled with the function $R(t) = t$. The theory of urn models states that de Casteljau's algorithm can be generalized to encompass other blending functions, and hence other curve schemes, by extending triangular graphs to allow arbitrary functional labels on the arcs, subject to three restrictions. With the notation that $L_i^j(t)$ and $R_i^j(t)$ denote the left and right labels incident upon the node $\mathbf{V}_i^j$, the restrictions can be stated as:

(*i*) All labels must be linear functions of the parameter $t$.
(*ii*) $L_i^j(t) + R_i^j(t) = 1$ for all $i$ and $j$.
(*iii*) $R_0^d(t) = t$.

Before justifying these restrictions, let us briefly examine how general triangular computations are used to define parametric curves. Just as for de Casteljau's algorithm, a point on a curve generated by a triangular computation with labels

$$L_i^j(t), R_i^j(t), \qquad j = 1, ..., d, \quad i = 0, ..., d - j,$$

is defined to be the value produced at the apex of the triangle. More precisely, if $\mathbf{A}(t)$ represents the parametrized curve described by the triangular computation, then

$$\mathbf{A}(t) \leftarrow \mathbf{V}_0^d$$

where

$$\mathbf{V}_i^j \leftarrow L_i^j(t)\mathbf{V}_i^{j-1} + R_i^j(t)\mathbf{V}_{i+1}^{j-1}, \qquad j = 1, ..., d, \quad i = 0, ..., d - j,$$

and where

$$\mathbf{V}_i^0 \leftarrow \mathbf{V}_i, \qquad i = 0, ..., d.$$

A general triangular computation can also be viewed as a geometric construction algorithm since a computed point $\mathbf{V}_i^j$ will divide the segment $\mathbf{V}_i^{j-1}\mathbf{V}_{i+1}^{j-1}$ into relative lengths $R_i^j(t)$ and $L_i^j(t)$, as shown in Figure 7 for $d = 2$.

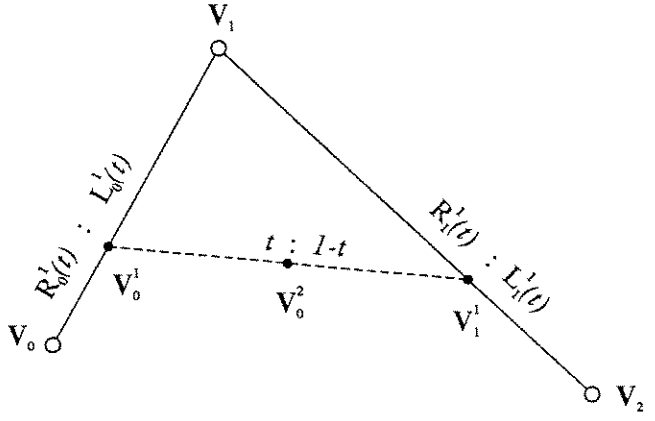



Figure 7. General Quadratic Construction

Restriction $(i)$ for the functional labels of a triangular computation guarantees that the parametrized curve produced by a $d$-level triangular graph will be a parametric polynomial of degree $d$. Restriction $(ii)$ guarantees that the relationship between the curve and its control polygon remains unchanged under rotation, translation, and scaling (in the jargon of computer-aided geometric design, restriction $(ii)$ guarantees that each of the intermediate points $\mathbf{V}_i^j$ are *affinely invariant* [6]). Finally, restriction $(iii)$ provides a normalization of the parameter range to the interval $[0, 1]$; that is, the curve segment is traced when $0 \leq t \leq 1$.

While triangular computations are one possible generalization of de Casteljau's algorithm, it must be demonstrated that they represent a *useful* generalization. To provide evidence that triangular computations are indeed useful, below we cite several results due to Goldman and others that show that virtually all of the curve techniques in widespread use in computer-aided geometric design can be obtained via triangular computations (or equivalently, via urn models).

A particular technique can be described by a triangular computation if it is possible to appropriately choose the functional labels so as to obtain an algorithm for computing points on such a curve. For instance, we have already seen that Bézier curves can be obtained by setting all left labels to $1 - t$, and all right labels to $t$, thereby resulting in de Casteljau's algorithm. Perhaps less obvious is the ability to generate arbitrary B-splines [6]. The functional labels necessary to generate B-splines are provided by the Cox-deBoor algorithm [6]. It is also possible to generate cubic Catmull-Rom curves [9], Pólya curves [16], generalized Pólya curves [1], and hence Lagrange curves [1, 5, 6] as a special case. A detailed assignment of labels for these curve schemes is given in Appendix 1.

## 3. The Basic Architecture

The Triangle is essentially a hardware implementation of triangular graphs, and is composed by associating a processing element (PE) per node in the triangular computation graph, connecting the processors as dictated by the arcs of the graph. Referring to Figure 8, all processors are identical, with each one receiving three inputs, $\mathbf{V}_L, \mathbf{V}_R$, and $t$, and generating one output $\mathbf{V}_{\text{out}}$. The resulting architecture for a cubic Triangle is shown in Figure 9.

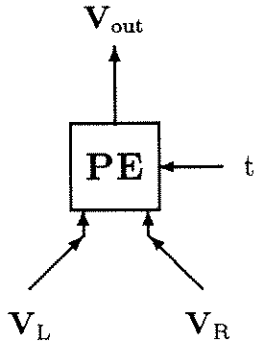



Figure 8. Processing Element

As an optional initialization step, the Triangle can be configured for a particular set of blending functions (*i.e.*, a particular curve scheme) by percolating appropriate linear functions into each PE. Actually, only one of the linear functions, say the one from the right, needs to be specified, the other being implied by restriction (*ii*). Since a linear function can be completely characterized by two scalar coefficients, each PE must possess two registers to record the labels incident upon it. If each PE stores the value of the right label for $t = 0$ and $t = 1$ in registers $R_0$ and $R_1$, respectively, then all other values of the label can be computed from $R(t) = (1-t)R_0 + tR_1$. Thus, the configuration procedure requires that the $i^{\text{th}}$ PE on level $j$ is given the values $R_i^j(0)$ and $R_i^j(1)$. A processing element can now be modeled by the pseudo-code:

$PE(\mathbf{V}_L, \mathbf{V}_R, t)$
**Local register** $R_0, R_1$

$R \leftarrow (1-t)R_0 + tR_1$
$\mathbf{V}_{\text{out}} \leftarrow (1-R)\mathbf{V}_L + R\mathbf{V}_R$
**return** $(\mathbf{V}_{\text{out}})$

Algorithm 3

High performance is achieved by synchronously pipelining the computation of many points of evaluation, treating each level of the Triangle as a stage in the pipeline. In general,

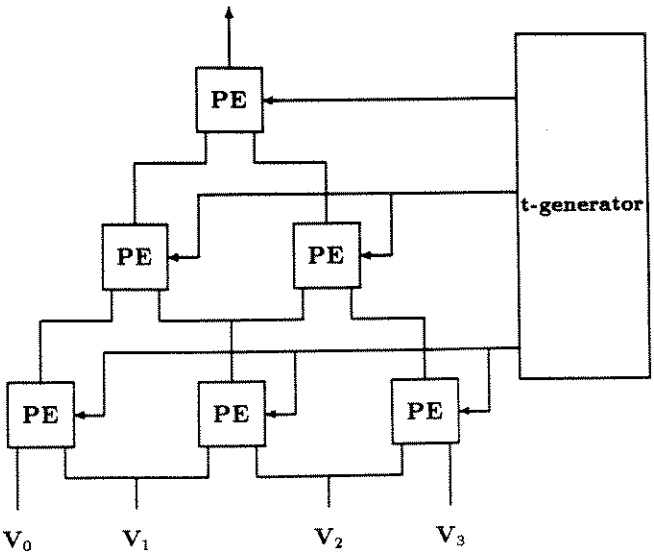



Figure 9. Basic Architecture

the points on the curve corresponding to a large number of parameter values $t_0, t_1, ..., t_k$ are required to achieve a dense dot-representation of the curve. In other words, if $\mathbf{Q}(t)$ is the curve generated by the Triangle, we seek the points $\mathbf{Q}(t_0), ..., \mathbf{Q}(t_k)$. By attaching an external "t-generator" as shown in Figure 9, parameter values can be supplied to each level of the Triangle according to the scheme shown in Figure 10. During the first time step, the processors on level one are fed the parameter value $t_0$. They perform the computation of Algorithm 3 with $t = t_0$, and pass their output up to the processors at level two by the end of time step one. During time step two, the processors on level two are fed $t_0$, while those on level one are fed $t_1$. The process proceeds in this lock-step fashion until at the beginning of time step four, the point $\mathbf{Q}(t_0)$ emerges from the apex of the Triangle, followed by the point $\mathbf{Q}(t_1)$ on the next time step, and so on.

There is a rather wide spectrum of choice for the implementation of the processing elements of a Triangle. The spectrum extends from microcomputers such as the NMOS Transputer [18] and Motorola 68020 on one end, through off-the-shelf MSI parts and gate arrays, to custom VLSI on the other end. Several possible implementations and estimates of their performance are developed in Section 8. The results of this analysis are summarized in Table 1.

Once the Triangle is "full", one point of evaluation per time step emerges from the apex of the Triangle — independent of the degree of the curve. For typical design applications, the degree of the curve is small, generally less than five, so the time to configure and fill the Triangle is small compared to the (roughly) 100 to 1,000 time steps that the Triangle runs at full throughput. For instance, assuming a Motorola 68020 host, we estimate that a custom VLSI Triangle can be configured for and generate 200 points on a cubic curve segment in $56\mu s$, implying that over 300 cubic curve segments can be completely redisplayed at real-time rates.

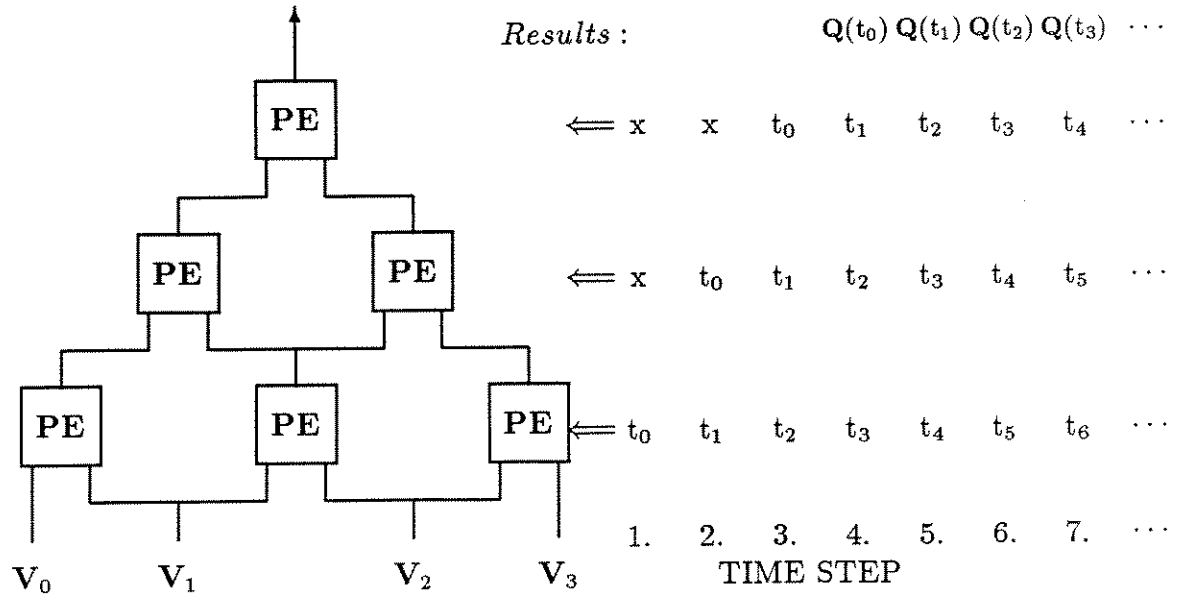



Figure 10. Pipelining Schemata (x = don't care)

| | Values/second | Processors/sq.in. |
|---|---|---|
| Transputer | 140,000 | 0.13 |
| MSI | 2,000,000 | 0.18 |
| Gate Array | 5,000,000 | 30 |
| CMOS VLSI | 1,000,000 | 50 |
| MAC | 13,300,000 | 0.18 |

Table 1: Performance Estimates

Transputer [18] estimates are for a software implementation of the scaled computation (see Section 8) executed on a model T414 running at 20 MHz. The binary search implementation described in Section 8.1 using Fairchild FAST [14] components provided the MSI estimates. Gate array estimates are for the implementation of this MSI design using an LSI Logics LCA10129 [8], with 129,042 gates. A 40% gate utilization is assumed. Estimates for custom VLSI are for this same binary search implementation, and are based on a comparison of the specifications for the above gate array and a $2\mu$m generic CMOS process. The MAC (multiplier/accumulator) estimates are from the example in Section 8.2.

Several remarks are in order at this point:

- The use of a Triangle obviates the need for specialized line and circle drawing hardware. Indeed, the line segment connecting two points $\mathbf{P}_0$ and $\mathbf{P}_1$ can be generated by

configuring the Triangle as shown in Figure 11; the generation of circles and ellipses will be presented in Section 4.

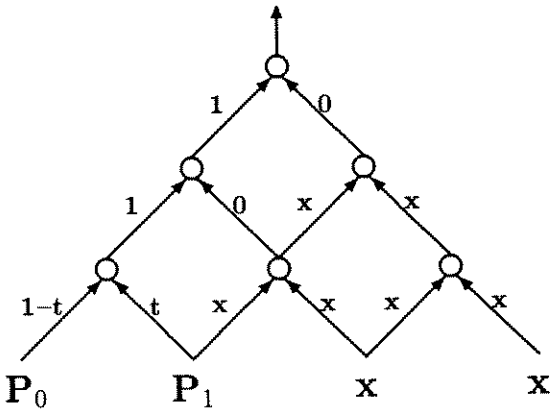



Figure 11. Labels for Line Generation (x = don't care)

- Referring to Algorithm 3, it is apparent that the calculation of the x- and y-components of $\mathbf{V}_{\text{out}}$ are independent. Thus, instead of building a Triangle for vector-valued computations, it is possible (and probably more desirable) to use two Triangles, each doing a scalar component computation (see Figure 12).

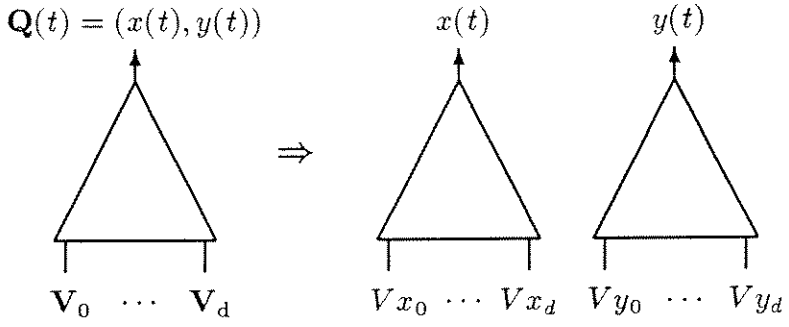



Figure 12. Scalar Triangles

- Although Algorithm 3 suggests that each PE must perform a floating point computation, by representing the control points directly in device coordinates the computation can be integerized in a number of ways (see Section 8). The particular integerized form can be chosen to optimize the capabilities of the PEs.

## 4. Generating Rational Polynomial Curves and Conic Sections

Although polynomial curves are quite flexible, they cannot exactly reproduce circles. In fact, the only conic section that can be exactly reproduced by polynomial curves is the parabola. However, by generalizing to rational polynomial curves, reproduction of all conic

sections is possible. For more information on the representation of conic sections by rational polynomial curves see Boehm *et al*[6] or Penna & Patterson [21].

If the polynomial curve

$$\mathbf{Q}(t) = \sum_{i=0}^{d} \mathbf{V}_i B_i(t)$$

can be generated by a triangular computation, then so too can the *rational polynomial* curve

$$\begin{aligned}\mathbf{P}(t) &= \frac{\sum_{i=0}^{d} w_i \mathbf{V}_i B_i(t)}{\sum_{i=0}^{d} w_i B_i(t)} \\ &= (X(t), Y(t))\end{aligned}$$

where $w_0, ..., w_d$ are scalar "weights" associated with each of the points [6, 21]. The rational polynomial curve $\mathbf{P}(t)$ can be generated using two dividers and three scalar Triangles (one for the x-component of the numerator, one for the y-component of the numerator, and one for the denominator) arranged as shown in Figure 13.

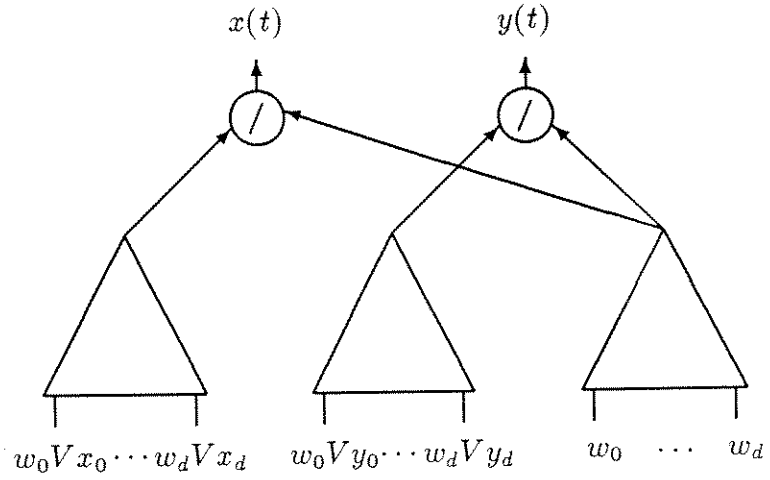



Figure 13. Generating Rational Polynomials

## 5. Generating Tensor Product Surfaces

The Triangle can also be used as a basic building block in the generation of tensor product surfaces such as bicubic B-spline and Bézier surfaces.

A *tensor product surface patch* $\mathbf{S}(t, u)$ (in the most general setting) is defined by two sets of blending functions, call them $A = (A_0(t), ..., A_n(t))$ and $B = (B_0(u), ..., B_m(u))$ of polynomial degree $n$ and $m$, respectively, together with a rectilinear net of control points $\{\mathbf{V}_{i,j}\}_{i=0}^{n}{}_{j=0}^{m}$ according to

$$\mathbf{S}(t, u) = \sum_{j=0}^{m} \sum_{i=0}^{n} \mathbf{V}_{i,j} A_i(t) B_j(u), \qquad (t, u) \in [0, 1] \times [0, 1]. \tag{5.1}$$

If curves defined by the set of blending functions $A$ and $B$ can be generated by triangular computations, then so too can the tensor product surface of Equation (5.1). To see this, we rewrite Equation (5.1) as

$$\mathbf{S}(t,u) = \sum_{j=0}^{m} \mathbf{Q}_j(t) B_j(u),$$

where

$$\mathbf{Q}_j(t) = \sum_{i=0}^{n} \mathbf{V}_{i,j} A_i(t),$$

showing that a tensor product surface can be thought of as blending the curves $\mathbf{Q}_j(t)$ together with the functions $B_0(u), ..., B_d(u)$. The required blending can be accomplished by a set of Triangles organized in the prismatic arrangement shown in Figure 14. Each Triangle in the triangular prism is configured to generate one of the curves $\mathbf{Q}_j(t)$, with the parameter values supplied by an external t-generator. The outputs of these are fed to another Triangle configured for the $B$ blending functions. Parameter values for the Triangle spanning the prism can be supplied by an external u-generator. The grid of control points is input on the bottom face of the prism and the points of evaluation of the surface appear at the apex of the Triangle that spans the prism.

Once the prism is configured, the entire surface patch can be generated without host intervention. Assuming a Motorola 68020 host and a VLSI implementation of the Triangles, we estimate that a prism can be configured for and generate 1,000 points on a bicubic patch in just over 500$\mu$s, implying that 65 surface patches can be completely redisplayed at real-time rates.

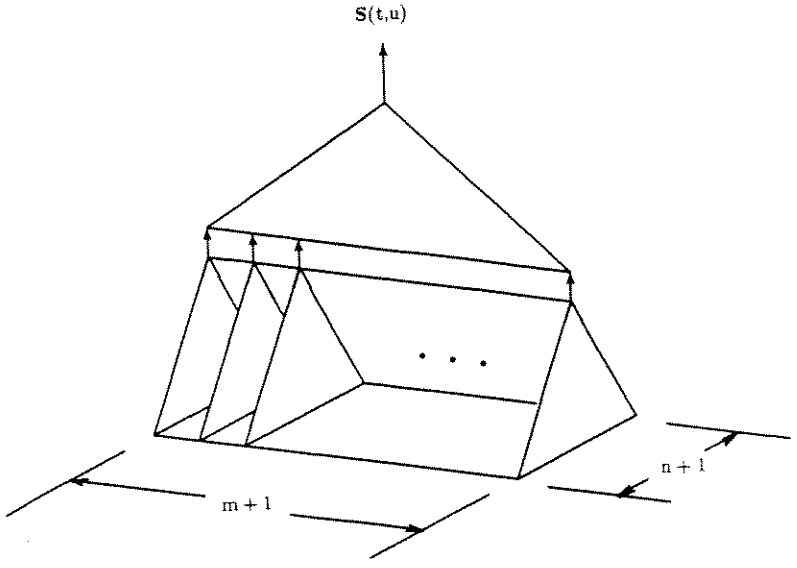



Figure 14. Prismatic Structure for Surface Generation

## 6. Generating Triangular Bézier Patches

In Section 5 is was shown that the Triangle could be used as a building block in the construction of a multiprocessor to generate points on tensor product Bézier surfaces. Although tensor product forms are well-known, another surface form known as the *triangular Bézier* patch is more appropriate for some applications (see Farin [13] for an excellent treatment of triangular Bézier patches). In this section we point out that by extending the triangular topology of the Triangle to a tetrahedral arrangement, points on triangular Bézier surfaces can be generated at the same rate as the Triangle can compute points on a Bézier curve.

Very briefly, a triangular Bézier patch of degree $d$ is defined by a triangular net of control points $\{\mathbf{V}_{i_1 i_2 i_3}\}$ where the indices $i_1, i_2, i_3$ take on all non-negative values such that $i_1 + i_2 + i_3 = d$. The control points are blended together to form surface patch according to

$$\mathbf{S}(t,u,v) = \sum_{i_1,i_2,i_3} \mathbf{V}_{i_1 i_2 i_3} B^d_{i_1 i_2 i_3}(t,u,v), \tag{6.1}$$

where

1. The summation is taken over all non-negative values of $i_1, i_2, i_3$ such that $i_1 + i_2 + i_3 = d$.
2. The parameters $(t,u,v)$ are restricted to be non-negative, and in addition, they must satisfy $t + u + v = 1$. Thus, although we have written the surface patch as a function of three parameters, the fact that they must sum to unity implies that only two of the parameters are independent.
3. The functions $B^d_{i_1 i_2 i_3}(t,u,v)$ are *bivariate Bézier* basis functions of degree $d$, and are defined by

$$B^d_{i_1 i_2 i_3}\ (t,u,v) = \frac{d!}{i_1! i_2! i_3!} t^{i_1} u^{i_2} v^{i_3}. \tag{6.2}$$

Just as for Bézier curves, there is a recursive algorithm due to de Casteljau for computing points on a triangular Bézier surface. The algorithm is a relatively simple extension of the de Casteljau algorithm for curves, and may be stated as [6]:

*TriangularBézier*( $\mathbf{V}_{d00}, \mathbf{V}_{d-1,1,0}, \ldots, \mathbf{V}_{00d}, t, u, v$)
  /* *Return* $\mathbf{S}(t,u,v)$ *using de Casteljau's Algorithm* */
  **for all** $i_1, i_2, i_3 \geq 0$ **such that** $i_1 + i_2 + i_3 = d$ **do**
    $\mathbf{V}^0_{i_1 i_2 i_3} \leftarrow \mathbf{V}_{i_1 i_2 i_3}$
  **endfor**
  **for** $j \leftarrow 1$ **to** $d$ **do**
    **for all** $i_1, i_2, i_3 \geq 0$ **such that** $i_1 + i_2 + i_3 = d - j$ **do**
      $\mathbf{V}^j_{i_1 i_2 i_3} \leftarrow t\mathbf{V}^{j-1}_{i_1+1,i_2,i_3} + u\mathbf{V}^{j-1}_{i_1,i_2+1,i_3} + v\mathbf{V}^{j-1}_{i_1,i_2,i_3+1}$
    **endfor**
  **endfor**
  **return** $\mathbf{V}^d_{000}$

Algorithm 4

This algorithm can be viewed as a data-flow graph of tetrahedral topology, as shown schematically in Figure 15. The implementation of the tetrahedral computation in hardware is straightforward, leading to a variant of the Triangle where each processing element accepts three input points and two parameters, $t$, and $u$, say, and generates one output.

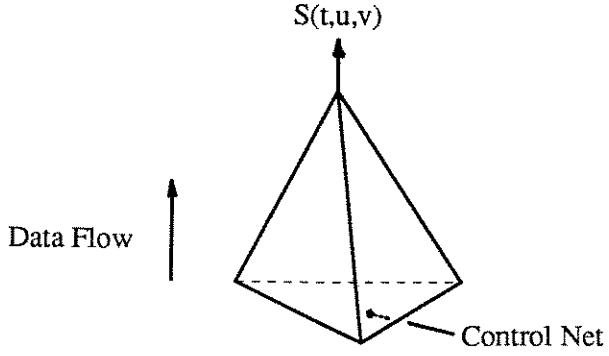



Figure 15. Tetrahedral Structure for Triangular Patch Generation

## 7. Comparison to Previous Work

Two alternatives to the Triangle for the generation of curves are architectures based on forward differencing and subdivision. In this section we briefly describe a parallel architecture for each method, and compare the performance of these architectures with the Triangle. Table 2 gives an implementation-independent summary of the results of this comparison for generating curves of degree $d$.

The first row of Table 2 corresponds to the time for initialization of the hardware and any necessary host precomputation; the second row corresponds to the time needed to re-initialize due to the movement of a control point; the third row corresponds to the time to compute a point of evaluation once the pipeline has been filled; the fourth row denotes the number of processing elements required; finally, the fifth row reflects the numerical stability of the computation.

The first three columns of Table 2 concern the performance of the Triangle. The first column refers to the generation of *uniform* urn models such as Bézier curves and uniform B-splines (in general, a curve technique is said to be uniform if the blending functions do not change from one segment to the next). Initialization for this case is done by simply down-loading the control points, requiring order $d$ time. Perturbation due to control point movement is also very fast since only the point that the designer has moved needs to be updated.

The second column refers to the generation of non-uniform urn models, *i.e.*, urn models such as non-uniform B-splines where the blending functions do change from one segment to

the next. In this case, the Triangle must be reconfigured by percolating in new labels, a process requiring $O(d^2)$ time.

The third column refers to the generation of a curve such as a Beta-spline that is not represented as an urn model. To generate such a curve with the Triangle, a change of basis algorithm must be performed by the host, requiring $O(d^3)$ time in general.

The fourth column of Table 2 is explained in Section 7.1, and the last two columns are explained in Section 7.2.

| | Triangle | | | Forward | Subdivision | |
|---|---|---|---|---|---|---|
| | Uniform | Non-uniform | Non-urn model | Difference | Bézier | Non-Bézier |
| Initialization Time | $O(d)$ | $O(d^2)$ | $O(d^3)$ | $O(d^3)$ | $O(d)$ | $O(d^3)$ |
| Perturbation Time | $O(1)$ | $O(d^2)$ | $O(d^3)$ | $O(d^3)$ | $O(1)$ | $O(d^3)$ |
| Computation Time/Value | $O(1)$ | $O(1)$ | $O(1)$ | $O(1)$ | $O(1)$ | $O(1)$ |
| Hardware Required | $O(d^2)$ | $O(d^2)$ | $O(d^2)$ | $O(d)$ | $O(d^2)$ | $O(d^2)$ |
| Numerical Stability | good | good | good | poor | good | good |

Table 2: Comparison of Parallel Architectures for Degree d Polynomials

### 7.1. Forward Differencing

Forward differencing is a well known and commonly used technique for generating points on curves and surfaces [5]. The basic forward differencing algorithm for computing the points of a degree $d$ polynomial $p(u)$ begins by computing $d+1$ values $p(ih)$, $i = 0, ..., d$, of the polynomial, and from these computes $d+1$ *forward differences.* This is essentially a change of basis step, requiring $O(d^3)$ time in general. Once the forward differences have been computed, the values $p((d+j)h)$, $j = 1, ..., N$, can be computed at a cost of $O(d)$ per point on a serial processor. However, it is possible to pipeline this computation so that a point can be generated in a constant time with $O(d)$ hardware.

While the forward differencing technique can rapidly generate points using very little hardware, it has some drawbacks. One of the major ones is that it is numerically unstable. As iteration along the curve proceeds, errors are propagated, potentially making the overall error large. Controlling this error requires that the computation be carried out with a large number of significant digits. For example, evaluating 256 points of a cubic polynomial for a 512x512 display will require in excess of 32 bits of accuracy [5]. Moreover, this cumulative error makes it difficult to generate curves with degree much greater than three [5]. In

contrast, the Triangle is numerically stable for common curve schemes, requiring at most 16 bits of accuracy for a 1024x1024 display (see Section 8).

The second difficulty with forward differencing is that the step size must be chosen small enough to make the curve appear smooth, yet must be maintained as large as possible to control cumulative error. It is not always possible to satisfy both of these opposing constraints, although there has been some promising recent work in this area [19].

Finally, the use of forward differencing always requires the host to perform a time consuming change of basis step.

### 7.2. The Subdivision Method

By applying de Casteljau's algorithm with $t = 1/2$ to the control polygon of a Bézier curve we can generate two new control polygons $\mathbf{L}$ and $\mathbf{R}$ that *subdivide* the curve. More precisely, the polygon $\mathbf{L}$ generates the first half of the original curve, and the polygon $\mathbf{R}$ generates the second half. Recursive application of the subdivision step leads to a divide and conquer approach to Bézier curve generation since the control points of the subdivided polygons are guaranteed to converge to the original curve.

A parallel implementation of the algorithm for Bézier curves is described in [10]. The basic architecture consists of a "subdivision processor", a stack, and some control. The subdivision processor is a triangular arrangement of adders that performs the operations of de Casteljau's algorithm, for $t = 1/2$, on the control point values entered at the base. Two new sets of control points forming the sub-polygons $\mathbf{L}$ and $\mathbf{R}$ are produced at the other two edges of the triangle. One of these polygons is pushed on the stack for later evaluation, and the other is routed back through the "subdivision" processor. This continues for a user specified number of iterations, at which point the resulting polygons are output for display. Computation then continues with the polygon on the top of the stack, terminating when the stack is empty.

For the generation of Bézier curves, no change of basis is required so initialization can be accomplished in $O(d)$ time. However, to generate curves defined by other blending functions we must first convert them to Bézier form, requiring $O(d^3)$ time in general.

Once initialized, each step in the above iteration takes time $O(d)$, and produces $O(d)$ points. If $k$ is the iteration depth, then $d\,2^{k+1}$ points are generated in a total of $2^{k+1}$ steps. On average, then, each point is produced in constant time. The hardware required for this architecture is $O(d^2)$ for the "subdivision processor" and also for the stack. The subdivision method is as numerically stable as the Triangle, so on the order of 16 bits of accuracy are required for a 1024x1024 display.

## 8. Implementation

The detailed architecture of the basic processing element shown in Figure 8 will depend on the exact formulation of the triangular computation of Algorithm 3, on the range of values we allow for labels, and on speed, area, and cost constraints imposed by the overall

system. In this section we will explore this space of architectural possibilities and examine the performance, area, and cost trade-offs involved in their implementation.

If the control point values input at the base of the Triangle represent points in a user coordinate system, then they will most likely be floating point quantities. In this case the processing element will need floating point hardware to achieve real-time performance. For example, we may use the T800 Transputer as a processing element. We estimate that a system constructed with this device would generate a point every 7.5 $\mu$sec.

However, it will generally be the case that control point values will represent points in the device coordinate system, and so will be integer quantities. It is then desirable to find a suitable "integerized" version of Algorithm 3. We will present two methods of accomplishing this integerization. Throughout this presentation we assume that the display is a frame-buffer type device with $2^r$ addressable pixels in each of the horizontal and vertical directions. Further, we assume that the curve segments to be computed using the Triangle have control points that are on or "near" the screen, and that each control point is represented as a pair $(x, y)$ where $x$ and $y$ are $r + e$ bit integers. The extra $e$ bits are used as guard bits and to represent points that are not on the screen but are near it, effectively surrounding the screen by a larger space of representable points. Without guard bits, each level of the Triangle could introduce up to one pixel error. The generation of a point on a $d$ degree polynomial could therefore accumulate up to $d$ pixels of error. This error can be eliminated by introducing a small number ($\log d$) additional guard bits.

The first method of integerizing Algorithm 3 is to simply take each quantity to be an integer fraction scaled up by $2^r$. Thus, if $R$ was in the range $[-n, n]$ it will now be in the range $[-n2^r, n2^r]$. With this integerization, we can perform the computation of Algorithm 3 with integer addition, subtraction, and multiplication. To be exact, two additions, two subtractions, and four multiplications are required. Alternatively, we can rearrange the computation as follows

$$R = R_0 + t(R_1 - R_0)$$

$$\mathbf{V}_{\text{out}} = \mathbf{V}_L + R(\mathbf{V}_R - \mathbf{V}_L)$$

thus reducing the number of multiplications to two. Also, $R_1 - R_0$ can be precomputed by the host, reducing the number of subtractions to one.

The second method of integerizing Algorithm 3 is to represent $t$ and $R$ as binary fractions, and to perform an integer computation using a binary search technique. To simplify the presentation of this method, let us for the moment consider a Triangle configured for generating a Bézier curve. In this case, each PE must perform the computation

$$\mathbf{V}_{\text{out}} \leftarrow (1 - t)\mathbf{V}_L + t\mathbf{V}_R \tag{8.1}$$

where $t \in [0, 1]$.

We use Equation (8.1) to provide a coordinate labeling of the line through $\mathbf{V}_L$ and $\mathbf{V}_R$. In particular, $\mathbf{V}_L$ is assigned the coordinate 0, $\mathbf{V}_R$ is assigned the coordinate 1, and in general, $\mathbf{V}_{\text{out}}$ is assigned coordinate $t$. In this context Equation (8.1) can be recast as a search

problem: each processor must *find* the point $\mathbf{V}_{\text{out}}$ with coordinate $t$ on the coordinatized line from $\mathbf{V}_L$ to $\mathbf{V}_R$.

With this interpretation of the problem, binary search enters as a natural solution. Let $t$ be represented by an $f$-bit binary fraction

$$t = .\mathrm{b}_1\mathrm{b}_2\cdots\mathrm{b}_f, \qquad \mathrm{b}_i \in \{0,1\}, \tag{8.2}$$

and let $\mathbf{M}$ denote the midpoint of the line segment $\mathbf{V}_L\mathbf{V}_R$

$$\mathbf{M} = \frac{\mathbf{V}_L + \mathbf{V}_R}{2}. \tag{8.3}$$

If $\mathrm{b}_1 = 0$, then $\mathbf{V}_{\text{out}}$ must be between $\mathbf{V}_L$ and $\mathbf{M}$. More precisely, $\mathbf{V}_{\text{out}}$ will be the point with coordinate $.\mathrm{b}_2\mathrm{b}_3\cdots\mathrm{b}_f0$ on the coordinatized line from $\mathbf{V}_L$ to $\mathbf{M}$. Symmetrically, if $\mathrm{b}_1 = 1$, then $\mathbf{V}_{OUT}$ will have coordinate $.\mathrm{b}_2\mathrm{b}_3\cdots\mathrm{b}_f0$ on the coordinatized line from $\mathbf{M}$ to $\mathbf{V}_R$. Recursive application of this process will have $\mathbf{M}$ converging to $\mathbf{V}_{\text{out}}$. Moreover, if all points are represented as pairs of $r+e$ bit integers, the process will converge after $r+e$ recursive invocations. An iterative version of this algorithm, one that is more efficiently implemented on standard hardware, can be stated as:

```
PE-Bezier( V_L, V_R, t)
    /* Processing element for a Bézier curve.  */
    /* Returned is the point on V_L V_R with coordinate t, where t ∈ [0,1]  */
    for i ← 1 to r + e do
        /* Compute Midpoint (≫ 1 means right shift 1 bit) */
        M ← (V_L + V_R) ≫ 1
        if high-order bit of t is 0 then
            /* V_out is between V_L and M */
            V_R ← M
        else
            /* V_out is between M and V_R */
            V_L ← M
        endif
        /* Left shift the representation of t left 1 bit */
        t ← t ≪ 1
    endfor
    return M
```

Algorithm 5

Coordinatized lines and binary search can also be used to refine the pseudo-code of Algorithm 3 for urn models other than Bézier curves. Note that as $t$ varies from 0 to 1, the

point $\mathbf{V}_{\text{out}}$ produced by Algorithm 3 varies from the point

$$\mathbf{V}_0 = (1 - R(0))\mathbf{V}_L + R(0)\mathbf{V}_R \tag{8.4}$$

to the point

$$\mathbf{V}_1 = (1 - R(1))\mathbf{V}_L + R(1)\mathbf{V}_R. \tag{8.5}$$

We can therefore rewrite the expression for $\mathbf{V}_{\text{out}}$ as

$$\mathbf{V}_{\text{out}} \leftarrow (1 - t)\mathbf{V}_0 + t\mathbf{V}_1. \tag{8.6}$$

For B-splines, the values $R(0)$ and $R(1)$ are guaranteed to be in the interval $[0, 1]$. Thus, Equations (8.4), (8.5), and (8.6) are of the form implemented by *PE-Bezier*, giving us the following code for the generation of B-splines:

*PE-Bspline*($\mathbf{V}_L$, $\mathbf{V}_R$, $t$)
**Local register** $R$, $\mathbf{V}_0$, $\mathbf{V}_1$

$\mathbf{V}_0 \leftarrow$ *PE-Bezier*($\mathbf{V}_L, \mathbf{V}_R, R(0)$)
$\mathbf{V}_1 \leftarrow$ *PE-Bezier*($\mathbf{V}_L, \mathbf{V}_R, R(1)$)
$\mathbf{V}_{\text{out}} \leftarrow$ *PE-Bezier*($\mathbf{V}_0, \mathbf{V}_1, t$)
**return** ($\mathbf{V}_{\text{out}}$)

Algorithm 6

In general, however, $R(0)$ and $R(1)$ may not be between 0 and 1 as required by *PE-Bezier*. For instance, if the processor is configured for Lagrange interpolation, then for all processors except the one at the apex of the Triangle $R(0) < 0$ and $R(1) > 1$. Fortunately, for most curve techniques, including Lagrange interpolation, the values of $R(0)$ and $R(1)$ are bounded by small signed integers. We therefore represent $R(0)$ and $R(1)$ as signed fixed-point numbers, *ie*,

$$R(0) = ROI.ROF$$
$$R(1) = R1I.R1F$$

where $ROI$ and $R1I$ are signed integers, and $ROF$ and $R1F$ are binary fractions. The code for *PE-Bézier* can now be generalized to find points whose coordinates are outside the interval $[0, 1]$:

```
BinSearch(A, B, I, F)
    /* Find the point on AB with coordinate I.F */
    if I > 0 then
        Δ ← B − A
        while I ≠ 0 do
            A ← B
            B ← A + Δ
            I ← I − 1
        endwhile
    else if I < 0 then
        Δ ← A − B
        while I ≠ 0 do
            B ← A
            A ← B + Δ
            I ← I + 1
        endwhile
    return (PE-Bezier(A, B, F))
```

Algorithm 7

An interized version of the pseudo-code for a processing element in the most general case can now be stated as:

```
PE-General(V_L, V_R, t)
Local register
        R0I,    /* Integer part of R(0) */
        R0F,    /* Fractional part of R(0) */
        R1I,    /* Integer part of R(1) */
        R1F,    /* Fractional part of R(1) */
        V_0, V_1
    V_0 ← BinSearch(V_L, V_R, R0I, R0F)
    V_1 ← BinSearch(V_L, V_R, R1I, R1F)
    V_out ← PE-Bezier(V_0, V_1, t)
    return (V_out)
```

Algorithm 8

These two integerized forms of the computation suggest quite different processor element architectures. An architecture for a scaled computation will require one or more multipliers and adders, while the binary search method requires only addition. A binary search architecture needs to either loop through the computation in a serial fashion using one

adder, or use a string of adders to pipeline the computation. The number of bits in $t$ and $R$ determines the number of times through the loop, or the length of the pipeline, respectively. By using a parallel multiplier for the scaled computation, the operations will be carried out in parallel with respect to the bits of $t$ and $R$. The scaled computation then has the potential of being faster, but the resulting architecture may require more area and be more costly than a binary search architecture . To further illustrate the design trade-offs, we will present two example architectures, one for each type of integerization. Note that in these examples our intention is to give a clear idea of the basic requirements for a Triangle implementation; consequently, we do not discuss the many possible optimizations.

### 8.1. A Binary Search Architecture

The first example we will consider is an architecture for the binary search integerization of the basic triangular computation restricted to the generation of cubic Bézier curves. This means that $R(t) = t$, so we need only perform the computation:

$$\mathbf{V}_{\text{out}} \leftarrow BinSearch01(\mathbf{V}_L, \mathbf{V}_R, t)$$

The architecture of a processing element is shown in Figure 16. Registers *l-reg* and *r-reg* hold the initial and intermediate values of $\mathbf{V}_L$ and $\mathbf{V}_R$. The adder uses the two values in these registers to produce $M$ on output bits $b_1 - b_{n+1}$, thus in effect performing the required shift. The *sel* control signal on the registers selects which of the two inputs will be loaded into the register. At the beginning of a search, new values are loaded from the level below. In this case both *l-ld* and *r-ld* are clocked. During the computation the feedback source is selected, and either *l-ld* or *r-ld* is clocked, depending on whether the high-order bit of $t$ is 0 or 1 respectively.

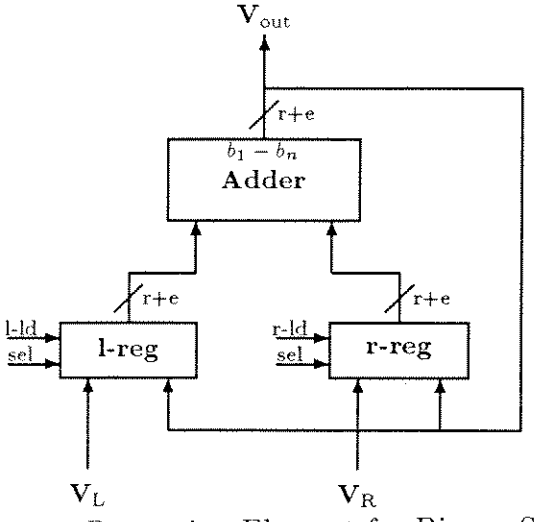



Figure 16. Processing Element for Binary Search

By combining twelve of these PE's to form two Triangles, we can compute the x and y coordinates of the points of a specified cubic Bézier curve. In such a system, the *ld* control

signals to the registers will be the same across a level in both Triangles, hence we need three copies of the logic required to generate these signals. The *sel* control signal will be the same throughout both Triangles, and so can be generated in the main control of the system. These building blocks along with a t-generator combine to produce a complete system as shown in Figure 17.

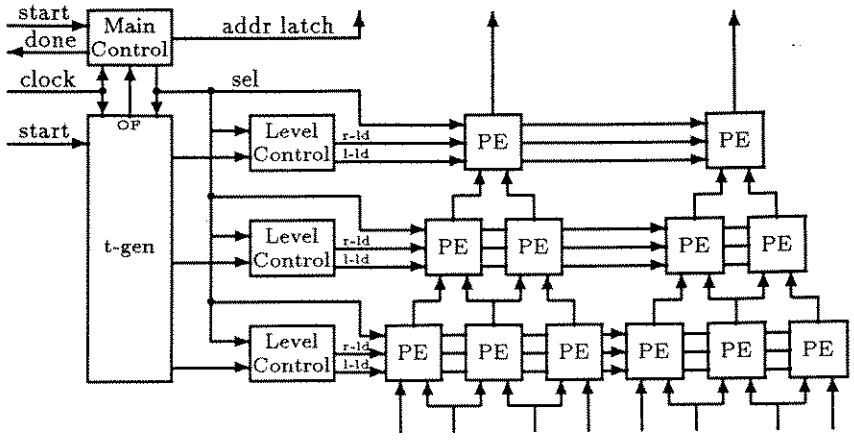



Figure 17. Complete Binary Search System

The logic to generate the *ld* signals, called the *level control*, requires three inputs, a signal to mark the beginning of a search, a clock to step the hardware through the search, and the high-order bit of the t-value for that level. The two *ld* signals are produced. Figure 18 shows the required logic and the timing relationships of the input and output signals.

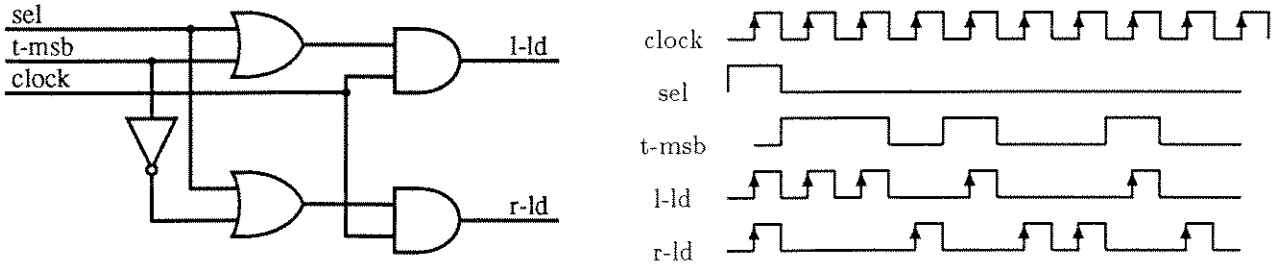



Figure 18. Level Control Logic

The details of the t-generator are shown in Figure 19. It consists of a counter and three shift registers, one for each level. The counter is initialized to zero via the *start* signal, and

is incremented at the beginning of each search by the *sel* signal. This continues until the counter overflows. The base shift register is also loaded at the beginning of each search, and all shift registers are shifted once for each step of the search. The t-values for each level are generated directly as the high-order bit of each shift register.

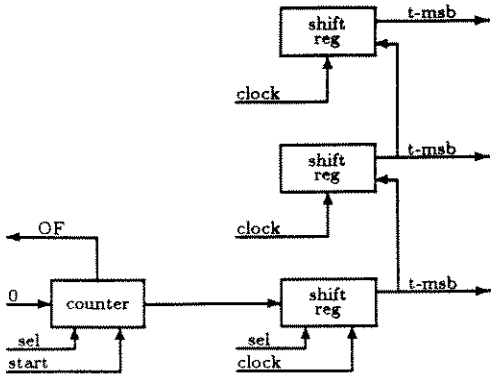



Figure 19. Binary Search T-Generator

The main control shown in Figure 20 is responsible for generating the *sel* and *clock* signals for the circuits described above. It also generates an address latch signal to the frame buffer to indicate a valid address, and a *done* signal back to the host. The inputs to the main control are the master clock and a *start* signal from the host. The 4-bit shift register is used to delay *addr-latch* generation until the first valid address emerges from the apex of the Triangle, and to delay at the end of the computation until the last address is generated.

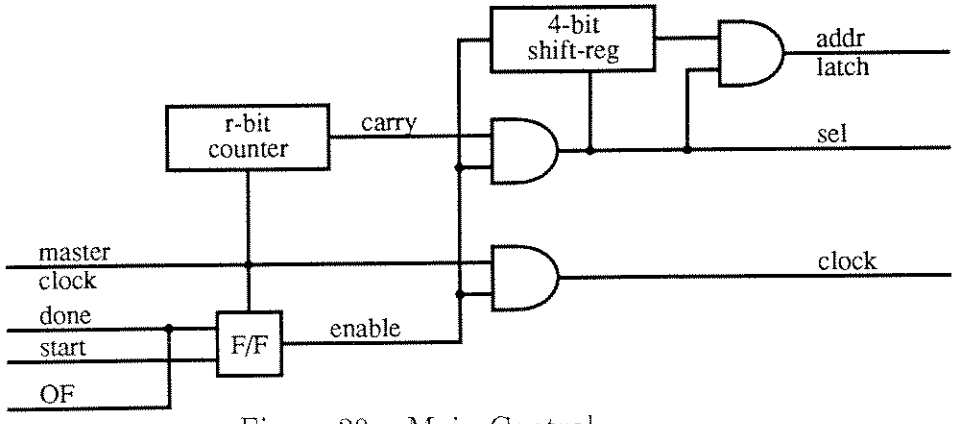



Figure 20. Main Control

As an inexpensive, yet high performance implementation of this binary search architecture, consider using off-the-shelf MSI components. In particular, we will use Fairchild FAST components [14]. The following table lists the required parts for a single processing element.

| | Part | Quantity |
|---|---|---|
| Registers | 74F399 | 8 |
| Adder | 74F381 | 4 |
| | 74F182 | 1 |

Processing Element Parts List

With these components the PE can perform one step of the search in 30 nsec., so overall performance for a system with a 16-bit $t$ value is one point every 480 nsec. For a 16-bit data path, and with standard DIP packages a PE takes 5.5 square inches. Thus, the twelve PE's to generate cubic Bézier curves will fit in an area of 65 square inches.

While this MSI implementation has very good performance, it does not have a very high level of integration. A good way to remedy this is to use a gate array. Consider the LSI Logics LCA10129 gate array [8], with 129,042 gates. A fair estimate of the number of PE's that will fit in this device can be obtained by counting the number of gates in the MSI implementation and comparing this to the number of gates in the array. We will assume a 40% utilization of gates in the device. In the MSI implementation described above there are approximately 730 gates per PE, while in the gate array there are approximately 50,000 gates. We can, then, fit at least 68 PE's in the gate array. For example, the 60 PE's required to generate a component of a degree 4 tensor product surface would easily fit in this device. In addition, the reduced gate delays of the LCA10129 will enable the PE's to operate two to three times faster, yielding a point generation time in the 160 to 240 nsec range.

## 8.2. A Scaled Computation Architecture

For our second example, we will present an architecture to compute Algorithm 3 in the form:

$$R \leftarrow R_0 + t(R_1 - R_0)$$

$$\mathbf{V}_{\text{out}} \leftarrow \mathbf{V}_L + R(\mathbf{V}_R - \mathbf{V}_L)$$

where all values are integers as discussed above. All quantities will be represented as 16-bit signed integers. This is sufficiently accuracy to allow values of R in the interval $[-4, 4]$, meaning that we can generate cubic Bézier, B-spline, Catmull-Rom, and Lagrange curves. The design of a complete system capable of generating these types of curves will be presented.

We begin by choosing a commercial component that can perform the required operations, and develop the processor element design around this device. We want this design to have very high performance, so we will choose the Bipolar Integrated Technology B3011 multiplier/accumulator (MAC) with a 15 nsec multiply-add time [20]. This device contains a 16x16 bit multiplier, a 32-bit adder, and two accumulators. By starting with this off-the-shelf component we simplify the task of implementation while achieving high performance and a high level of integration.

Figure 21 shows the internal architecture of the B3011 MAC, and Figure 22 shows how this device is used to form a processing element. The operations performed by this PE to carry out the required computation are given below. The operations on each line are performed in parallel, and in one clock cycle.

Initialize:

$$XB \leftarrow R_i^j(0);\ YB \leftarrow R_i^j(1) - R_i^j(0)$$

Loop:

$$XA \leftarrow \mathbf{V}_L;\ YA \leftarrow \mathbf{V}_R$$
$$A \leftarrow XA$$
$$YA \leftarrow YA - XA$$
$$B \leftarrow XB$$
$$XA \leftarrow B + t(YB)$$
$$\text{output } A + XA(YA)$$

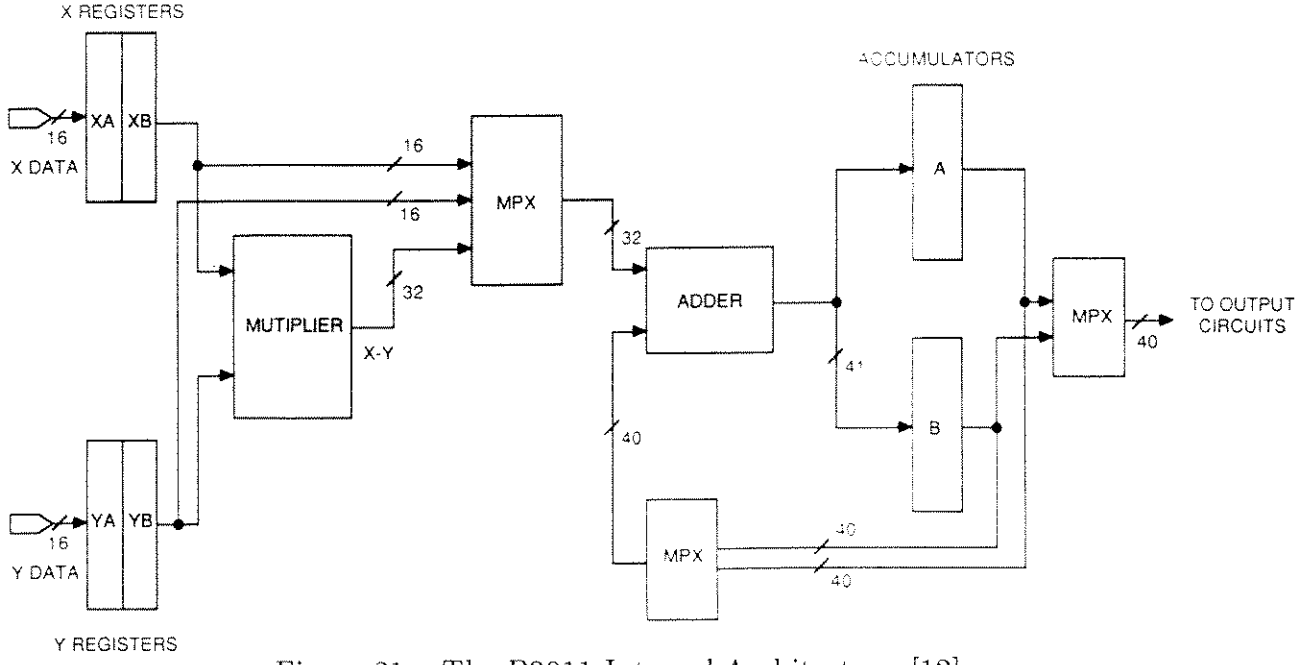



Figure 21. The B3011 Internal Architecture [12]

The initialization step is used to load the labels for a curve segment. Note that the quantity $(R(1) - R(0))$ may be precomputed by the host or may be computed in the PE as an additional initialization step. Figure 23 shows how these label values are loaded into the

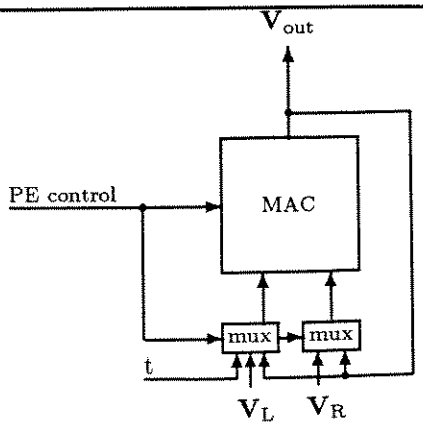
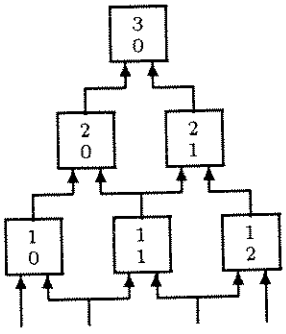



Figure 22. Processing Architecture for Scaled Computation

system, assuming that the subtraction is performed in the PE. The twelve labels are loaded in three time steps, then all of the PE's perform the subtraction in parallel overwriting R(1) with the result.




$t = 0:$ $R_0^3(0)$ $\Delta_0^2$ $\Delta_1^1$ $\Delta_0^3$

$t = 1:$ $R_0^2(0)$ $R_1^1(0)$ $R_1^2(0)$ $\Delta_1^2$

$t = 2:$ $R_0^1(0)$ $\Delta_0^1$ $R_2^1(0)$ $\Delta_2^1$

Figure 23. Label Initialization ($\Delta_i^j = R_i^j(1) - R_i^j(0)$)

In the first step of the loop we store the values output from the level below. Next, $\mathbf{V}_L$ is moved to accumulator A, and then $\mathbf{V}_R - \mathbf{V}_L$ is computed and stored back in register YA, overwritting the value of $\mathbf{V}_R$. Now we initialize accumulator B with the value of $R(0)$ and compute $R$ by loading $t$ in the left port. The value of R is then loaded into register XA in preparation for the final multiplication. The final output value is loaded into the next level by the first step of the next loop, so the loop takes five steps.

Now consider building a complete system with this processing element. There will be two triangles, each with six PE's interconnected as shown in Figure 24. Since all PE's perform the same operations, only one control circuit, the main control, is necessary. This is

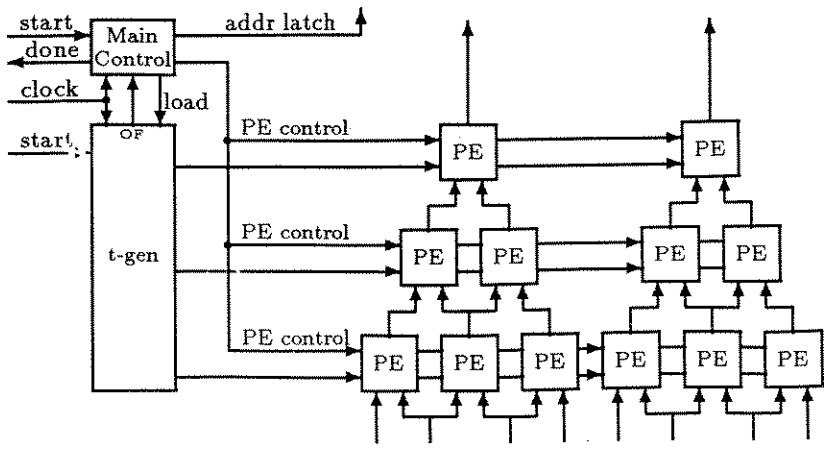



Figure 24. Complete System for Scaled Computation

in contrast to the previous example in which each level required a separate control circuit. The t-generator is also different for this example, as shown in Figure 25. This t-generator has three registers, one for each level, with the register at the base level capable of being incremented.

Conservatively, this implementation can produce a point every 75 nsec, or 13.3 million points per second. Compared to the previous example as implemented in MSI, it is about five times faster, takes about the same area, and can generate a much wider range of curve types. Like the MSI example, this architecture could benefit from a higher level of integration. A complete report on a VLSI implementation of the Triangle is forthcoming.

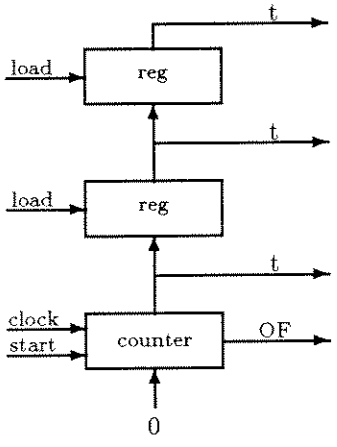



Figure 25. T-Generator

## 8.3. System Integration

Given a Triangle system as described above, it is a straight-forward matter to integrate this into a graphics system. If the host system has a single bus, the Triangle would be

connected as shown in Figure 26. The major drawback with this configuration is that the Triangle has to contend with the host for bus cycles when accessing the frame buffer. To alleviate this problem, we can add a separate bus for the frame buffer as shown in Figure 27. This configuration is typical of high performance graphics systems.

Refering to Figure 27, the host interface allows the loading of label and control point values into the Triangle via the host bus. This interface will provide the basic bus interface plus the logic necessary to decode addresses and load registers. Only two control signals are required between the host and the Triangle. The *start* signal to the Triangle indicates that all labels and control points have been loaded and the computation may begin, and the *done* signal from the Triangle to the host indicates the computation has finished.

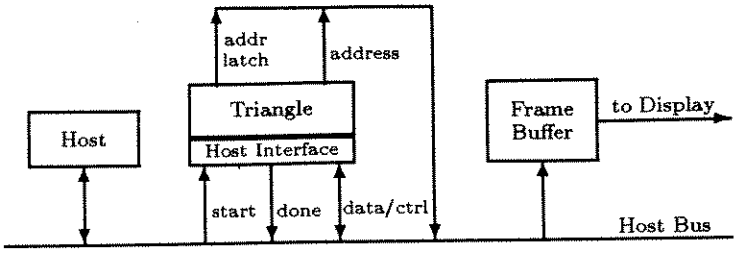



Figure 26. System Integration

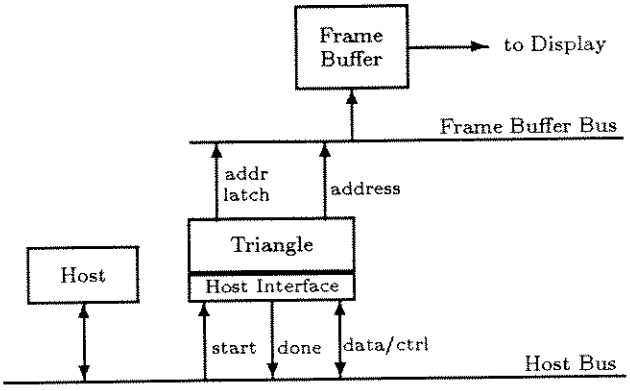



Figure 27. System Integration

The interface to the frame buffer is also quite simple. If the Triangle is given control points in device coordinates, then frame buffer addresses are formed simply by concatenating the x and y coordinates of the values produced at the apex of the Triangle. Viewing transformations such as perspective projection of surfaces can be accomplished with a small amount of additional arithmetic logic (*e.g.*, two dividers).

The only control signal between the Triangle and the frame buffer is a signal to indicate to the buffer that the next address is ready. The frame buffer is assumed to be at least as fast as the Triangle.

## 9. Conclusions

In this paper we have described the Triangle, a pipelined, parallel architecture that can be used as the basic building block of a multiprocessor for fast, numerically stable generation of a wide variety of geometric primitives including line segments, circles, spline curves, triangular patch surfaces, and tensor product surfaces.

Previously described processors have been capable of generating curves represented by a single, fixed polynomial basis. Consequently, host applications were forced to convert the basis being manipulated by the designer into the one implemented by the hardware. The Triangle, however, is unique in its ability to directly generate curves represented in a wide range of polynomial bases. Since virtually all bases in widespread use are directly implementable, host applications rarely need to execute costly change of basis algorithms, thereby improving the total throughput of the application.

A fully custom VLSI implementation of the Triangle is currently underway by a group of graduate students under the direction of Profs. Carl Ebeling and Lawrence Snyder at the University of Washington. Preliminary results suggest that a cubic scalar Triangle can be fabricated on a single chip running at approximately 2MHz, implying that a bicubic prism could be built with 15 chips. The design is expected to be completed and fully simulated sometime in the summer of 1987.

# Appendix 1: Labels for Common Curve Schemes

## Non-Uniform B-splines

In the most general form, a B-spline curve $\mathbf{Q}(t)$ of degree $d$ is a piecewise polynomial curve constructed from a sequence of control points $\mathbf{V}_k$, $k = 0, ..., m$, and sequence of non-decreasing real numbers $\bar{t} = \{t_r\}$, $r = 0, ..., m + d + 1$, according to [6]:

$$\mathbf{Q}(t) = \sum_{k=0}^{m} \mathbf{V}_k \, N_k^d(\bar{t}; t), \qquad t \in [t_d, t_{m+1}] \tag{9.1}$$

where the functions $N_k^d(\bar{t}; t)$ are the B-spline blending functions of degree $d$, which can be defined recursively by

$$N_k^d(\bar{t}; t) = \frac{t - t_k}{t_{k+d} - t_k} N_k^{d-1}(\bar{t}; t) + \frac{t_{k+d+1} - t}{t_{k+d+1} - t_{k+1}} N_{k+1}^{d-1}(\bar{t}; t). \tag{9.2}$$

where

$$N_k^0(\bar{t}; t) = \begin{cases} 1 & \text{if } t \in [t_k, t_{k+1}] \\ 0 & \text{otherwise.} \end{cases}$$

The sequence $\bar{t}$ is called the *knot vector*, and each value $t_r$ is known as a *knot*.

As mentioned above, $\mathbf{Q}(t)$ is a piecewise curve, the $\ell^{\text{th}}$ such piece $\mathbf{Q}_\ell(t)$ being swept out when the parameter $t$ varies on the interval $[t_{\ell+d}, t_{\ell+d+1}]$. Since the B-spline blending functions have *local support*, that is, $N_k^d(\bar{t}; t)$ is non-zero only over $d+1$ parametric intervals, the $\mathbf{Q}_\ell(t)$ depends only on a subset of the control points. In particular, $\mathbf{Q}_\ell(t)$ depends only on the control points $\mathbf{V}_\ell, ..., \mathbf{V}_{\ell+d}$.

With this notation, the Cox-deBoor algorithm [6] for generating B-spline curves can now be invoked to assign labels in the triangular computation for $\mathbf{Q}_\ell$. Specifically, the Cox-deBoor algorithm can be used to show that the right labels are given by

$$R_i^j(t) = \frac{t - t_{\ell+i+j}}{t_{\ell+d+i+1} - t_{\ell+i+j}} \tag{9.3}$$

for $j = 1, ..., d$ and $i = 0, ..., d - j$; each of the left labels can then be determined from

$$L_i^j(t) = 1 - R_i^j(t),$$

where $t$ is allowed to vary over the interval $[t_{\ell+d}, t_{\ell+d+1}]$. Since this parameter range violates restriction *(iii)* of labels (see Section 2), the curve must be reparametrized so that $t$ varies over the unit interval $[0, 1]$. The necessary reparametrization is

$$t \mapsto t_{\ell+d} + t(t_{\ell+d+1} - t_{\ell+d}).$$

Substituting this into Equation (9.3) yields

$$R_i^j(t) = \frac{t(t_{\ell+d+1} - t_{\ell+d}) + (t_{\ell+d} - t_{\ell+i+j})}{t_{\ell+d+i+1} - t_{\ell+i+j}}.$$

Finally, the base of the triangular computation consists of the control points $\mathbf{V}_\ell, ..., \mathbf{V}_{\ell+d}$.

In the special case of uniform B-splines, that is when the knots are uniformly spaced along the parameter line, the right labels for $\mathbf{Q}_\ell(t)$ simplify to

$$R_i^j(t) = \frac{t-i-j+d}{d-j+1}.$$

For the most common case of a uniform cubic B-spline (ie, $d = 3$), the $\ell^{\text{th}}$ segment is generated by the triangular computation shown in Figure 28.

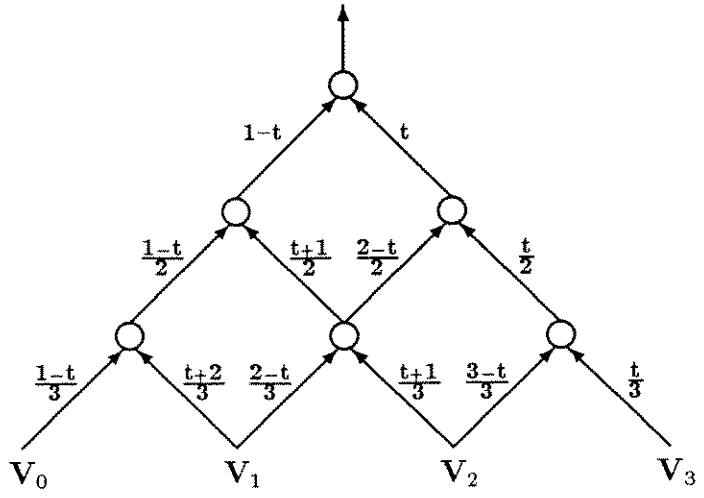



Figure 28. Labels for a Uniform Cubic B-spline

### Uniform Lagrange Curves

A uniform Lagrange curve $\mathbf{Q}(t)$ of degree $d$ is defined by a set of control points $\mathbf{V}_0, ..., \mathbf{V}_d$ according to

$$\mathbf{Q}(t) = \sum_{k=0}^{d} \mathbf{V}_k \mathrm{L}_k^d(t), \qquad t \in [0,1], \tag{9.4}$$

where

$$\mathrm{L}_k^d(t) = \prod_{\substack{r=0 \\ r \neq k}}^{d} \frac{dt-j}{i-j} \tag{9.5}$$

are the uniform Lagrange polynomials (see Boehm *et al*[6]). Aitken's algorithm can be used to compute points on Lagrange curves using triangular computations where the right labels are given by

$$R_i^j(t) = \frac{dt-i}{j}, \qquad j = 1, ..., d, \quad i = 0, ..., d-j,$$

and the left labels are given by

$$L_i^j(t) = 1 - R_i^j(t).$$

The labels for a uniform cubic Lagrange triangle are shown in Figure 29.

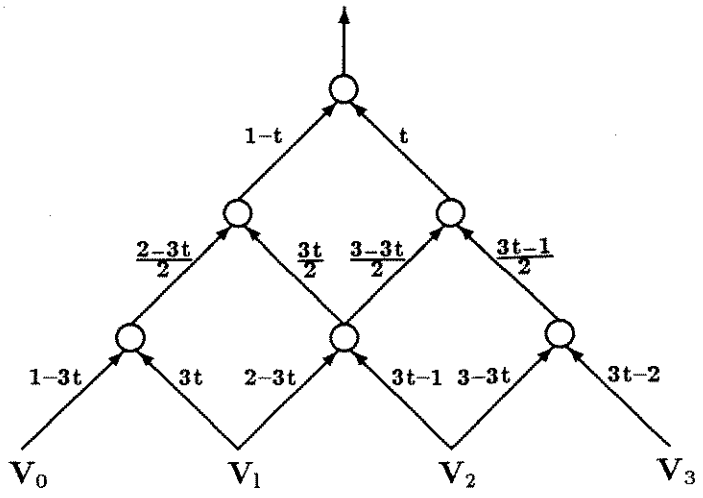



Figure 29. Labels for a Uniform Cubic Lagrange Curve

## Cubic Catmull-Rom Curves

A segment $\mathbf{Q}(t)$ of a cubic Catmull-Rom curve is defined by a set of control points $\mathbf{V}_0, \mathbf{V}_1, \mathbf{V}_2, \mathbf{V}_3$ according to

$$\mathbf{Q}(t) = \sum_{k=0}^{3} \mathbf{V}_k \phi_k(t), \qquad t \in [0,1] \tag{9.6}$$

where

$$\begin{aligned}
\phi_0(t) &= -\frac{t - 2t^2 + t^3}{2} \\
\phi_1(t) &= \frac{2 - 5t^2 + 3t^3}{2} \\
\phi_2(t) &= \frac{t + 4t^2 - 3t^3}{2} \\
\phi_3(t) &= -\frac{t^2 - t^3}{2}
\end{aligned}$$

are the cubic Catmull-Rom blending functions [9, 11]. These functions can be factored so that $\mathbf{Q}(t)$ as defined in Equation (9.6) can be generated using the triangular computation shown in Figure 30.

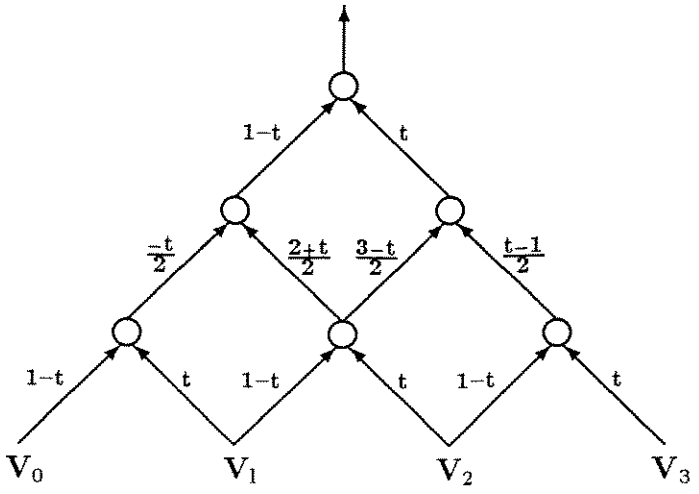



Figure 30. Labels for a Cubic Catmull-Rom Curve

## References


1. Phillip J. Barry, *Urn Models, Recursive Schemes, and Computer Aided Geometric Design*, Ph.D. Thesis, Department of Mathematics, University of Utah, Salt Lake City, Utah (June, 1987).

2. Phillip Barry, Tony DeRose, and Ronald Goldman, "B-splines, Pólya Curves, and Duality." Submitted for publication.

3. Brian A. Barsky, *The Beta-spline: A Local Representation Based on Shape Parameters and Fundamental Geometric Measures*, Ph.D. Thesis, University of Utah, Salt Lake City, Utah (December, 1981).

4. Brian A. Barsky, *Computer Graphics and Geometric Modelling Using Beta-splines*, Springer-Verlag, Tokyo (to appear).

5. Richard H. Bartels, John C. Beatty, and Brian A. Barsky, *An Introduction to Splines for Use in Computer Graphics and Geometric Modelling*, Morgan-Kaufmann Publishers, Inc., Los Altos, California (to appear).

6. Wolfgang Boehm, Gerald Farin, and Jürgen Kahmann, "A Survey of Curve and Surface Methods in CAGD," *Computer Aided Geometric Design*, Vol. 1, No. 1, July, 1984, pp. 1–60.

7. Wolfgang Boehm, "Curvature Continuous Curves and Surfaces," *Computer Aided Geometric Design*, Vol. 2, No. 4, 1985, pp. 313-323.

8. W. Carney and I. Curtin, "50K Gate Array Meets Tomorrows Design Challenges," *Digital Design*, March 1986, pp. 84–88.

9. Edwin E. Catmull and Raphael J. Rom, "A Class of Local Interpolating Splines," pp. 317-326 in *Computer Aided Geometric Design,* ed. Robert E. Barnhill and Richard F. Riesenfeld, Academic Press, New York (1974).

10. Fuhua Cheng, Kuen-Rong Hsieh, Rei-Ron Huang, and Yeh-Hao Chin, "Bézier Curve Generator: A Hardware Approach to Curve Generation," Proceedings of the *Second International Symposium of VLSI Technology, Systems, and Applications*, May 1985, Taipei, Taiwan, pp. 278–281.

11. Tony D. DeRose and Brian A. Barsky, "Geometric Continuity and Shape Parameters for Catmull-Rom Splines (Extended Abstract)," pp. 57-62 in *Proceedings of Graphics Interface '84,* Ottawa (27 May – 1 June, 1984).

12. *EDN*, June 11, 1987, pp. 115–126.

13. Gerald Farin, "Triangular Bernstein - Bézier Patches," *Computer Aided Geometric Design,* Vol. 3, No. 2, August 1987, pp. 83-127.

14. *FAST, Fairchild Advanced Schottky TTL*, Fairchild Corp., 1985.

15. Ronald N. Goldman, "An Urnful of Blending Functions," *IEEE Computer Graphics and Applications,* Vol. 3, No. 7, July 1983, pp. 49-54.

16. Ronald N. Goldman, "Pólya's Urn Model and Computer Aided Geometric Design," *SIAM Journal on Algebraic and Discrete Methods,* Vol. 6, No. 1, January, 1985, pp. 1–28.

17. Ronald N. Goldman, "Urn Models, Approximations, and Splines," *Journal of Approximation Theory,* to appear.

18. *INMOS Databook,* INMOS Corp., 1986.

19. Sheue-Ling Lien, Michael Shantz, and Vaughan Pratt, "Adaptive Forward Differencing for Rendering Curves and Surfaces," *Computer Graphics,* Vol. 21, No. 4, July 1987, pp. 111-118.

20. Robert E. Owen, "ECL MAC slashes system processing times by 80%," *Electronic Design,* October 30, 1986.

21. M. A. Penna and R. R. Patterson, *Projective Geometry and its Applications to Computer Graphics,* Prentice-Hall, Englewood Cliffs, N.J., 1985.