A PIPELINED ARCHITECTURE FOR RAY/BÉZIER PATCH INTERSECTION COMPUTATION

Renwei Wang[1]
School of Electrical Engineering and Computer Science
Washington State University

Robert R. Lewis
School of Electrical Engineering and Computer Science
Washington State University

Donald Hung
Department of Computer Engineering
San Jose State University

Keywords and Phrases: **pipelining, architecture, ray tracing, Bézier surfaces**
ACM Computing Classification categories: C.1, C.3, I.3.1, I.3.5

Abstract

We describe an algorithm for computing ray/Bézier patch intersections from a hardware design aspect. This algorithm uses patch subdivision and other geometrical techniques to find a given maximum number of intersection points nearest to the ray origin. We propose a pipeline-based hardware architecture, verify the number of pipeline stages required by simulation, and estimate the performance of a load-balanced implementation based on a state-of-the-art digital signal processor (DSP).

1 Introduction

Ray tracing is one of the most important rendering techniques in computer graphics for rendering high quality and photorealistic pictures. In it, at least one ray is cast into the scene for each pixel of the image. If the ray hits an object, it may produce other rays, such as reflection, shadow, and refraction rays to determine the color of the associated pixel (see [1] for more details). Ray tracing deals with thousands, if not millions, of such ray-object intersection calculations in each image and is thus well known for its exorbitant computation time.

[1] **currently at Synopsys, Inc.**

Xpatch is a software program that uses this graphical technique to model an incident radar signal and generate the resulting radar cross-sections of various objects. The object surfaces are modeled by different shape primitives such as meshes and parametric patches. Each type of primitive has its own algorithm to determine intersection with a given ray. Much work has been conducted at the Air Force Research Lab (AFRL) to determine which part of *Xpatch* is the most time consuming, finding that the majority of the time (between 50% and 80%) is spent in executing these algorithms.

Among commonly-used modelling primitives, Bézier patches stand out for being able to provide high accuracy of curved surfaces. Nevertheless, ray/Bézier patch intersection algorithms are computationally expensive. Many attempts have been made to improve them. The Bézier clipping algorithm [9] introduces an iterative geometric algorithm that finds all solutions of the ray-patch intersection problem up to an user definable accuracy, but it is hard to determine the number of iterative steps needed for a given patch. The Chebyshev boxing algorithm [5] replaces an original patch by many bilinear approximating patches, which will be subdivided into pieces repeatedly until the original patch has been well approximated. The preprocessing procedure required by this algorithm uses a recursive method to subdivide a given patch. The bounding volume hierarchy algorithm [3] combines the previous two by first carrying out the preprocessing procedure in the Chebyshev boxing algorithm and then calculating the tight bounding volumes.

In [12], one of us demonstrated that the depth of recursive calls using the Bézier clipping algorithm may vary greatly from patch to patch and from ray to ray. One patch may need a depth of three calls while another a depth of eleven, even though both resulted in a single intersection. All the above algorithms behave similarly, being based on recursive methods. Even if we were to implement the recursion by redundant hardware, they would still not be suitable for hardware acceleration because of the unpredictable resource allocation required.

Significant efforts to acclerate intersectio computation have also been made on the hardware side. As reported by Arvo and Kirk in 1989 [1], these include:

- LINKS-1, a 64-node (Intel 8086/8087) multiprocessor system that can be configured as a set of parallel pipelines to render a sequence of images.
- Kobayashi's work distributing the world database among a set of intersection processors.
- Goldsmith's work on a general-purpose multiprocessor system for ray-tracing on a hypercube.

More recent efforts include:

- Bouatouch's [2] exploitation of cache and virtual memory techniques to reduce message passing, yielding a speedup of about 60 on a 64-node MIMD
- Kin and Kyung's [8] ring-structured multiprocessors, allowing a linear speedup by partitioning objects among processors.

All the aforementioned hardware solutions are based on general-purpose architectures that are suitable for executing any kind of ray/object intersection problems.

Earlier research conducted by one of us at AFRL in Summer 1998 [6] optimized a ray/triangular facet intersection algorithm for hardware realization and also proposed the corresponding hardware architecture. This article may therefore be considered an extension of that work, focussing on a similar approach for accelerating ray/Bézier patch intersection. As before, efforts have been made to:

1. Develop an algorithm suitable for hardware/software acceleration.
2. Define the algorithm-specific system architecture for physical implementation in the future.

We therefore present a new algorithm based on a set of subtasks including patch subdivision and related geometric techniques to find a given maximum number of intersection points that are nearest to the ray origin. We also propose a pipelined system architecture based on this algorithm. The number of coarse-grained pipeline stages was verified by extensive simulation. For performance estimation, the pipeline was refined, load-balanced and mapped into multiple digital signal processors (DSPs) and custom-designed hardware. The results show that by reducing the grain scale of the pipeline and supporting the inherent parallelism of the computing tasks within its stages, a system throughput of over 10^6 ray/patch intersections per second is achievable.

Section 2 presents the proposed ray/Bézier patch intersection algorithm and reports our verification of that algorithm. Section 3 describes the functional blocks required for executing the algorithm, and proposes the overall system architecture without explicit specifications on software/hardware boundaries. Section 4 describes the performance estimation based on implementation models relying on off-the-shelf DSPs and limited ASIC for an identified computational bottleneck. Section 5 concludes the paper.

2 The Ray/Bézier Patch Intersection Algorithm

In this section, we will develop the mathematics of the patch intersection algorithm.

2.1 Background and Overview

Our study has been focused on a bicubic, nonrational, parametric Bézier patch, which we for simplicity will refer to hereafter as a 'patch'.

A ray, defined by an origin $\boldsymbol{O}$ and a direction $\boldsymbol{D}$., intersects a patch as shown in Figure 1. There are a maximum of 18 possible intersection points when a ray pierces a patch [7], but from a ray tracing standpoint, we are only interested in finding the intersection closest to $\boldsymbol{O}$. We therefore arbitrarily limit our interest here to successively finding and then refining (at most) N_i intersection points that are nearest to $\boldsymbol{O}$ and then selecting the closest of these as the answer.

One of the major difficulties of the ray/patch intersection problem is that exact solutions to the problem require finding roots of very high-degree polynomials, which cannot in general be done analytically and whose numerical solution is error-prone. Hence, we must look for approximate solutions.

Figure 2 shows how a mesh of 16 control points define a patch. The convex hull property states that the patch must lie inside its convex hull - the smallest (intuitively, "shrink-wrapped") polyhedron containing all the control points [13]. It is therefore a necessary condition for a ray to intersect a patch that the ray must intersect the patch's convex hull. It is not, however, a sufficient condition: a ray that passed through the convex hull may still miss the patch, but if we subdivide the patch into smaller patches and then consider the intersection of the ray with these smaller patches, missing the patch becomes less likely as the union of the convex hulls of the subpatches approaches the original patch. When a ray hits the convex hull of a sufficiently small patch, the center of the patch can be considered as a good approximation to the ray/patch intersection.

To find a convex hull intersection, then, we first consider the projection-based approach illustrated in Figures 3 through 6. Imagine that a ray is contained in the intersection of two perpendicular planes U and V, and that these, the patch, and its convex hull are all projected onto a third plane perpendicular to the ray direction. We use the U and V intersections to define a coordinate system whose origin is the projection of the ray.

In the projected 2D coordinate system, if the 16 projected control points are located on the same side of either one of the two axes, the ray *must* have missed the convex hull of the testing patch (Figures 4 and 5). Otherwise the ray *may* hit the convex hull (Figure 6).

Based on the above discussion, we outline the ray/patch intersection algorithm we will use:

1. Divide an input patch into $4N_i$ subpatches. (If N_i is 4, this could be done with two four-way subdivision levels.)

2. Find all subpatches that are possibly intersected by the ray, based on the convex hull property.

3. If no subpatches pass the convex hull test, exit with a null intersection result.

4. From all possibly-intersecting subpatches, find the (at most) $\mathrm{N_i}$ subpatches that are closest to the origin of the ray.

5. If the subpatches are sufficiently small, exit with the center of the closest patch to the origin as the intersection.

6. Subdivide the $\mathrm{N_i}$ subpatches to obtain $4\mathrm{N_i}$ smaller subpatches.

7. Repeat steps 2 through 7.

2.2 Notation and Preliminaries

A ray **R** is represented in the form $\boldsymbol{R} = \boldsymbol{O} + \boldsymbol{D} t$. For our purposes, we will assume that **D** is normalized. A plane in Euclidean 3-space is defined (as usual) by the equation

$$Ax + By + Cz + E = 0 \tag{1}$$

Within a nonzero scaling factor, the normal to such a plane is the vector (A, B, C) .

A nonrational parametric Bézier patch of degree *m* is defined by

$$\boldsymbol{Q}(u,v) = \sum_{i=0}^{n} \sum_{j=0}^{m} \boldsymbol{P}_i, j B_j, m(u) B_i, n(v) \quad \text{for} \quad 0 \leq u, v \leq 1 \tag{2}$$

where $\boldsymbol{P_{ij}} = (x_{ij}, y_{ij}, z_{ij})$ are the patch's control points, and the blending functions are the Bernstein polynomials:

$$B_j, m(u) = \binom{m}{j} u^j (1-u)^{(m-j)} \qquad \binom{m}{j} \equiv m \frac{!}{j!(m-j)!} \quad , \tag{3}$$

In matrix form, (2) becomes

$$\boldsymbol{Q}(u,v) = \boldsymbol{U} \boldsymbol{M} \boldsymbol{P} \boldsymbol{M}^t \boldsymbol{V} \tag{4}$$

where $\boldsymbol{U} = \begin{bmatrix} u^m & u^{m-1} & \ldots & 1 \end{bmatrix}$, $\boldsymbol{V} = \begin{bmatrix} v^m & v^{m-1} & \ldots & 1 \end{bmatrix}$, ***P*** is the control point matrix, ***M*** is the Bézier *basis matrix*, and ***M***^{t} is the transpose of ***M***.

The bicubic patch is a special case of (4) with *n* = *m* = 3 where

$$\boldsymbol{M} = \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \quad \text{and} \quad \boldsymbol{P} = \begin{bmatrix} P_{00} & P_{01} & P_{02} & P_{03} \\ P_{10} & P_{11} & P_{12} & P_{13} \\ P_{20} & P_{21} & P_{22} & P_{23} \\ P_{30} & P_{31} & P_{32} & P_{33} \end{bmatrix} \quad .$$

2.3 Plane Generation

Given a ray **R**, we need to find two mutually perpendicular planes U and V that contain the ray. If **N**$_u$ and **N**$_v$ are the normals of the two planes, respectively, then **N**$_u$, **N**$_v$, and **D** are all mutually perpendicular, as illustrated in Figure 79.

Of course, an infinite number of vectors are perpendicular to **D**. A simple and robust algorithm to choose two mutually perpendicular ones is:

1. let ξ be x, y, or z such that $D_\xi = min\left(|D_x|,|D_y|,|D_z|\right)$

2. let **T** be the unit vector in the ξ direction (i.e., $T_i = \delta_{i\xi}$)

3. $\boldsymbol{N_u} = \boldsymbol{T} \times \boldsymbol{D}$

4. $\boldsymbol{N_v} = \boldsymbol{N_u} \times \boldsymbol{D}$

This could easily be implemented in hardware, especially since the first cross-multiplication is simply a rearrangement of the elements of **D**.

2.4 Patch Subdivision

The patch subdivision algorithm described here is based on the de Casteljau algorithm [5]. Control points of all the subpatches are generated by matrix multiplications which are easily and parallelizably performed in hardware. We decompose the algorithm into two steps.

In the first step, we subdivide the range of the parameter *u*, deriving left ($0 \le u < 1/2$) and right ($1/2 \le u < 1$) subpatch control vertices P^L and P^R respectively. The left subpatch $\mathbf{Q}^L$ is defined by:

$$\boldsymbol{Q}^L(u,v) = \boldsymbol{Q}\left(\frac{u}{2},v\right) \quad . \tag{5}$$

Using (4), we can rewrite this as

$$\boldsymbol{U}\boldsymbol{M}\boldsymbol{P}^{L}\boldsymbol{M}^{t}\boldsymbol{V}=\begin{bmatrix}\frac{u^3}{8} & \frac{u^2}{4} & \frac{u}{2} & 1\end{bmatrix}\boldsymbol{M}\boldsymbol{P}\boldsymbol{M}^{t}\boldsymbol{V}=\boldsymbol{U}\frac{1}{8}\begin{bmatrix}1&0&0&0\\0&2&0&0\\0&0&4&0\\0&0&0&8\end{bmatrix}\boldsymbol{M}\boldsymbol{P}\boldsymbol{M}^{t}\boldsymbol{V} \tag{6}$$

Since this must be true for all **U** and **V**, we infer that

$$\boldsymbol{P}^{L}=\boldsymbol{M}^{-1}\frac{1}{8}\begin{bmatrix}1&0&0&0\\0&2&0&0\\0&0&4&0\\0&0&0&8\end{bmatrix}\boldsymbol{M}\boldsymbol{P}=\boldsymbol{D}^{L}\boldsymbol{P}\ \text{ where }\ \boldsymbol{D}^{L}=\frac{1}{8}\begin{bmatrix}8&0&0&0\\4&4&0&0\\2&4&2&0\\1&3&3&1\end{bmatrix}\ . \tag{7}$$

The right subpatch $\mathbf{Q}^{R}$ is defined by:

$$\boldsymbol{Q}^{R}(u,v)=\boldsymbol{Q}\left(\frac{u+1}{2},v\right)\ . \tag{8}$$

Again using (4), we find that

$$\boldsymbol{P}^{R}=\boldsymbol{M}^{-1}\frac{1}{8}\begin{bmatrix}1&0&0&0\\3&2&0&0\\3&4&4&0\\1&2&4&8\end{bmatrix}\boldsymbol{M}\boldsymbol{P}=\boldsymbol{D}^{R}\boldsymbol{P}\ \text{ where }\ \boldsymbol{D}^{R}=\frac{1}{8}\begin{bmatrix}1&3&3&1\\0&2&4&2\\0&0&4&4\\0&0&0&8\end{bmatrix}\ . \tag{9}$$

In the second step, we apply the same method in the *v* direction, creating the top and bottom patch control vertices P_B^T and P_B^B . It is easy to show that these are related to a control vertex matrix **P'** via $\mathbf{P}^{T}$ = $\mathbf{P'}\mathbf{D}^{T}$ and $\mathbf{P}^{B}$ = $\mathbf{P'}\mathbf{D}^{B}$, where

$$\boldsymbol{D}^{T}=\frac{1}{8}\begin{bmatrix}8&4&2&1\\0&4&4&3\\0&0&2&3\\0&0&0&1\end{bmatrix}=\left[\boldsymbol{D}^{L}\right]^{t}\ \text{ and }\ \boldsymbol{D}^{B}=\frac{1}{8}\begin{bmatrix}1&0&0&0\\3&2&0&0\\3&4&4&0\\1&2&4&8\end{bmatrix}=\left[\boldsymbol{D}^{R}\right]^{t}\ . \tag{10}$$

Combining the two steps, we derive the control vertex arrays of all four subpatches as

$$\begin{bmatrix} \boldsymbol{P}^{TL} & \boldsymbol{P}^{TR} \\ \boldsymbol{P}^{BL} & \boldsymbol{P}^{BR} \end{bmatrix} = \begin{bmatrix} \boldsymbol{D}^{L} & \boldsymbol{D}^{R} \end{bmatrix} \boldsymbol{P} \begin{bmatrix} \boldsymbol{D}^{T} \\ \boldsymbol{D}^{B} \end{bmatrix} \ , \tag{11}$$

which is easily done in parallel.

2.5 Hull Classification

In Section 2.1 we showed how to determine whether a given ray hits the convex hull of a given patch by looking at the allocation of the projected patch control points in the ray-based projected 2D coordinate frame. While conceptually useful, we can be more efficient at implementation by making four observations.

First, we do not need to do the projections. We need only determine whether a given control point lies above or below the U or V plane. This is a simple evaluation of the plane equation using the control point. Recall that in a 3D space, the distance of a point **P** at (*x, y, z*) to a plane as given in (1) is

$$d = \frac{Ax + By + Cz + D}{\sqrt{A^2 + B^2 + C^2}} \tag{12}$$

This distance is either negative or positive depending on which side of the plane the point **P** is located. The normal defines the positive direction. The distance is zero if **P** is on the plane.

Second, we do not need the actual distance, only need its sign to determine whether the point is above or below the plane. (We consider the infrequently occurring case where the control point is contained within the plane to be "above" for hull classification purposes.)

Since we are only interested in the sign of *d*, which is solely determined by the numerator of the right-hand side of (12), we can define the *sign distance* as

$$d^s = Ax + By + Cz + D \tag{13}$$

or, in matrix form,

$$d^s = \begin{bmatrix} A & B & C & D \end{bmatrix} \begin{bmatrix} \boldsymbol{P} \\ 1 \end{bmatrix} \tag{14}$$

Our hull classification algorithm says that, for either U or V, if the signs of the sixteen sign distances are either all positive or all negative, the ray misses the patch. If this is untrue for both planes, the ray may hit the convex hull of the patch and hence it may also intersect the patch itself.

A third observation is that we do not necessarily need to test all 16 points. Finding two points on opposite sides of the plane would imply possible intersection, but as this requires sequential evaluation, we cannot take advantage of this observation.

Fourth, the sign calculations for both planes can also be done simultaneously.

The explicit computations for calculating the 16 sign distances with respect to the planes U and V are

$$\begin{aligned} \boldsymbol{d_u^s} &= \begin{bmatrix} A_u & B_u & C_u & D_u \end{bmatrix} \begin{bmatrix} p_{00x} & p_{01x} & \cdots & p_{32x} & p_{33x} \\ p_{00y} & p_{01y} & \cdots & p_{32y} & p_{33y} \\ p_{00z} & p_{01z} & \cdots & p_{32z} & p_{33z} \\ 1 & 1 & \ldots & 1 & 1 \end{bmatrix} . \\ \boldsymbol{d_v^s} &= \begin{bmatrix} A_v & B_v & C_v & D_v \end{bmatrix} \begin{bmatrix} p_{00x} & p_{01x} & \cdots & p_{32x} & p_{33x} \\ p_{00y} & p_{01y} & \cdots & p_{32y} & p_{33y} \\ p_{00z} & p_{01z} & \cdots & p_{32z} & p_{33z} \\ 1 & 1 & \ldots & 1 & 1 \end{bmatrix} \end{aligned} \tag{15}$$

We define the function:

$$Sgnsum(\boldsymbol{d}^s) = \sum_{i=0}^{15} \begin{cases} -1 & \text{if } d_i^s < 0 \\ 1 & \text{if } d_i^s \geq 0 \end{cases} \tag{16}$$

If either $\left|Sgnsum(\boldsymbol{d}_u^s)\right|$ or $\left|Sgnsum(\boldsymbol{d}_v^s)\right|$ is equal to 16, the ray cannot intersect the patch. Otherwise, the patch needs to be further divided into smaller subpatches for a more accurate approximation.

2.6 Patch Center

As stated in Section 2.1, it is necessary to find the center of a given patch for two reasons. First, an intersection is approximated by the center of a sufficiently small patch. Second, in finding the approximated intersection points, our algorithm must sort out four subpatches (from all possibly-hit subpatches) closest to the origin of the ray. We assume here that "closeness" means the distance between the origin of the ray and the center of a given patch.

One way to define the center of the patch would be to take the arithmetic mean of the control points. This would be simple to compute, but we have no guarantee that this point actually lies on the patch, a desirable feature for the final intersection approximation.

Instead, we observe from (4) that for a given patch **Q**(u, v), an unambiguous "parametric center" ***C*** can be defined:

$$\boldsymbol{C} \equiv \boldsymbol{Q}\left(\frac{1}{2}, \frac{1}{2}\right) = \boldsymbol{U}\boldsymbol{M}\boldsymbol{P}\boldsymbol{M}^t\boldsymbol{V} \text{ where } \boldsymbol{U} = \boldsymbol{V}^t = \begin{bmatrix} \frac{1}{8} & \frac{1}{4} & \frac{1}{2} & 1 \end{bmatrix} . \tag{17}$$

Based on the above, we can derive

$$C = \frac{1}{64}\begin{bmatrix}1 & 3 & 3 & 1\end{bmatrix}\begin{bmatrix}\boldsymbol{p}_{00} & \boldsymbol{p}_{01} & \boldsymbol{p}_{02} & \boldsymbol{p}_{03}\\ \boldsymbol{p}_{10} & \boldsymbol{p}_{11} & \boldsymbol{p}_{12} & \boldsymbol{p}_{13}\\ \boldsymbol{p}_{20} & \boldsymbol{p}_{21} & \boldsymbol{p}_{22} & \boldsymbol{p}_{23}\\ \boldsymbol{p}_{30} & \boldsymbol{p}_{31} & \boldsymbol{p}_{32} & \boldsymbol{p}_{33}\end{bmatrix}\begin{bmatrix}1\\3\\3\\1\end{bmatrix} \tag{18}$$

2.7 Patch Distance

We define *patch distance* to be the distance between the origin of the ray and the center of a given patch. Given two points $\boldsymbol{P_1} = (x_1, y_1, z_1)$ and $\boldsymbol{P_2} = (x_2, y_2, z_2)$ in 3-space, the distance between them is normally defined by an L_2 (Euclidean) metric:

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2} \tag{19}$$

However, to simplify the computation, we redefine the distance as an L_1 (Manhattan) metric:

$$d = |x_2 - x_1| + |y_2 - y_1| + |z_2 - z_1| \tag{20}$$

The distance computation from the ray origin must also include a sign. Positive signs indicate positive ray parameter (*t*) values. If the cosine of the angle between **D** and the vector from **O** to **C** (given by $\frac{(\boldsymbol{C} - \boldsymbol{O}) \cdot D}{|\boldsymbol{C} - \boldsymbol{O}||D|}$**)** is positive, the ray points towards (i.e., may hit) the patch, otherwise it points away from (i.e., definitely misses) the patch.

Again, since we are only interested in the sign of the cosine, all we need to calculate is **(C-O)**·**D**, so we can sort patches using the *signed ray/patch distance*

$$d^p = \begin{cases} d & \text{if } (\boldsymbol{C} - \boldsymbol{O}) \cdot \boldsymbol{D} \geq 0 \\ -1 & \text{if } (\boldsymbol{C} - \boldsymbol{O}) \cdot \boldsymbol{D} < 0 \end{cases} \tag{21}$$

Using -1 as a flag allows us to remove that particular candidate from further processing.

2.8 The Culler

In our intersection algorithm, each of four patches is divided into four subpatches, a total of 16 subpatches. Each subpatch is then evaluated to see if its convex hull intersects with the ray (see Section 2.1). From all qualifying subpatches, the N_i (at most) with the shortest ray/patch distance will be selected for further approximation. The sorting task is accomplished using *Quicksort*, a standard sorting algorithm with $\Theta(n \lg n)$ run time efficiency on an input array of size *n*. This combination of sorting and selection constitutes the "Culler" module.

2.9 Summary: An Algorithm for Finding the Ray/ Patch Intersection

We now present an implementation of the ray/patch intersection algorithm described in Section 2.1. As mentioned above, we have taken N_i to be 4. We also use intermediate buffers to hold patches.

1. Subdivide the input patch into four subpatches (Section 2.4) and put them into *buffer1*, which can store at most four subpatches. Generate planes U and V from the ray parameters (Section 2.3).

2. If the desired number of iterations is reached, go to Step 5. Otherwise, subdivide each patch in *buffer1* into four subpatches (Section 2.4) and store them in *buffer2*, which can hold at most 16 subpatches.

3. Evaluate each patch in buffer2 by hull classification (Section 2.5). Calculate the centers of the patches (Section 2.6) and their distances to the origin of the ray (Section 2.7).

4. Sort the patches in *buffer2* to find the four patches closest to (but not "behind", i.e. $t \geq 0$) the ray origin (Section 2.8) and put them into *buffer1*. Then go back to Step 2.

5. Output the center of the closest of the four subpatches.

Assuming that there is an intersection point, after *n* iterations the resulting subpatch will be approximately $1/2^n$ of the original (linear) patch dimensions. As long as the input patch is reasonably small, the four final subpatches will converge to a tiny area, the center of the closest of which will be considered the intersection point.

2.10 Test of the Proposed Algorithm

In order to verify our algorithm, we developed a user interface in which the user can specify arbitrary rays and patches. Using the standard "teapot" patches as the benchmark, we also compared the results obtained from our algorithm with those from Nishita's algorithm [9] (based on the Bézier clipping technique). We found that the intersection points generated by the two algorithms are within the accuracy specified for the Bézier clipping approach.

3 Functional Building Blocks and the System Architecture

In this section, we first define the functional building blocks required by the algorithm discussed in Section 2. We then propose a pipelined system architecture to implement this. Our purpose here is to specify the functionality of each building block as well as the structure and operation of the overall system, rather than the hardware/software boundaries and implementation details.

3.1 Functional Building Blocks

All the functional blocks required by the proposed ray/patch intersection algorithm are listed in Table 2. After coding them as C functions, we obtained the execution times shown in Figure 9. These were measured on a PC Workstation with a 350MHz Intel Pentium II-MMX processor using the Windows^{TM} NT 4.0 operating system with networking activities disabled and the Visual C++ compiler version 6.0 with the *Maximize Speed* option on.

These data reveal the load distribution among the different computing tasks and serve as guidelines for balancing the pipeline stages and determining the hardware/software partition.

3.2 System Architecture

Based on these functional building blocks, the overall system architecture for physical implementation of our algorithm is shown in Figure 10, where the block M2 contains three functional blocks listed in Table 1: HullClassification, PatchCenter, and PatchDistance.

The proposed architecture contains eleven pipelined stages. Stage 0 executes Step 1 of the algorithm; Stages 1 to 10 execute the loop body of the algorithm. They are identical except that stage 10 outputs centers of the last four subpatches that will be taken as the intersection points. Within each stage, the inherent parallelism of the algorithm is exploited by the functional building blocks. The number of stages was determined based on observations of the algorithm's convergence rate during the software emulation process. Throughput of the overall system will be determined by the latency of a single stage.

Note that this architecture illustrates the datapath stream of our algorithm with coarse-grained pipeline stages. It is open to refined intra-stage pipelining and hardware/software partitioning, based on specific performance requirements and implementation constraints determined by different applications.

4 Performance Analysis

Our evaluation results were obtained from experiments except for the culler module (which was identified as the computational bottleneck and thus implemented in hardware). All the other functional building blocks were implemented in software and executed on off-the-shelf TMS320C6701 (hereafter “6701”) digital signal processors (DSPs). We chose the 6701 because it is representative to the latest generation of high-performance DSPs, and its development tools are readily available. Although our analysis is based on a specific implementation model, it provides a good insight for future efforts in physical implementation.

4.1 Code Optimization for the Functional Blocks

All function calls listed in Table 1 were coded and optimized using the 6701 development tool set Code Composer Studio. When profiling the function calls, all data are declared as single precision floating-point numbers, and all data storage was kept word-aligned. We took multiple steps to optimize the function calls. The functional block *PatchSubdivider* is used as an example to show the procedure and effectiveness of the optimization techniques. Table 2 lists the optimization techniques applied and Figure 8 shows the function’s 6701 runtime after the application of each technique.

Source code for all the other functional blocks was optimized in a similar fashion. Runtimes of the optimized function calls on 6701 are listed in Table 3.

4.2 An ASIC Realization of the Culler Module

From Table 3, it is clear that Culler optimized for the 6701 is still the computational bottleneck of the proposed algorithm. Consequently, a hardware implementation of that module became necessary.

To increase the sorting efficiency and reduce the memory requirements, ray/patch distance is used as the sorting key. In order to identify its corresponding patch, each sorting key must carry a tag, as shown in Figure 11: the ray/patch distance in IEEE 754 single precision floating-point format occupies bits 0 to 31 and a 4-bit *Patch Index* is prepended to the beginning (bits 32 to 35).

Two basic functional units are defined for the sorting task: *upsort* and *downsort*. Both units take two sorting keys *a* and *b* as inputs, and have two outputs labeled *up* and *down*. For the unit *upsort*, its *up* output is max(*a*, *b*) and the *down* output is min(*a*, *b*); for the unit *downsort,* its outputs are defined oppositely. As the sorting task has been identified as a computational bottleneck (see Figure 9), symbols, schematics and function tables of the *upsort* and the *downsort* are shown in Figure 12 for possible ASIC realization. Note that, since a negative ray/patch distance (when bit 31 in Figure 11 is 1) means the ray runs away from the patch (see Section 2.7), it is always treated as "greater" than a positive distance and therefore will be eliminated by the *Culler*.

The logic expressions for the signal *upsel* are $upsel = \overline{a[31]}(b[31] + \overline{\mathrm{GT}})$ and $downsel = \overline{b[31]}(a[31] + \overline{\mathrm{GT}})$ in *upsort* and *downsort*, respectively.

Using upsort and downsort as elements, a bitonic sorting network is constructed as shown in Figure 20. It inputs 16 sorting keys and outputs the tags of the four keys with the smallest ray/patch distance values. Sorting elements in the dashed blocks are not required since only four (= N_i) outputs are needed.

Based on the bitonic sorting network, *Culler* can be completed as shown in Figure 13.

Without affecting timing, a scaled-down (12-bit) version of *Culler* was implemented using Xilinx FPGA devices (XC4025E).

The total delay of the 10-stage sorting network is at most 800ns without any intra-stage pipelining.

4.3 Performance Estimation

Using the overall system architecture proposed in Section 3.2 as the framework, different implementation cases have been studied.

In this implementation, performance estimation based on a fine-grain pipelined implementation is discussed here. Each of the pipeline stages shown in Figure 22 is divided into five substages and contains multiple 6701s except Substage 5 (the Culler), which is implemented in hardware as discussed in Section 4.2. The refined pipeline stages are illustrated in Figure 13, and the detailed information on Stage 1 is given in Table 4, where the performance estimation is based on the execution time required by Substages 1 and 2 as they have the largest latency.

This implementation leaves a lot of room for further expedition of the execution. For instance, optimizing the code for the function *SplitInHalf* at the assembly language level could reduce the number of 6701 cycles required by Substages 1 and 2 and thus the pipeline cycle time could be shortened.

5 Conclusion

In this study, we have presented an algorithm for solving the ray/Bézier patch intersection problem. Although its computational complexity is on the same order of some other known algorithms, the proposed algorithm has the advantage of being suitable for parallel and pipelined execution. The corresponding systems architecture for physical realization has also been developed with all the functional building blocks specified.

We profiled computing tasks required by this algorithm on the 6701 DSP and identified the computational bottleneck. To eliminate the bottleneck, we proposed and verified a hardware solution. We obtained performance data based on two implementation models built with the 6701s, with and without additional custom hardware. These show that with the proposed algorithm, more than one million ray-patch intersections per second could be achieved on specially-designed computing systems based on the proposed architecture, with the pipeline stages being refined and the inherent parallelism of computing tasks inside the pipeline stages being supported.

The data also suggest that higher performance can be achieved by optimizing the code at assembly level (for functional blocks implemented in software) or more efficiently, by replacing the functional building blocks with custom-designed hardware.

We anticipate that with the current level of rapid technological advances, practical systems based on this study could be built in the near future.

Acknowledgments

This work was partially supported by the U. S. Air Force under AFOSR contract number F49620-93-C-0063.

We are indebted to Alain Fournier for his assistance in deriving the patch subdivision formulae, to Wayne Cochran for his initial implementation of the Nishita algorithm, and to Shuyang Li for her help in adapting the Nishita algorithm and in implementing the software version of the proposed algorithm.

References

[1] J. Arvo and David Kirk, “A Survey of Ray Tracing Acceleration Techniques”, Chapter 6 in *An Introduction to Ray Tracing,* (A.S. Glassner Ed.), Academic Press, 1989.

[2] Didier Badouel and Kadi Bouatouch, “Distributing Data and Control for Ray Tracing in Parallel”, *IEEE Computer Graphics and Applications*, 14(4), pp. 69-77, July 1994.

[3] Swen Campagna, Philipp Slusallek, and Hans-Peter Seidel, “Ray Tracing of Spline Surfaces: Bézier Clipping, Chebyshev Boxing, and Bounding Volume Hierarchy - A Critical Comparison With New Results”, *The Visual Computer*, Vol. 13, pp. 265-282, 1997.

[4] James D. Foley, Andries van Dam, Steven Feiner, and John Hughes, *Computer Graphics: Principles and Practice*, 2nd Edition, Addison Wesley, 1997.

[5] Alain Fournier and John Buchanan, “Chebyshev Polynomials for Boxing and Intersections of Parametric Curves and Surfaces”, *Computer Graphics Forum*, 13(3) (EUROGRAPHICS ’94 Proceedings), pp. 129-142.

[6] Donald L. Hung, "A Study on Accelerating the Ray/Triangular Facet Intersection Computation in Xpatch", Technical Report to the U.S. Air Force, August, 1998.

[7] James T. Kajiya, “Ray Tracing Parametric Patches”, *Computer Graphics*, 16(3), pp. 245-254, July 1982.

[8] Hyun-Joon Kim and Chong-Min Kyung, “A New Parallel Ray-Tracing System Based on Object Decomposition”, *The Visual Computer*, Vol. 12, pp. 244-253, 1996.

[9] Tomoyuki Nishita, Thomas W. Sederberg, and Masanori Kakimoto, “Ray Tracing Trimmed Rational Surface Patches”, *Computer Graphics*, 24(4), pp. 337-345, August 1990.

[10] *TMS320C62x/67x Programmer’s Guide*, Texas Instruments, 1999.

[11] *TMS320C6000 Code Composer Studio Tutorial*, Texas Instruments, 1999.

[12] Renwei Wang, *A Study on Accelerating the Ray/Bézier-Patch Intersection Calculation in Xpatch*, MS Thesis, School of EECS, Washington State University, 2000.

[13] Alan Watt and Mark Watt, *Advanced Animation and Rendering Techniques: Theory and Practice*, Addison Wesley, 1998.

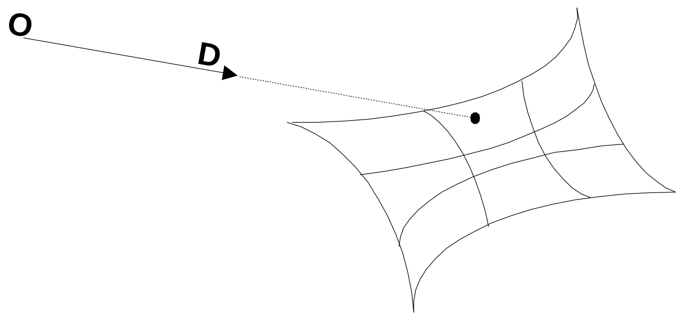

Figure 1: A Ray/Patch Intersection

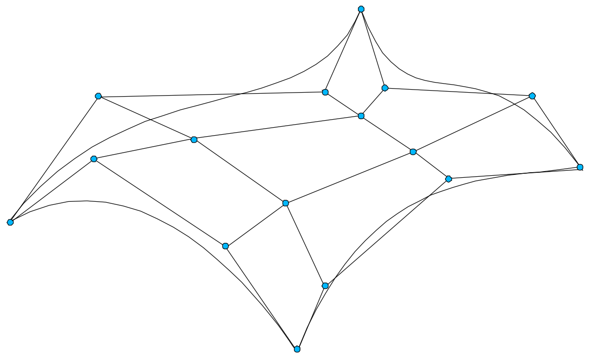

Figure 2: The Control Mesh of a Bézier Patch

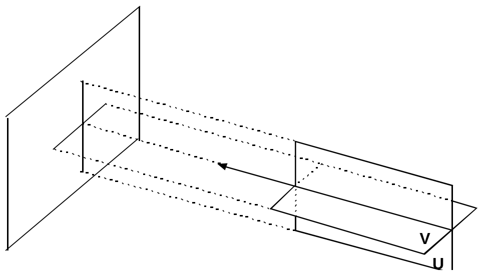

Figure 3: The Concept of Virtual Projection

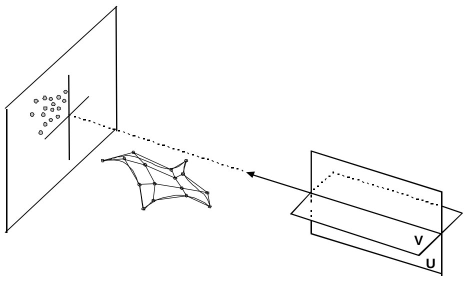

Figure 4: Case 1 - The Ray Misses the Patch Vertically

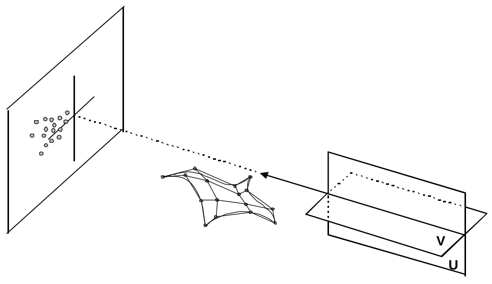

Figure 5: Case 2 - The Ray Misses the Patch Horizontally

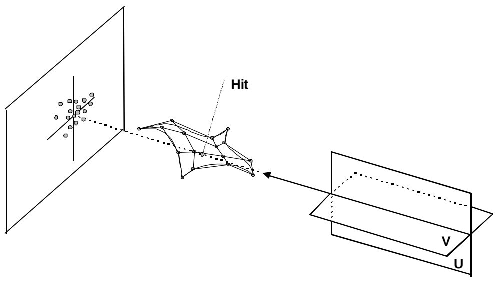

Figure 6: Case 3 - The Ray May Hit the Convex Hull of the Patch. (In this case, it does hit it.)

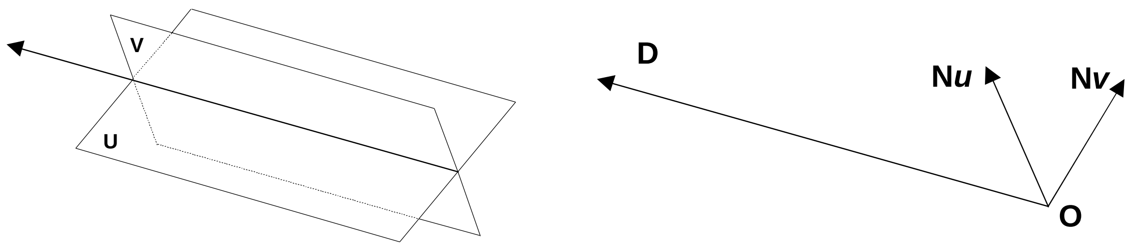

Figure 7: Plane Generation

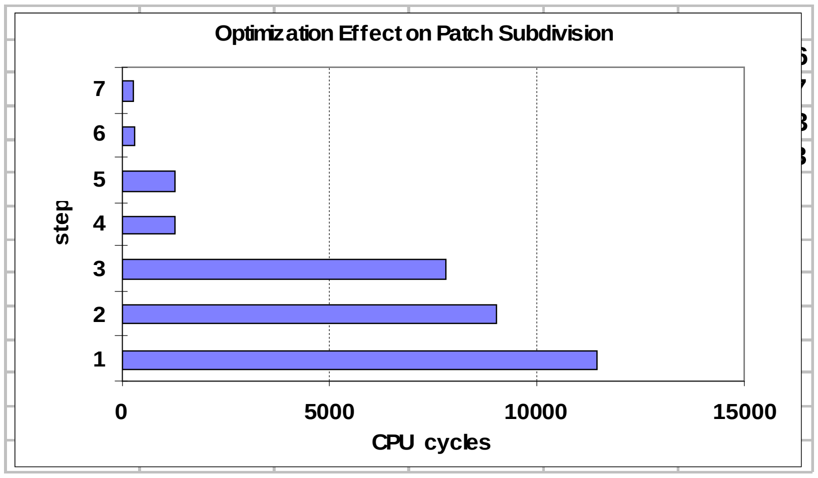

Figure 8: The Effects of Code Optimization on the Function Call PatchSubdivider

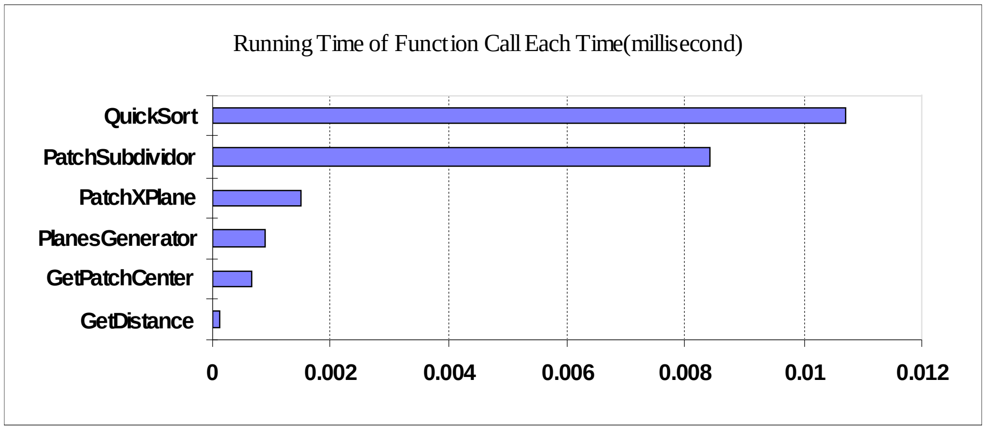

Figure 9: Execution Times of Various Functions

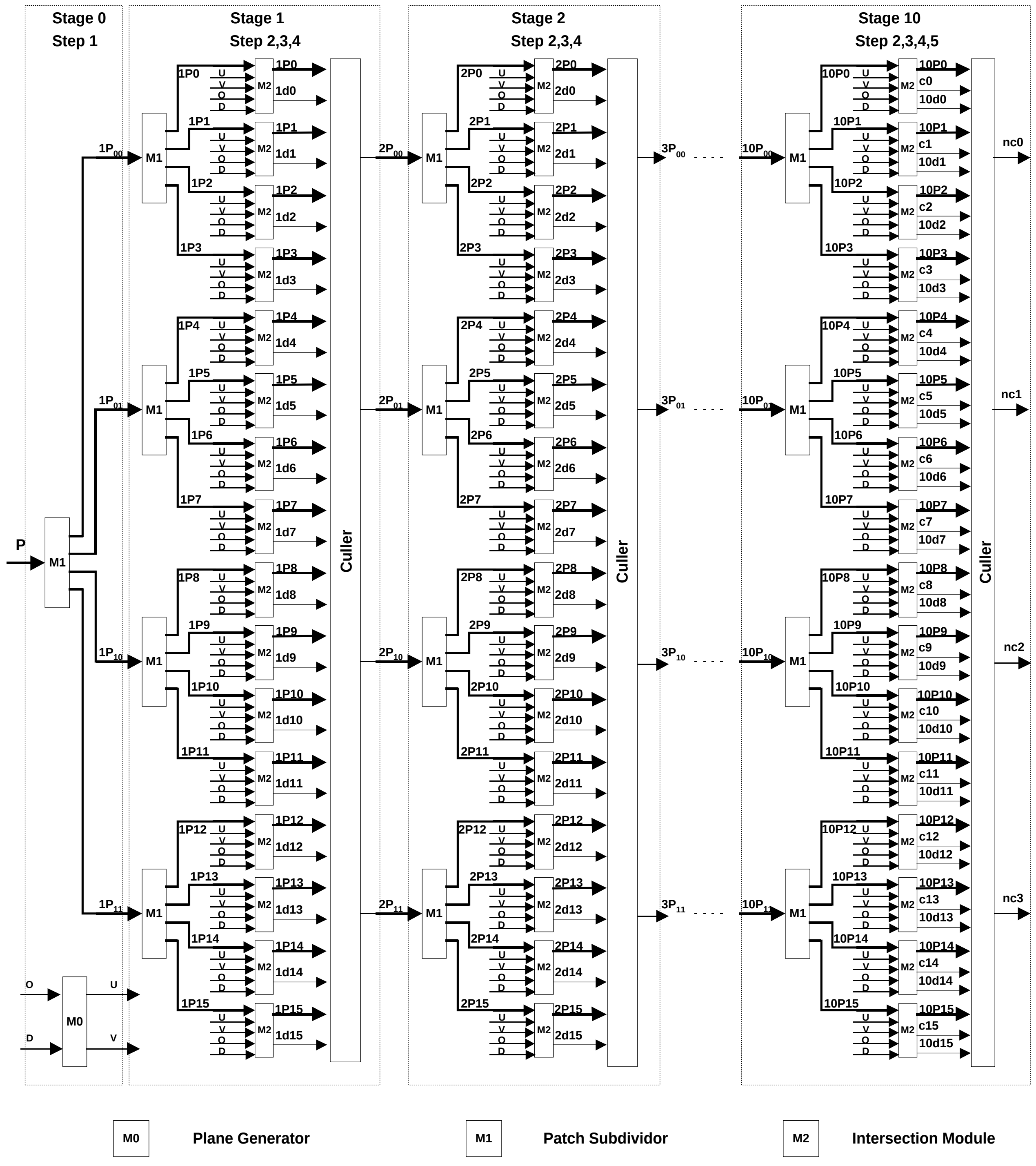

Figure 10: Proposed System Architecture

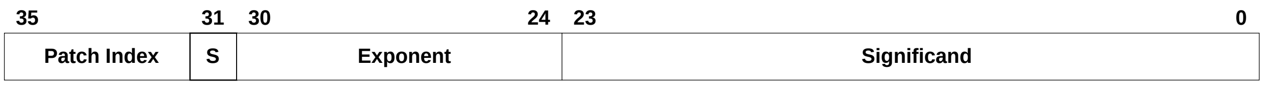

Figure 11: Format of the Sorting Key

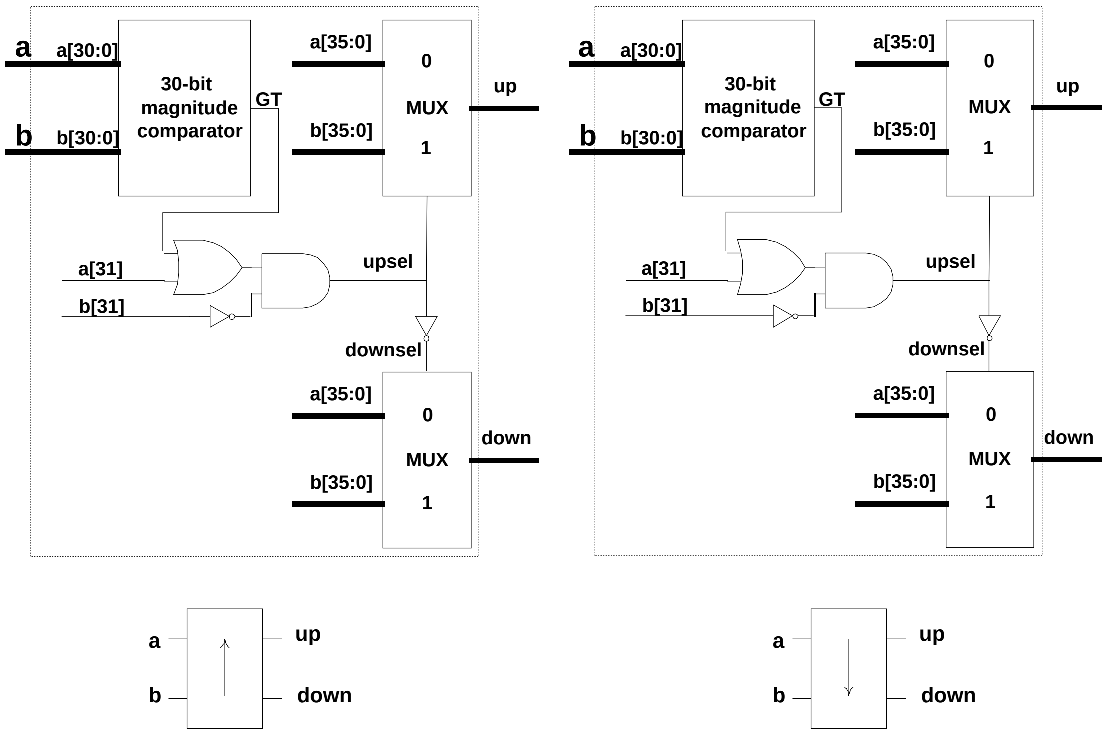

Figure 12: Symbols and Schematics for Functional Units *upsort* (left) and *downsort* (right)

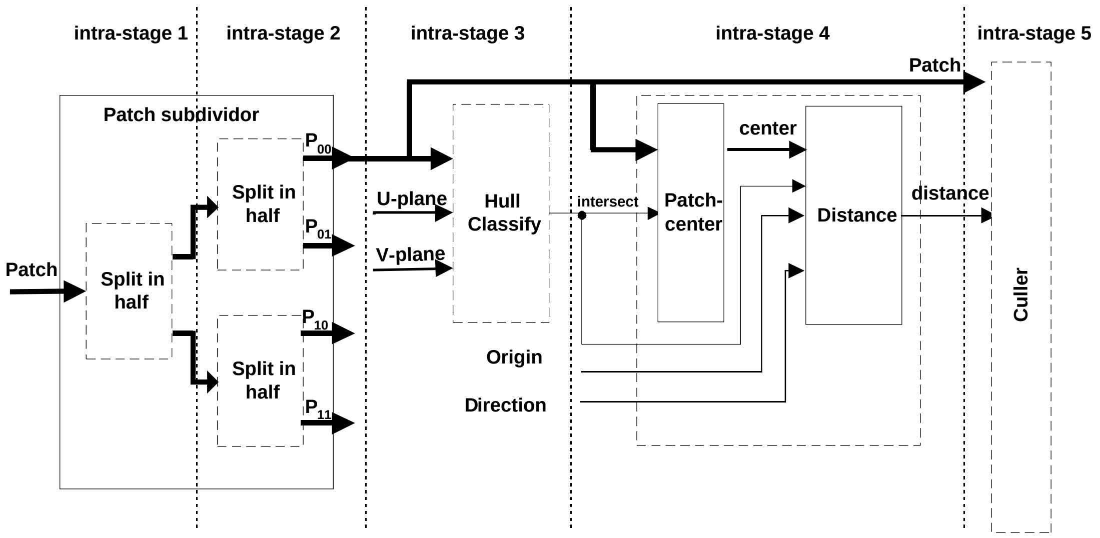

Figure 13: Substages Inside a Typical Stage in Implementation Case 2

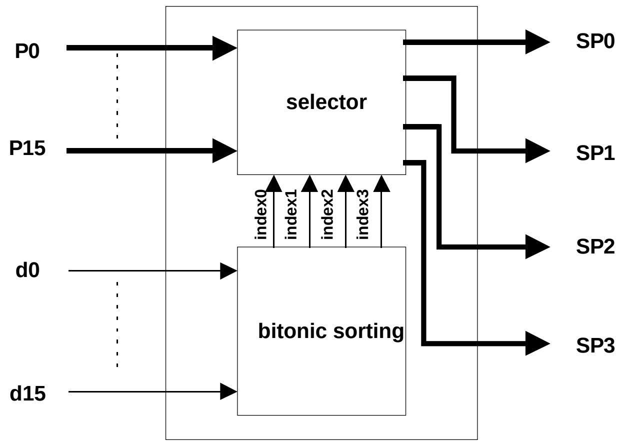

Figure 14: The Culler Module

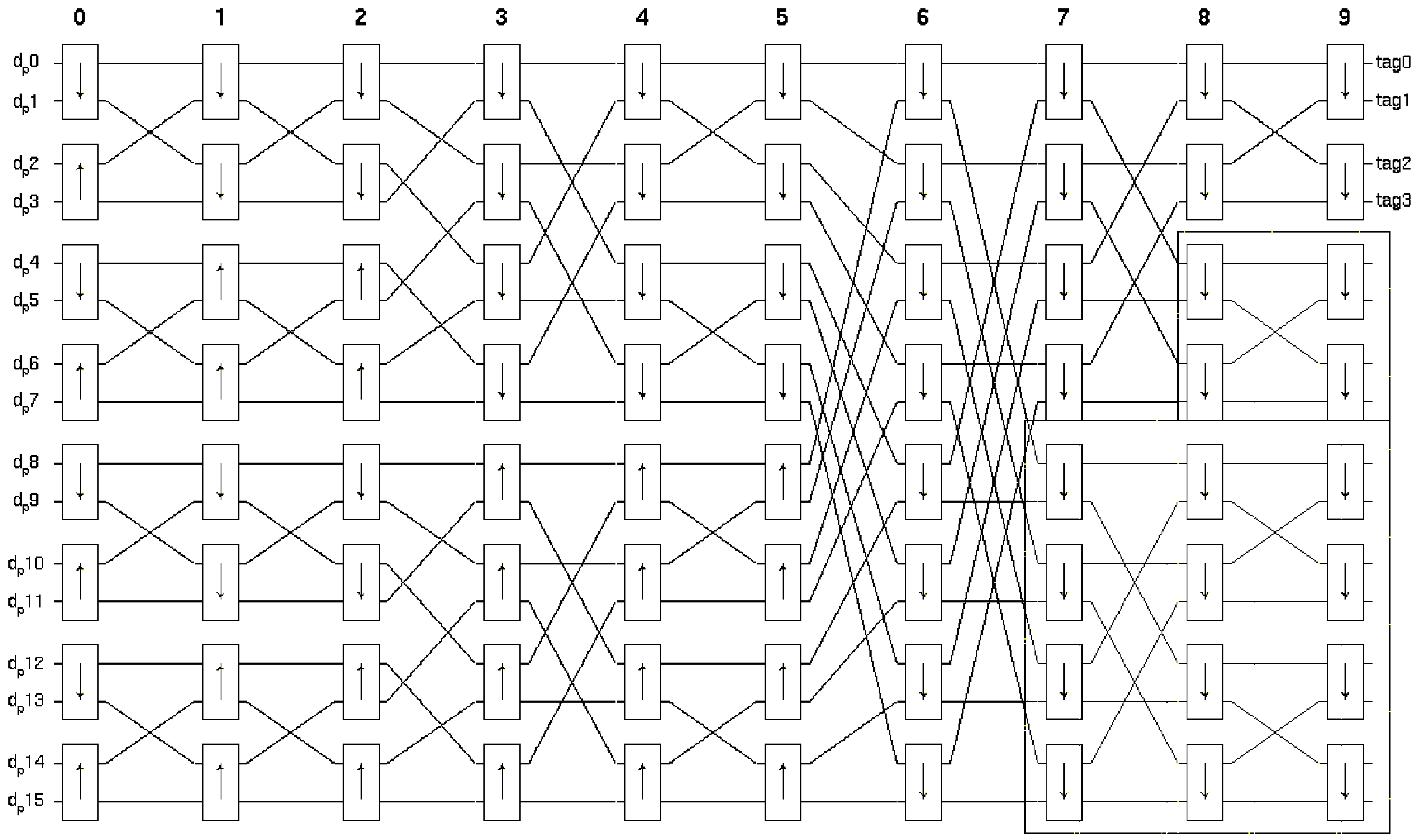

Figure 20. The Sixteen Floating-Number Inputs Bitonic Sorting Network

Name of Function	**Description**
PlaneGenerator	**generate the two planes U and V from a given ray**
PatchSubdivider	**divide an input patch into four subpatches**
HullClassification	**execute the task of hull classification in Step 3 of the algorithm**
PatchCenter	**calculate the center of a given patch**
PatchDistance	**calculate the distance from the patch center to the ray origin**
Culler	**sort out 4 subpatches nearest to the ray origin from 16 candidates**

Table 1: Functional Building Blocks for the Ray/Patch Intersection Algorithm

Step	6701 CPU cycles	Optimization Methods Applied
1	11478	None
2	9018	declare parameters as const type
3	7803	file-level optimization by compiler
4	1284	invoke software-pipelining ; eliminate divide routine call by replacing multiply
5	1257	loop unrolling
6	317	force constant in expression to float
7	243	speculative execution

Table 2: Optimization Procedures on the Function Call PatchSubdivider

Name of Function Call	**6701 CPU Cycles**	**Comments**
PatchSubdivider	**243**	
PatchCenter	**102**	
PlaneGenerator	**36**	
HullClassification	**112**	***on one plane***
PatchDistance	**28**	
Culler	**3081**	

Table 3: Runtime of Optimized Function Calls on the 6701 DSP

Substage s in Stage 1	**Number of 6701s used in each substage**	**Function calls executed by each 6701**	**Number of 6701 clock cycles used (6 ns/cycle)**	**Pipeline Cycle in seconds**	**Overall system throughput in RPI/s**
1	**4**	***SplitInHalf***	**182**	**1.092×10^{-6}**	**9.16×10^{5}**
2	**8**	***SplitInHalf***	**182**		
3	**32**	***HullClassification***	**112**		
4	**16**	***GetPatchCenter GetDistance***	**130**		
5	**None (done in ASIC)**	***Culler***	**N. A. (800 ns delay)**		

Table 4: Performance Estimation Based on Implementation Case 2