Fast Dynamic Tessellation of Trimmed NURBS Surfaces[1]

Salim S. Abi-Ezzi and Srikanth Subramaniam[2]

SunSoft, Inc., 2550 Garcia Ave., Mountain View, CA 94043–1100, USA.
Salim@Eng.Sun.COM and Srikanth@Radiance.COM.

Abstract

Trimmed NURBS (non-uniform rational B-splines) surfaces are being increasingly used and standardized in geometric modeling applications. Fast graphical processing of trimmed NURBS at interactive speeds is absolutely essential to enable these applications, which poses some unique challenges in software, hardware, and algorithm design. This paper presents a technique that uses *graphical compilation* to enable fast *dynamic tessellation* of trimmed NURBS surfaces under highly varying transforms.

We use the concept of graphical data compilation, through which we preprocess the NURBS surface into a compact, *view-independent* form amenable for fast per-frame extraction of triangles. Much of the complexity of processing is absorbed during compilation. Arbitrarily complex trimming regions are broken down into simple regions that are specially designed to facilitate tessellation before rendering. Potentially troublesome cases of degeneracies in the surface are detected and dealt with during compilation. Compilation enables a clean separation of algorithm-intensive and compute-intensive operations, and provides for parallel implementations of the latter. Also, we exercise a classification technique while processing trimming loops, which robustly takes care of geometric ambiguities and deals with special cases while keeping the compilation code simple and concise.

Keywords: NURBS, trimming, curved surfaces, dynamic tessellation, compilation.

1. Introduction

We address the problem of the speedy tessellation of *trimmed* NURBS surfaces into triangles for display purposes. We assume highly varying modeling and viewing transformations; for example, the zoom effect of the view is assumed to vary considerably. This assumption made us focus on dynamic tessellation techniques, where triangles are extracted per rendered frame, in contrast with static tessellation techniques that extract and store triangles for subsequent frames.

Today's graphics accelerators are characterized by having special VLSIs for rendering triangles, complemented with microprogrammable floating-point processors for floating point intensive tasks [4]. While it is cost effective to produce special VLSI for triangles, being a common denominator primitive, it is not so for curved surfaces, and trimming accentuates this point. Therefore, it becomes important to tessellate trimmed surfaces to triangles that get fed into triangle processing VLSI, and to do so at a rate comparable to that of triangle consumption, which today is in the order of 100,000 triangles per second on commonly available mid-range accelerators. This poses serious challenges in algorithm design and load balancing.

---

1 Patent pending.

2 Now at Radiance Software, Berkeley, CA.

This paper concludes a series of results on the dynamic tessellation technique, [1, 2, 13, 14], and focuses on the trimming aspects of the problem. Trimming is extremely important for using NURBS in real life geometric modeling situations, and at the same time adds a considerable dimension of difficulty to the problem of dynamic tessellation of curved surfaces.

We use the concept of compilation to absorb the majority of complexity in trimming into a onetime step, that vastly simplifies a trimmed NURBS into a view-independent form, which is amenable for rapid tessellation into triangles. During compilation we reduce arbitrarily complex trimming regions into simple ones, and we detect and resolve all cases of degeneracies in the original surface primitive. Also, we do the proper formulation so that the actual tessellation of the simple regions can proceed in parallel.

The technique we describe makes a clear distinction between algorithmically complex but computationally cheap algorithms, versus algorithmically simple but computationally expensive algorithms. The latter algorithms are parallelizable and lend themselves to firmware implementations.

We currently have a complete and industrial quality implementation of these algorithms, which is shipping in products.

2. Background

In this section we give some introductory material on previous related results, on trimmed NURBS, and on the dynamic tessellation technique.

2.1. Previous Related Work

Shantz and Chang [12] describe a direct hardware rendering technique of trimmed surfaces based on the adaptive forward differencing (AFD) method. Their technique is very interesting but not practical due to the pragmatics of special graphics VLSI.

Rockwood et al [10] present a dynamic tessellation technique for trimmed surfaces, which is similar to our own approach. They break down the trimmed regions into simpler regions before tessellation, and they dynamically tessellate the trimming curves. However, they do not use graphical compilation, which is central to our results, and have not clearly separated the compute-intensive and algorithm-intensive steps. In our experience this will have efficiency problems and will result with load imbalance problems.

In both of the references above the treatment remains at a high level, while here we tackle some of the challenging details and offer a complete solution in breadth and depth. As practitioners in this area testify, the challenge when it comes to trimming is in the details.

2.2. Trimmed NURBS

For an overview of the mathematics of NURBS, please refer to [5]. NURBS are piecewise rational surfaces with a rectangular 2D parameter space. *Trimming loops* are specified in parameter space to identify an arbitrarily complex trimming region; see Figure 2. According to the PHIGS PLUS specification [8], each trimming loop consists of one or more *trimming curves*, which are 2D NURBS curves, joined head-to-tail to form a closed loop. These curves must be C^0 continuous, and should not intersect themselves or each other; however, they may touch.

The significant region of the trimmed surface is doubly defined through the *odd-winding rule* and the *left-handedness rule*. The odd-winding rule states that for a point to be inside a significant region a semi-infinite ray cast from the point must intersect the trimming loops in an odd number of times. The left-handedness rule states that the significant region is always to the left of directed trimming loops. Clearly, for any desired significant region, we can find a set of trimming loops that describe the region while satisfying the rules above.

2.3. The Dynamic Tessellation Technique

Figure 1 shows the stages of the NURBS graphical processing pipeline. Compilation is a onetime step that takes the NURBS geometry input, and produces a view-independent form that is amenable for fast view-dependent tessellation. Per-frame processing is split into two phases: an algorithm intensive phase, and a compute-intensive phase.

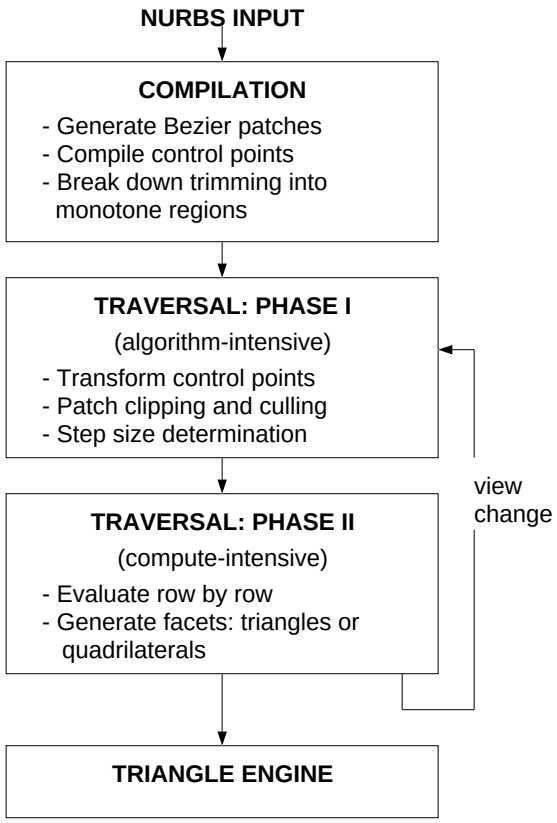

Figure 1: Stages in the NURBS pipeline

We tessellate the NURBS surface in the lighting coordinate system [1], which provides for accurate lighting computations and at the same time, for efficient transformation of the resulting triangles into Display Coordinates. We use tessellation step sizes in LC that insure meeting size and/or deviation thresholds specified in DC. We obtain the step sizes in LC by transforming the DC thresholds into LC, and by consolidating them with precomputed bounds of first and second order derivatives [2]. We perform front/back face culling at the patch level and before tessellation by using the *cone of normals* technique [13].

3. Data Compilation

A classical technique in computational geometry involves preprocessing raw geometric data into a form that makes further repeated processing much faster [9]. Our contribution is to apply the technique effectively for the graphical processing of trimmed NURBS.

The geometric input data in our case consists of:

1. The 2D array of a NURBS control points (de Boor points).
2. The mathematical U and V orders of the surface.
3. The U and V knot vectors in parameter space.
4. A set of trimming loops, each of which is a closed chain of trimming curves (NURBS curves in 2D parameter space).

We first reduce the NURBS surface into its *Bézier patches* using the knot insertion algorithm, see [5], and we use the Bézier control points for further computations. In this section we discuss how the trimming information is reduced accordingly into separate patches, and further simplified in each patch into monotonic regions.

In compiling the trimming information we simplify our code while insuring its correctness, by exercising a technique that we call *geometric classification* for handling geometric ambiguities and special cases. We systematically deploy well-defined conventions for dealing with special occurrences which are rare but possible. We abstract out the specialness and treat these cases like normal ones, while at the same time keeping record of their specialness through classification. This vastly simplifies the downstream processing at every stage. More importantly, these conventions insure the completeness and correctness of the algorithms. Due to the nature of the trimming problem, geometric classification proved to be extremely useful.

3.1. Processing Trimming Loops

Figure 2 shows trimming loops in parameter space, cutting across a 2D grid of Bézier patches. We compile these trimming loops into trimming information per-patch. Each patch is classified as COMPLETE (completely in the significant region), EMPTY (is not part of the significant regions), or TRIMMED (properly trimmed). For each TRIMMED patch we determine an ordered list of U-V monotone chains of the original trimming loops, and some classification information that will be used for further processing.

3.1.1. Traversing Trimming Loops

We first reduce each trimming NURBS loop into its Bézier components. We traverse each resulting Bézier loop sequentially, using a state machine to keep track of past behavior. In one pass of processing, we identify patches that are properly trimmed (i.e., labeled TRIMMED), and we split the loop at patch boundaries and at U and V extremes.

The *vertices* at which we break the loop are classified according to Figures 3 and 4, depending on if they are U or V extremes, or boundary intercepts, and depending on the direction of their corresponding trimming segment. The resulting vertices with their labels are stored with the patch that they belong to. Each boundary vertex is stored with both corresponding neighboring patches and with two different labels; see Figure 4. As a result the trimming loop is simplified into per-patch *monotone* chains consisting of a *starting vertex*, a list of Bézier curves (possibly linear) in between, and an *ending vertex*. Each monotone chain is labeled INC or DEC, based on whether it is increasing or decreasing.

For example in Figure 10 a trimming loop starts at vertex P1; extremes P2 and P3 are identified; then the loop crosses the patch at intercept I2; then extremes P4, P5 and P6 are identified; and finally the loop intersects the boundary at vertex I1. Each of the pieces (P1,P2), (P2,P3), (P3,I2) and so on are U-V monotone (possibly curved) chains of trimming curves.

After processing all trimming loops in the above manner, we need to classify the non-TRIMMED patches as either EMPTY or COMPLETE. We exploit the odd-winding rule mentioned in Section 2.2 and the number of intercepts per TRIMMED patch boundary to make this classification. For every non-TRIMMED patch, we compute the number of intercepts its top V knot line contains starting from the left boundary of parameter

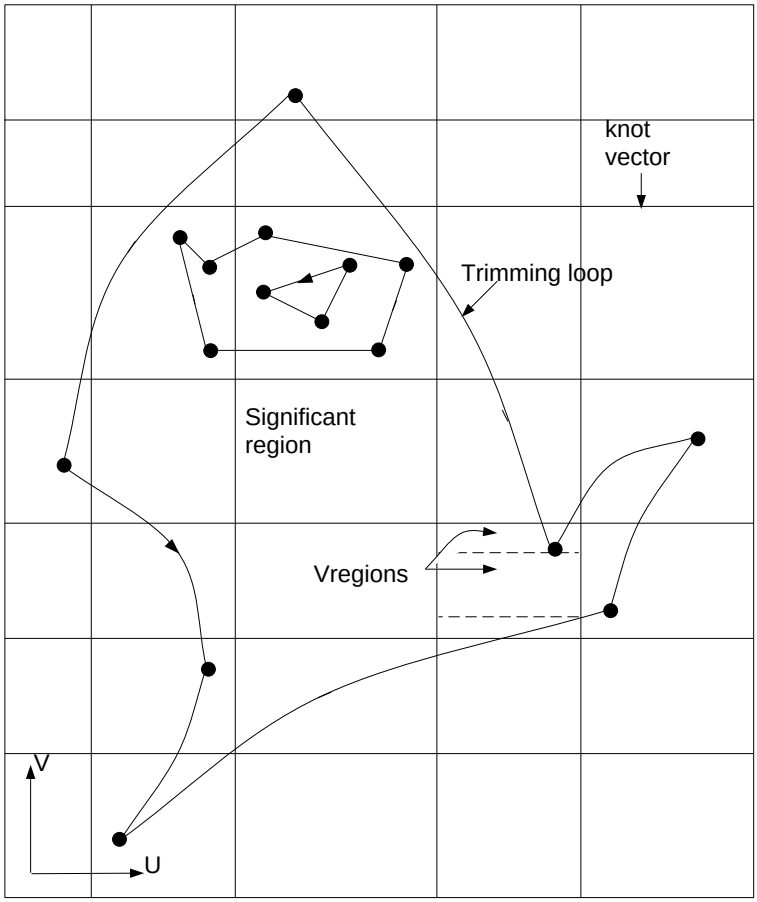

Figure 2: Trimming loops in parameter space.

space and before reaching the top left corner of the patch. If this number is odd, it follows from the odd-winding rule that the corner is within the significant region, and hence the corner is labeled as INSIDE and the patch is labeled as COMPLETE. If that number is even, the patch is labeled as EMPTY and is ignored.

After processing the trimming loops and classifying the patches, we can now treat each patch separately. Each trimmed patch now has the following information:

1. A list of U-V monotone trimming chains that run through the patch. Each monotone chain consists of a list of trimming segments, and is bounded by labeled vertices.
2. A list of the extremes and left/right border vertices sorted in decreasing V order.
3. A list of border vertices for the four boundaries sorted left to right, or top to bottom.

In our example of Figure 10, the resulting list of trimming chains is { (I3,I4), (I1,P1), (P1,P2), (P2,P3), (P3,I2) }. The sorted list of vertices is { I3, P2, I2, P3, I4, I1, P1 }. The LEFT, RIGHT and TOP border vertex lists are { I4 }, { I2, I1 }, and { I3 }, respectively.

3.1.2. Processing Trimming Curves

The operations described above involve the following operations on trimming segments: determining their U and V extremes, computing their intersections with U/V knot lines, and classifying them based on

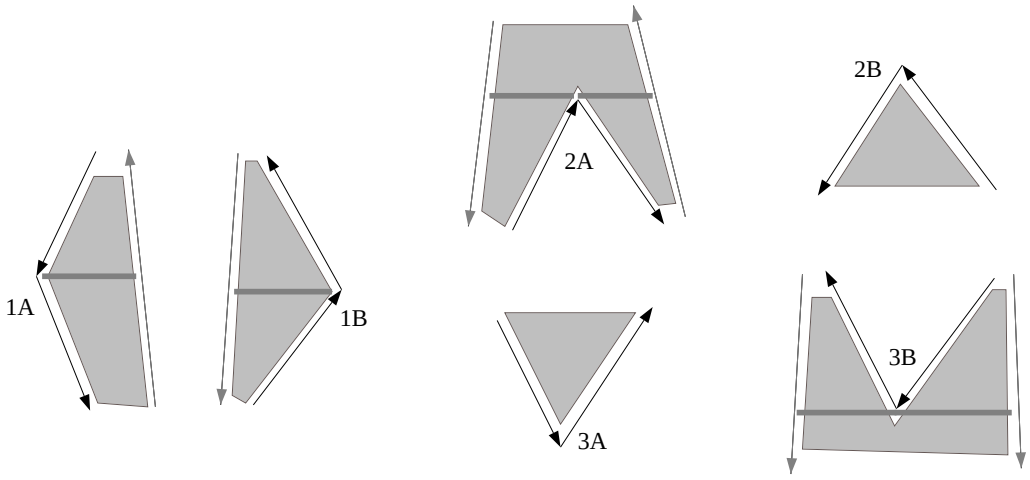

Figure 3: Vertex classification

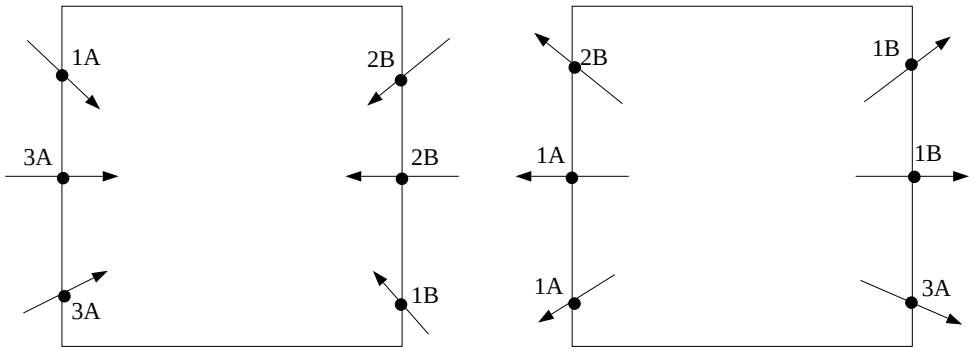

Figure 4: Classification of vertices at border crossings

their U/V orientation. While these operations are straightforward for linear segments, they are nontrivial for curved segments.

Bézier curves exhibit certain properties that make them very useful in this regard. We utilize the convex hull property, in that we search for an intersection with a knot line only if the control polygon intersects the knot line. We exercise the variation diminishing property, in that we search for extremes only if the control polygon itself has extremes. We exploit the end point interpolation and tangentiality property of Bézier curves in order to classify their U/V orientation.

At the heart of these operations, we use a Bézier root-solving algorithm [10, 11], which uses an approach similar to the Newton-Raphson method in finding estimates of roots based on the control polygons. At every stage, the curve is subdivided at the estimated root using the de Casteljau algorithm [5]. At the end, a by-product of finding the exact root is that we also have the control points of the two subcurves obtained by subdividing at the root.

To compute the intersection of a Bézier curve with an arbitrary U or V knot line, we temporarily shift the Bézier curve to a new coordinate system where the knot line is zero, and compute the root. To compute extremes within a Bézier curve, we take the derivative of the Bézier curve in U and/or V, and apply the root-solver. We choose the first parameter value that is an extreme. To find more extremes, we apply the same operations on the second sub-curve.

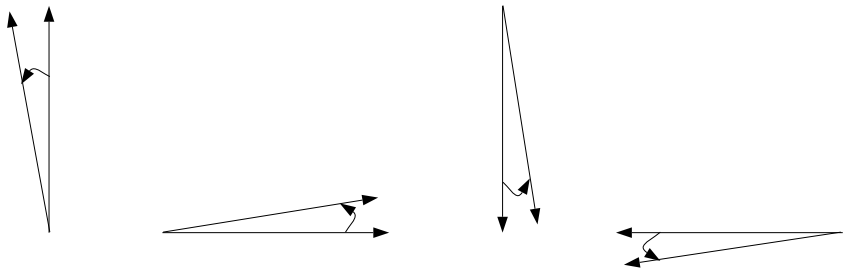

Figure 5: Horizontal and vertical segments.

3.1.3. Conventions for Handling Special Cases

While traversing the trimming loops, a number of special cases may come up. Examples include horizontal or vertical trimming segments, trimming segments that end at patch boundaries or at patch corners, trimming segments that overlap patch boundaries, and combinations thereof[3]. We identify a set of general conventions that enable us to handle all these cases unambiguously without giving special treatment (code) to any of them. The idea is to conceptually modify the entity at hand so that it subscribes to one of the common classes, instead of its special class. This conceptual modification is done strictly for classification purposes and without altering the value of the geometry (coordinates). This approach can be contrasted with identifying all the special cases and developing code for them, which leads to a combinatorial explosion to track and to provide for.

Our conventions are based on the left-handedness trimming rule (the significant region always lies to the left of a trimming curve). The conventions are as follows:

1. A horizontal or a vertical trimming segment is classified and treated as if it is pivoted to the left side (this is well defined, given that the trimming segment is oriented). This convention also accommodates the case where the trimming segment overlaps a knot line; see Figure 5.

2. A trimming segment that ends on a knot line is treated as if it falls infinitesimally short of the knot line. As a consequence, if the next consecutive trimming segment continues into the adjacent patch, it is treated as if it properly intersects the knot line. On the other hand, if the next segment does not cross the knot line, then both segments are assumed to be inside the patch (no intersection); see Figure 6.

3. A trimming segment that crosses a corner point (an intersection of knot lines), is treated as if it is shifted slightly to the left hand side (again, this is well defined given that the trimming segment is oriented); see Figure 7.

4. If the starting point of a trimming loop coincides with a corner or lies on a boundary then it is treated as if it is shifted slightly from the knot line(s) it lies on, in a manner dependent on the orientation of the last trimming segment leading back to the starting point. The point is shifted so that the last trimming segment does not intersected the knot line(s) that the point lies on. If the point lies on the parameter space boundary, it is treated as if it is shifted inside the boundary patch.

5. In some (rare) cases, two extreme vertices on trimming loops (either the same loop or two different ones), may be coincident. We use a lookup table for ordering such vertices based on their vertex types.

6. All combinations of the above cases are treated using a combination of the conventions. For example, consider a trimming segment that overlaps a horizontal knot line and that ends at a corner. The segment

[3] When using floating-point numbers for modeling geometric entities, the notion of equality based on identity is not adequate, instead we use a notion of equality to within a certain minute threshold.

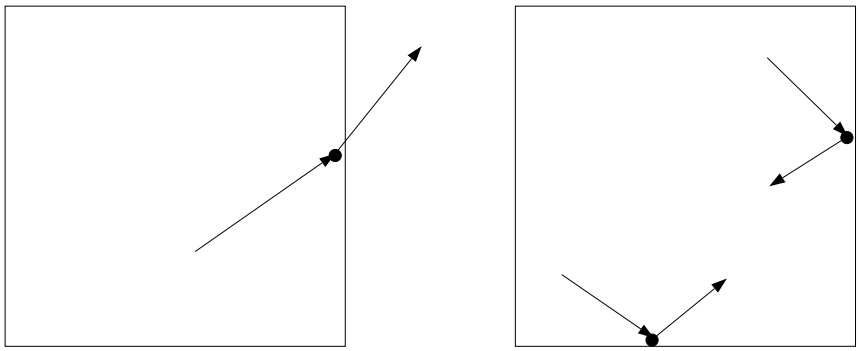

Figure 6: Ending at knot lines.

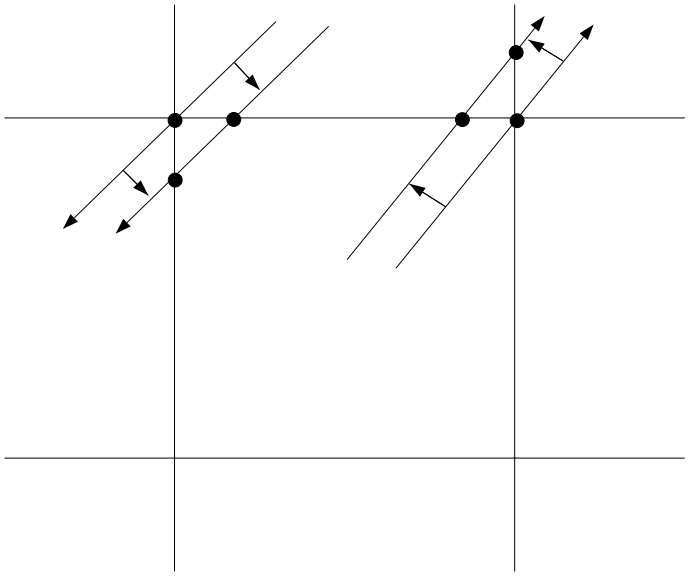

Figure 7: Crossing corner points.

is treated as if it is pivoted left and as if it is short of the perpendicular knot line that it ends on. See Figure 8.

3.2. Generating the Vregions

Using the information generated in Section 3.1.1, we split the significant region of each patch into U-V monotone regions that we call *Vregions*. A Vregion is trapezoidal in shape, and consists of horizontal top and bottom bases, and a left and right sides (typically curved) that are monotone in both U and V. Vregions are amenable for the efficient evaluation of long sequences of points along U iso-lines.

Each base is of type KNOT in case it is a complete patch boundary, or GAP in case it has vertices of type 2A or 3B lying on it, and hence has potential gaps that need to be prevented during tessellation, or REGULAR otherwise.

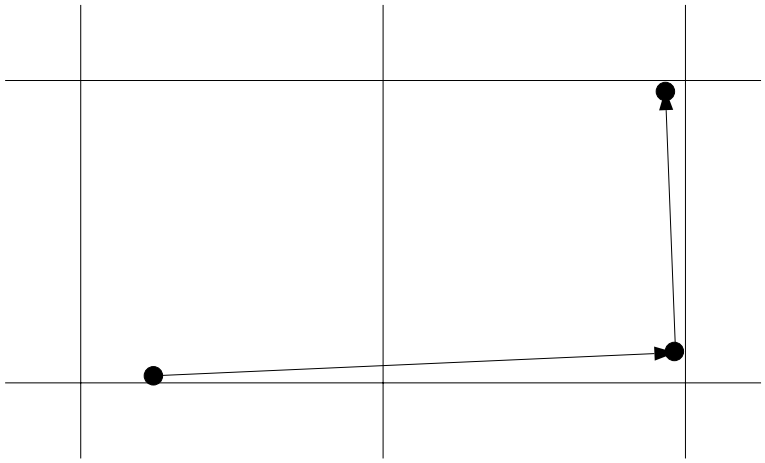

Figure 8: Combination of cases.

The left and right sides each consist of a chain of U-V monotone trimming segments (linear or curved). The side is classified as DEC (decreasing in U), INC (increasing in U), or BORDER (part of left or right patch boundary). As a result we classify each Vregion into one of nine classes; seen Figure 9.

3.2.1. The Generation of Vregions

We use an adaptation of an algorithm that reduces arbitrary polygons into trapezoids due to Fournier and Montuno [6]. As we enumerate the steps of the algorithm, we walk through the example in Figure 10:

1. Get the sorted list of vertices for the LEFT, RIGHT and TOP patch borders. If a patch corner is labeled INSIDE, see 3.1.1, then add it to the two corresponding border lists, after which all border lists will have an even number of vertices. Construct segments on the LEFT, RIGHT, and TOP borders by pairing vertices in the corresponding sorted lists. In Figure 10, the LEFT, RIGHT and TOP segments are { (I4,C1) }, { (C3,I2), (I1,C2) }, and { (I3,C3) } respectively.

2. Use the resulting TOP segments to start a collection of active Vregions. Active Vregions have their top base and two sides identified, and remain active until their bottom base is determined. In Figure 10, the starting active Vregion is { (I3,I4), (C3,I2) }.

3. Traverse the patch's sorted list of vertices, and process them based on their labels, see Figures 3 and 4, by ending and starting Vregions on the fly:

    a. A vertex of type 1A or 1B indicates the completion of an active Vregion, and the activation of a new one.
    b. A vertex of type 2A indicates the completion of an active Vregion, and the activation of two new ones. Package this vertex along with the bottom base of the completed Vregion, because it could potentially cause a gap during tessellation. The base is labeled as type GAP as a result.
    c. A vertex of type 2B indicates the activation of a new Vregion.
    d. A vertex of type 3A indicates the completion of an active Vregion.
    e. A vertex of type 3B indicates the completion of two Vregions and start of a new Vregion. Package this vertex with the top base of the new Vregion as a GAP vertex.

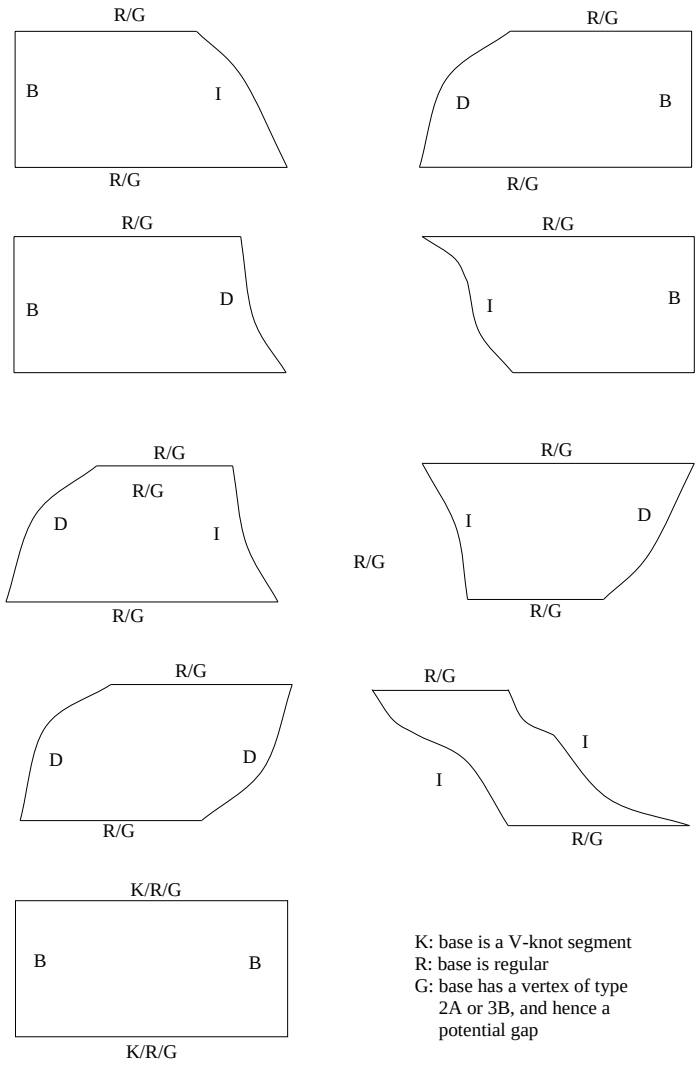

Figure 9: Vregion classification

In the example in Figure 10, the vertices are processed as follows: Process vertex P2 (of type 2A). So cast a horizontal line, create Q-intercepts Q1 and Q2, and close off Vregion { (I3,Q1), (C3,Q2) }. Start Vregions { (Q1,I4), (P2,P1) } and { (P2,P3), (Q2,I2) }. Process vertex I2 (of type 1B). Cast a line to the left, and close off Vregion { (P2,Q3), (Q2,I2) }. Start Vregion { (Q3,P3), (I2,P3) }. Process vertex P3 (of type 3B). Close off Vregion { (Q3,P3), (I2,P3) }. The next vertex is I4 (of type 1A). Close off Vregion { (Q1,I4), (P2,Q4) }. Start Vregion { (I4,C1), (Q4,P1) }. The next vertex is I1 (of type 2B). Note that even though vertices I1 and P1 have the same V value, I1 is processed first because the segment (I1,P1) is treated as slightly pivoted to its left. I1 starts the Vregion { (I1,P1), (I1,C2) }. The next vertex is P1 (of type 2A). Cast lines on both sides, and close off Vregions { (I4,Q5), (Q4,P1) } and { (I1,P1), (I1,Q6)[4] }. Start Vregion { (Q5,C1), (Q6,C2) }.

4. Complete the remaining active Vregions at the bottom boundary. In Figure 10, close off Vregion { (Q5,C1), (Q6,C2) }.

[4] This Vregion is degenerate which is a consequence of the conventions that we use. So, the conventions while they drastically simplify the code, they occasionally result with a vregion that is degenerate in parameter space.

5. Generate lists of "Q-points" on the LEFT and RIGHT boundaries. These are lists of vertices sorted in decreasing V. They consist of the original intercept vertices, and the new points that are generated on the border, due to the generation of Vregions. These lists of Q-points are needed for avoiding gaps between adjacent patches. In Figure 10, The LEFT and RIGHT Q-point lists are {I4, Q5, C1} and {C3, Q2, I2, I1, Q6, C2} respectively.

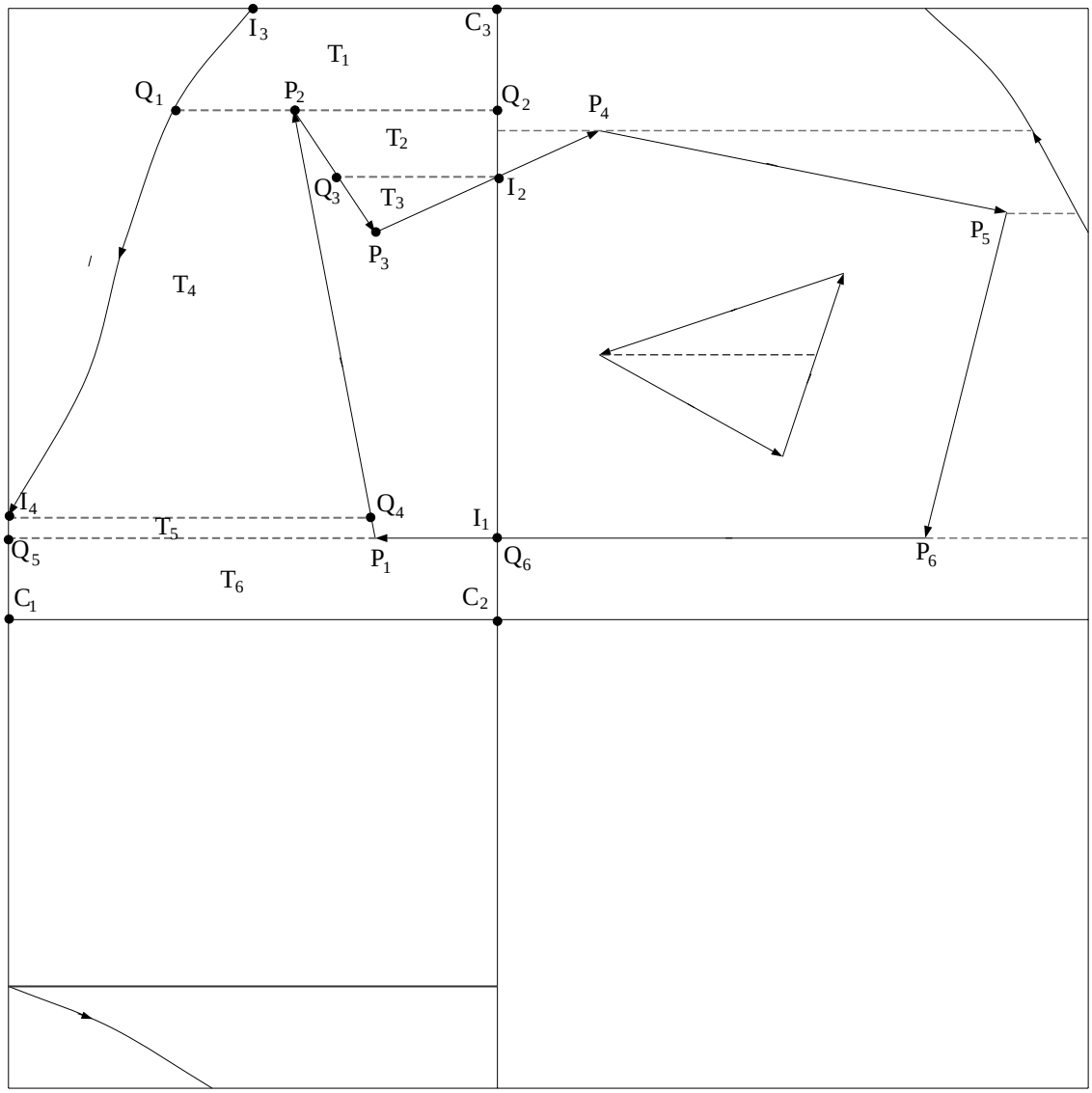

Figure 10: An example of trimming compilation

3.2.2. Packaging Vregions for Traversal

Once the patch has been broken down into Vregions, each Vregion can now be processed (tessellated and rendered) separately. We first eliminate degenerate Vregions (Vregions of zero height), which do not

have any GAP vertices. We then reverse the order of segments on the left sides so that both sides have segments going from bottom to top. We transform the (u,v) coordinates of each trimming point to the *local* coordinates of the patch parameter space (since trimming loops are defined in the parameter space of the whole B-spline). We evaluate the surface at the trimming points (in modeling coordinates). These trimming points will always occur in the triangulation even in the case of dynamic tessellation, and hence we evaluate these points during compilation, so that we only transform them during per-frame traversal.

3.2.3. Handling Patch Degeneracies

A patch degeneracy occurs when a certain geometric configuration of control points results in undefined normals (and hence, undefined lighting) at one or more points on the surface. The normal at a point on the surface is computed as the normalized cross product of two tangents evaluated at that point.

Examples of commonly occurring degeneracies are:

1. An entire patch boundary collapsed to a point as in the case of a cone. In this case the normal is undefined along that boundary.
2. A patch corner and its neighboring control points on both sides are colinear. In this case, the normal is undefined at that patch corner.
3. A boundary row (or column) of control points is exactly the same as its next inner set of points. In this case normals are undefined for points on that boundary.

We detect the above conditions at compile time by inspecting the control points. We then isolate the degenerate portions of the patch into a Vregion. Figure 11 shows the Vregion created for an edge degeneracy. The degenerate Vregion is marked for special handling during traversal, in which case the normals are computed from the triangulation and not from the tangents of the underlying patch. This ensures both correctness and efficiency during traversal, since the traversal process does not need to test for and handle degeneracies at all. It processes the "healthy" portion of the patch using the usual techniques.

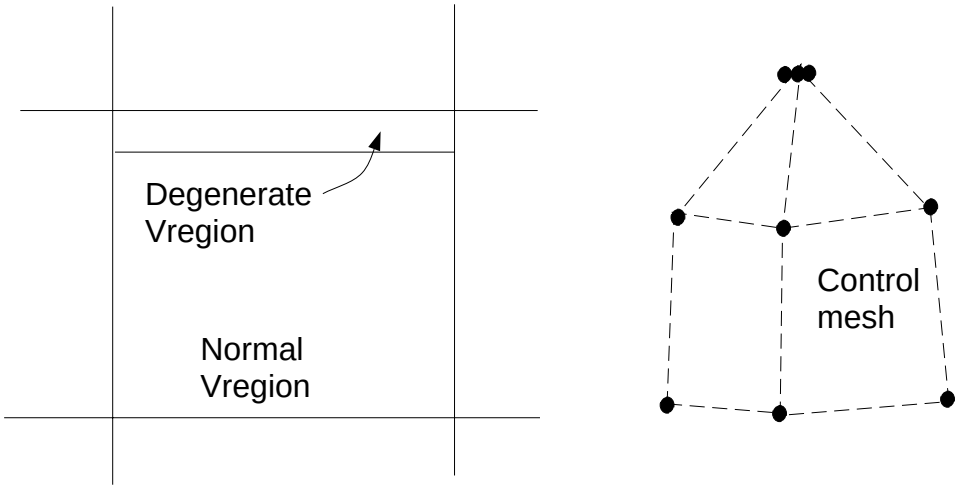

Figure 11: An example of patch degeneracy: an entire boundary collapsed to a point

4. Traversal Processing

So far we've described the compilation of the NURBS input into a view-independent form. For every frame, we extract triangles from this form for rendering. We perform traversal in two phases, in Phase I, we perform all the algorithmically intensive, but computationally cheap operations such as step size determination. This phase performs view dependent operations that prepare the surface for actual tessellation. Phase II consists of floating-point intensive, but simple and repetitive operations- a candidate for parallel implementation on special-purpose accelerators. Phase II performs the actual tessellation and processing of the triangles.

4.1. Phase I Traversal

We give a brief description of Phase I operations.

4.1.1. Transformation, Culling and Step Size Determination

In [3] we've proposed a coordinate system called "lighting coordinates (LC)" situated between world coordinates (WC) and device coordinates (DC) as appropriate for tessellation. Control points are transformed from WC to LC, and tessellated triangles are transformed from LC to DC by a sparse matrix and hence at a reduced cost.

Patches are clip culled by inspecting their control points, since the tessellated patch will always lie within the convex hull. Further, if the eye point lies in the front or back region of the "cone of normals" which is computed during compilation [13], the patch is face culled.

Within a single Bézier patch, we determine uniform step sizes in U and V for meeting a specified threshold. Step sizes typically vary from patch to patch, [1] gives the mathematical details of step size determination for meeting deviation and size thresholds, while [2] gives an approach for scaling an approximation threshold defined in DC to LC.

4.1.2. Patch Subdivision and Combination

Because we use a uniform step size throughout a single patch, some portions of the patch may be over-tessellated just because other portions have a complex curvature, or are nearer to the view point. Also, the patch may be partially clipped. In these cases, we "recursively subdivide" the patch into four "subpatches", each of which is tessellated (if at all) with different step sizes. Subdivision reduces the resulting number of triangles considerably when the user zooms into specific portions of the patch.

Sometimes the opposite happens. There may be a large number of patches on the screen, each covering only a few pixels. In this case, we "combine" adjacent small patches, and reduce the entire group into a quadrilateral (two triangles). We do patch combining in situations where screen coverage is sufficiently small, so that ignoring trimming will have no influence on the resulting rendering.

4.2. Phase II Traversal

In Phase II, we perform the actual extraction of triangles. Trimmed patches (consisting of Vregions) are treated differently from untrimmed patches. In this phase we need to insure that there are no gaps (missed pixels) in the rendering of the surface. We first look at a general mechanism for surface evaluation and triangle generation, before going into tessellation of trimmed surfaces.

4.2.1. Surface Evaluation

We generate sample points on the surface based on the step size computed in Phase I. For a row of points, we first compute the univariate equation of the curve, and then evaluate this curve according to the corresponding step size. For more efficient evaluation, we convert the Bézier basis to a "symmetric power basis" [11]. However, if the patch is small (number of steps in U and V are small), then we evaluate in the Bézier basis and save on the overhead of converting to a different basis.

We compute normals by evaluating the U and V tangent surfaces, taking the cross-product and normalizing the result (using an inverse square root lookup table). However, if the application gives additional hints about the nature of the surface, we compute normals using cheaper methods. SunSoft's XGL programmer interface [15] allows the application to specify additional hints about the NURBS surface (such as spherical, cylindrical, conical or planar geometry information). In the case of a spherical NURBS surface, for example, the normal at a point is simply the vector from the center to the given point. We obtain 1.5 to 3 times speedup in the overall tessellation time when such hints are given. We also found that a majority of the NURBS surfaces used in MCAD applications are of these special types.

4.2.2. Triangle Generation

We pass a vertex (coordinates and normals) to the triangle engine as soon as it is determined. The engine constructs triangles based on the new vertex and two previously stored vertices, and then lights and shades the triangles. Our cases always involve the triangulation of a region between two piecewise linear lines (U or V isolines or U-V monotones), and hence lend themselves to a simple triangulation mechanism. This mechanism handles all our cases of triangle generation: untrimmed patches, vregions, adjacency, and stitching strips (as we will see later in the latter two cases).

We label the two piecewise linear lines that define the region to be triangulated, as the "A" and "B" lines. The implementation has three registers, A, B and N (for new). The mechanism works as follows:

```
Init (CW_flag);
Count := 0;
SendA/SendB (point) {
    COUNT++;
    N := point;
    if (Count > 2) FormTriangle (A, B, N);
    A/B := N;
    }
```

As new A or B points are obtained, they are submitted by the SendA and SendB facilities, respectively. The control routine that drives this Send facility needs to be careful in selecting an appropriate order in which points are sent. Also, the CW_flag is used to indicate which side of the triangle is front-facing. If this flag is set, then the front face is the side in which the vertices A, B, and C are clockwise.

4.2.3. Avoiding Gaps Between Patches

Because adjacent patches may have different step sizes, there is a potential for gaps in the tessellation. Gaps may also occur due to the splitting of trimmed patches into Vregions which are processed independently. There are two solutions for this: the first "avoids" the gaps, and the second "stitches" the gaps by plugging extra triangles. In both cases, we maintain enough information in every patch about its neighbors, so that each patch can be processed independently in Phase II. Along each common boundary only one of the neighboring patches need to intervene to solve the adjacency problem, we have conventions that determine which of the two patches does so.

An "adjacency strip" is used to avoid gaps that may be created when adjacent patches have different step sizes (Figure 12). This is the more efficient scheme, but can only be used in simple cases such as two neighboring untrimmed patches. The patch with the larger step size uses an adjacency strip in its last tessellation row or column to blend with the neighbor's tessellation. Figure 13 shows the bottom patch with an adjacency strip on its boundary. Using the Send facility described above, all the points in the top row are sent using SendA, and the bottom row using SendB. A special ordering of these sends is chosen to minimize the sizes in parameter space of the resulting triangles.

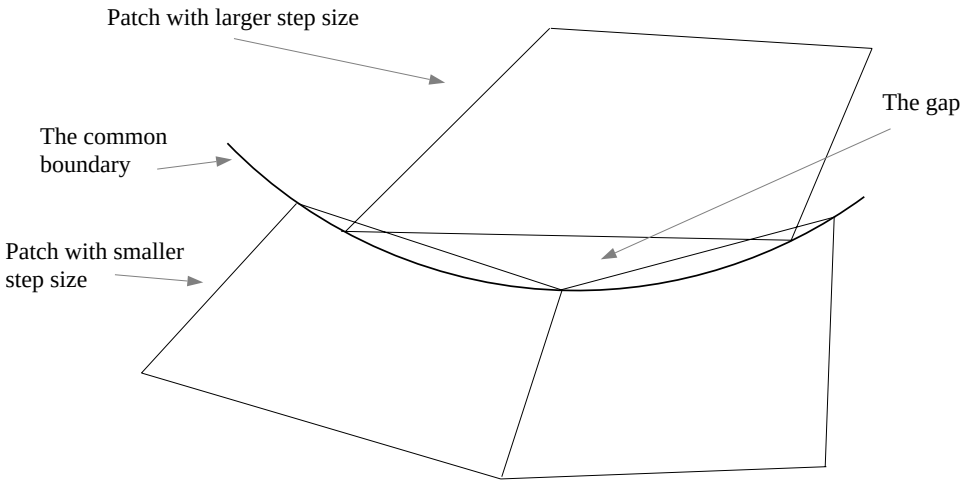

Figure 12: Gaps between patches that have different step sizes.

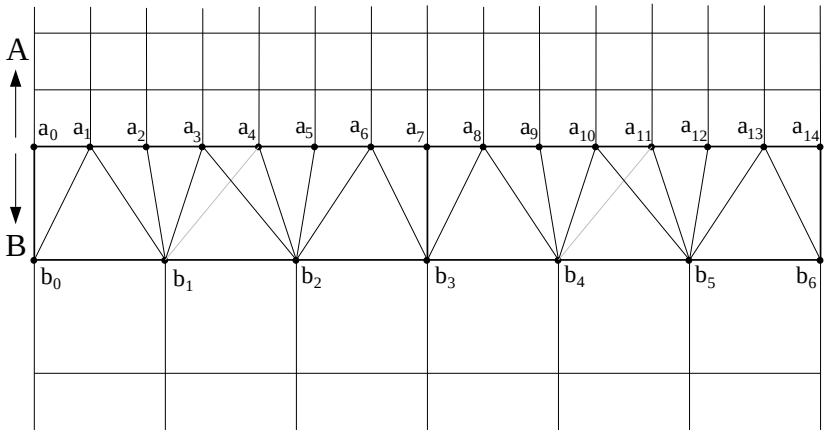

Figure 13: An adjacency strip.

A stitching strip (Figure 14) is a more general, but a slightly less efficient mechanism to plug gaps between two patches of different step sizes. It is used in more complex cases like trimmed patches. A stitching strip has zero width in parameter space, and hence consists of near degenerate triangles in modeling

space. In Phase I, we identify the points on either patch that need to be stitched. These points are generated from the Q-points obtained during compilation, see Section 3.2. Based on the Q-points and the step sizes on both sides of the boundary, two sorted lists of V coordinates are generated. The points in the two lists are then merged, and duplicates are consolidated. The resulting points are labelled as A (belongs to patch A), B (patch B), or C (duplicate). This merged list of coordinates is used to generate stitching triangles as in Figure 14, and by making use of the Send facility discussed above.

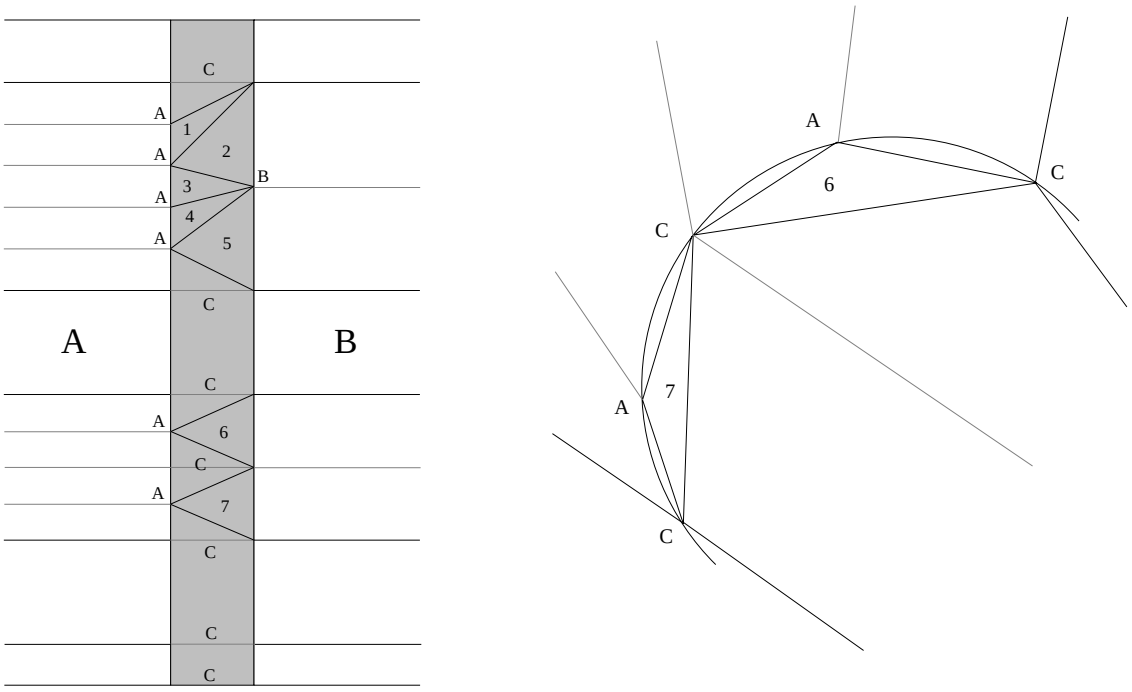

Figure 14: A stitching strip in parameter space and in object space.

4.2.4. Tessellating Trimmed Patches

Untrimmed patches are tessellated in a quadrilateral mesh fashion, with up to four adjacency strips along the boundaries. Depending on the underlying hardware and memory considerations, the triangles can either be immediately rendered, or accumulated and rendered as a quadrilateral mesh. Trimmed patches, on the other hand, are more complex. Each Vregion and the stitching strips are processed separately. This section describes how a Vregion is tessellated.

If the Vregion sides consist of Bézier curves, tessellate the curves in parameter space. The step size is determined based on a size criterium in parameter space, as well as the U and V step sizes of the underlying patch. The points are evaluated as "stand-alone" surface points (in general they don't lie on an evaluation isoline). Since this is expensive, we cache these points in MC and reuse them (by transforming them) as long as the step sizes for the curves don't change.

The classifications of the left/right sides and top/bottom bases of the Vregion dictate how the tessellation will be performed (see Figures 15 and 16). Processing a Vregion involves the following steps:

1. Determine a uniform step size in V, based on the patch step size and height of the Vregion (this will insure an integral number of steps along the width of the Vregion).

2. The Vregion is divided and processed as "strips" going from bottom to top. For each tessellation row, compute the intersections of the row with the left and right sides, and evaluate the surface points on the row (both the intersection points and the intermediate grid points).

3. Each strip consists of up to three "zones" (or groups of cells): a left-trimmed zone, an untrimmed or doubly trimmed zone, and a right trimmed zone. These zones are again triangulated separately.

4. If the top or bottom bases are of type GAP, then evaluate the gap points and triangulate them with adjacent grid points to avoid any potential gaps in the rendering.

5. If the top or bottom bases are of type KNOT (i.e., they are a whole patch boundary), then we simply use an adjacency strip.

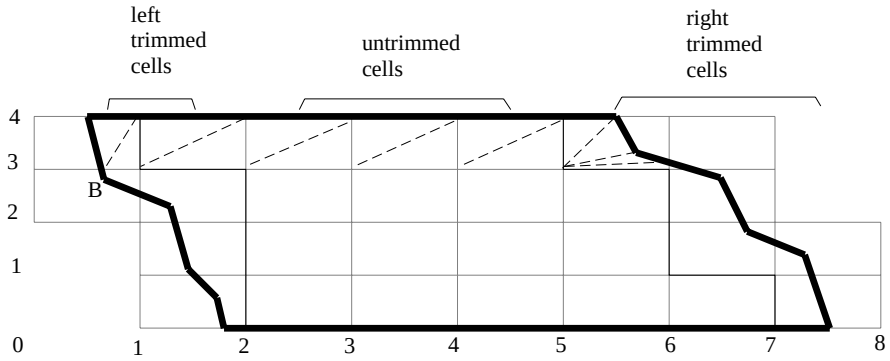

Figure 15: Vregion tessellation for untrimmed and left/right trimmed cells

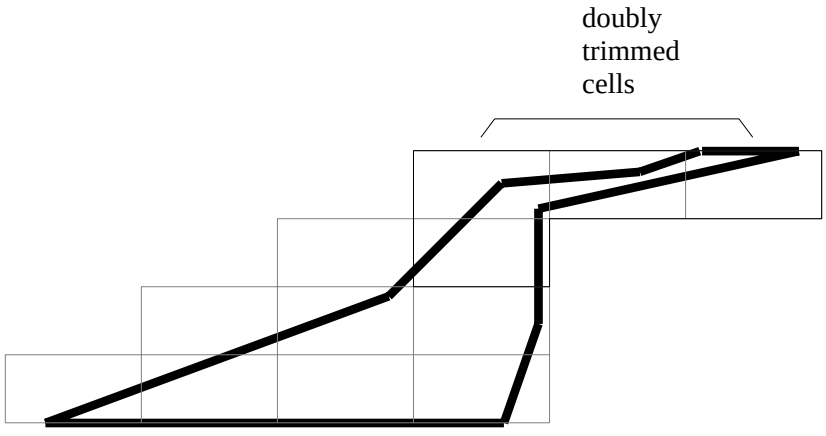

Figure 16: Vregion tessellation for doubly trimmed cells

Each Vregion strip consists of a top and bottom row of grid points, and left and right trim sides. As stated before, the strip consists of up to three zones:

1. Left trimmed cells: These cells have the left trim side going through them (but not the right trim side).

2. In between cells could be either:

   a. Untrimmed cells: These cells do not have any trim sides going through them.
   b. Doubly trimmed cells: These cells have both the left and right trimmed cells going through them.

3. Right trimmed cells: These cells have only the right trim side going through them.

A sequence of left trimmed cells is triangulated by generating a triangle star using either the lower right corner (if side is DEC) or the upper right corner (INC); see Figure 17. Right trimmed cells are handled similarly.

A sequence of untrimmed cells is triangulated as a triangle strip (each cell is sent as two triangles).

There are two cases for tessellating doubly trimmed cells. First, if the U extents of the two sides do not overlap, which is always the case if the two sides have the opposite direction. In this case, we vertically split the cell into a left trimmed cell and a right trimmed cell, which get tessellated as above (see Figure 18). Note that we have to generate a triangle to plug the gap created by splitting the cell.

Second, if the U extents of the two sides do overlap, which could only happen if the two sides have the same orientation, we use a more general scheme for triangulation. This scheme, though quite simple, is the most algorithmically complex triangulation scheme used in traversal Phase II. We use an adaptation of the O(n) algorithm by Garey et al [7] for the triangulation of a monotone polygon; see Figure 19. This algorithm involves sorting the polygon into two monotone sides (in our case, the points are already sorted), and using a stack to send points one by one, based on the side and angle subtended. Here again, the send facility described in Section 4.2.2 is quite handy.

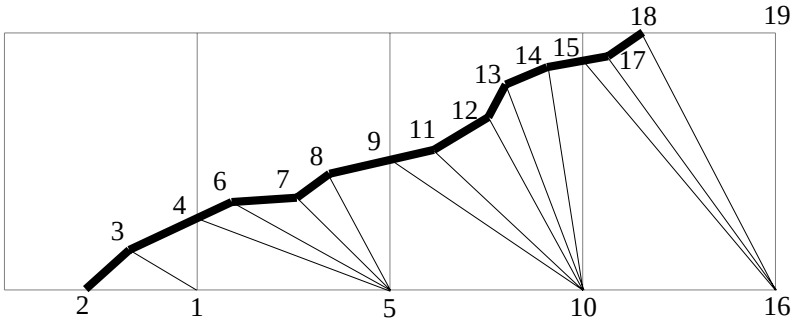

Figure 17: Triangulation of left trimmed cells.

5. Implementation and Results

The NURBS technology we've discussed in this paper is mature and shipping in three products at this point. The products include the XGL [15] foundational graphics library, the SunPHIGS graphics library, and a parallel implementation of the Phase II algorithms in firmware as part of the ZX graphics accelerator [4]. Furthermore, we've filed three related patents one of which has been approved.

For the dynamic tessellation technology to be practical on a platform like the ZX we had to be able to generate triangles as fast as the ZX is able to consume them, otherwise the tessellation will be the bottleneck. We were able to achieve a well balanced solution by performing the phase II algorithms in the four special purpose floating-point processors in the ZX. This was ideal, since the phase II algorithms are simple and hence lend themselves to a firmware implementation, are parallelizable and hence lend themselves to the architecture of the ZX, and are floating point intensive and hence it is best to perform them on special floating point processors as opposed to burdening the general purpose CPU.

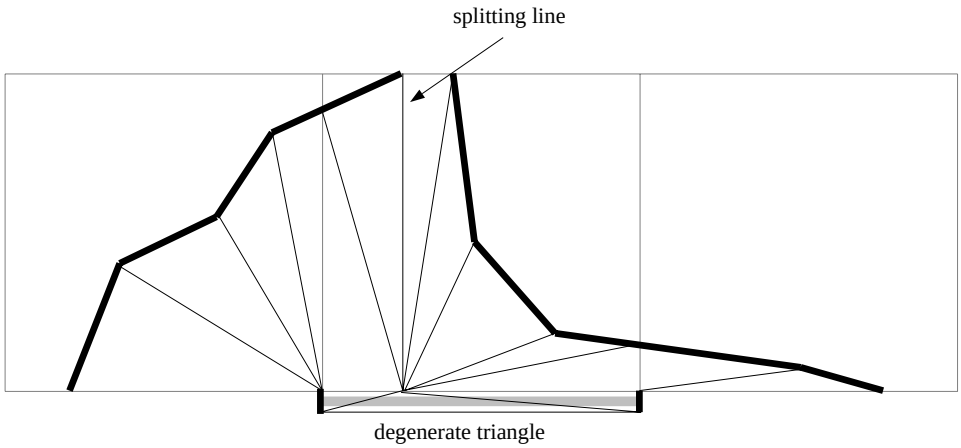

Figure 18: Triangulation of a doubly trimmed cell, the simple case.

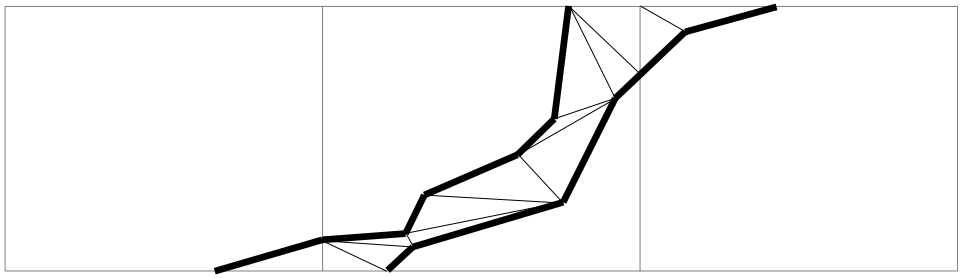

Figure 19: Triangulation of a doubly trimmed cell, the complex case.

The ZX accelerator can process approximately 214,000 precomputed and lit (statically tessellated) triangles/second. These performance numbers are based on one ambient and two directional lights, with ambient and diffuse lighting enabled. In contrast, the ZX can dynamically tessellate and process up to 110,000 triangles/second, generated from untrimmed non-rational bicubic or conic (quadratic rational and where normal computations are simplified) NURBS surfaces. For trimmed surfaces, the performance depends on the complexity and nature of trimming. In general, we've obtained 50,000–80,000 triangles/second from trimmed NURBS.

There are no industry benchmarks for measuring NURBS rendering performance. One metric is the number of triangles dynamically generated and rendered per second. This metric indicates how well the tessellation system keeps up with the triangle rendering hardware; however, it is not an accurate measure of the overall performance of NURBS rendering. A system that over-tessellates a NURBS surface (generates more triangles than needed), may process more triangles per second and still be slower for the overall problem (than a system that generates a more optimal number of triangles). As a consequence, in highly dynamic situations our technique not only produces a higher quality rendering by honoring a dynamic tessellation criteria but it runs *actually faster* than static tessellation. This is the case since we adaptively pick a proper number of triangles for the situation at hand, which is typically much less than the number of triangles resulting from, the usually overly conservative, static tessellation techniques.

For a quantitative analysis of our algorithms we define the CSF (Compile Speedup Factor) as the ratio of execution time of (Compile + Phase I + Phase II) / (Phase I + Phase II), and the FSF (Firmware Speedup Factor) as the ratio of (Phase I + Phase II) / (Phase I). The former is to indicate the benefit of the compilation phase which applies regardless of firmware acceleration, and the later is to indicate the benefit of performing

Phase II in firmware on special hardware. By determining these factors based on a collection of industrial MCAD models covering a variety of options and under different viewing conditions, we obtained a CSF of between 2 and 3 and a FSF of between 4 and 7, for a compounded speedup factor of roughly an order of magnitude.

6. Conclusion

There are clear advantages for having a unified representation for geometric modeling, and there are clear advantages for using NURBS for such a purpose. By providing a complete and efficient solution for the graphical processing of trimmed NURBS, and by the effective support of such primitives at the graphics API level, we hope that we've dropped an important obstacle for the establishment of NURBs as a common geometric representation form.

Bibliography

[1] Salim Abi-Ezzi and Leon Shirman. The tessellation of curved surfaces under highly varying transformations. In *Proc. Eurographics '91*, 1991.

[2] Salim Abi-Ezzi and Leon Shirman. The scaling behavior of a viewing transformation. *IEEE Computer Graphics and Applications*, 1993.

[3] Salim Abi-Ezzi and Michael Wozny. Factoring a homogeneous transformation for a more efficient graphics pipeline. In *Proc. Eurographics '90*, 1990.

[4] Michael Deering and Scott Nelson. Leo: A system for cost-effective 3d shaded graphics. *Computer Graphics. Proceedings of Siggraph'93*, 1993.

[5] Gerald Farin. *Curves and Surfaces in Computer Aided Geometric Design: A Practical Guide*. Academic Press, 1990.

[6] Alain Fournier and Delfin Montuno. Triangulating simple polygons and equivalent problems. *ACM Transactions on Graphics*, 1984.

[7] M. Garey, D. Johnson, F. Preparata, and R. Tarjan. Triangulating a simple polygon. *Information Processing Letters*, 1978.

[8] ISO. Computer graphics: Programmer's hierarchical interactive graphics system (phigs). part 4 - plus lumiere und surfaces, phigs plus. Technical report, International Organization for Standardization (ISO), 1991.

[9] Franco P. Preparata and Michael M. Shamos. *Computational Geometry: an Introduction,*. Springer-Verlag, 1985.

[10] Alyn Rockwood, Kurt Heaton, and Tom Davis. Real-time rendering of trimmed surfaces. *Computer Graphics. Proceedings of Siggraph'89*, 1989.

[11] Philip Schneider. *A Bezier Curve-Based Root-Finder, in Graphics Gems*, chapter 8. Academic Press, 1990.

[12] Michael Shantz and Sheue-Ling Chang. Rendering trimmed nurbs with adaptive forward differencing. *Computer Graphics. Proceedings of Siggraph '88*, 1988.

[13] Leon Shirman and Salim Abi-Ezzi. The cone of normals for fast processing of curved patches. In *Proc. Eurographics '93*, 1993.

[14] Srikanth Subramaniam, Salim S. Abi-Ezzi, and Leon Shirman. Dynamic tessellation of trimmed nurbs surfaces in sunsoft's graphics apis. In *First Annual PHIGS User's Group Conference*, 1993.

[15] SunSoft. *Solaris XGL 3.0 Programmer's Guide*.