PCT

WORLD INTELLECTUAL PROPERTY ORGANIZATION
International Bureau

INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(51) International Patent Classification 5 :
G06F 15/72

A1

(11) International Publication Number: WO 94/17486

(43) International Publication Date: 4 August 1994 (04.08.94)

(21) International Application Number: PCT/US94/00194

(22) International Filing Date: 6 January 1994 (06.01.94)

(30) Priority Data:
08/007,952 22 January 1993 (22.01.93) US

(71) Applicant: TALIGENT, INC. [US/US]; 10201 N. de Anza Boulevard, Cupertino, CA 95014 (US).

(72) Inventors: PETERSON, John; 724 Arastradero Road, #115, Palo Alto, CA 94306 (US). HO, Hsuen, Chung; 10100 N. Blaney, Cupertino, CA 95014 (US).

(74) Agent: STEPHENS, Keith; Taligent, Inc., 10201 N. de Anza Boulevard, Cupertino, CA 95014 (US).

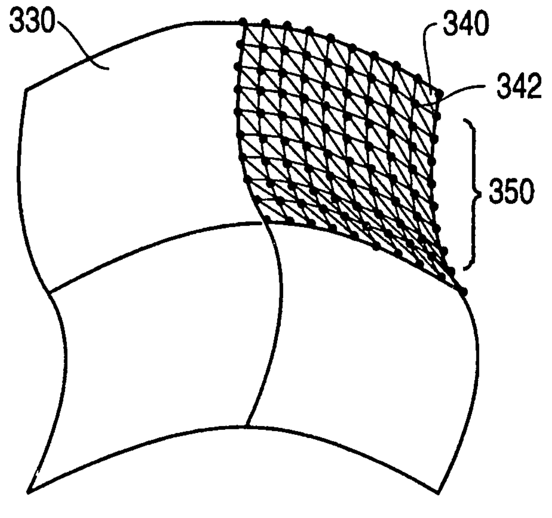
(81) Designated States: AT, AU, BB, BG, BR, BY, CA, CH, CZ, DE, DK, ES, FI, GB, HU, JP, KP, KR, KZ, LK, LU, LV, MG, MN, MW, NL, NO, NZ, PL, PT, RO, RU, SD, SE, SK, UA, UZ, VN, European patent (AT, BE, CH, DE, DK, ES, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE), OAPI patent (BF, BJ, CF, CG, CI, CM, GA, GN, ML, MR, NE, SN, TD, TG).

Published
With international search report.
Before the expiration of the time limit for amending the claims and to be republished in the event of the receipt of amendments.

(54) Title: TESSELLATION SYSTEM

(57) Abstract

The present invention provides a method and apparatus for tessellating three-dimensional spline surfaces, which are separated into columns having a series of subpatches, into shards. This is accomplished by undertaking a series of evaluations upon each of the subpatches of a selected column. The intermediate results of such evaluations are stored within caches. Such evaluations include continuity between subpatches, visibility of subpatches and granularity for discretization of the subpatches. Once the evaluations are completed, a grid which holds the discretized points of each subpatch, is computed by dynamic selection of an algorithm. Thereafter, any crack between subpatches of the selected column or the selected column and the previously selected column are removed. Ultimately, the previously selected column is rendered for illustration. Each of the columns is handled in the same manner so as to ultimately render a set of adjacent columns that form the two-dimensional representation.



TESSELLATION SYSTEM

COPYRIGHT NOTIFICATION

CROSS-REFERENCE TO RELATED PATENT APPLICATION

This patent application is related to the patent application entitled Object Oriented Framework System, by Debra L. Orton, David B. Goldsmith, Christopher P. Moeller, and Andrew G. Heninger, filed 12/23/92, and assigned to Taligent, the disclosure of which is hereby incorporated by reference.

FIELD OF THE INVENTION

This invention generally relates to a system which converts three-dimensional surfaces into two-dimensional illustrations, and more particularly, to a system which economically converts three-dimensional spline surfaces into a series of shards that represent a two-dimensional figure.

BACKGROUND OF THE INVENTION

Tessellation is the process of decomposing a three-dimensional curved surface into small discrete triangles which are referred to as shards. The size of each shard is determined by the screen space projection of the representation.

Tessellation refers to the process of breaking a high level surface description, such as a curved surface, mesh or fractal, into simple two-dimensional representations. A tessellator for a particular type of geometry contains routines for separating that particular geometry into renderable shards.

Samples are actual points on the surface of an object which is being tessellated. Each sample is one of three types of points: (i) a three-dimensional point on the surface, (ii) a normal, and (iii) a parametric coordinate of the object. Device specific samples contain additional information such as color and fixed point screen coordinates. A shard is formed by simply linking a set of three samples for a given area of a surface.

Current graphic systems undertake a complex and lengthy process, usually involving the large amounts of computation, in order to convert a three-dimensional surface into a two-dimensional display. This follows since current methods employ approaches that result in the creation and utilization of different and redundant code for various geometric primitives. Such graphic systems also may depict cracks or require costly computationally intensive techniques to remove the cracks between different areas of the same surface areas and between similar surfaces which are joined together. Thus, the prior art that applicant is aware of provides a graphic system that is inefficient for rendering two-dimensional representations from three-dimensional surfaces.

SUMMARY OF THE INVENTION

The present invention overcomes the aforementioned deficiencies of the prior art by providing a method and apparatus for converting three-dimensional surfaces into two-dimensional representations through the formation and execution of specified conversion programs.

The present invention consists of a system that converts three-dimensional spline surfaces, which are separated into columns having a series of subpatches, into shards. This is accomplished by undertaking a series of evaluations upon each of the subpatches of a selected column. The intermediate results of such evaluations are stored within caches. Such evaluations include continuity between subpatches, visibility of subpatches and granularity for discretization of the subpatches. Once the evaluations are completed, a grid which holds the discretized points of each subpatch, is computed by dynamic selection of an algorithm. Thereafter, any crack between subpatches of the selected column or between the selected column and the previously selected column are removed. Ultimately, the previously selected column is rendered for illustration. Each of the columns is handled in the same manner so as to ultimately render a set of adjacent columns that form the two-dimensional representation.

The invention also provides a method for removing portions of a surface that is either facing away from a viewer or outside of the display screen. This method only requires the examination of the basic surface specification.

Unlike the methods employed by the prior art, the present invention renders a two-dimensional representation that is free of cracks and redundant operations.

BRIEF DESCRIPTION OF THE DRAWINGS

Figure 1 is a block diagram of a computer system in accordance with the present invention;

Figure 2A illustrates a two-dimensional illustration of a NURB having twelve subpatches in accordance with the subject invention;

Figure 2B illustrates the surface parameter space of a NURB surface having twelve subpatches in accordance with the subject invention;

Figure 3A illustrates a two-dimensional surface with control points and a control mesh in accordance with the subject invention;

Figure 3B illustrates a two-dimensional surface with a shard, discretized point, grid and subpatch in accordance with the subject invention;

Figure 4 illustrates a flowchart of the logic operation of the present invention.

Figure 5 illustrates a three-dimensional surface with its accompanying control mesh;

Figure 6 illustrates adjacent tessellation grids of different densities before crack prevention occurs in accordance with the present invention; and

Figure 7 illustrates adjacent tessellation grids of different densities after crack prevention has occurred in accordance with the present invention.

DETAILED DESCRIPTION OF THE INVENTION

COMPUTER SYSTEM

A representative hardware environment is depicted in Figure 1, which illustrates a suitable hardware configuration of a workstation **40** in accordance with the present invention. The workstation **40** has a central processing unit **10**, such as a conventional microprocessor, and a number of other units interconnected via a system bus **12**. The illustrated workstation **40** shown in Figure 1 includes a Random Access Memory **14** (RAM), a Read Only Memory **16** (ROM), an I/O adapter **18** for connecting peripheral devices such as disk units to the bus **12**, a user interface adapter **22** for connecting a keyboard **24**, a mouse **26**, a speaker **28**, a microphone **32**, and/or other user interface devices such as a touch screen device (not shown) to the bus **12**. The workstation **40** may also have a communications adapter **34** for connecting the workstation **40** to a data processing network **30** and a display adapted **36** for connecting the bus **12** to a display device **38**.

INPUT PARAMETERS

The present invention operates on Non-Uniform Rational B-Spline (hereinafter "NURB") surfaces. The tessellator operates on NURB surfaces. A NURB surface is defined by the formula:

$$F(u,v) = \frac{\sum_{i=0}^{m} \sum_{j=0}^{n} V_{i,j} B_{i,k}(u) B_{j,l}(v)}{\sum_{i=0}^{m} \sum_{j=0}^{n} w_{i,j} B_{i,k}(u) B_{j,l}(v)}$$

$$B_{i,1}(u) = \begin{cases} 1 & u_i \leq u < u_{i+1} \\ 0 & \text{otherwise} \end{cases}$$

$$B_{i,r}(u) = \frac{u - u_i}{u_{i+r-1} - u_i} B_{i,r-1}(u) + \frac{u_{i+r} - u}{u_{i+r} - u_{i+1}} B_{i+1,r-1}(u) \Bigg|_{r=2,3,K,k}$$

The variables u and v denote the parameters of the surface. The variables $V_{i,j}$ and $w_{i,j}$ represent the $n \times m$ array of control points and rational weights, respectively. The functions $B_{i,k}(u), B_{i,l}(v)$ are the B-spline blending functions of order k and l having knot vectors $\{u_p\}_{p=0}^{n+k-1}$ and $\{v_q\}_{q=0}^{m+l-1}$, respectively.

The present invention operates by dividing each NURB surface into subpatches. Each NURB surface is divided into subpatches by the positioning of the knot vectors. A subpatch is defined as the parametric interval in the knot vector extending from u_i to u_{i+1} (where $u_i < u_{i+1}$) and extending from v_i to v_{i+1} (where $v_i < v_{i+1}$).

For instance, a NURB surface may have a knot vector in u of {0,0,0,1,1,2,3,3,3} and a knot vector in v of {0,0,0,1,2,3,4,4,4}. Referring to Figure 2A, such a NURB surface is illustrated. The resulting twelve subpatches are referred to with reference numerals **200, 202, 204, 206, 208, 210, 212, 214, 216, 218, 220,** and **222**. It should be noted that a parametric discontinuity, such as discontinuity **230**, can only occur at subpatch boundaries.

Referring to Figure 2B, the surface parameter space of the NURB surface is graphically illustrated. Since the knot vector in u reaches four, five knot vectors are formed in parallel to the u vector. Accordingly, four rows of subpatches are formed in perpendicular to the u vector, that is, in the v direction. The four rows of subpatches across in the v direction have intervals between 0 and 1, 1 and 2, 2 and 3, and 3 and 4. In contrast, four knot vectors are formed in perpendicular to the v vector direction since the knot vector in v reaches three. This translates into the formation of three columns of subpatches in perpendicular to the v vector, that is, in the u direction. The resulting three columns of subpatches across the u direction have intervals between 0 and 1, 1 and 2, and 2 and 3. The combinations of four rows and three columns yields the aforementioned twelve subpatches.

In addition to the surface input parameters, the present invention also accepts the following input parameters: (a) the resolution of the display screen; (b) a three-dimensional transformation matrix to convert the three-dimensional surface coordinates into two-dimensional screen coordinates; (c) clipping information to determine if a three-dimensional coordinate will be outside of the two-dimensional screen display area; and (d) a flag indicating whether or not backfacing portions of surfaces may be culled.

GENERAL LOGIC OPERATION

The present invention operates by analyzing the three-dimensional surface in a number of steps. First, a column of subpatches is selected. Columns are selected from left to right in the parameter space of the surface. Second, each subpatch within the column is analyzed. The result of the analysis is the computation of a set of grids which correspond to the set of subpatches within the column. A grid is a rectangular array of discretized points on a surface. A crack is a separation or gap between a mesh of shards. A shard is a triangular portion of a surface with a set of vertices defined by a set of discretized points on the surface. The set of grids are subsequently removed of any: (i) cracks as between each subpatch in the column, and (ii) cracks as between the selected column and the previously selected column (the column to the left of the currently selected column). Since the present invention generates columns of grids from left to right, a particular column of grids can only be drawn when its neighboring column(s) have been computed. Thus, after computation of the grids for the selected column, the previous column of grids is drawn. Thereafter, the adjacent right column of subpatches is selected and the entire process is repeated.

Figure 3A illustrates a two-dimensional surface with control points and a control mesh and Figure 3B illustrates a two-dimensional surface with a shard, discretized point, grid and subpatch in accordance with the subject invention. The control points 300 and 302 define the dimensions of the control mesh 310 and the control mesh facets 320. Examples of subpatches are also provided at 330. A shard is presented at 340 and is defined by a set of discretized points 342. A grid of discretized points is also illustrated at 350.

Referring to Figure 4, a flowchart of the logic operation of the present invention is illustrated. Initially, as indicated by logic block **400**, caches are allocated for storing intermediate results. Thereafter, as indicated by block **402**, an interval in the *u* direction is selected. Intervals in the *u* direction are selected from left to right, that is, in order in which they appear from the origin of the *u* vector. For instance, the NURB surface shown in figures 2 and 3 would first be selected and analyzed with respect to the interval between 0 and 1. Such interval contains subpatches **200, 202, 204** and **206**. The NURB surface would subsequently be selected and analyzed with respect to the interval between 1 and 2, and then ultimately with respect to the interval between 2 and 3.

Once a column composing an interval in the *u* direction has been selected, each interval in the *v* direction which exists within the selected interval in the *u* direction is analyzed. Accordingly, a subpatch within the selected column is selected in incremental order from the *v* vector. This is denoted by block **408**. For instance, subpatch **200** of the NURB surface shown in figures 2 and 3 would first be selected and analyzed when the interval in the *u* direction between 1 and 2 of the NURB surface has been selected. Subsequently, subpatches **202, 204** and **206**, in that order, would be analyzed. Hence, the selection of subpatches is a nested loop within the selection of an interval in the *u* direction.

The selected subpatch is first analyzed to determine whether the subpatch is visible. That is, determination is made as to whether (i) the subpatch is backface culled, or (ii) whether the subpatch is clipped from the display. This is indicated by block **410**. An affirmative determination of backface culling or clipping leads the flow of control to block **408** wherein a subsequent subpatch is selected and thereafter analyzed.

When it is determined that the selected subpatch is not backface culled or clipped, then, the flow of control leads to further analysis of the already selected subpatch by block **412** through block **428**. Block **412** represents a determination of granularity for the discretization of the selected subpatch. Once granularity is determined, the continuity of the subpatch with its adjacent subpatch within the selected column is determined. This is indicated by block **414**. Continuity is determined as between the selected subpatch and the adjacent subpatch in the *v* direction. In other words, continuity is determined between the selected subpatch and the next subpatch to be selected within the logic of block **408**. Upon a determination of continuity, the cache basis functions of the subpatch are determined as indicated by block **416**.

Still further analysis of the selected subpatch includes determination of (i) the existence of a high granularity of discretization as indicated by block **418**, and (ii) the position of the subpatch with respect to the edges of the surface as indicated by block **424**. A determination of a high granularity of discretization directs that the grid of discretized points for the subpatch be computed by a forward differencing algorithm. Such is indicated by block **422**. In contrast, a determination that the granularity of discretization is low results in the grid of discretized points of the subpatch being determined by a direct evaluation algorithm. This is denoted by block **420**.

Once the grid of discretized points of the selected subpatch has been computed, either by forward differencing or direct evaluation, a determination of the position of the subpatch with respect to the edges of the surface is undertaken. This is indicated by block **424**. If the subpatch is on the edge of the surface, then the edge is computed independently as indicated by block **426** and control passes to function block **428**. If no portion of the subpatch is determined to be along the surface, then the flow of control leads directly to block **428**.

Block **428** calls for the sealing of all cracks existing between the grid formed from the selected subpatch and the grid formed from the previously selected subpatch. That is, the sealing of any crack as determined by block **414** is implemented. Thereafter, the flow of control returns to block **408** for selection of a subsequent subpatch within the interval selected by block **402**.

Once each subpatch within the column selected by block **402** has been analyzed, the selected interval is determined for continuity as between itself and the adjacent interval in the *u* direction that was previously selected and analyzed. This is indicated by block **404**. For instance, the interval in the *u* direction between 2 and 3 of the NURB surface, as illustrated in figures 2 and 3, would be analyzed for continuity as between it and the interval in the *u* direction between 1 and 2.

After a determination of continuity between the selected column and the previously selected column, all determined cracks are sealed. This is indicated by block **424**. Since the previously selected column has now been removed of cracks shared with both of its adjacent columns, the column of grids for the previously selected column is drawn. In addition, the column of grids just computed now becomes the previous column with regard to future analysis upon a newly selected column. This is indicated by block **430**. Thereafter, the flow of control is remanded to block **402** wherein an adjacent column in the *u* direction is selected for analysis.

Since the tessellator processes the subpatches one interval at a time, crack prevention and output of the first column can only occur when at least two columns are generated. It follows that the output is always one column behind the computation or generation of a column of grids. Accordingly, the previously generated column of grids, for which a rendering now exists, is flushed as indicated by block **440**.

DETAILED LOGIC OPERATION OF THE PRESENT INVENTION

Various applications invoke the present invention for tessellation of surfaces. Three dimensional modeling and computer automated design ("CAD") applications are examples of applications that invoke a the present invention. Such applications form a control mesh defining a three-dimensional surface that is sought to be tessellated.

Referring to Figure 5, a three-dimensional surface is illustrated with its control mesh. The three-dimensional surface **500** has a control mesh which has frontal facet **502**, rear facet **504**, and side facet **506**. Frontal facet **502** of the control mesh is composed of lower triangle **510** and upper triangle **512**. Rear facet **504** of the control mesh has lower triangle **514** and upper triangle **516**. Side facet **506** of the control mesh has lower triangle **518** and upper triangle **520**.

Upon receiving input parameters, the present invention allocates various information within caches. The caches are employed to store intermediate result of various evaluations. The following caches are formed. First, a cache of integer breakpoint intervals which are the indices into the knot vectors that mark the boundaries between the subpatches is formed. This avoids subsequent determinations of subpatch boundaries. Second, caches for the basis functions of the splines in *u* and *v*, and their derivatives are also formed. Since the basis functions are constant across the surface of a particular value of *u* or *v*, redundant computation is avoided by formation of this cache. Third, a cache is formed to retain the transformed control points of one subpatch. This cache will be utilized for subsequent culling and clipping tests. Fourth, a cache is formed to retain two columns of computed grids. Each grid is composed of the discretized points of a subpatch. The shards, for a particular subpatch, are then rendered from the discretized points of the grid which correspond to the subpatch.

Once all the caches are formed, the present invention transforms all of the surface control points by the three-dimensional transformation matrix. Thereafter, the present invention selects a first interval in the *u* direction. The first interval is in essence a column. Each subpatch of the selected column is then analyzed to ultimately generate a column of subpatches with no cracks as between the subpatches of the column.

The present invention initially analyzes each subpatch to determine if the subpatch is visible. A subpatch is visible unless it is backface culled or completely clipped from the display screen. Backface refers to portions of the surface which are facing away from a viewer after the surface has been transformed for viewing on the display.

The evaluation of backfacing is carried out by an algorithm that examines portions of the control mesh for backfacing elements. The existence of backfacing is determined by initially inspecting the $k \times l$ grid of control points that defines the particular subpatch. The triangles of each set of adjoining control points are then examined with respect to backfacing. In the event that all of the triangular facets of a particular control mesh are backfacing, then the entire subpatch is regarded as backfacing. Consequently, that particular control mesh will not be rendered if backfacing surfaces are to be culled.

During the backfacing analysis of the control mesh facets, two flags are kept. One flag is set to true if any one of the control mesh facets are determined to be backfacing. When any one of the control mesh facets are backfacing, a lower level rendering code must perform backfacing analysis upon each individual shard of the subpatch associated with the facet. The second flag is set to true when all facets of a control mesh are backfacing. When all of the facets are determined to be backfacing, the entire subpatch is ignored if backfacing surfaces are not to be rendered.

After the present invention has undertaken the backfacing analysis, the control points of each subpatch are compared to the clipping fustrum for a desired viewing perspective. An algorithm is employed to determine if the convex hull for a portion of the surface is outside the bounds of the display screen. In the event that the bounding box enclosing the usable display area is not within the bounds of the display screen, then the subpatch is clipped and thus not rendered. As with backfacing evaluations, the present invention identifies and recalls the clipped subpatches so as to avoid redundant tests on individual shards for subsequent renderings.

Subpatches that are determined to be invisible, whether because of being backface culled or completely clipped from the display screen, are denoted by an appropriate flag. The flagged subpatches are not further analyzed with respect to discretization and shading.

After all invisible subpatches are determined, a determination of granularity for discretization is undertaken. Granularity refers to the density of discretized points in the grid. Thus, if the subpatch appears large on the display screen and the subpatch is highly curved, it is likely that the granularity will be high. Conversely, if the surface is relatively small and flat, the granularity will be low.

The present invention makes a granularity determination by examining the magnitude of the second derivative of the surface in both the u and v directions. The measurement of magnitude is taken in a number of places to find the maximum degree of curvature. This parameter is employed as the number of discrete grid samples to be generated for the subpatch. A grid is then allocated for the subpatch that is capable of storing the discretized points.

Upon a determination of granularity, a continuity analysis is performed with regard to each subpatch in the selected column. Each subpatch is analyzed with respect to each of its neighboring subpatch in the column for continuity. The continuity is determined by examining the knot vector in the u and v parameters. The results obtained from the continuity examination are employed during the process of crack prevention to provide for a correct formulation of the seams between the subpatches of a column.

Where the number of rows or columns of control points exceeds the order of the surface in the u or v direction, such surface must be rendered in multiple segments. Each segment is an individual polynomial curve. These segments are joined together with different continuities, which are referred to by the symbol C_n. The variable n refers to the order of the continuity. A change in position is a C_0 discontinuity; thus a C_0 discontinuity is a break in the curve. A change in tangent is a C_1 discontinuity; thus a C_1 discontinuity is an abrupt change in the slope of a curve. A change in speed is a C_2 discontinuity; thus a C_2 discontinuity is a change in the speed.

Surfaces having a Bézier knot vector must be treated in a different fashion than other surfaces. This results since a knot vector may indicate a C_1 joint even though the control points are co-linear. In this instance, the subpatches should be smoothly joined and the joint marked as C_2 continuous. The continuity of the joint is used to guide the crack suppression.

After determining whether the subpatch possessed a high or low granularity, the grid for selected subpatch is computed by either of two algorithms. Initially, the caches for the spline basis value are initialized for the particular subpatch in accordance with the knot vector of the spline. If the large number of samples is required, then a forward differencing algorithm is undertaken. Specifically, forward differencing is employed when the number of samples in the u or v direction, as determined by the granularity measurements, are greater than two times the order of the direction (where k and l are the orders of the u and v surfaces, respectively). Thus, if the granularity measurements determines that the number of samples in u is greater than $2k$, or the number of samples in v is greater than $2k$, then the forward differencing algorithm is employed. Otherwise, direct evaluation algorithm is employed.

The forward differencing algorithm initially evaluates the $k \times l$ points of the subpatch so as to form an array of forward difference values. The points of the array are generated by evaluating a surface column in the v direction. Subsequent columns of points are obtained by forward differencing the forward difference values of the evaluated surface column.

The present invention generates normal vectors from the control mesh by averaging the normals from the triangles surrounding a particular point. The normals of the edges of the surface are, however, computed independently by direct evaluation.

Upon computation of the grid by selection and application of the appropriate algorithm, a determination of whether the subpatch lies on the perimeter of the surface is undertaken. This determination is undertaken to prevent an equivalent edge of two adjacent subpatches from tessellating an equivalent edge differently. Two edges are equivalent if their control points, knot vectors and orders are the same. Thus, the sampling density for all subpatches on the edge of a surface are computed independently.

A final step in the present invention is crack prevention wherein cracks are removed. In the event that the knot vector indicates the existence of a C_0

discontinuity (a surface break), no crack prevention is undertaken and the discontinuity is denoted by a flag. Similarly, C_1 discontinuities are denoted by a flag so that the code which is implemented to remove cracks ignores these discontinuities. Otherwise, the C_0 and C_1 discontinuities would be removed to produce an incorrect rendering.

Cracks of each subpatch within a selected column are initially removed. Thereafter, the cracks between the selected column and the previously selected column are determined and removed.

Cracks occur in a rendering when two adjacent subpatches are sampled at different densities. To prevent the cracks, the present invention matches each point of the less dense grid with the points of the dense grid. Points of the more dense grid which are unable to be matched are forced to remain on the line between the edges of the grids.

Referring to Figure 6, adjacent tessellation grids of different densities are illustrated before crack prevention occurs. The edge line **650** which separates the edges of the two grids contains thirteen points. The edge line **650** contains a top shared end point, a bottom shared end point and a shared mid-point, which are referred to by reference numerals **600**, **602**, and **604** respectively. Four points on the edge line **650**, which are referred to by reference numerals **606**, **608**, **610**, and **612**, are the unshared edge points of the less dense grid. Six points on the edge line **650**, which are referred to by reference numerals **614**, **616**, **618**, **620**, **622**, and **624**, are the unshared edge points of the more dense grid.

Referring to Figure 7, adjacent tessellation grids of different densities are illustrated after crack prevention occurs. After the crack prevention procedure has been undertaken, four shared points are formed upon the edge line **650**. Namely, points **606**, **608**, **610**, and **612**, which were formerly the unshared edge points of the less dense grid, have been joined with points **614**, **618**, **620**, and **624**, which were formerly the unshared edge points of the more dense grid. Accordingly, joined points **702**, **704**, **706**, and **708** are illustrated. Points **616** and **622** are projected to a line as the unshared edge points of the more dense grid.

Given arrays of points "left" and "right" having "numleft" and "numright" points, respectively, the following algorithm is employed for eliminating cracks between these set of points:

```
procedure linkup (left, right: pt_array_t; numleft, numright: integer);
   var
      i, j, curInd, lastInd, minpts: integer;
      curDist, ratio: float;
      minside, maxside: pt_array_t;
      tmp: point_t;
begin
```

```
    if numleft < numright then
      begin
        ratio := (numright - 1.0) / (numleft - 1.0);
        minside := left;
        maxside := right;
        minpts := numleft;
      end
    else
      begin
        ratio := (numleft - 1.0) / (numright - 1.0);
        minside := right;
        maxside := left;
        minpts := numright;
      end;
    curDist := 0.0;
    curInd := 0;
    maxside[0] := minside[0];
    for i := 1 to minpts - 1 do
      begin
        curDist := curDist + ratio;
        lastInd := curInd;
        curInd := round(curDist);
        for j := lastInd + 1 to curInd - 1 do
          begin
            tmp := maxside[j];
            ProjectToLine(minSide[i - 1], minSide[i], tmp);
            maxside[j] := tmp;
          end;
        maxside[curInd] := minside[i];
      end;
  end;
```

The crack prevention procedure initially examines the edge line between a first column of grids, which has only one neighboring column, and a second generated column. Once the crack prevention examination has been completed as between the first and second columns, the first column of tessellation grids is rendered. Thereafter, the third column is generated and the edge line between the second column and the third column is examined. Once crack prevention has been completed as between the second and third columns, the second column of grids is rendered. This continues until all edge lines have been examined for crack prevention.

In addition to crack prevention between grids, crack prevention procedures is also employed as between adjacent NURB surfaces. The procedure for examining cracks between NURB surfaces is similar to the aforementioned procedure of crack prevention between grids. Since the surfaces are rendered independently of one another, it is possible for two adjacent surfaces that have an identical edge to tessellate the edge differently.

Ultimately, the previously selected column is rendered. This follows since a column can only be rendered upon generation of itself as well as upon the generation of adjacent columns. Otherwise, crack prevention between the adjacent columns could not take place. The last generated column then replaces the previously generated column for utilization within the evaluation of the next selected column. The aforementioned process is then repeated upon the next column selected. Such repetition continues until all columns have been rendered.

An embodiment of the invention developed by the inventor to render three-dimensional surfaces is provided below to clarify the detailed logic of the present invention.

```
#ifndef __PIPELINE3D__
#include <Pipeline3D.h>
#endif
#ifndef __TESSELATION__
#include <Tesselation.h>
#endif
#ifndef __PRSPLINEROUTINES__
#include "prSplineRoutines.h"
#endif
#ifndef __STDLIB__
#include <StdLib.h>
#endif
// for qprintf only:
#ifndef __RUNTIME__
#include <RunTime.h>
#endif
#ifndef __MATH__
#include <Math.h>        // For fabs
#endif
#ifndef __STORAGE__
#include <Storage.h>     // for tracking memory leaks
#endif
#ifndef __KERNELCALLS__
#include <kernelCalls.h>        // For OpusBug()
#endif
#ifndef __SURFACE3D__
#include <Surface3D.h>          // for TGSurface3D
#endif
///////////// Generic TPipeline3D defs ///////////////////
TPipeline3D::TPipeline3D( Boolean   cullBackFaces )
{
        fCullBackFaces = cullBackFaces;
}
////////////////// TSampleGrid defs ///////////////////
TGridEdge::TGridEdge()
{
        fSamples = NIL;
        fEdgeMax = 0;
}
TGridEdge::~TGridEdge()
{
        delete fSamples;
        fEdgeMax = 0;
}
Boolean TGridEdge::IsSet()
{
        return (fSamples != NIL);
}
TSampleGrid::TSampleGrid()
{
        fMaxU = fMaxV = 0;
        fSamples = NIL;
}
TSampleGrid::~TSampleGrid()
{
        delete fSamples;
        fSamples = NIL;
}
//
// Allocate a subgrid of sampled points.  If the orginal
// storage is too small, extend it.
//
void TSurfaceTessellator::AllocSampleGrid( TSampleGrid& grid, long newV,
long newU )
{
        long n = (newV + 1) * (newU + 1);
        if (! grid.fSamples)
        {
                grid.fSamples = new TSurfaceSample[n];
        }
        else
                if ( (grid.fMaxU + 1) * (grid.fMaxV + 1) < n )
                {
                        grid.fSamples = (TSurfaceSample *)
                                realloc( grid.fSamples,
                                                                (size_t) (sizeof(
TSurfaceSample ) * n ));
```

```
        }
    grid.fMaxU = newU;
    grid.fMaxV = newV;
}
////////// Constructor/Destructor /////////////
TSurfaceTessellator::TSurfaceTessellator( const TGSurface3D& surface,
                    TPipeline3D * tessInfo,
                    GCoordinate resolution,
                    GCoordinate C1Tolerence,
                    long          minSteps )
{
    register long i, brkPoint;

    fTessInfo = tessInfo;
    fResolution = resolution;
    fC1Tolerence = C1Tolerence;
    fMinSteps = minSteps;

    // disgorge the surface
    fNumPointsU = surface.GetNumberPoints( TGSurface3D::kuDirection );
    fNumPointsV = surface.GetNumberPoints( TGSurface3D::kvDirection );
    fOrderU     = surface.GetOrder( TGSurface3D::kuDirection );
    fOrderV         = surface.GetOrder( TGSurface3D::kvDirection );
    fKnotsU     = surface.CopyKnots( TGSurface3D::kuDirection );
    fKnotsV     = surface.CopyKnots( TGSurface3D::kvDirection );
    fPoints         = surface.CopyPoints();

    // Re-scale the parameter range to 0..1

    u0 = fKnotsU[0];
    u1 = fKnotsU[fNumPointsU];
    uInc = 1.0 / (u1 - u0);
    for (i = 0; i < fNumPointsU + fOrderU; i++)
        fKnotsU[i] = (fKnotsU[i] - u0) * uInc;

    v0 = fKnotsV[0];
    v1 = fKnotsV[fNumPointsV];
    vInc = 1.0 / (v1 - v0);
    for (i = 0; i < fNumPointsV + fOrderV; i++)
        fKnotsV[i] = (fKnotsV[i] - v0) * vInc;

    // Flag basis caches as empty
    maxGranV = -1;  maxGranU = -1;
    basisCacheV = basisCacheDV = NIL;

    // Temp arrays used for vertical curves, forward      diff scratch
```

```
        i = max( fNumPointsU, fNumPointsV ) + 1;
        temp = new TGRPoint3D[i];
        dtemp = new TGRPoint3D[i];

        //
        // Make a cache for the screen space
        // ctl points to do the test.
        //
        fMeshSubPatch = new TGPoint3D[fOrderU * fOrderV];

        //
        // Precalculate the breakpoint intervals in U and V
        // (should add flags to record c0 discontinuties)
        // Size of KV is worst case size.
        //

        brkPointsU = new long[fNumPointsU + fOrderU];
        brkPointsV = new long[fNumPointsV + fOrderV];

        brkPoint = FindInterval( fKnotsU[fOrderU - 1], fKnotsU, fNumPointsU,
fOrderU );
        numBrksU = 0;
        while( fKnotsU[brkPoint] < fKnotsU[fNumPointsU] )
        {
                brkPointsU[numBrksU] = brkPoint;
                brkPoint = FindNextInterval( brkPoint, fKnotsU, fNumPointsU );
                numBrksU++;
        }

        brkPoint = FindInterval( fKnotsV[fOrderV - 1], fKnotsV, fNumPointsV,
fOrderV );
        numBrksV = 0;
        while( fKnotsV[brkPoint] < fKnotsV[fNumPointsV] )
        {
                brkPointsV[numBrksV] = brkPoint;
                brkPoint = FindNextInterval( brkPoint, fKnotsV, fNumPointsV );
                numBrksV++;
        }
        // For caching the basis functions in U

        basisCacheU      = new GCoordinate[fOrderU + 1];
        basisCacheDU     = new GCoordinate[fOrderU + 1];

        // The forward difference matrix
        deltas = new TGRPoint3D*[fOrderU];
        for (i = 0; i < fOrderU; i++)
                deltas[i] = new TGRPoint3D[fOrderV];

        //
        // For each subpatch of the surface, a "subgrid" of points is computed.
        // We keep the last column of subgrids around, so any potential cracks
        // can be patched up if the two grids are sampled at different densities.
        //

        lastGrids     = new TSampleGrid[numBrksV];
        curGrids      = new TSampleGrid[numBrksV];      // Calls constructors
}
//
// Destructor
//
TSurfaceTessellator::~TSurfaceTessellator()
{
        register long i;

        delete brkPointsU;
        delete brkPointsV;
        delete temp;
        delete dtemp;

        delete[numBrksV] curGrids;
        delete[numBrksV] lastGrids;
        delete fMeshSubPatch;

        for (i = 0; i < fOrderU; i++)
                delete deltas[i];
        delete deltas;

        for (i = 0; i <= maxGranV; i++)
        {
                delete basisCacheV[i];
                delete basisCacheDV[i];
        }
        delete basisCacheV;
        delete basisCacheDV;

        delete basisCacheU;
        delete basisCacheDU;

        delete fPoints;
        delete fKnotsU;
        delete fKnotsV;
}
////////// Granularity determination //////////////////
//
// Determine if the current subpatch of a surface is visible
//
Boolean
TSurfaceTessellator::SubPatchVisible(
        long Vbase,
        long Ubase,
        unsigned long& clipInCode,
        eFrontFace& frontFaceCode )
{
        register long vi, ui;
        unsigned long clipCode, outCode = 0xFF;    /* All out */
        TGPoint3D meshPt;
        Boolean frontFace, anyFrontFace, allFrontFace;

        clipInCode = 0;     /* All in */

        for (vi = 0; vi < fOrderV; vi++)
                for (ui = 0; ui < fOrderU; ui++)
                {
                        meshPt = fPoints[(Vbase + vi)*fNumPointsU+(Ubase +
ui)].DivW();
                        clipCode = fTessInfo->GetClippingCode( meshPt );
                        outCode &= clipCode;
                        clipInCode |= clipCode;
                        if (fMeshSubPatch) fMeshSubPatch[vi * fOrderU + ui] =
meshPt;
                }
        if (! (outCode == 0))               /* if code == 0, then something was visible
*/
                return( FALSE );
        // Determine back/front facing of surface by looking at the
        // facets formed by the control mesh

        allFrontFace = TRUE;
        anyFrontFace = FALSE;
        // Find plane equations for all facets of the mesh...
        for (vi = 0; vi < fOrderV-1; vi++)
                for (ui = 0; ui < fOrderU-1; ui++)
                {
                        frontFace = fTessInfo->IsFrontSurface(
                                                fMeshSubPatch[vi * fOrderU + ui],
                                        fMeshSubPatch[(vi+1) * fOrderU +
ui],
                                        fMeshSubPatch[vi * fOrderU + ui +
1] );

                    allFrontFace = (allFrontFace && frontFace);
                    anyFrontFace = (anyFrontFace | | frontFace);

                    frontFace = fTessInfo->IsFrontSurface(
                                        fMeshSubPatch[(vi+1) * fOrderU + ui
+ 1],
                                        fMeshSubPatch[vi * fOrderU + ui +
1],
                                        fMeshSubPatch[(vi+1) * fOrderU +
ui] );
                    allFrontFace = (allFrontFace && frontFace);
                    anyFrontFace = (anyFrontFace | | frontFace);
               }


     if (allFrontFace) frontFaceCode = TSampleGrid::kAllFront;
     else
     if (!anyFrontFace) frontFaceCode = TSampleGrid::kAllBack;
     else
     frontFaceCode = TSampleGrid::kSomeFront;

     return( anyFrontFace | | !(fTessInfo->fCullBackFaces) );
}
//
// Find the 2nd derivative, used for computing deviation tolerance
//
GCoordinate
TSurfaceTessellator::GetDerivMag(
                         GCoordinate u,
                         long brkPoint,
                         long order,
                         const GCoordinate * const kv,
                         long rowOrCol,          // The row or column index
                         Boolean rowflag )
{
     const kMaxLocal = 6;
     GCoordinate local_bvals[kMaxLocal], local_dvals[kMaxLocal];
     GCoordinate * bvals, * dvals;
     TGPoint d, cp;
     long i;
```

```
        if (order > kMaxLocal)
        {
                bvals = new GCoordinate[order];
                dvals = new GCoordinate[order];
        }
        else
        {
                bvals = local_bvals;
                dvals = local_dvals;
        }

        /* Note: splinevals should be rigged so that if bvals
         * aren't needed (i.e., just use the deriv) then they
         * aren't computed.
         */
        ComputeSplineValues( u, brkPoint, order, kv, bvals, NULL, dvals );

        d = TGPoint::kOrigin;
        for (i = 0; i < order; i++)
        {
                // Project the 3D control points to 2D to find the "curvature"
                // of the projected surface along this isoline.  What should
                // this do with W???
                if (rowflag)
                        cp = fTessInfo->DeviceProjection(
fPoints[rowOrCol*fNumPointsU+(brkPoint-i)] );
                else
                        cp = fTessInfo->DeviceProjection( fPoints[(brkPoint -
i)*fNumPointsU+rowOrCol] );
                d.fX += cp.fX * dvals[i];
                d.fY += cp.fY * dvals[i];
                // d.fW += cp.fW * dvals[i];
        }

        if (bvals != local_bvals)
        {
                delete bvals;
                delete dvals;
        }
        return( sqrt( d.fX * d.fX + d.fY * d.fY ) );
}
//
// Determine the step size for a given breakpoint interval on the curve.
// This uses a crude bounds on the second derivative.  Needs More Thought.
//
long
```

```
TSurfaceTessellator::GetGran(
        long brkPointV,
        long brkPointU,
        Boolean rowflag )
{
        GCoordinate * kv;
        long brkPoint, index, numRows, row, i, order;
        GCoordinate u, mag, maxMag;

        kv = (rowflag ? fKnotsU : fKnotsV);
        if (rowflag)
        {
                order = fOrderU;
                brkPoint = brkPointU;
                index = brkPointV - (fOrderV - 1);
                numRows = fOrderV;
        }
        else
        {
                order = fOrderV;
                brkPoint = brkPointV;
                index = brkPointU - (fOrderU - 1);
                numRows = fOrderU;
        }
        maxMag = -1e30;

        /* Need special case for twisted bi-linears... */
        if (order == 2)                 /* Linear, interval is flat */
                return( 2 );

        for (row = 0; row < numRows; row++)
                for (i = 0; i < order-2; i++)
                {
                        if (order == 3)
                                u = kv[brkPoint];
                        else
                                u = kv[brkPoint] + (kv[brkPoint + 1] - kv[brkPoint]) * (i /
(order-3));
                        mag = GetDerivMag( u, brkPoint, order, kv, index + row,
rowflag );
                        maxMag = max( mag, maxMag );
                }
        i = (long)sqrt( maxMag / fResolution );
        i = max( fMinSteps ? fMinSteps : (rowflag ? fOrderU : fOrderV), i );
        return(i);
}
```

```
//
// Like GetGran, but only measures along the edge.  Assuming edges that meet
// are exactly the same, measuring just the edge curve should ensure like
// edges Tessellate the same, and thus don't crack.
//
long
TSurfaceTessellator::GetEdgeGran(
        long brkPoint,
        long rowOrCol,
        Boolean rowflag )
{
        GCoordinate mag, backmag;
        long gran, order;

        order = rowflag ? fOrderU : fOrderV;
        /* Special case for linear */
        if (order == 2)
                return 2;

        if (rowflag)
                mag = GetDerivMag( fKnotsU[brkPoint], brkPoint, fOrderU,
fKnotsU, rowOrCol, TRUE );
        else
                mag = GetDerivMag( fKnotsV[brkPoint], brkPoint, fOrderV,
fKnotsV, rowOrCol, FALSE );

        // 2nd derivative is constant for quadratic and less, but may
        // change for cubic and larger.  Need to measure both ends in
        // case two opposing curves are next to each other.
        backmag = 0.0;
        if (order >= 4)
        {
                if (rowflag)
                        backmag = GetDerivMag( fKnotsU[brkPoint+1L], brkPoint,
fOrderU, fKnotsU, rowOrCol, TRUE );
                else
                        backmag = GetDerivMag( fKnotsV[brkPoint+1L], brkPoint,
fOrderV, fKnotsV, rowOrCol, FALSE );
        }

        mag = fmax( mag, backmag );
        gran = (long)sqrt( mag / fResolution );
        gran = max( rowflag ? fOrderU : fOrderV, gran );
        return gran;
}
///////////////////////// Evaluation ///////////////////
```

```
//
// Given a position (step index, actually) along the subpatch in U,
// compute a curve with orderV control points going down. The control
// points (and derivs) are placed in temp (dtemp). Points that lie
// on this curve are also on the surface.
//
void
TSurfaceTessellator::ComputeVCurve(
        long stepU,
        long deriv )
{
        GCoordinate u;
        register long stepV, j;

        u = u0 + stepU * uInc;
        ComputeSplineValues( u, curBrkU, fOrderU, fKnotsU, basisCacheU,
                              (deriv ? basisCacheDU : NULL), NULL );
        /* Find control points of a curve going down surface from those going
across */

        j = curBrkV-(fOrderV-1);

        if (deriv)
                for (stepV = 0; stepV < fOrderV; stepV++)
                {
                        BasisMul( curBrkU, fOrderU, basisCacheU,
                                        fPoints+(j + stepV)*fNumPointsU,
temp[stepV + j] );
                        BasisMul( curBrkU, fOrderU, basisCacheDU,
                                        fPoints+(j + stepV)*fNumPointsU,
dtemp[stepV + j] );
                }
        else
                for (stepV = 0; stepV < fOrderV; stepV++)
                        BasisMul( curBrkU, fOrderU, basisCacheU,
                                        fPoints+(j + stepV)*fNumPointsU,
temp[stepV + j] );
}
//
// Allocate the cache for the basis functions. If it's already
// there, make sure it's big enough, enlarging if not.
//
void
TSurfaceTessellator::AllocBasisCache(
                GCoordinate ** & cachePtr,
                long order,
                long newSteps,
                long& maxSteps )
{
        long i;

        if (! cachePtr)
        {
                cachePtr = new GCoordinate*[newSteps + 1];
                for (i = 0; i <= newSteps; i++)
                        cachePtr[i] = new GCoordinate[order];
        }
        else    /* must enlarge cache */
        {
                cachePtr = (GCoordinate **) realloc( cachePtr,
                                                                        (size_t)
(sizeof( GCoordinate * ) * (newSteps + 1)) );
                for (i = maxSteps+1L; i <= newSteps; i++)
                        cachePtr[i] = new GCoordinate[order];
        }
        maxSteps = newSteps;
}
////////////////// Crack Prevention //////////////////
//
// Return TRUE if the points rp0, rp1 & rp2 are colinear "enough".
//
Boolean
TSurfaceTessellator::TestColinear(
                const TGRPoint3D& rp0,
                const TGRPoint3D& rp1,
                const TGRPoint3D& rp2 )
{
        const GCoordinate EPSILON = 1.0e-12;
        TGPoint3D p0, p1, p2;
        TGPoint3D perp, vec0, vec1;
        GCoordinate dist0, dist1, costheta;

        p0 = rp0.DivW();
        p1 = rp1.DivW();
        p2 = rp2.DivW();
        vec0 = p0 - p1;
        vec1 = p2 - p1;

        /*
         * Normalize by hand.  If the vectors are zero length, then return true
         * (co-located == co-linear)
         */
        dist1 = vec1.Normalize();
        if (dist1 < EPSILON)
                return( TRUE );

        dist0 = vec0.Normalize();
        if (dist0 < EPSILON)
                return( TRUE );
        costheta = vec0.DotProduct( vec1 );
        if (fabs( costheta + 1.0 ) > fC1Tolerence)
                return( FALSE );
        else
                return( TRUE );
}
//
// This is a sad story.  StitchEdges needs to know if a curve is
// c1 or c2 at a seam between subgrids.  If it's c2, then both the
// normals and the points on the seam must be aligned.  If it's
// just c1 continous, then only the points should be aligned.
// Unfortunatly, many piecewise Bézier curves are c1 according to
// the knot vector, but really c2 because the control points are
// colinear.  This needs to be checked for so that StitchEdges can
// do the right thing.
//
long
TSurfaceTessellator::IsReallyC1(
                                        long brkPtU,
                                        long brkPtV,
                                        Boolean rowflag )
{
        long i;
        Boolean isColinear = TRUE;

        if (rowflag)
        {
                for (i = 0; isColinear && i < fOrderV; i++)
                        isColinear = TestColinear( fPoints[(brkPtV-
i)*fNumPointsU+(brkPtU - 1)],
                                                                fPoints[(brkPtV-
i)*fNumPointsU+(brkPtU)],
                                                                fPoints[(brkPtV-
i)*fNumPointsU+(brkPtU + 1)] );
        }
        else
        {
                for (i = 0; isColinear && i < fOrderU; i++)
```

```
                    isColinear = TestColinear( fPoints[(brkPtV -
1)*fNumPointsU+(brkPtU-i)],
                                                fPoints[(brkPtV)
*fNumPointsU+(brkPtU-i)],
                                                fPoints[(brkPtV +
1)*fNumPointsU+(brkPtU-i)] );
        }
        return( isColinear ? 2 : 1 );
}
//
// Given a line defined by firstPt and lastPt, project midPt onto
// that line.
//
void
TSurfaceTessellator::ProjectToLine(
                            const TGPoint3D& firstPt,
                            const TGPoint3D& lastPt,
                            TGPoint3D& midPt )
{
        TGPoint3D base, v0, vm;
        GCoordinate fraction, denom;

        base = firstPt;

        v0 = lastPt - base;
        vm = midPt - base;
        denom = v0.DotProduct( v0 );
        fraction = (denom == 0.0) ? 0.0 : (v0.DotProduct( vm ) / denom);

        midPt = base + v0 * fraction;
}
//
// The side with more samples (the "maxSide") has its points
// set to the side with fewer samples (the "minSide"). Points
// on the maxSide that fall in between are projected onto the line
// defined by the two nearby minSide points.
//
void
TSurfaceTessellator::StitchSide(
                        TSurfaceSample * minSide,
                        TSurfaceSample * maxSide,
                        long minStep,
                        long maxStep,
                        long minPts,
                        GCoordinate ratio,
                        long cont )
```

```
{
        GCoordinate curDist = 0.0;
        long lastInd, curInd = 0;
        register long i, j;

        /*
         * If C0, don't do this.  If C1, just copy point.  If C2, copy normal too.
         */
        if (! cont)             /* C0 discontinous means two separate surfaces... */
                return;
        if (cont >= 2)
                /* C2 continous, so force normals, parameters, etc. to be the same */
                *maxSide = *minSide;
        else
        {
                maxSide->fPoint = minSide->fPoint;          /* Just C1 continous,
don't copy normals */
                maxSide->fUV = minSide->fUV;
        }
        for (i = 1; i <= minPts; i++)
        {
                curDist += ratio;
                lastInd = curInd;
                curInd = (long) (curDist + 0.5);
                for (j = lastInd+1; j < curInd; j++)
                        ProjectToLine( minSide[(i - 1) * minStep].fPoint,
                                        minSide[i * minStep].fPoint,
                                        maxSide[j * maxStep].fPoint );
                if (cont >= 2)
                        maxSide[curInd * maxStep] = minSide[i * minStep];
                else
                {
                        maxSide[curInd * maxStep].fPoint = minSide[i *
minStep].fPoint;
                        maxSide[curInd * maxStep].fUV = minSide[i * minStep].fUV;
                }
        }
}
//
// Stitch together a column (or two) of subgrids.  Since the subpatchs of
// a surface can be sampled at different rates, the samples may not connect
// at the edges.  If this is the case, then the edge of a grid with more
// samples than its neighbor has its points forced onto the edge defined
// by that neighbor.  Much like the crack prevention in subdivision.
//
void
```

```
TSurfaceTessellator::StitchEdges(
                              TSampleGrid * curcol,
                              TSampleGrid * lastcol,
                              long colsize )
{
    TSampleGrid * top, * bottom, * left, * right;
    long c, minPts, minStep, maxStep, cont;
    GCoordinate ratio;
    TSurfaceSample * minSide, * maxSide;

    top = curcol;
    bottom = curcol;
    bottom++;

    /*
     * Stitch tops to bottoms
     */
    for (c = 1; c < colsize; c++)
    {
        if ((top->fIsVisible) && (bottom->fIsVisible) && (top->fMaxU !=
bottom->fMaxU))
        {
            cont = top->fContV;
            if (top->fMaxU < bottom->fMaxU)
            {
                /* Note: All these assignments can go away, and just
be passed
                 * straight to StitchSide.  But for now, this makes
debugging easier.
                 */
                minSide = &(top->fSamples[top->fMaxV]);
                maxSide = bottom->fSamples;
                minStep = top->fMaxV + 1;
                maxStep = bottom->fMaxV + 1;
                minPts = top->fMaxU;
                ratio = (double)bottom->fMaxU / (double)top->fMaxU;
            }
            else
            {
                minSide = bottom->fSamples;
                maxSide = &(top->fSamples[top->fMaxV]);
                minStep = bottom->fMaxV + 1;
                maxStep = top->fMaxV + 1;
                minPts = bottom->fMaxU;
                ratio = (double)top->fMaxU / (double)bottom->fMaxU;
            }
```

```
                    StitchSide( minSide, maxSide, minStep, maxStep, minPts,
ratio, cont );
                }
                top++;
                bottom++;
        }

        if (! lastcol)
                return;

        left = lastcol;
        right = curcol;
        for (c = 0; c < colsize; c++)
        {
                if ((left->fIsVisible) && (right->fIsVisible) && (left->fMaxV != right-
>fMaxV))
                {
                        minStep = 1;
                        maxStep = 1;
                        cont = left->fContU;
                        if (left->fMaxV < right->fMaxV)
                        {
                                minSide = &(left->fSamples[left->fMaxU * (left->fMaxV
+ 1)]);
                                maxSide = right->fSamples;
                                minPts = left->fMaxV;
                                ratio = (double)right->fMaxV / (double)left->fMaxV;
                        }
                        else
                        {
                                minSide = right->fSamples;
                                maxSide = &(left->fSamples[left->fMaxU * (left-
>fMaxV + 1)]);
                                minPts = right->fMaxV;
                                ratio = (double)left->fMaxV / (double)right->fMaxV;
                        }
                        StitchSide( minSide, maxSide, minStep, maxStep, minPts,
ratio, cont );
                }
                left++;
                right++;
        }
}
// Interpolate samples along the edge
static void
SampleInterp(
        const TSurfaceSample& p0,
        const TSurfaceSample& p1,
        GCoordinate u,
        TSurfaceSample& result )
{
        result.fNormal = (p1.fNormal - p0.fNormal) * u + p0.fNormal;
        result.fTangentU = (p1.fTangentU - p0.fTangentU) * u + p0.fTangentU;
        result.fTangentV = (p1.fTangentV - p0.fTangentV) * u + p0.fTangentV;
}
//
// If two adjacent surfaces that meet have exactly the same control points
// and knot vectors along the edge, they should Tessellate to the same point,
// even if the over-all surfaces require different Tessellations along those
// subpatches. To ensure this, the granularity along the edge is measured
// separately. If it's different, then the edge is discretized and triangulated
// to the internal Tessellation.
//
void TSurfaceTessellator::AddEdge(
        TSampleGrid& grid,
        /* TGridEdge::*/eEdgeInd edge,
        long gran,
        long rowOrCol,
        Boolean rowflag )
{
        GCoordinate Inc, val;
        long maxorder = max( fOrderU, fOrderV );
        TGCoordinateStash bvals( maxorder );
        register long i, j, sampInd;
        TSurfaceSample * edgePts;
        GCoordinate gridStep, gridPos;
        TGRPoint3D p;

        edgePts = new TSurfaceSample[ gran + 1 ];

        if (rowflag)
        {
                Inc = (u1 - u0) / ((double) gran);
                gridStep = ((double) grid.fMaxU) / ((double) gran);
                gridPos = 0.0;

                for (i = 0; i <= gran; i++)
                {
                        val = u0 + i * Inc;
```

```
                    ComputeSplineValues( val, curBrkU, fOrderU, fKnotsU, bvals
);
                    BasisMul( curBrkU, fOrderU, bvals, &fPoints[rowOrCol *
fNumPointsU], p );
                    edgePts[i].fPoint = p.DivW();
                    // We already have points on the curve from computing the
subgrid
                    // points.  These already have normals, tangents, etc.  Rather
than
                    // compute a whole new set of normals, we interpolate the
ones already
                    // available on the subgrid.

                    sampInd = (rowOrCol) ? grid.fMaxV : 0L;
                    if ((long)gridPos == grid.fMaxU)
                    {
                          edgePts[i].fNormal = grid.fSamples[sampInd +
grid.fMaxU * (grid.fMaxV + 1L)].fNormal;
                          edgePts[i].fTangentU = grid.fSamples[sampInd +
grid.fMaxU * (grid.fMaxV + 1L)].fTangentU;
                          edgePts[i].fTangentV = grid.fSamples[sampInd +
grid.fMaxU * (grid.fMaxV + 1L)].fTangentV;
                    }
                    else
                          SampleInterp( grid.fSamples[sampInd + ((long)
gridPos) * (grid.fMaxV + 1)],
                                          grid.fSamples[sampInd + ((long) gridPos
+ 1) * (grid.fMaxV + 1)],
                                          gridPos - (long)gridPos,
                                          edgePts[i] );

                    edgePts[i].AdjustNormal();
                    edgePts[i].fUV.fX = val;
                    edgePts[i].fUV.fY = rowOrCol ? v1 : v0;

                    gridPos += gridStep;
              }
       }
       else
       {
              Inc = (v1 - v0) / ((double) gran);
              gridStep = ((double) grid.fMaxV) / ((double) gran);
              gridPos = 0.0;

              for (i = 0; i <= gran; i++)
              {
                    val = v0 + i * Inc;

                    ComputeSplineValues( val, curBrkV, fOrderV, fKnotsV, bvals
);

                    // Fetch the column of control points
                    for (j = 0; j < fNumPointsV; j++)
                        temp[j] = fPoints[j * fNumPointsU + rowOrCol];

                    BasisMul( curBrkV, fOrderV, bvals, temp, p );
                    edgePts[i].fPoint = p.DivW();
                    /* Grab normal from existing samples rather than recompute
it */

                    sampInd = (rowOrCol) ? grid.fMaxU * (grid.fMaxV + 1) : 0;
                    if ((long)gridPos == grid.fMaxV)
                    {
                        edgePts[i].fNormal = grid.fSamples[sampInd +
grid.fMaxV].fNormal;
                        edgePts[i].fTangentU = grid.fSamples[sampInd +
grid.fMaxV].fTangentU;
                        edgePts[i].fTangentV = grid.fSamples[sampInd +
grid.fMaxV].fTangentV;
                    }
                    else
                        SampleInterp( grid.fSamples[sampInd + ((long)
gridPos)],
                                grid.fSamples[sampInd + ((long) gridPos
+ 1)],
                                gridPos - (long)gridPos,
                                edgePts[i] );
                    edgePts[i].AdjustNormal();
                    edgePts[i].fUV.fX = rowOrCol ? u1 : u0;
                    edgePts[i].fUV.fY = val;

                    gridPos += gridStep;
                }
        }
        grid.fEdges[edge].fSamples = edgePts;
        grid.fEdges[edge].fEdgeMax = gran;
}
/////////////////////////// Normal Computations
////////////////////
//
// If a normal has collapsed to zero (dist == 0.0) then try
// and fix it by looking at its neighbors. If all the neighbors
// are sick, then re-compute them from the plane they form.
// If that fails too, then we give up...
//
void
TSurfaceSample::FixNormals(
        TSurfaceSample& s0,
        TSurfaceSample& s1,
        TSurfaceSample& s2 )
{
        Boolean goodnorm;
        long i, j, ok;
        double dist;
        TSurfaceSample * V[3];
        TGPoint3D norm;
        V[0] = &s0; V[1] = &s1; V[2] = &s2;

        /* Find a reasonable normal */
        for (ok = 0, goodnorm = FALSE; (ok < 3L) && !(goodnorm = (V[ok]->fDist
> 0.0)); ok++);

        if (! goodnorm)             /* All provided normals are zilch, try and invent
one */
        {
                norm.fX = 0.0; norm.fY = 0.0; norm.fZ = 0.0;

                for (i = 0; i < 3L; i++)
                {
                        j = (i + 1L) % 3L;
                        norm.fX += (V[i]->fPoint.fY - V[j]->fPoint.fY) * (V[i]->fPoint.fZ
+ V[j]->fPoint.fZ);
                        norm.fY += (V[i]->fPoint.fZ - V[j]->fPoint.fZ) * (V[i]->fPoint.fX
+ V[j]->fPoint.fX);
                        norm.fZ += (V[i]->fPoint.fX - V[j]->fPoint.fX) * (V[i]->fPoint.fY
+ V[j]->fPoint.fY);
                }
                dist = norm.Normalize();
                if (dist == 0.0)
                        return;                 /* This sucker's hopeless... */
                for (i = 0; i < 3; i++)
                {
                        V[i]->fNormal = norm;
                        V[i]->fDist = dist;
                }
        }
        else                    /* Replace a sick normal with a healthy one nearby */
        {
            for (i = 0; i < 3; i++)
                if ((i != ok) && (V[i]->fDist == 0.0))
                    V[i]->fNormal = V[ok]->fNormal;
        }
        return;
}
//
// Normalize the normal in a sample.  If it's degenerate,
// flag it as such by setting the dist to 0.0
//
void
TSurfaceSample::AdjustNormal()
{
        /* If it's not degenerate, to the normalization now */
        fDist = fNormal.Normalize();
        if (fDist < 0.005)              /* VALUE NEEDS FIGURING OUT... */
            fDist = 0.0;
}

//
// Given the point's w and the tangents du and dv, compute
// the normal, taking care of the w's (uses division rule
// of derivatives).
//
void
TSurfaceTessellator::RatNormal(
                                const TGRPoint3D& r,
                                const TGRPoint3D& du,
                                const TGRPoint3D& dv,
                                TSurfaceSample * samp )
{
        GCoordinate wsqrdiv;

        /*
         * If coords are rational then the division rule must be carried
         * out.  Possible optimization: do test at a higher level
         */
        if ((r.fW == 1.0) && (du.fW == 0.0) && (dv.fW == 0.0))
        {
            samp->fTangentU = du.DropW();
            samp->fTangentV = dv.DropW();
        }
        else
        {
            wsqrdiv = 1.0 / (r.fW * r.fW);
            samp->fTangentU.fX = (r.fW * du.fX - du.fW * r.fX) * wsqrdiv;
            samp->fTangentU.fY = (r.fW * du.fY - du.fW * r.fY) * wsqrdiv;
            samp->fTangentU.fZ = (r.fW * du.fZ - du.fW * r.fZ) * wsqrdiv;

            samp->fTangentV.fX = (r.fW * dv.fX - dv.fW * r.fX) * wsqrdiv;
            samp->fTangentV.fY = (r.fW * dv.fY - dv.fW * r.fY) * wsqrdiv;
            samp->fTangentV.fZ = (r.fW * dv.fZ - dv.fW * r.fZ) * wsqrdiv;
        }
        samp->fNormal = samp->fTangentV.CrossProduct( samp->fTangentU );
        samp->AdjustNormal();
}
//
// Forward diff can't easily find the normals, so compute them
// from the mesh of points generated.  The normals around the
// edge of the mesh are found via evaluation.
//
void
TSurfaceTessellator::ComputeInteriorNormals( TSampleGrid& sg )
{
        register long i, j;
        TSurfaceSample * c00, * c10, * c11, * c01;
        TGPoint3D v0, v1;

        /* Compute the normals of the plane formed by each
         * square in the grid.
         */
                                                          /*   c00 --- c01
        */
        c01 = &(sg.fSamples[sg.fMaxV+1]);  /*        | / |      ^     */
        c10 = &(sg.fSamples[1]);                     /*      | / |     v1    */
        c11 = &(c01[1]);                             /*    c10 --- c11  */
                                                          /*             <v0
            */
        for (i = 1; i <= sg.fMaxU; i++)
        {
            for (j = 1; j <= sg.fMaxV; j++)
            {
                v0 = c10->fPoint - c11->fPoint;
                v1 = c01->fPoint - c11->fPoint;
                c11->fNormal = v1.CrossProduct( v0 );
                c01++; c10++; c11++;
            }
            c01++; c10++; c11++;
        }

        /*
        * Find the normal for each sample by averaging the normals
        * of the four planes surrounding it.
        */

        c00 = &(sg.fSamples[sg.fMaxV+2]);
        c01 = &(c00[sg.fMaxV+1]);
        c10 = &(c00[1]);
        c11 = &(c01[1]);

        for (i = 1; i < sg.fMaxU; i++)
        {
                for (j = 1; j < sg.fMaxV; j++)
                {
                        c00->fNormal = (c00->fNormal + c01->fNormal + c10-
>fNormal + c11->fNormal)
                                                * 0.25;
                        c00->AdjustNormal();
                        c00++; c01++; c10++; c11++;
                }
                c00++; c01++; c10++; c11++;
                c00++; c01++; c10++; c11++;
        }
}
//
// The interior normals for forward differencing are approximated via
// plan normals, but we can't do this for normals at the edges (not enough
// facets to average). So they are computed directly via evaluation.
//
void
TSurfaceTessellator::ComputeEdgeNormals()
{
        TGRPoint3D rp, rdu, rdv;
        TSurfaceSample * smplPtr;
        long stepU, stepV;
        for (stepU = 0; stepU <= granU; stepU++)
        {
                ComputeVCurve( stepU, 1 );

                smplPtr = &(curGrids[baseV].fSamples[stepU * (granV + 1)]);

                /* Do whole column at the left and right edge */
                if ((stepU == 0) || (stepU == granU))
                {
                        for (stepV = 0; stepV <= granV; stepV++)
                        {
                                BasisMul( curBrkV, fOrderV, basisCacheV[stepV],
temp, rp );
                                BasisMul( curBrkV, fOrderV, basisCacheV[stepV],
dtemp, rdu );
                                BasisMul( curBrkV, fOrderV, basisCacheDV[stepV],
temp, rdv );
                                RatNormal( rp, rdu, rdv, smplPtr );
                                smplPtr++;
                        }
                }
                else    /* Compute top and bottom in between the edges */
                {
                        BasisMul( curBrkV, fOrderV, basisCacheV[0], temp, rp );
                        BasisMul( curBrkV, fOrderV, basisCacheV[0], dtemp, rdu );
                        BasisMul( curBrkV, fOrderV, basisCacheDV[0], temp, rdv );
                        RatNormal( rp, rdu, rdv, smplPtr );
                        smplPtr = &(smplPtr[granV]);
                        BasisMul( curBrkV, fOrderV, basisCacheV[granV], temp, rp );
                        BasisMul( curBrkV, fOrderV, basisCacheV[granV], dtemp,
rdu );
                        BasisMul( curBrkV, fOrderV, basisCacheDV[granV], temp,
rdv );
                        RatNormal( rp, rdu, rdv, smplPtr );
                }
        }
}
//////////////// Forward Differencing ///////////////////
//
// Take an orderV x orderU set of samples in deltas and
// turn them into a forward difference matrix for the surface.
//
void
TSurfaceTessellator::ComputeDeltas()
{
        long i, j, stepU, stepV;

        /* First, find the deltas in V for each Ku sample of them */

        for (stepU = 0; stepU < fOrderU; stepU++)
        {
                for (stepV = 0; stepV < fOrderV; stepV++)
                        temp[stepV] = deltas[stepU][stepV];

                for (i = 1; i < fOrderV; i++)
                {
                        for (j = 0; j < fOrderV - i; j++)
                        {
                                temp[j].fX = temp[j + 1].fX - temp[j].fX;
                                temp[j].fY = temp[j + 1].fY - temp[j].fY;
                                temp[j].fZ = temp[j + 1].fZ - temp[j].fZ;
                                temp[j].fW = temp[j + 1].fW - temp[j].fW;
                        }
                        deltas[stepU][i] = temp[0];
                }
        }
        /*
         * Now make the "forward differences of forward differences"
         * used to step down the patch. (See Cook's "Patch Work" paper).
         */
        for (stepV = 0; stepV < fOrderV; stepV++)
        {
                for (stepU = 0; stepU < fOrderU; stepU++)
                        temp[stepU] = deltas[stepU][stepV];

                for (i = 1; i < fOrderU; i++)
                {
                        for (j = 0; j < fOrderU - i; j++)
                        {
                                temp[j].fX = temp[j + 1].fX - temp[j].fX;
                                temp[j].fY = temp[j + 1].fY - temp[j].fY;
                                temp[j].fZ = temp[j + 1].fZ - temp[j].fZ;
                                temp[j].fW = temp[j + 1].fW - temp[j].fW;
                        }
                        deltas[i][stepV] = temp[0];
                }
        }
}
//
// Forward difference one column of the patch
//
void
TSurfaceTessellator::ForwardStepCol(
                TSurfaceSample * sptr,
                GCoordinate u )
{
        long i, j, stepV;
        GCoordinate v = v0;
        /* Now forward difference the surface */

        for (i = 0; i < fOrderV; i++)
                temp[i] = deltas[0][i];
        sptr->fPoint = temp->DivW();
        sptr->fUV.fX = u;
        sptr->fUV.fY = v;
        sptr++;

        for (stepV = 0; stepV < granV; stepV++)
        {
                /* Step down the patch */

                for (j = 0; j < fOrderV - 1; j++)
                {
                        temp[j].fX += temp[j + 1].fX;
                        temp[j].fY += temp[j + 1].fY;
                        temp[j].fZ += temp[j + 1].fZ;
                        temp[j].fW += temp[j + 1].fW;
                }
                v += vInc;

                sptr->fPoint = temp->DivW();
                sptr->fUV.fX = u;
                sptr->fUV.fY = v;
                sptr++;
        }

        /* Step across patch (note we only use granU-1 steps) */

        for (stepV = 0; stepV < fOrderV; stepV++)
                for (j = 0; j < fOrderU - 1; j++)
                {
                        deltas[j][stepV].fX += deltas[j + 1][stepV].fX;
                        deltas[j][stepV].fY += deltas[j + 1][stepV].fY;
                        deltas[j][stepV].fZ += deltas[j + 1][stepV].fZ;
                        deltas[j][stepV].fW += deltas[j + 1][stepV].fW;
                }
}
//
// Triangulate between an edge and the subgrid.
//
void
TSurfaceTessellator::DrawSealedEdge( const TSampleGrid& grid,
                                                /* TGridEdge::*/ eEdgeInd
edge,
                                                long gridBase )
{
        register long i, curInd;
        register GCoordinate curDist, ratio, nudge;
```

```
    TSurfaceSample * sp[3];
    long ind2, ind3;
    long maxEdge = grid.fEdges[edge].fEdgeMax;
    TSurfaceSample * edgeSamp = grid.fEdges[edge].fSamples;
    TSurfaceSample * gridSamp = &(grid.fSamples[gridBase]);

    long gridSkip = (edge < TGridEdge::kEdge0v) ? 1 : grid.fMaxV + 1;
    long maxGrid = (edge < TGridEdge::kEdge0v) ? grid.fMaxV : grid.fMaxU;

    if ( maxEdge > maxGrid )
        nudge = 1.0 / (maxEdge + 1L);
    else
        nudge = 1.0 / (maxGrid + 1L);

    if ((edge == TGridEdge::kEdge0u) | | (edge == TGridEdge::kEdge1v))
    {
        ind2 = 1;           // Goofy indexing because triangle orientation
switches
        ind3 = 0;           // if we're on the left or bottom.
    }
    else
    {
        ind2 = 0;
        ind3 = 1;
    }

    ratio = ((GCoordinate) maxEdge) / ((GCoordinate) maxGrid);
    curDist = ratio + nudge;
    curInd = (long) curDist;

    for (i = 1; i <= maxGrid; i++)
    {
        sp[0] = &(gridSamp[(i - 1) * gridSkip]);
        sp[1 + (1-ind2)] = &(edgeSamp[curInd]);
        sp[1 + (1-ind3)] = &(gridSamp[i * gridSkip]);
        TSurfaceSample::FixNormals( *(sp[0]), *(sp[1]), *(sp[2]) );
        fTessInfo->RenderShard( sp, 3, grid.fClipInCode, grid.fFrontFace );
        curDist += ratio;
        curInd = (long) curDist;
    }
    ratio = ((double) maxGrid) / ((double) maxEdge);
    curDist = ratio - nudge;
    curInd = (long) curDist;

    for (i = 1; i <= maxEdge; i++)
    {
            sp[0] = &(edgeSamp[i - 1]);
            sp[1 + ind2] = &(gridSamp[curInd * gridSkip]);
            sp[1 + ind3] = &(edgeSamp[i]);
            fTessInfo->RenderShard( sp, 3, grid.fClipInCode, grid.fFrontFace );
            curDist += ratio;
            curInd = (long) curDist;
        }

        // Clear edge now so it's not re-used if it's not needed
        delete grid.fEdges[edge].fSamples;
        grid.fEdges[edge].fSamples = NIL;
        grid.fEdges[edge].fEdgeMax = 0;
}
//
// Draw the set of samples
//
void
TSurfaceTessellator::DrawSubGrid( TSampleGrid& samples )
{
        register long       i, j;
        TSurfaceSample   * c00, * c10, * c11, * c01;
        TSurfaceSample * sp[3];
        /*TGridEdge::*/eEdgeInd edge;
        long startU, endU, startV, endV, nextRowBump, extraSamples;

        startU = 0;
        startV = 0;
        endU = samples.fMaxU;
        endV = samples.fMaxV;
        nextRowBump = 1;
                                                                        //
Orientation of the points is thus:
        c00 = samples.fSamples;                                  //   c00 ---
c01
        c01 = &(samples.fSamples[samples.fMaxV+1]);  //        | / |
        c10 = &(samples.fSamples[1]);                          //       | /  |

        c11 = &(c01[1]);                                          //      c10 --- c11

        //
        // If edges must be triangulated separately, adjust the loop
        // parameters.
        //
        if (samples.fEdges[TGridEdge::kEdge0u].IsSet())
        {
            startU++;
            c00 += samples.fMaxV+1; c01 += samples.fMaxV+1;
            c10 += samples.fMaxV+1; c11 += samples.fMaxV+1;
        }
        if (samples.fEdges[TGridEdge::kEdge1u].IsSet())
        {
            endU--;
        }
        if (samples.fEdges[TGridEdge::kEdge0v].IsSet())
        {
            startV++;
            nextRowBump++;
            c00++; c01++;
            c10++; c11++;
        }
        if (samples.fEdges[TGridEdge::kEdge1v].IsSet())
        {
            endV--;
            nextRowBump++;
        }

        // Do device-specific setup (convert to screen coords, etc.)
        extraSamples = 0;
        for (edge = TGridEdge::kEdge0u; edge < TGridEdge::kNumEdges; edge++)
            if (samples.fEdges[edge].IsSet())
                extraSamples += samples.fEdges[edge].fEdgeMax + 1;

        fTessInfo->ReallocDevSamples( (samples.fMaxU + 1) * (samples.fMaxV +
1) + extraSamples );
        fTessInfo->PrepareSamples( samples.fSamples, 0,
                                                    (samples.fMaxU+1) *
(samples.fMaxV+1),
                                                    samples.fClipInCode );

        // Triangulate the interior
        for (i = startU; i < endU; i++)
        {
            for (j = startV; j < endV; j++)
            {
                TSurfaceSample::FixNormals( *c00, *c10, *c01 );
                sp[0] = c00; sp[1] = c10; sp[2] = c01;
                fTessInfo->RenderShard( sp, 3, samples.fClipInCode,
samples.fFrontFace );
                TSurfaceSample::FixNormals( *c10, *c11, *c01 );
                sp[0] = c10; sp[1] = c11; // sp[2] = c01;
                fTessInfo->RenderShard( sp, 3, samples.fClipInCode,
samples.fFrontFace );
```

```
                c00++; c01++; c10++; c11++;
            }

            for (j = 0; j < nextRowBump; j++)
            {
                c00++; c01++;
                c10++; c11++;       // move to next column
            }
        }
        //
        // Seal edges by triangulating the "edge" strips with the
        // row of points in the grid just in from the edge.
        //
        long devSampleBase = (samples.fMaxU+1) * (samples.fMaxV+1);
        long gridBase;
        for (edge = TGridEdge::kEdge0u; edge < TGridEdge::kNumEdges; edge++)
            if (samples.fEdges[edge].IsSet())
            {
                fTessInfo->PrepareSamples(
                                        samples.fEdges[edge].fSamples,
devSampleBase,
                                        samples.fEdges[edge].fEdgeMax + 1,
samples.fClipInCode );
                devSampleBase += samples.fEdges[edge].fEdgeMax + 1;

                switch (edge)
                {
                case TGridEdge::kEdge0u:
                    gridBase = samples.fMaxV+1;
                    break;
                case TGridEdge::kEdge1u:
                    gridBase = (samples.fMaxU-1) * (samples.fMaxV+1);
                    break;
                case TGridEdge::kEdge0v:
                    gridBase = 1;
                    break;
                case TGridEdge::kEdge1v:
                    gridBase = samples.fMaxV-1;
                    break;
                case TGridEdge::kNumEdges:
                    OpusBug( "Huh? kNumEdges is just here to shut up
Cfront" );
                }
                DrawSealedEdge( samples, edge, gridBase );
            }
}
```

```
///////////////////////// Tessellation ///////////////////
//
// Main entry point
//
void
TSurfaceTessellator::Tessellate(Boolean alwaysDirectEvaluate)
{
	register long i, stepU, stepV;
	long junk;
	long disconU, realDisconU = 2, disconV;
	GCoordinate u, v;
	TGRPoint3D rp, rdu, rdv;
	TSurfaceSample * smplPtr;
	TSampleGrid * gridPtr;
	long edgeGran;
	//
	// Put the points in the right space (I'd prefer to do this in the
	// constructor, but C++ can't call virtual subclass functions from
	// constructors. Grrrrrr...)
	//
	fTessInfo->TransformPoints( fPoints, fNumPointsU * fNumPointsV );
	/*
	 * The curve is processed in vertical bands, one for each U interval.
	 * It's done in this order because it's slightly more natural to do the
	 * evaluations this way (simply pass rows to CalcLinePt). Each subpatch
	 * is done individually, to optimize step sizes.
	 */
	for (baseU = 0; baseU < numBrksU; baseU++)
	{
		curBrkU = brkPointsU[baseU];
		u0 = fKnotsU[curBrkU];
		u1 = fKnotsU[curBrkU + 1];

		/* Check discontinuity */
		if (baseU < numBrksU - 1)
		{
			disconU = 1;
			while ((disconU < fOrderU) && (fKnotsU[curBrkU + 1 +
disconU] == u1))
				disconU++;
			disconU = fOrderU - disconU;
		}
		for (baseV = 0; baseV < numBrksV; baseV ++)
		{
			curBrkV = brkPointsV[baseV];
			gridPtr = &(curGrids[baseV]);
```

```
                        /* Don't bother with subpatches that are completely off
screen */

                        gridPtr->fIsVisible = SubPatchVisible( curBrkV-(fOrderV-1),

        curBrkU-(fOrderU-1),

        gridPtr->fClipInCode,

        gridPtr->fFrontFace );
                        if (! gridPtr->fIsVisible)
                                continue;

                        v0 = fKnotsV[curBrkV];
                        v1 = fKnotsV[curBrkV + 1];
                        /* Check discontinuity */
                        if (baseV < numBrksV - 1)
                        {
                                disconV = 1;
                                while ((disconV < fOrderV) && (fKnotsV[curBrkV + 1 +
disconV] == v1))
                                        disconV++;
                                disconV = fOrderV - disconV;
                                if (disconV == 1)
                                        disconV = IsReallyC1( curBrkU, curBrkV, FALSE
);
                        }
                        if (baseU < numBrksU - 1)
                        {
                                if (disconU == 1)
                                        realDisconU = IsReallyC1( curBrkU, curBrkV,
TRUE );
                                else
                                        realDisconU = disconU;
                        }
                        granU = GetGran( curBrkV, curBrkU, TRUE );
                        granV = GetGran( curBrkV, curBrkU, FALSE );

                        /* Allocate storage for the grid of generated points */

                        AllocSampleGrid( *gridPtr, granV, granU );
                        gridPtr->fContU = realDisconU;      /* realDisconU */
                        gridPtr->fContV = disconV;
                        smplPtr = gridPtr->fSamples;

                        /* Pre-compute and stash the basis functions in V */
```

```
			if (granV > maxGranV)
			{
				junk = maxGranV;
				AllocBasisCache( basisCacheV, fOrderV, granV, junk );
	/* HACK! */
				AllocBasisCache( basisCacheDV, fOrderV, granV,
maxGranV );
			}

			vInc = (v1 - v0) / ((double) granV);	// Parameterization
increments
			uInc = (u1 - u0) / ((double) granU);

			for (stepV = 0; stepV <= granV; stepV++)
			{
				v = v0 + stepV * vInc;
				ComputeSplineValues( v, curBrkV, fOrderV, fKnotsV,
							basisCacheV[stepV],
basisCacheDV[stepV], NULL );
			}

			/*
			 * Some basic timing tests show it takes roughly(!) 2*order
samples before
			 * the faster speed of forward differencing pays off the cost of
setting
			 * up the forward difference table (the "deltas"). Thus, if fewer
samples
			 * are required, the surface is evaluated directly.
			 */

			u = u0;
			if (alwaysDirectEvaluate || (granU <= fOrderU * 2) ||
(granV <= fOrderV * 2))
			{
	/***** D I R E C T   E V A L U A T I O N *****/

				for (stepU = 0; stepU <= granU; stepU++)
				{
					ComputeVCurve( stepU, 1 );
					v = v0;

					for (stepV = 0; stepV <= granV; stepV++)
					{
						BasisMul( curBrkV, fOrderV,
basisCacheV[stepV], temp, rp );
```

```
                                        BasisMul( curBrkV, fOrderV,
basisCacheV[stepV], dtemp, rdu );
                                        BasisMul( curBrkV, fOrderV,
basisCacheDV[stepV], temp, rdv );

                                        smplPtr->fPoint = rp.DivW();
                                        smplPtr->fUV.fX = u;
                                        smplPtr->fUV.fY = v;
                                        v += vInc;
                                        RatNormal( rp, rdu, rdv, smplPtr );
                                        smplPtr++;
                                }
                                u += uInc;
                        }
                }
else    /***** F O R W A R D   D I F F E R E N C E *****/
                {
                        /*
                        * Compute the initial points that the deltas are
computed from.
                        * Each subpatch (i.e., one span in U and V) needs Ku *
Kv samples
                        * to work from.
                        */
                        for (stepU = 0; stepU < fOrderU; stepU++)
                        {
                                ComputeVCurve( stepU, 0 );

                                /* Find points on the surface */

                                for (stepV = 0; stepV < fOrderV; stepV++)
                                        BasisMul( curBrkV, fOrderV,
basisCacheV[stepV],
                                                        temp, deltas[stepU][stepV] );
                        }

                        ComputeDeltas();

                        /* Now forward difference the surface */
                        for (stepU = 0; stepU <= granU; stepU++)
                        {
                                ForwardStepCol( smplPtr, u );
                                u += uInc;
                                smplPtr = &(smplPtr[granV + 1]);
                        }
```

```
                    /*
                     * Find the normals - interior via averaging facet
normals,
                     * edges via evaluation.  Interior must be done first
because
                     * it uses the edges as temp storage.
                     */
                    ComputeInteriorNormals( *gridPtr );
                    ComputeEdgeNormals();
                }

                //
                // If a subpatch is at the boundary of the surface, then seal
the edge
                // of the surface.
                //
                if ((baseU == 0)
                 && (granV != (edgeGran = GetEdgeGran( curBrkV, 0, FALSE
))))
                    AddEdge( *gridPtr, TGridEdge::kEdge0u, edgeGran, 0,
FALSE );
                if ((baseU == numBrksU-1)
                 && (granV != (edgeGran = GetEdgeGran( curBrkV,
fNumPointsU-1, FALSE ))))
                    AddEdge( *gridPtr, TGridEdge::kEdge1u, edgeGran,
fNumPointsU-1, FALSE );

                if ((baseV == 0)
                 && (granU != (edgeGran = GetEdgeGran( curBrkU, 0, TRUE
))))
                    AddEdge( *gridPtr, TGridEdge::kEdge0v, edgeGran, 0,
TRUE );

                if ((baseV == numBrksV-1)
                 && (granU != (edgeGran = GetEdgeGran( curBrkU,
fNumPointsV-1, TRUE ))))
                    AddEdge( *gridPtr, TGridEdge::kEdge1v, edgeGran,
fNumPointsV-1, TRUE );
            }

            /* Fill in the cracks between the two columns of subgrids, and
             * output the previous column. */

            StitchEdges( curGrids, (baseU ? lastGrids : NULL), numBrksV );
            if (baseU != 0)
            {
                gridPtr = lastGrids;
                for (i = 0; i < numBrksV; i++, gridPtr++)
                    if (gridPtr->fIsVisible)
                        DrawSubGrid( *gridPtr );
            }
            gridPtr = lastGrids;
            lastGrids = curGrids;
            curGrids = gridPtr;}
        gridPtr = lastGrids;        /* Flush the last column */
        for (i = 0; i < numBrksV; i++, gridPtr++)
            if (gridPtr->fIsVisible)
                DrawSubGrid( *gridPtr );
```

While the invention has been described in terms of a preferred embodiment in a specific system environment, those skilled in the art recognize that the invention can be practiced, with modification, in other and different hardware and software environments within the spirit and scope of the appended claims.

CLAIMS

Having thus described our invention, what we claim as new, and desire to secure by Letters Patent is:

1. A method for tessellating a surface having a plurality of subpatches, comprising the steps of:
(a) selecting a column of subpatches;
(b) selecting a subpatch from the selected column of subpatches;
(c) determining whether the selected subpatch is visible;
(d) computing a grid of discretized points if the selected subpatch is visible; and
(e) removing any cracks between the selected subpatch and a previously selected subpatch.

2. A method as recited in claim 1, including the steps of:
(a) removing any cracks between the selected column of grids and a previously selected column of grids; and
(b) rendering the previously selected column of grids.

3. A method for tessellating a surface having a plurality of subpatches organized within columns, comprising the steps of:
(a) selecting a column and a subpatch within the column;
(b) determining whether the selected subpatch is visible;
(c) computing a grid of discretized points for the selected subpatch by dynamic selection of an algorithm; and
(d) removing any crack between the computed grid and a grid computed immediately before.

4. The method as recited in claim 3, including the step of repeating steps (a) through (e) for each subpatch within the selected column to form a computed column of grids.

5. The method as recited in claim 4, including the steps of:
(a) removing any crack between the computed column of grids and a column of grids computed immediately before; and
(b) rendering the column of grids computed immediately before.

6. A method for tessellating a surface having a plurality of subpatches organized within columns, comprising the steps of:
   (a) selecting a column and a subpatch within the column;
   (b) determining whether the selected subpatch is visible;
   (c) computing a grid of discretized points for the selected subpatch by dynamic selection of an algorithm, the algorithm selection being dependent upon a determination of granularity; and
   (d) removing any crack between the computed grid and a grid computed immediately before.

7. A method as recited in claim 4, including the step of repeating steps (a) through (e) for each subpatch within the selected column so as to form a computed column of grids.

8. A method as recited in claim 4, including the steps of:
   (a) removing any cracks between the computed column of grids and a column of grids computed immediately before; and
   (b) rendering the column of grids computed immediately before.

9. A method for tessellating a surface having a plurality of subpatches organized within columns, comprising the steps of:
   (a) selecting a column and a subpatch within the column;
   (b) determining whether the selected subpatch is visible;
   (c) computing a grid of discretized points for the selected subpatch by selection of either a direct evaluation algorithm or a forward differencing algorithm, the selection being dependent upon a granularity value;
   (d) removing any crack between the computed grid and a grid computed immediately before;
   (e) repeating steps (a) through (e) for each subpatch within the selected column so as to form a computed column of grids;
   (f) removing any crack between the computed column of grids and a column of grids computed immediately before;
   (g) rendering the column of grids computed immediately before; and
   (h) repeating steps (a) through (g) for a column adjacent to the selected column that was not selected.

10. A method for tessellating a surface having a plurality of subpatches organized within columns, comprising the steps of:
    (a) selecting a column and a subpatch within the column;
    (b) determining whether the selected subpatch is visible;
    (c) computing a grid of discretized points for the selected subpatch by dynamic selection of an algorithm;
    (d) removing any crack between the computed grid and a grid computed immediately before;
    (e) repeating steps (a) through (e) for each subpatch within the selected column so as to form a computed column of grids;
    (f) removing any crack between the computed column of grids and a column of grids computed immediately before;
    (g) rendering the column of grids computed immediately before; and
    (h) repeating steps (a) through (g) for a column adjacent to the selected column that was not selected.

11. A method for tessellating a surface having a plurality of subpatches organized within columns, comprising the steps of:
    (a) selecting a column and a subpatch within the column;
    (b) determining whether the selected subpatch is visible;
    (c) computing a grid of discretized points for the selected subpatch by dynamic selection of an algorithm;
    (d) removing any crack between the computed grid and a grid computed immediately before;
    (e) repeating steps (a) through (e) for each subpatch within the selected column so as to form a computed column of grids;
    (f) removing any crack between the computed column of grids and a column of grids computed immediately before;
    (g) rendering the column of grids computed immediately before;
    (h) maintaining the grid of columns for the selected column; and
    (i) repeating steps (a) through (h) for a column adjacent to the selected column that was not selected.

12. A method for tessellating a surface having a plurality of subpatches organized within columns, comprising the steps of:
    (a) selecting a column and a subpatch within the column;
    (b) determining whether the selected subpatch is visible;
    (c) computing a grid of discretized points for the selected subpatch by dynamic selection of an algorithm;

(d) removing any crack between the computed grid and a grid computed immediately before; and
(e) displaying the rendered surface on a display.

13. An apparatus for tessellating a surface having a plurality of subpatches, comprising:
(a) processor means for selecting a column of subpatches;
(b) processor means for selecting a subpatch from the selected column of subpatches;
(c) processor means for determining whether the selected subpatch is visible;
(d) processor means for computing a grid of discretized points if the selected subpatch is visible; and
(e) processor means for removing any cracks between the selected subpatch and a previously selected subpatch.

14. An apparatus as recited in claim 13, including:
(a) processor means for removing any cracks between the selected column of grids and a previously selected column of grids; and
(b) processor means for rendering the previously selected column of grids.

15. An apparatus for tessellating a surface having a plurality of subpatches organized within columns, comprising:
(a) processor means for selecting a column and a subpatch within the column;
(b) processor means for determining whether the selected subpatch is visible;
(c) processor means for computing a grid of discretized points for the selected subpatch by dynamic selection of an algorithm; and
(d) processor means for removing any crack between the computed grid and a grid computed immediately before.

16. The apparatus as recited in claim 15, including the step of repeating steps (a) through (e) for each subpatch within the selected column to form a computed column of grids.

17. The apparatus as recited in claim 16, including:

(a) processor means for removing any crack between the computed column of grids and a column of grids computed immediately before; and

(b) processor means for rendering the column of grids computed immediately before.

18. An apparatus for tessellating a surface having a plurality of subpatches organized within columns, comprising:

(a) processor means for selecting a column and a subpatch within the column;

(b) processor means for determining whether the selected subpatch is visible;

(c) processor means for computing a grid of discretized points for the selected subpatch by dynamic selection of an algorithm, the algorithm selection being dependent upon a determination of granularity; and

(d) processor means for removing any crack between the computed grid and a grid computed immediately before.

19. An apparatus as recited in claim 18, including processor means for repeating steps (a) through (d) for each subpatch within the selected column so as to form a computed column of grids.

20. An apparatus as recited in claim 19, including:

(a) processor means for removing any cracks between the computed column of grids and a column of grids computed immediately before; and

(b) processor means for rendering the column of grids computed immediately before.

21. An apparatus for tessellating a surface having a plurality of subpatches organized within columns, comprising:

(a) processor means for selecting a column and a subpatch within the column;

(b) processor means for determining whether the selected subpatch is visible;

(c) processor means for computing a grid of discretized points for the selected subpatch by selection of either a direct evaluation algorithm or a forward differencing algorithm, the selection being dependent upon a granularity value;

(d) processor means for removing any crack between the computed grid and a grid computed immediately before;

(e) processor means for repeating steps (a) through (e) for each subpatch within the selected column so as to form a computed column of grids;

(f) processor means for removing any crack between the computed column of grids and a column of grids computed immediately before;

(g) processor means for rendering the column of grids computed immediately before; and

(h) processor means for repeating steps (a) through (g) for a column adjacent to the selected column that was not selected.

22. An apparatus for tessellating a surface having a plurality of subpatches organized within columns, comprising:

(a) processor means for selecting a column and a subpatch within the column;

(b) processor means for determining whether the selected subpatch is visible;

(c) processor means for computing a grid of discretized points for the selected subpatch by dynamic selection of an algorithm;

(d) processor means for removing any crack between the computed grid and a grid computed immediately before;

(e) processor means for repeating steps (a) through (e) for each subpatch within the selected column so as to form a computed column of grids;

(f) processor means for removing any crack between the computed column of grids and a column of grids computed immediately before;

(g) processor means for rendering the column of grids computed immediately before; and

(h) processor means for repeating steps (a) through (g) for a column adjacent to the selected column that was not selected.

23. An apparatus for tessellating a surface having a plurality of subpatches organized within columns, comprising:

(a) processor means for selecting a column and a subpatch within the column;

(b) processor means for determining whether the selected subpatch is visible;

(c) processor means for computing a grid of discretized points for the selected subpatch by dynamic selection of an algorithm;

(d) processor means for removing any crack between the computed grid and a grid computed immediately before;

(e) processor means for repeating steps (a) through (e) for each subpatch within the selected column so as to form a computed column of grids;

(f) processor means for removing any crack between the computed column of grids and a column of grids computed immediately before;

(g) processor means for rendering the column of grids computed immediately before;

(h) storage means for maintaining the grid of columns for the selected column; and

(i) processor means for repeating steps (a) through (h) for a column adjacent to the selected column that was not selected.

24. An apparatus for tessellating a surface having a plurality of subpatches organized within columns, comprising:

(a) processor means for selecting a column and a subpatch within the column;

(b) processor means for determining whether the selected subpatch is visible;

(c) processor means for computing a grid of discretized points for the selected subpatch by dynamic selection of an algorithm;

(d) processor means for removing any crack between the computed grid and a grid computed immediately before; and

(e) processor means for displaying the rendered surface on a display.

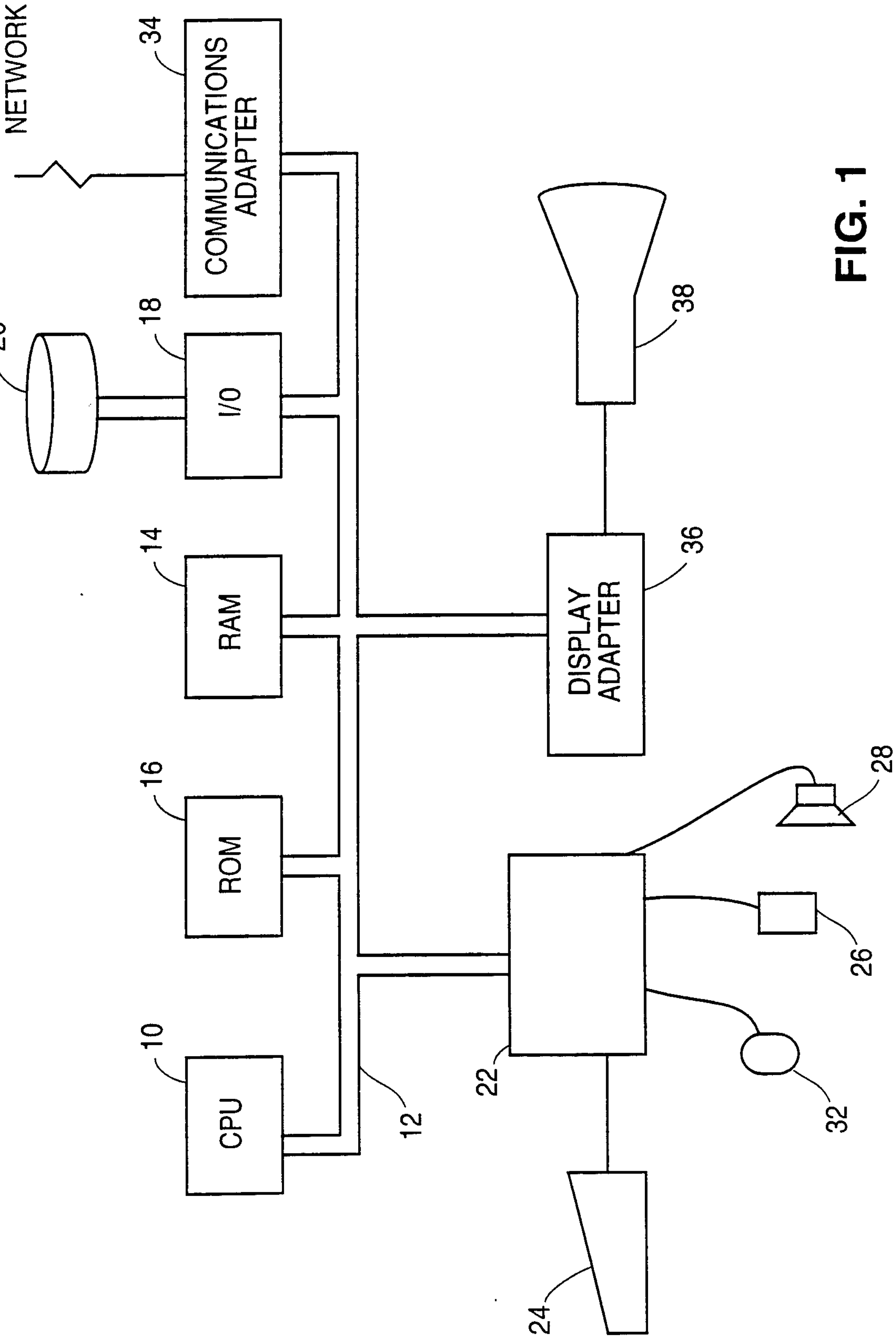

NETWORK
COMMUNICATIONS ADAPTER
34
20
18
I/O
14
RAM
16
ROM
10
CPU
12
22
24
32
26
28
DISPLAY ADAPTER
36
38

FIG. 1

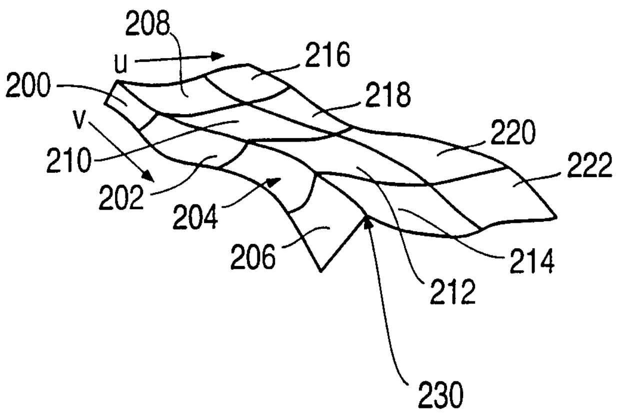

208
u
216
200
218
v
220
210
222
202
204
206
214
212
230

FIG. 2A

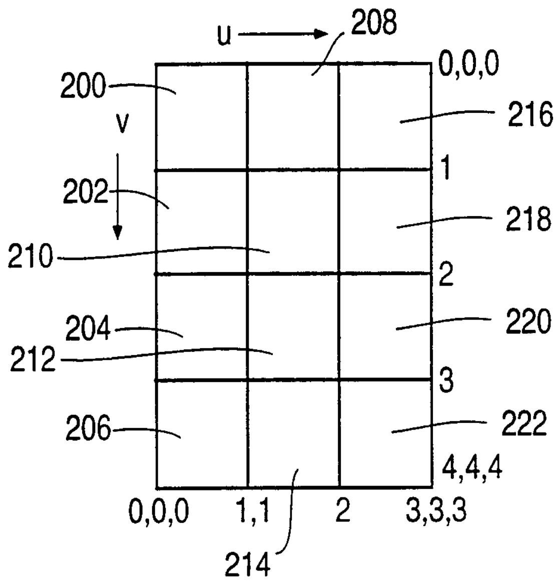

u
208
0,0,0
200
216
v
1
202
218
210
2
204
220
212
3
206
222
4,4,4
0,0,0
1,1
2
3,3,3
214

FIG. 2B

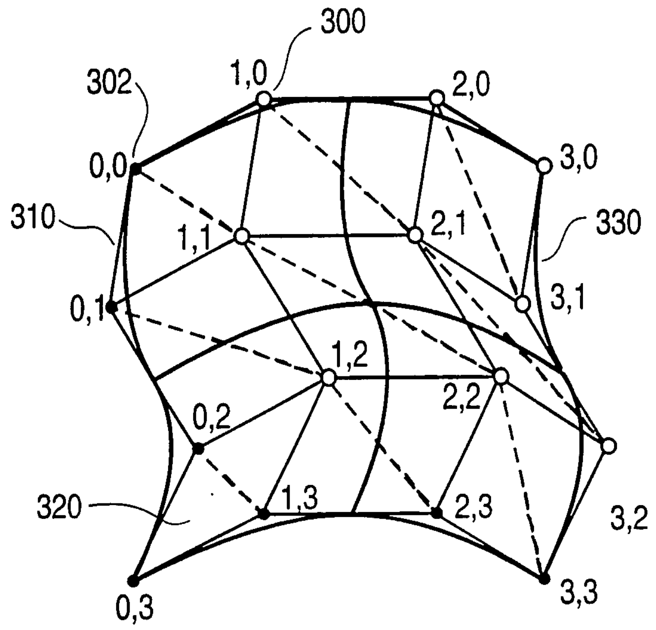

300
302
1,0
2,0
0,0
3,0
310
1,1
2,1
330
0,1
3,1
1,2
2,2
0,2
320
1,3
2,3
3,2
0,3
3,3

FIG. 3A

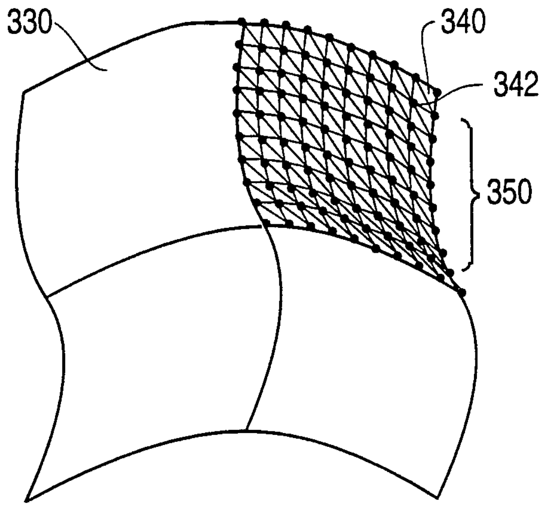

330
340
342
350

FIG. 3B

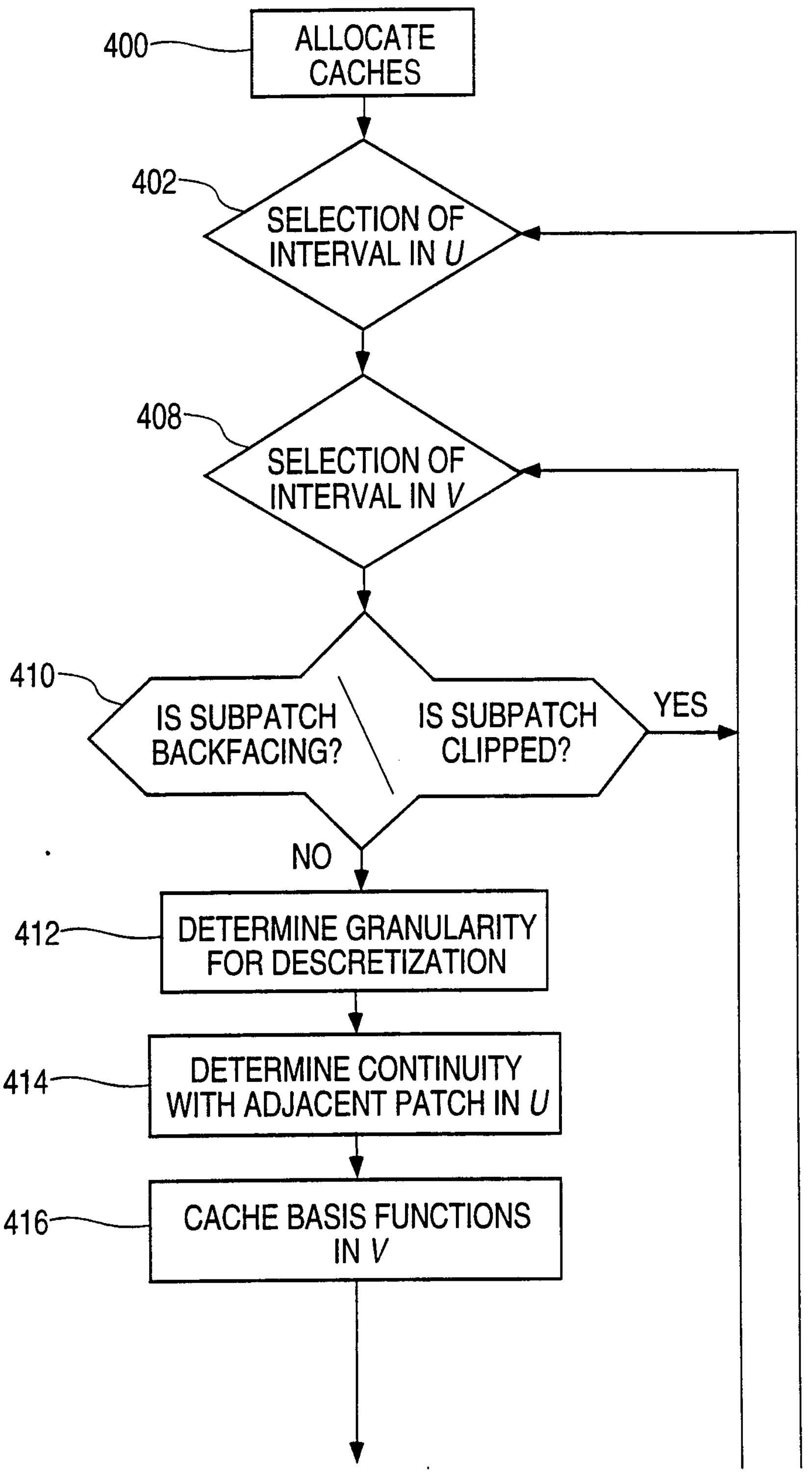

400
ALLOCATE CACHES
402
SELECTION OF INTERVAL IN U
408
SELECTION OF INTERVAL IN V
410
IS SUBPATCH BACKFACING?
IS SUBPATCH CLIPPED?
YES
NO
412
DETERMINE GRANULARITY FOR DESCRETIZATION
414
DETERMINE CONTINUITY WITH ADJACENT PATCH IN U
416
CACHE BASIS FUNCTIONS IN V

FIG. 4A

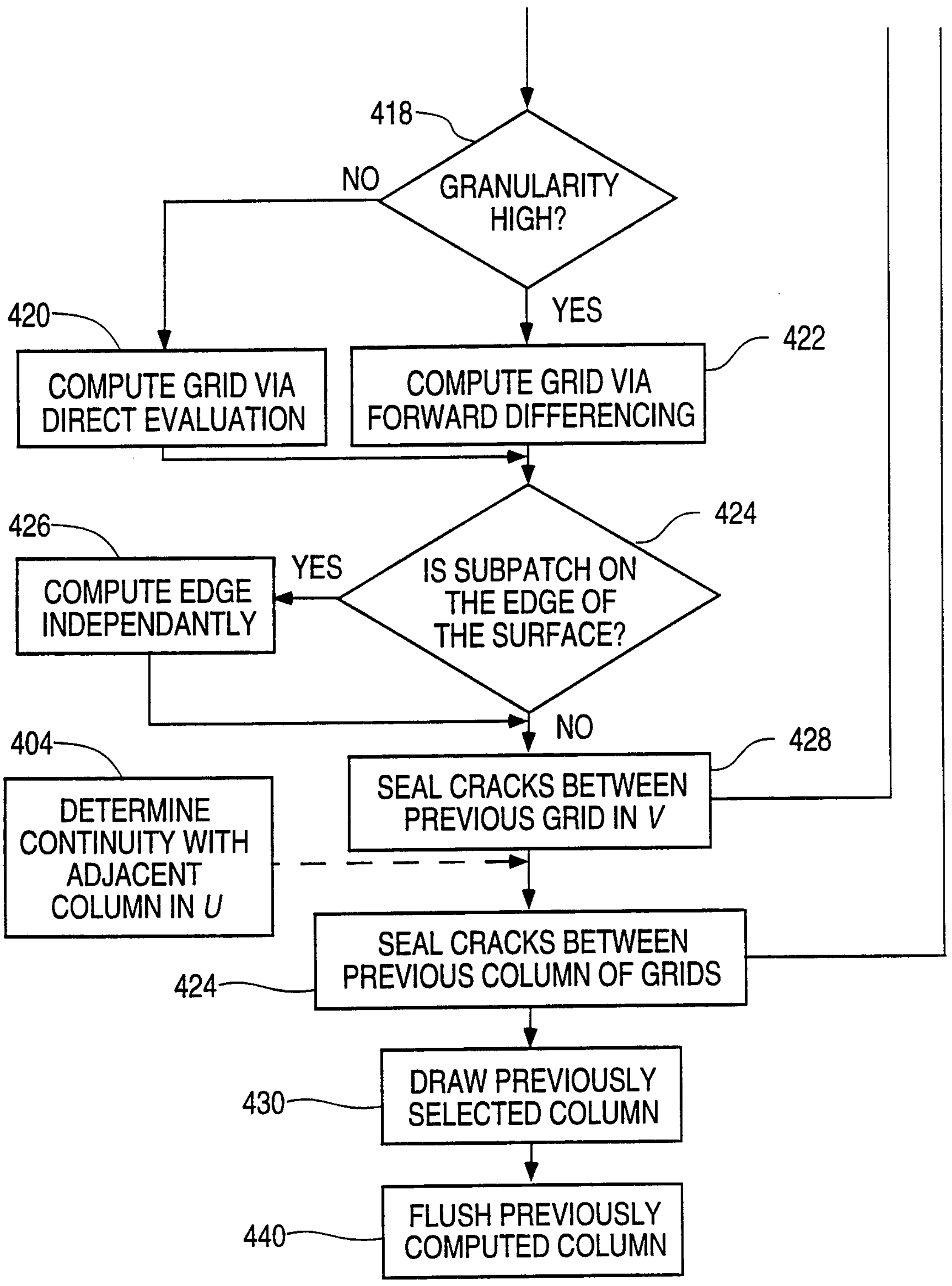

418
GRANULARITY HIGH?
NO
YES
420
COMPUTE GRID VIA DIRECT EVALUATION
422
COMPUTE GRID VIA FORWARD DIFFERENCING
424
IS SUBPATCH ON THE EDGE OF THE SURFACE?
YES
426
COMPUTE EDGE INDEPENDANTLY
NO
404
DETERMINE CONTINUITY WITH ADJACENT COLUMN IN U
428
SEAL CRACKS BETWEEN PREVIOUS GRID IN V
424
SEAL CRACKS BETWEEN PREVIOUS COLUMN OF GRIDS
430
DRAW PREVIOUSLY SELECTED COLUMN
440
FLUSH PREVIOUSLY COMPUTED COLUMN

FIG. 4B

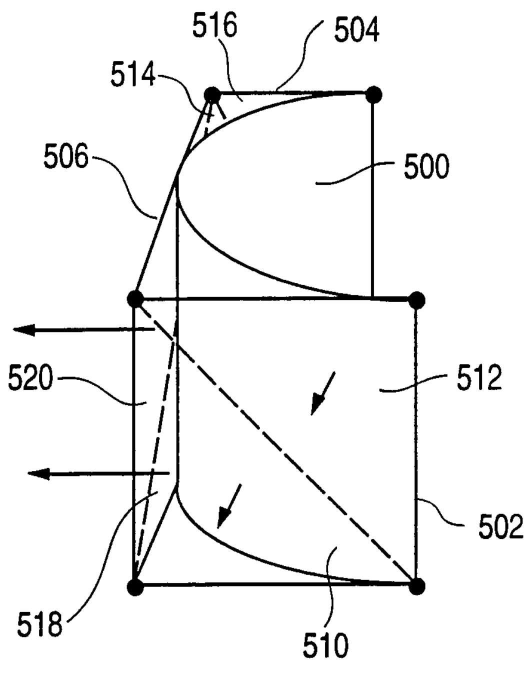


FIG. 5

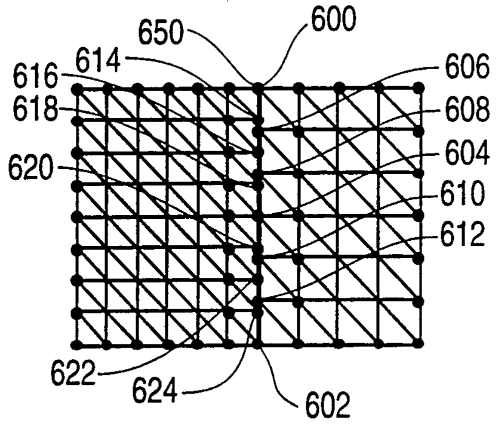

650
600
614
616
606
618
608
604
620
610
612
622
624
602

FIG. 6

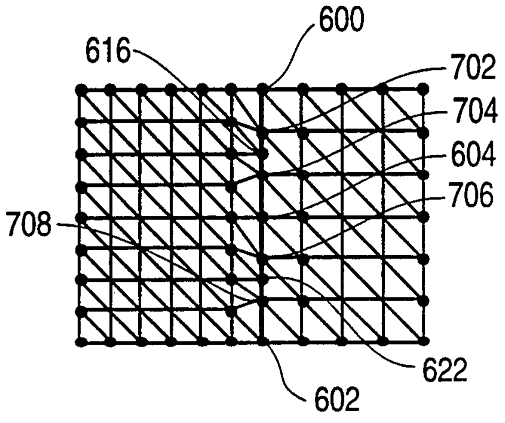

600
616
702
704
604
706
708
622
602

FIG. 7

INTERNATIONAL SEARCH REPORT

Inten nal Application No
PCT/US 94/00194

A. CLASSIFICATION OF SUBJECT MATTER
IPC 5 G06F15/72

According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)
IPC 5 G06F

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic data base consulted during the international search (name of data base and, where practical, search terms used)

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category °	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
Y	COMPUTER VISION, GRAPHICS AND IMAGE PROCESSING vol. 28, no. 3 , December 1984 , USA pages 303 - 322 KOPARKAR ET AL. 'computational techniques for processing parametric surfaces' *paragraph: "3.estimating planarity"* *paragraph:"4.local visibility"* see figure 4 --- -/--	1,2,13, 14

[X] Further documents are listed in the continuation of box C.

[X] Patent family members are listed in annex.

° Special categories of cited documents :

"A" document defining the general state of the art which is not considered to be of particular relevance

"E" earlier document but published on or after the international filing date

"L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)

"O" document referring to an oral disclosure, use, exhibition or other means

"P" document published prior to the international filing date but later than the priority date claimed

"T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention

"X" document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone

"Y" document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art.

"&" document member of the same patent family

1

Date of the actual completion of the international search	Date of mailing of the international search report
11 May 1994	03.06.94

Name and mailing address of the ISA	Authorized officer
European Patent Office, P.B. 5818 Patentlaan 2 NL - 2280 HV Rijswijk Tel. (+ 31-70) 340-2040, Tx. 31 651 epo nl, Fax: (+ 31-70) 340-3016	Perez Molina, E

Form PCT/ISA/210 (second sheet) (July 1992)

Interna ıl Application No
PCT/US 94/00194

C.(Continuation) DOCUMENTS CONSIDERED TO BE RELEVANT

Category °	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
Y	PROCEEDINGS.GRAPHICS INTERFACE`90 14 May 1990 , CANADA pages 1 - 8 FORSEY ET AL. 'an adaptative subdivision algorithm for crack prevention in the display of parametric surfaces.'	1,2,13, 14
Y	*paragraph:"5.summary"* ---	3-8, 10-12, 15-20, 22-24
Y	EP,A,0 456 408 (IBM) 13 November 1991 *abstract* see page 3, line 36 - line 42 see page 4, line 18 - line 24 see page 5, line 9 - line 38; figures 2-3 ---	3-8, 10-12, 15-20, 22-24
A	EP,A,0 366 463 (TEKTRONIX) 2 May 1990 -----	

1

INTERNATIONAL SEARCH REPORT

Information on patent family members

Intern al Application No
PCT/US 94/00194

Patent document cited in search report	Publication date	Patent family member(s)		Publication date
EP-A-0456408	13-11-91	US-A-	5163126	10-11-92
		JP-A-	4229387	18-08-92
EP-A-0366463	02-05-90	US-A-	5142617	25-08-92
		JP-A-	2171877	03-07-90