COMPUTER-AIDED DESIGN & APPLICATIONS, 2017
VOL. 14, NO. 5, 622–631
http://dx.doi.org/10.1080/16864360.2016.1273579

T-spline local refinement as a belief revision system: a rule-based implementation

Abdulwahed Abbas [a] and Ahmad Nasri [b]

[a]The University of Balamand, Lebanon; [b]Fahad Bin Sultan University, Saudi Arabia

ABSTRACT
As originally conceived, T-splines generalize both NURBS and Subdivision surfaces. Central to T-splines is the knot refinement algorithm, which seems to successfully import the local characteristic of B-spline and NURBS curve knot insertion. However, the mathematical decisiveness manifested in curve knot insertion is nowhere to be seen in previously published versions of T-spline local refinement. In this respect, this paper gives a tutorial exposition of T-spline local refinement, interpreted in the spirit of a belief-revision metaphor. It also provides a detailed implementation of that, designed following the architecture of rule-based systems. Both of these are classical topics in traditional Artificial Intelligence Research.

KEYWORDS
T-spline surfaces; local refinement; belief revision; rule-based systems

1. Introduction

The best known methods (B-splines, NURBS [7] and recursive subdivision [5], for examples), for modeling geometric surfaces, start from a so-called control mesh on the basis of which the desired surface may be constructed. In this sense, T-splines[1] [9, 10, and 11] are no different, because modeling starts from an initial control mesh. However, the initial control mesh here is abstracted away through having its constituents (i.e., vertices, edges and faces) embedded inside a two-dimensional grid called a T-mesh, presumably considered within the knot domain.

By comparison with other modeling schemes, a T-mesh permits fewer connections between pairs of control points, thus causing the appearance of so-called T-junctions on the T-mesh. Less vertices, edges and faces are adopted without jeopardizing the expressive power of the scheme. Quite the contrary, this seems instead to enhance the flexibility of T-splines by allowing more freedom and ease of basic manipulations. At the same time, this may be considered as more economical in terms of space usage and in terms of processing time.

Furthermore, the T-mesh structure allows for the automatic inference of the knot information associated with its constituent control points and, more importantly perhaps, it comes with a refinement routine whose local effects make it comparable to the classical curve knot insertion.

However, T-spline local refinement is cast in an algorithmic language lacking the mathematic decisiveness manifested in the curve knot insertion, and keeping in the shadow many details that need to be available for implementation purposes.

Accordingly, this paper gives a tutorial exposition of this routine portrayed using a belief revision metaphor and further provides a rule-based implementation of that. The flexibility of this interpretation motivates the introduction of more features (such as edge-insertion) that are never explicit anywhere in previously published versions of this routine.

The rest of this paper is structured as follows: Section 2 includes a description of classical knot insertion of B-spline and NURBS curves. Section 3 presents an alternative formulation of knot insertion as a rule-based system, included for the purpose of motivating the alternative implementation of T-spline local refinement. Section 4 provides a review of the original T-splines literature including its characterizing features: automatic knot inference and local refinement.

Section 5 presents a detailed account of the implementation of T-spline local refinement including the required data structure. Section 6 includes a description of the execution of this algorithm on a selected example as well as a brief mention of an application that made use of T-spline local refinement and the edge insertion feature as formulated in this paper. The final section concludes with a

CONTACT Abdulwahed Abbas abbas@balamand.edu.lb

summary, a discussion and some suggestions for further work.

2. Knot insertion for B-spline curves

This section provides a review of the classical knot insertion algorithm for curves intended to make the material that follows more self-contained (see references in [3], for further elaboration on that).

2.1. B-spline basis functions

Given a positive integer k (the order of the basis function) and a sequence (τ) of knots $t_0, t_1, \ldots, t_m$, such that $t_i \le t_{i+1}$, for all i where $0 \le i < m$, N_i^k is the B-spline basis function defined as follows:

- $N_i^1(t) = 1$ when $t_i \le t < t_{i+1}$and 0 otherwise.
- When $k > 1$, $N_i^k(t)$ is defined by the following expression: $\frac{t-t_i}{t_{i+k-1}-t_i} N_i^{k-1}(t) + \frac{t_{i+k}-t}{t_{i+k}-t_{i+1}} N_{i+1}^{k-1}(t)$

B-spline basis functions return nonnegative values (i.e. $N_i^k(t) \ge 0$) satisfying: $0 \le N_i^k(t) \le 1$. They all have *partition of the unity* property: $\sum_{i=0}^n N_i^k(t) = 1$.

2.2. B-spline and nurbs curves

Given a *knot vector* (τ) and a sequence (π) of control points $p_0, p_1, \ldots, p_m$ forming a control polygon, a B-spline curve is defined through the following equation:

$$p(t) = \sum_{i=0}^{n} N_i^k(t) p_i, \text{where } t_{min} \le t < t_{max} \qquad (1)$$

This satisfies the *fundamental identity* $m = n + k$, where $m + 1$, $n + 1$ and k are respectively the number of knots, the number of control points, and the order of the B-spline curve. Note also that, since $N_i^k(t) = 0$ only when $t < t_i$ or $t \ge t_{i+k}$, a control point p_i influences the curve only when $t_i \le t < t_{i+k}$. This means that at most k consecutive control points influence the curve in any given knot span.

NURBS curves directly generalize B-spline curves by associating every point p_i of the control mesh with a weight coefficient w_i, thus leading to the following equation for the curve:

$$c(t) = \frac{\sum_{i=0}^n N_i^k(\mathrm{t}) w_i p_i}{\sum_{i=0}^n w_i}, \text{where } t_{min} \le t < t_{max} \qquad (2)$$

2.3. The knot insertion routine

As classically portrayed, the *knot insertion* routine adds a new knot to an existing knot vector *without altering the shape* of the corresponding curve.

Inserting a new knot increases the number of knots by one. Thus, one way of maintaining the *fundamental identity* mentioned above would be by an identical increase of the number of control points. In actual fact, some existing control points are replaced by new ones through *corner cutting*. This way, if the order of the curve is k, *knot insertion* affects at most k consecutive control points (see Fig. 1). This *locality* characteristic in particular is what makes *knot insertion* interesting.

When the new knot t is such that $t_i \le t < t_{i+1}$, it leads to the insertion of k-1 new control points: q_i on edge $p_{i-1}p_i$, q_{i-1} on edge $p_{i-2}p_{i-1}$, ... , and q_{i-k+2} on edge $p_{i-k+1}p_{i-k+2}$. Thus, the old sequence between p_{i-k+1} and p_i is replaced by a new sequence $p_{i-k+1}q_{i-k+2} \ldots q_i p_i$ by cutting the corners of the old polygon at $p_{i-k+2} \ldots p_{i-1}$respectively. No other control point is affected, which means that $k - 2$ control points of the original control polygon are replaced by $k - 1$ new points. In fact, each new control point q_j, on edge $p_{j-1}p_j$, is determined by: $q_j = (1 - \alpha_j)p_{j-1} + \alpha_j p_j$, where:

$$\alpha_j = \frac{t - t_j}{t_{j+k-1} - t_j}, \text{where } i - k + 2 \le j < i \qquad (3)$$

From now on, and without loss of generality, the discussion and examples will be restricted to cubic B-spline curves; i.e. degree 3 (order 4). Consequently, each control point will be associated with a subsequence consisting

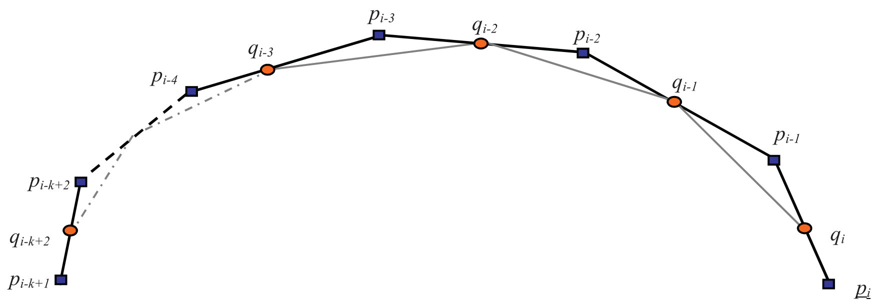

Figure 1. B-spline Curve Knot Insertion.

of five consecutive elements (that will be called the local knot vector) of the global knot vector (τ). Accordingly, equation (2) may be rewritten as:

$$p(t) = \sum_{i=0}^{n} N_0^4[t_i, t_{i+1}, t_{i+2}, t_{i+3}, t_{i+4}](t)p_i, \text{ where } t_i \le t < t_{i+4} \tag{4}$$

Where t_i refers here to the i^{th}element of (τ). Thus, for simplicity, when the local knot vector is known, both subscript and superscript of the basis function will be suppressed without causing any ambiguities.

3. Knot insertion as a rule-based system

In many respects, the material presented in this paper may be considered as an elaboration on ideas that were merely suggested in [2] and [3].

3.1. The metaphor

Very briefly, the memory of an artificial intelligence agent [8] may be considered as a set of consistent beliefs. Such beliefs are interrelated in their consequences in the sense that they involve varying relationships. Thus, in the face of a new incoming belief, the system will undergo a revision of all its beliefs, in order to preserve their consistency. The nature of revision and the depth of possible modifications that may be undergone are related to the interdependency of existing beliefs (and of their consequences) with respect to the new incoming belief.

In the same vein, truth maintenance refers, among other things, to the process the memory has to undergo to preserve its consistency when one (or more) of its belief elements is no longer true.

Accordingly, at initialization, a control polygon is composed of a sequence of control points (π) will be in a consistent state with respect to a global knot vector (τ), in the sense that it possesses the appropriate information to calculate the summation in Eqn. (4). However, when a new knot t is inserted in the sequence (τ), inconsistency may be introduced into the summation in Eqn. (4), in the sense that one or more of the knot local vectors now have invalid knot values caused by this insertion.

In this sense, the vocabulary used in the description of local refinement in [9] may be interpreted against the same spirit to that used in the above metaphors. In other words, when a knot is inserted, the whole setup will undergo a process similar to belief revision or to truth maintenance, whose purpose is to restore consistency, while keeping the value of the summation in Eqn. (4) intact.

3.2. The rules

Given a local knot vector $\boldsymbol{t} = [t_0, t_1, t_2, t_3, t_4]$ and $N(t) = N[t_0, t_1, t_2, t_3, t_4](t)$, when a knot t_* is inserted, one of the following rules (taken from [9]) should fire:

Rule 1. If $\boldsymbol{t}' = [t_0, t_*, t_1, t_2, t_3, t_4]$ then $N(t) = c_1N[t_0, t_*, t_1, t_2, t_3](t) + d_1N[t_*, t_1, t_2, t_3, t_4](t)$ where $c_1 = (t_* - t_0)/(t_3 - t_0)$and $d_1 = 1$
Rule 2. If $\boldsymbol{t}' = [t_0, t_1, t_*, t_2, t_3, t_4]$ then $N(t) = c_2N[t_0, t_1, t_*, t_2, t_3](t) + d_2N[t_1, t_*, t_2, t_3, t_4](t)$ where $c_2 = (t_* - t_0)/(t_3 - t_0)$ and $d_2 = (t_4 - t_*)/(t_4 - t_1)$
Rule 3. If $\boldsymbol{t}' = [t_0, t_1, t_2, t_*, t_3, t_4]$ then $N(t) = c_3N[t_0, t_1, t_2, t_*, t_3](t) + d_3N[t_1, t_2, t_*, t_3, t_4](t)$ where $c_3 = (t_* - t_0)/(t_3 - t_0)$ and $d_3 = (t_4 - t_*)/(t_4 - t_1)$
Rule 4. If $\boldsymbol{t}' = [t_0, t_1, t_2, t_3, t_*, t_4]$ then $N(t) = c_4N[t_0, t_1, t_2, t_3, t_*](t) + d_4N[t_1, t_2, t_3, t_*, t_4](t)$ where $c_4 = 1$ and $d_4 = (t_4 - t_*)/(t_4 - t_1)$
Rule 5. If $t_* \le t_0$ or $t_* \ge t_4$, $N(t)$ does not change

These rules specify *how the inconsistent knot vector should be split to counter the effects of the inserted knot and the weights that should be* associated with each split component in order to preserve the integrity of the original state this vector is coming from (see Eqn. (4)).

From now on, and for ease of presentation, in the case of curves, the term *vertex* will be used to refer to the triplet: control point, weight coefficient and local knot vector. However, in the case of surfaces, this will also include a second local knot vector.

3.3. The data structure

When performing knot insertion, the control polygon needs to hold the control points of the curve together with the corresponding local knot vectors, since individual curve points can then be calculated on the basis of such information, through Eqn. (4).

The process that is able to reproduce knot insertion can manipulate the summation in Eqn. (4) directly and simply substitute equals for equals. However, the description of that should start from the underlying data structure (see Fig. 2).

Accordingly, given the initial sequence of control points (π) and the initial knot vector (τ), a sequence $T[0..n+4]$ is constructed such that each element holds a sequence of vertices $< p, \boldsymbol{t} >$, where p is a control point and $\boldsymbol{t}$ is a local vector consisting of 5 consecutive knots of the global knot vector (τ).

At initialization, for any given index i of the sequence T, $T[i]$ will contain only a single vertex $< p_i, \boldsymbol{t}_i >$, where the first element of $\boldsymbol{t}_i$ should precisely be t_i.

In the treatment that follows, it would perhaps be more suggestive to associate each point of the sequence (π)

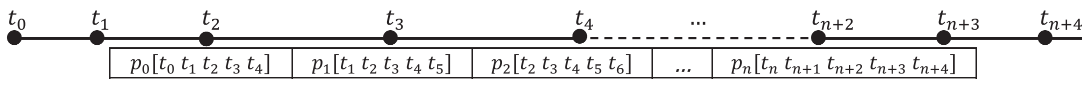
t_0
t_1
t_2
t_3
t_4
...
t_{n+2}
t_{n+3}
t_{n+4}
$p_0[t_0\ t_1\ t_2\ t_3\ t_4]$
$p_1[t_1\ t_2\ t_3\ t_4\ t_5]$
$p_2[t_2\ t_3\ t_4\ t_5\ t_6]$
...
$p_n[t_n\ t_{n+1}\ t_{n+2}\ t_{n+3}\ t_{n+4}]$

Figure 2. Data Structure for Alternative Formulation of Knot Insertion.

with the middle knot of the corresponding local knot vector (see Fig. 2).

3.4. The routine

Given the sequence of Fig. 2, when an inserted knot t_* falls within the span of a local knot vector of a vertex in this sequence, an appropriate rule should be invoked on this local vector (see Fig. 3). This will replace this vertex by pair of vertices thereby restoring consistency to this particular slot of the sequence.

Each component of the replacing pair will be stored in the appropriate slot of the sequence specified by the middle knot of the corresponding knot vector. This means that some of these slots will need to accommodate more than a single vertex. However, various vertices within the same slot of the sequence will end up having the same local knot vector.

The final step of the routine will add the vertices of each slot together thus forming the refined control polygon emanating from the original one the process started off with in Fig. 2. Note here that, the presence of t_* among the knots of Fig. 3 will simply mean that $m = n + 1$. From the point of view off efficiency, this alternative algorithm is still comparable to the classical knot insertion routine.

4. B-spline, NURBS and T-spline surfaces: a brief overview

This section gives a brief overview of B-spline, NURBS and T-spline surfaces which seems to be necessary before delving into the main focus of this paper.

4.1. B-spline surfaces

Given two parameters u and v, a B-spline surface may be obtained as a direct generalization of equation (2) by taking a tensor product form:

$$\sum_{i=0}^{m} \sum_{j=0}^{n} N_i^k(u) N_j^l(v) p_{ij} \text{ where } u_{min} \le u < u_{max} \text{ and } v_{min} \le v < v_{max} \tag{5}$$

In the same spirit, inserting a knot pair $< u, v >$ on the $(m+1) \times (n+1)$ control mesh involves inserting the knot u on every row and inserting the knot v on every column of the control mesh. This ends up with adding a whole row and a whole column of control points to the control mesh in the respective places.

4.2. NURBS surfaces

NURBS directly generalize B-spline surfaces by associating every point p_{ij} of the control mesh with a weight w_{ij} thus leading to the following expression for the surface:

$$\frac{\sum_{i=0}^{m} \sum_{j=0}^{n} N_i(u) N_j(v) w_{ij} p_{ij}}{\sum_{i=0}^{m} \sum_{j=0}^{n} w_{ij}}, \text{ where } u_{min} \le u < u_{max} \text{ and } v_{min} \le v < v_{max} \tag{6}$$

Aside from weight association with individual control points, everything else to do with NURBS surfaces remains very much in the same spirit as those of B-spline surfaces, including knot insertion.

4.3. T-spline surfaces

T-spline surfaces may be considered as a generalization of NURBS surfaces in several respects. In fact, while T-spline surface points are calculated in the same way as NURBS surface points *via* Eqn. (6), T-spline surfaces rely on several other distinctive basic concepts.

4.3.1. T-meshes

T-splines take modeling up to a higher level of abstraction called a *T-mesh*, which is a two-dimensional structure presumably embedded in the (uw) parameter domain. The control points of the T-spline surface are thus embedded in the planar T-mesh, where edges (each connecting two points) are restricted to run along a given row or a given column of the T-mesh (see Fig. 4(a)).

Furthermore, contrary to what is usually encountered in B-splines and NURBS surfaces, an edge line in a T-mesh does not have to run continuously from one border of the T-mesh to the opposite border. Rather, it can be

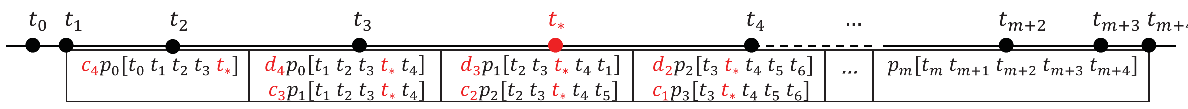
t_0
t_1
t_2
t_3
t_*
t_4
...
t_{m+2}
t_{m+3}
t_{m+4}
$c_4 p_0[t_0\ t_1\ t_2\ t_3\ t_*]$
$d_4 p_0[t_1\ t_2\ t_3\ t_*\ t_4]$
$c_3 p_1[t_1\ t_2\ t_3\ t_*\ t_4]$
$d_3 p_1[t_2\ t_3\ t_*\ t_4\ t_1]$
$c_2 p_2[t_2\ t_3\ t_*\ t_4\ t_5]$
$d_2 p_2[t_3\ t_*\ t_4\ t_5\ t_6]$
$c_1 p_3[t_3\ t_*\ t_4\ t_5\ t_6]$
...
$p_m[t_m\ t_{m+1}\ t_{m+2}\ t_{m+3}\ t_{m+4}]$

Figure 3. The effects of Knot Insertion using appropriate rules.

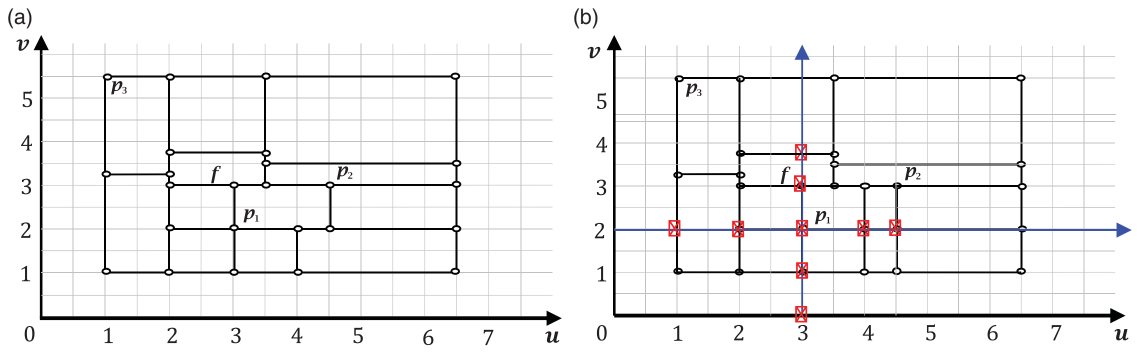

Figure 4. T-spline: (a) A T-mesh (b) Knot Vector Inference.

broken at an inner control point of the T-mesh resulting in a T-junction (see Fig. 4(a)). Thus, a T-spline surface can in fact be composed of considerably less control points than their counterparts in B-splines and NURBS surfaces.

4.3.2. The knot inference mechanism

Another use of the T-mesh is that it comes equipped with a knot inference mechanism that is able to automatically generate the two local knot vectors associated with any control point of the T-mesh (see Fig. 4(b)).

In fact, since edges of the T-mesh run parallel to one or the other of the (*uw*) coordinate system, the knot vectors associated with any point of the T-mesh may be inferred as follows: take a local coordinate system centered at this point with axes parallel to those of the main coordinate system. The knot vectors of this point then simply correspond to the nearest intersections to the center of the local system of the axis of this system with the control points or with the edges of the T-mesh. In this sense, an edge on the T-mesh does not seem to have any other role to play other than determining the next knot value with respect to a given control point on the T-mesh.

Fig. 4(b) shows five intersections in each direction of the two axes. Therefore, the corresponding point has two knot vectors composed of five knots each, simply because the T-spline surface being considered here is assumed to be bi-cubic (see Fig. 5). Moreover, contrary to the B-spline and NURBS surfaces, the knot vectors might not be the same for any two distinct points of the T-mesh.

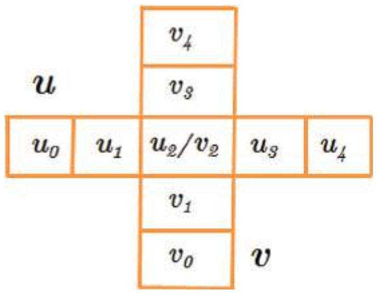

Figure 5. Local Knot Vectors.

4.3.3. Local refinement

The two published versions of the T-spline local refinement algorithm [10, 9] largely miss the mathematical decisiveness of the classical knot insertion algorithm for curves, as they are rather expressed in a language more often encountered in the symbolic computing and automatic theorem proving literature as briefly reviewed in section 3.1.

The first version [10] bears a striking resemblance to the so-called backward-chaining algorithm [8] often used in those contexts. In fact, that version of the algorithm insists that all knot vectors of vertices encountered in the neighborhood of where the knot pair $< u, v >$ is inserted, should not be disturbed. Otherwise, the algorithm will embark on a chain of control point insertions in order to remedy the possible effects of that insertion.

Similar to backward chaining, this chain of insertions usually resorts to recursive calls to the original insertion process. This can run arbitrarily deep, depending on the initial setting of the T-mesh. This second version of the algorithm [9], which represents the main focus of this paper, bears more resemblance to forward chaining [8].

The problems that this process is called to remedy (which are referred to in section 3.1 as *inconsistencies*) are called *conflicts* in [9]. In this respect, the first kind of conflicts arises when the inserted knot interferes with the knot vector of an existing control point, which is the kind that occurs in the case of knot insertion for curves (see section 3.4).

This conflict is resolved using knot vector splitting via one of the rules listed in section 3.2. However, in the case of T-splines, the other knot vector of the control point, carried with one of split halves of the knot vector to a different location, potentially causing a new kind of conflicts in the new location. In fact, one of the shifted knots might miss its reason of existence in its new location.

One obvious and seemingly inexpensive way to resolve this second kind of conflict would be to explicitly insert a new knot there. Thus, in contrast to the situation of curve case, this carries the risk of the refinement process expanding its influence beyond the immediate neighborhood of the position where the original knot is inserted.

In conclusion, the above details seem to favor an implementation of the T-spline local refinement as a rule-based system [2].

5. T-spline local refinement as a rule-based system

This section is concerned with an implementation of the T-spline local refinement as a rule-based system. However, before we carry on, it should perhaps be emphasized here that this is an exposition of an in-principle implementation that responds to the basic requirements of the algorithm as posed in Sederberg [9]. In this respect, particular attention will be paid to the locality characteristic of this algorithm.

5.1. The data structure

In the case of B-spline and NURBS surfaces, the mesh will be composed of a (full) regular grid of points, where the knot vectors associated with each point are easily derived.

Regularity stems from the fact that the U-knots are the same for each point on the same column, and the V-knots are the same for each point on the same row (see Fig. 6(a)). By Comparison, the T-spline mesh (T-mesh) is composed of a (sparse) grid of nodes (see Fig. 6(b)).

5.1.1. T-mesh data structure

Accordingly, the data structure associated with this process is partly depicted in Fig. 6(b), which is designed on the basis of the requirements of the problem at hand [1]. It is composed of the following items:

- Grid dimensions: M and N, where M is the number of columns and N is the number of rows.
- The global knot vectors: $U$and V, where U has M knots and V has N knots. These vectors are extendable to permit the accommodation of possibly more knots.
- The grid G is simply an $M \times N$ sparse matrix of nodes. This matrix is sparse in the sense that a sizable number of its elements are non-existent. Moreover, in the same way as the vectors U and V, the grid is row-wise and column-wise extendible, as this may be needed in order to accommodate more nodes that are required to be inserted on new rows and/or columns.
- Note that, in order to preserve consistency, a newly inserted row into the grid needs to correspond to a new knot that is inserted in the knot vector V. Similarly, a newly inserted column into the grid needs to correspond to a new knot that is inserted in the knot vector U.
- Thus, a node element existing at position $<i, j>$ of the matrix will also have knot coordinates $< U[i], V[j] >$. These latter coordinates will remain constant no matter how many times and in how many different ways the grid is extended.
- Moreover, this matrix representation provides a low-cost method of navigation between various elements of the grid, which will need to be performed very frequently during the local refinement process.
- A node has four associated knot fields: L (*left*), R (*right*), U (*up*) and D (*down*). These are used to mark the existence of an edge joining a pair of node elements of the grid. A default value is assigned to the corresponding field in case such an edge is non-existent.
- Finally, a node is also associated with a sequence of *vertices* residing at that position of the grid. Initially, such a sequence has only one element having two local knot vectors u and v determined by the knot inference mechanism explained in section *4.3.3*. This sequence indicates that the data structure being described here is a straightforward generalization of that described for the curve case in section 3.3.
- At termination, all vertex elements of each sequence will have identical local knot vectors, which permit their collapse into a single *vertex* through addition.

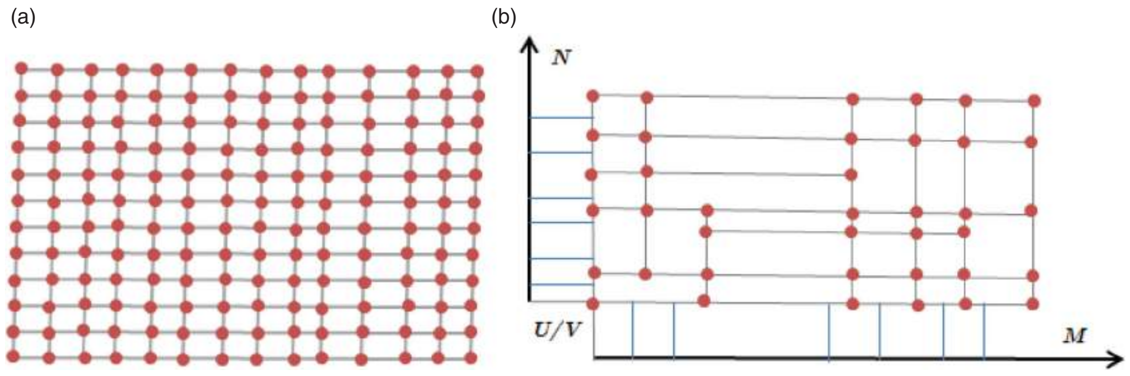

Figure 6. Control Meshes: (a) B-spline and NURBS Surfaces Meshes (b) The T-spline Mesh (T-Mesh).

5.1.2. The revision list

The main task of the local refinement algorithm is to detect then remove conflicts that may arise as consequences of the first knot insertion. A key idea for the success of this endeavor is the reduction of the amount of checking for conflicts that will frequently be carried out during the process. In other words, it would be impractical to make an exhaustive scan of the whole T-mesh looking for conflicts every time the need arises. Instead, the algorithm will have at its disposal a list of vertices that are the only possible suspects for causing conflicts, which should represent a small proportion of the total number of vertices constituting the T-mesh.

In this regard, each element of the revision list will be composed of a vertex plus a (u or v) indicator as to which of the two associated local knot vectors is the potential cause of the conflict, if any. This revision list will be continuously updated during the local refinement process.

In fact, a conflict may arise precisely when a new element (an edge or a node) is inserted in the T-mesh. For instance, when a newly inserted edge crosses the knot span of any of the two local knot vectors of an existing vertex of the T-mesh then this vertex will need to be added to the revision list indicating that the corresponding knot vector will need to be checked for causing a possible conflict.

On the other hand, when a new node is inserted, two cases may arise:

- If the node contain a vertex then that must have got to its current position along a knot direction. This would mean that this vertex will need to be added to the revision list indicating that its other knot vector will need to be checked for causing a possible conflict.
- On the other hand, if this node is empty (i.e. does not possess any vertex yet) and falling within the knot span of any of the two local knot vectors of an existing vertex of the T-mesh then this vertex will need to be added to the revision list indicating that the corresponding knot vector will need to be checked for causing a possible conflict.

It should perhaps be noted here that the effort spent to carry both tasks is minimal in these cases.

5.2. The control structure

In this section, some of the routines of concern are described below in a **C**-like pseudo-code. The code refers to the knot position of insertion **< *u*, *v* >**, a revision list ***R*** and a T-mesh ***T***.

5.2.1. Knot insertion

```
void Insert( < u, v > , R, T) {
  if position < u, v > is on an edge E then
      InsertOnEdge( < u, v > , E, R, T)
  else {
  if there exists a nearest position < u', v'> to < u,
      v > on a horizontal or a vertical edge E'
  if < u', v'> are not the knot coordinates of an
      existing point in T then
  InsertOnEdge( < u', v' > , E', R, T)
  else {
      find a nearest position < u', v'> to < u, v > on
      a populated row or column of T
      Insert( < u', v' > , R, T)}
    InsertNode( < u, v > , R, T)
    InsertEdge( < u, v > , < u', v' > , R, T)}
   BookKeeping( < u, v > , R, T) }
  void InsertNode( < u, v > , R, T) {
   make an empty node N
   add N at position < u, v > of T
   add all affected vertices to R}
  void InsertEdge( < u, v > , < u',v' > , R, T) {
   connect positions < u, v > and < u', v'> via an
      edge in T
   add all affected vertices to R}
  void InsertOnEdge( < u, v > , E, R, T) {
    InsertNode( < u, v > , R, T);
    split E into two edges at < u, v > }
  void BookKeeping( < u, v > , R, T) {
    for any remaining nearest positions < u', v'> of an
      existing point in T
      on the same row or column still unconnected
        to < u, v > via an edge
      InsertEdge( < u, v > , < u',v' > , R, T)}
```

5.2.2. The matching process

When a potential cause of conflict is attributed to a given local knot vector (say $\boldsymbol{u} = [u_0 u_1 u_2 u_3 u_4]$) attached to a given vertex of the T-mesh, it is essential to determine the type of conflict in order to decide on a suitable resolution. This is done by first inferring the current state of this vector (say $\boldsymbol{c} = [c_0 c_1 c_2 c_3 c_4]$) then running both vectors through a matching process that is able to detect the offending knot (if any) that is a reason of this mismatch.

Because, the middle knots of $\boldsymbol{u}$ and $\boldsymbol{c}$ must be the same, since they are the knot coordinates of the same vertex, matching can be confined to the first (and the last) two pairs of knots of these two vectors. We will present here the matching the first two pairs. The other two pairs may be done in very much the same way:

- If $c_1 > u_1$ then the offending knot is c_1.
- Otherwise if $c_1 < u_1$ then the offending knot is u_1.

- Otherwise if $c_0 > u_0$ then the offending knot is c_0.
- Otherwise if $c_0 < u_0$ then the offending knot is u_0.

This, by itself, is sufficient to determine the cause of the conflict, and therefore the way to resolve it (see the following subsection).

5.2.3. *The revision process*

```
// revision depends on the flag of s which is could be u or v
// let us suppose that it is u, the other case of v can be handled similarly
void revise(s, R, T) {
    Infer new knot vector u' of s
    match u' against u
    if the offending knot t belongs to u then
        Insert( < t, v[2] > , R, T)
    else if the offending knot t belongs to u' {
        split s into two halves s' and s" at t
        insert the first half in place of s and add all
            affected vertices to R
        insert the second half at < t, v[2] > and
            add all affected vertices to R}}
```

5.2.4. *Local refinement*

```
void main () {
    readMesh(T); // initialize with given dimensions,
        points and edges (see Fig. 6(b))
    Infer the local knot vectors of each point and add that
        to the corresponding points in T
    print-T-mesh(T); //printing done with reference to
        the knot coordinates < u[2], v[2] > of each node
    printSurface(T); // done with reference to the
        Cartesian coordinates < x, y, z > of each point.
    read u and v;
    if < u, v > are the polar coordinates of an existing
        node in T then stop.
    else {
    if either u or v is not on an existing row or column
        then insert missing row or column in T.
    Insert( < u, v > , R, T); // this is the core function in
        the whole process
    }
    While (R != empty) { // revise
    s = select-vertex-from(R);
    revise(s, R, T);
    remove(s, R);
    }
    tidy-up(T);
    print-T-mesh(T);
    printSurface(T);}
```

6. An example and an application

In this example, the starting configuration of the T-mesh is depicted in Fig. 7.

Insert at knot position **< 10, 9 >** in **Fig.** 7 (a).
Vlist: {**10, 11, 12, 13**} along the ***u*** direction
Revise 13: match old **u: [9 12 15 18 21]**
and new **u: [10 12 15 18 21]** of **13**
Split u into: **[9 10 12 15 18]** and **[10 12 15 18 21]** $\Rightarrow$ no further revision is needed here since same ***v*** in all newly arising cases
Revise 12: match old u: **[7 9 12 15 18]**
and new **u: [9 10 12 15 18]** of **12**
Split u into: **[7 9 10 12 15]** and **[9 10 12 15 18]** $\Rightarrow$ no further revision is needed here since same ***v*** in all newly arising cases
Revise 11: match old **u: [5 7 9 12 15]**
and new **u: [5 7 9 10 12]** of **11**
Split u into: **[5 7 9 10 12]** and **[7 9 10 12 15]** $\Rightarrow$ no further revision is needed for first half since same ***v*** in all newly arising cases, besides, **24** is now fully initialized
Revision for second half: match old v: **[5 7 9 12 15]**
and new **v: [5 7 9 15 18]** of **24**
Insert at knot position **< 10, 12 >** in Fig. 7 (b): insert **edge** to closest node and in Fig. 7 (c).
Vlist: {**7 and 8**} along the ***u*** direction, no further revision is needed here since same ***v*** in all newly

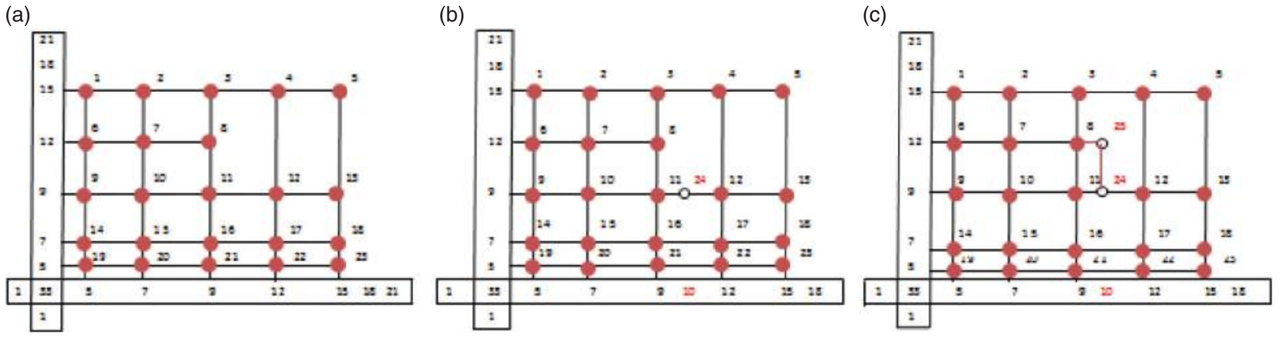

Figure 7. The T-spline Local Refinement Example: (a) Initial State (b) Intermediate State (c) Final State.

arising cases and only {**24**} along the ***v*** direction, no further revision is needed here since same ***u*** in this newly arising case.

6.1. An application

Polygonal complexes [4] are structures that can be embedded within a control mesh of, ultimately, any modeling scheme. Their purpose is to make it simple to interpolate a B-spline curve by such surfaces [6]. This notion has successfully been utilized within B-spline, NURBS and several types of subdivision surfaces, and lately within T-spline surfaces.

In the T-spline domain, this seems to require repeated applications of local refinement, where edge insertion seems to play a critical role, which is the reason for their mention in this context.

7. Conclusions and suggestions for further work

Given the nature of this algorithm and the frequency of its use in any given application [5], its implementation has to respond to a number of concerns:

- Efficiency in terms of time and space
- The conflicts that a knot causes should be detected easily preferably without causing too many checks, especially when these checks could possibly become exhaustive.
- Determining the nature of the conflict should be performed efficiently so as to quickly determine how to resolve it.
- The effects of the algorithm should remain local with respect to entirety of the T-mesh.
- Finally during processing, when the system reaches a situation where more than one alternative may be taken, the system should be able to decide on the optimal route to take, since the algorithm does not have the means to backtrack from the position it is currently at.

One might say that the above issues fall in the domain of implementation and, as such, it should not belong to the domain of the theory. However, an implementation issue immediately turns into a theoretical issue if it cannot be suitably and efficiently resolved.

In addition to that, the data structure needed by the algorithm is drawn on the basis of these requirements [1]. In many respects, this pays more attention to transparency of the basic operations of the algorithm, perhaps at the cost of the efficiency.

Furthermore, the new additional edge-insertion feature is explicitly treated in this version of the implementation, something that, to our knowledge, has never been addressed explicitly before in this context. This feature should have a role to play with regard to the locality characteristic of the algorithm as a whole, in addition to other sub-routines of this algorithm.

As stated in [9], and as it is formulated there, T-spline local refinement would have a worst case scenario, where the chain of insertions will extend till the T-mesh landscape in Fig. 6(b) will reach a limit similar to that depicted in Fig. 6(a). That is, the initial T-mesh will end up looking like that of a NURBS surface. This comes as no surprise; in fact, the resemblance of T-spline local refinement to artificial intelligence forward chaining means that, in the worst case, it will sometimes suffer the same fate; i.e. trying all the possibilities before reaching a solution (see Fig. 8).

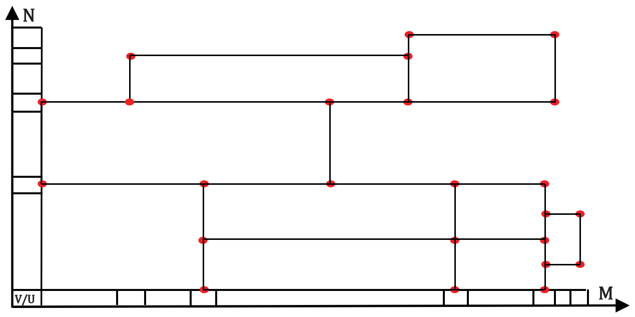

Figure 8. A Potential Source of a Worst Case Scenario.

Further research should examine whatever heuristic can be employed to assign priorities to whatever decisions are made during the search so as to avoid the worst case scenario. In our opinion, this would revolve around the function called **select-vertex-from**, in section 5.2.4.

Note

1. Our reference to T-splines here is restricted to the specifications of that mentioned in ref. [9] and [10], thus excluding any new features that may have been added to this technology since then [11].

Acknowledgement

The research work reported herein is supported by a grant from the Lebanese Council for Scientific Research, for the academic year 2015-2016. Thanks are also due to Lama Ouayjan for implementing the programming system that generated the figures included in this paper.

ORCID

Abdulwahed Abbas http://orcid.org/0000-0002-3954-7076
Ahmad Nasri http://orcid.org/0000-0002-2047-6693

References

[1] Abbas, A.; Nasri, A.: Synthesizing data structure requirements from algorithm specifications: case studies from recursive subdivision for computer graphics and animation, ACS/IEEE International Conference on Computer Systems and Applications, 2003. http://dx.doi.org/10.1109/aiccsa.2003.1227520

[2] Abbas, A.: A Framework for Rule-Based Systems in CAGD Software, the 7th International CAD Conference and Exhibition, Honolulu, Hawaii, 2007.

[3] Abbas A.: A Generic Rule-Based Formulation of Knot Insertion across CAD Applications, Computer-Aided Design and Applications, 5(6), 2008, 831-840. https://doi.org/10.3722/cadaps.2008.831-840

[4] Abbas, A.: T-spline Polygonal Complexes, Computer Aided Design & Applications, 12(4), 2014, 465–474. http://dx.doi.org/10.1080/16864360.2014.997643

[5] Catmull, E.; Clark, J.: Recursively Generated B-spline Surfaces On Arbitrary Topological Meshes, Seminal Graphics, Ed. Rosalee Wolfe, ACM Press, ISBN 1-58113-052-X, 1998, 183-188.

[6] Nasri, A. H.; Sinno, K.; Zheng, J.: Local T-spline surface skinning, The Visual Computer, 28(6–8), 2012, 787–797. http://dx.doi.org/10.1007/s00371-012-0692-1

[7] Piegl, L.; Tiller, W.: The NURBS Book, Second Edition, Springer Verlag, 1997. http://dx.doi.org/10.1007/978-3-642-59223-2

[8] Russell S.; Norvig P.: Artificial Intelligence: a modern approach (2nd Edition), Prentice Hall, 2003.

[9] Sederberg, T.-W.; Cardon, D.-L.; Finnigan, G.-T.; North, N.-S.; Zheng, J.; Lyche, T.: T-spline Simplification and Local Refinement, ACM Transactions on Graphics, 23(3), 2004, 276–283. http://dx.doi.org/10.1145/1015706.1015715

[10] Sederberg, T.-W.; Zheng, J.; Bakenov, A.; Nasri, A.: T-splines and T-NURCCS, ACM Transactions on Graphics, 22(3), 2003, 477–484. http://dx.doi.org/10.1145/882262.882295

[11] T-splines, http://www.tsplines.com