Can local NURBS refinement be achieved by modifying only the user interface?

Neil A. Dodgson*, Jiří Kosinka

The Computer Laboratory, University of Cambridge, 15 J. J. Thomson Avenue, Cambridge, UK CB3 0FD

Abstract

NURBS patches have a serious restriction: they are constrained to a strict rectangular topology. This means that a request to insert a single new control point will cause a row of control points to appear across the NURBS patch, a global refinement of control. We investigate a method that can hide unwanted control points from the user so that the user's interaction is with local, rather than global, refinement. Our method requires only straightforward modification of the user interface and the data structures that represent the control mesh, making it simpler than alternatives that use hierarchical or T-constructions. Our results show that our method is effective in many cases but has limitations where inserting a single new control point in certain cases will still cause a cascade of new control points to appear across the NURBS patch.

Keywords: NURBS, user interface, local refinement, surface design, hierarchical B-spline, control mesh

1. Introduction

NURBS are the standard mechanism for modelling in CAD. For decades [1], there has been interest in producing hierarchical NURBS, NURBS with T-junctions, and other NURBS variants that allow for local refinement of a NURBS patch (Section 3). None of these solutions, however, has yet been widely adopted in the CAD industry. Some require significant changes to the underlying NURBS engine. We investigate whether it is possible to construct a mechanism that provides local refinement to the designer by modifying *only* the user interface, leaving the underlying NURBS engine unchanged (Sections 5 and 6).

Our motivation is that providing local refinement through the user interface alone would allow CAD software providers to add the extra functionality without the need to make expensive additions and changes to the underlying NURBS engine. Our investigation shows that our method does deliver such functionality but that it suffers from inescapable limitations (Section 8). Nevertheless, this idea provides an interesting intermediate option between the status quo and adoption of a new engine.

2. The challenge

Bivariate NURBS patches are composed, in parameter space, as the tensor product of univariate NURBS. It is well known that, in the univariate case, a NURBS curve can be locally refined arbitrarily often in arbitrary locations (Figure 3). A NURBS patch cannot be refined arbitrarily often at arbitrary point locations, owing to its tensor product nature. Any refinement of the NURBS patch will stretch from one side of the patch to the other (Figure 1).

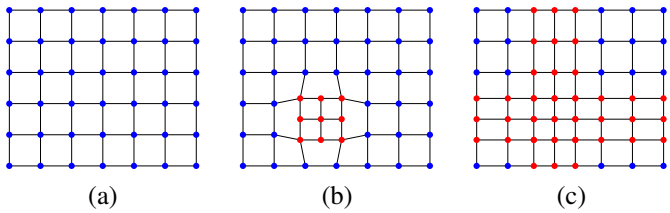

Figure 1: (a) A NURBS patch showing control points in a regular grid. (b) The desirable approach to local refinement, with new control points in a local area only. (c) Attempting to do this with NURBS introduces new control points across the mesh owing to the tensor product nature of NURBS patches.

Our basic idea is to provide a mechanism whose user interface shows only the desired control points to the designer. That is, it hides unwanted control points. We implement this as a series of tensor product control meshes, each of which we call a *layer*. Each layer is a refinement of the layer above in which a single knot is added. Some points from a given layer may be visible to the user and some may be hidden. The rationale here is that the positions of the hidden control points, in the refined layer, can be calculated from control points in the previous layer without altering the shape of the surface. This is just basic knot insertion where, in the univariate case, inserting a single knot in a curve of order k (degree $k-1$) causes one new control point to be introduced and $k-2$ existing control points to be moved without changing the shape of the curve.

Our basic idea is illustrated in Figure 2. Figure 2(a) shows what the user sees in the user-interface. Figure 2(b)–(f) shows how this can be implemented as a series of layers, each of which introduces a single new knot. The bottom-most layer, Figure 2(f), is a tensor-product NURBS that is passed to the underlying NURBS engine. There are three types of points: *visible* control points available in the user interface (coloured circles), *replaced* control points (grey circles) that have been su-

*Corresponding author. Tel: +44-1223-334417 Fax: +44-1223-334678
Email address: nad@cl.cam.ac.uk (Jiří Kosinka)

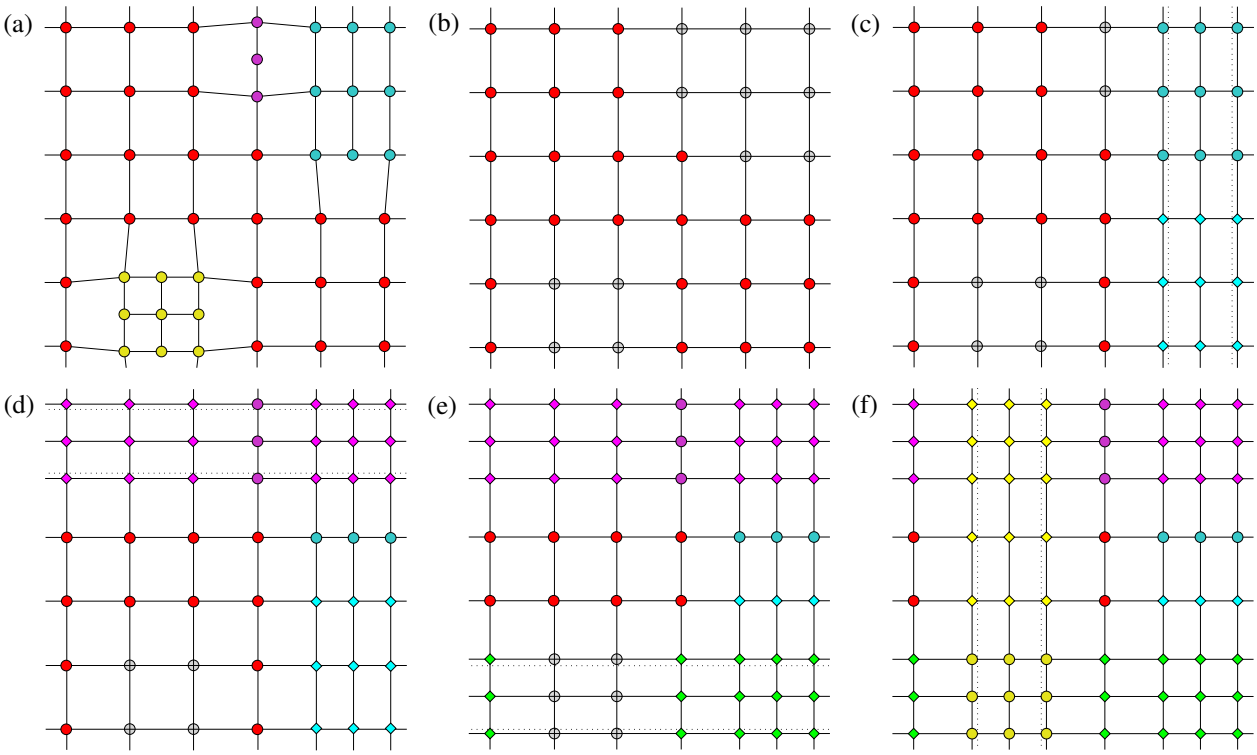

Figure 2: An example of the basic idea in the cubic case. (a) The control mesh as seen by the user. In this example, the original mesh comprised the red control points. The user then requested a row of new points (blue, upper right), a single new point on a single edge (purple, middle top), and a square of new points (yellow, lower left). This is implemented as a series of layers. (b) The top layer is the original mesh. The grey points are those replaced by *visible* points in lower layers. (c) Inserting a row of new points requires a single new vertical knot. Diamonds show control points that are calculated from points in the layer above and circles show points that are revealed to the user for further manipulation. (d) Inserting a single control point still requires the insertion of an entire knot line. (e) A square of control points requires inserting two new knots. The first is inserted horizontally. (f) The second knot is inserted vertically. (f) shows the final tensor-product set of control points. The user-interface, (a), comprises the *visible* control points drawn from all the layers (coloured circles). Those control points in the higher layers that are not used are marked as *replaced* (grey circles) and are not manipulable and not used in any further calculations. Diamonds show *hidden* points calculated from the layer above.

perseded by points in a lower layer, and *hidden* points (coloured diamonds) that are calculated from points in the layer above. Note that every layer is a tensor-product arrangement, while the control mesh visible to the user is not necessarily tensor-product and is constructed by building a mesh from the control points that are marked as *visible* in the various layers.

3. Related work

Since their invention in the 1970s, NURBS [2], a non-uniform rational extension of B-splines, have become a universal standard for representing free form curves and surfaces in computer aided design. Modelling is facilitated by control points whose positions determine the desired shape. NURBS possess many features that make them attractive for various applications such as geometric modelling, analysis, and approximation. However, NURBS suffer from a major drawback: control points need to form a rectangular topological grid.

We are interested in using NURBS for designing models in three-dimensional space. Consider the situation when a fine detail needs to be added to an existing coarse model. This is a typical operation performed in practice, for example, when adding a small ear detail to a face model. The structure of NURBS does not allow this to be performed as a local operation. If a new control point needs to be introduced, a whole strip of control points, running across the whole patch, has to be added. Otherwise, control points would no longer lie in a rectangular grid. Thus, requesting only a single new control point causes many unwanted control points to be introduced into the model. This fact complicates design and produces unnecessary overhead for the designer.

One of the earliest studies addressing this shortcoming led to the framework called hierarchical B-splines (HB-splines) [1]. Using nested spaces, the framework allows for locally refined patches that can represent finer detail. Later, a basis for these nested spaces was found and its stability studied [3]. More recently, HB-splines were studied from the point of view of iso-geometric analysis [4], a finite element framework [5]. By construction, the new basis functions formed by coarse and fine level B-splines do not sum to unity: weights need to be introduced. An improved construction, truncated hierarchical B-splines, which avoids the need for weights and provides a strongly stable basis, was recently discovered [6]. The truncated basis functions are convex combinations of B-splines.

Independently, spline spaces over T-meshes have been investigated [7, 8]. In this approach to local refinement of B-splines,

a T-mesh in the parameter space forms a foundation for the method. The most recent construction that addresses local refinement was coined locally refined (LR) B-splines [9].

The most widely known and used T-construction is T-splines [10, 11]. T-splines are basically B-splines whose control meshes allow T-junctions. Local refinement is supported. Nevertheless, some local changes trigger a whole chain of local refinements [11, §4.3][12, §3.2.5]. T-splines were originally introduced for degree three and later generalised to arbitrary degrees [13].

All of the T-constructions mentioned above are based on B-splines and thus can be converted and generalised to NURBS. This led us to the question, answered in this paper, of whether we could achieve the desired local control by changing *only* the user interface, without introducing T-splines or hierarchical B-splines. This would allow existing NURBS software to be used, with all of its optimisations and functionality, with modifications required only to the user interface.

4. The mathematical framework

The B-spline patch is a bivariate generalisation of the univariate B-spline curve. A B-spline curve is defined by a sequence of n *control points*, $\mathbf{P}_i$, and their associated *basis functions*, $N_{i,k}$.

$$\mathbf{P}(t) = \sum_{i=1}^{n} N_{i,k}(t)\mathbf{P}_i, \;\; t_{\min} \le t < t_{\max}. \tag{1}$$

The basis functions are determined by a sequence of knots, the knot vector $[t_1, t_2, \ldots, t_{k+n}]$, where knots are a non-decreasing sequence of real numbers, $t_i \le t_{i+1} \;\; \forall i$, in a parameter space spanned by t. The *order* of a B-spline curve is given by k, which is one higher than the *degree* of the curve [14].

NURBS are a generalisation of B-splines in which the operations are conducted in a four-dimensional (4D) homogenous coordinate space, where the extra coordinate is a weight associated with the control point. Displaying a NURBS curve in standard three-dimensional (3D) space requires the straightforward projection from this 4D homogenous space to 3D [14, §5–13]. In common with many other authors, we use "NURBS" and "B-spline" interchangeably, with the understanding that NURBS operations require this 4D to 3D projection.

It is straightforward to introduce a new control point into a B-spline or NURBS curve, without affecting the shape of the curve at all. This allows subsequent manipulation of the curve, at a finer level of detail, in the neighbourhood of the new point.

When a new knot is introduced to the knot vector, the locations of the new control points, $\mathbf{Q}_i$, are calculated by simple linear interpolation of the existing control points, $\mathbf{P}_1, \mathbf{P}_2, \ldots, \mathbf{P}_n$. In general, given a knot vector $[t_1, t_2, \ldots, t_{k+n}]$ and a new knot value w to be inserted between knots t_j and t_{j+1}, we find [15]:

$$\begin{array}{ll} \mathbf{Q}_i = \mathbf{P}_i, & i \le j-k+1 \\ \mathbf{Q}_i = (1-\alpha_i)\mathbf{P}_{i-1} + \alpha_i \mathbf{P}_i, & j-k+1 < i \le j \\ \mathbf{Q}_i = \mathbf{P}_{i-1} & j < i \end{array} \tag{2}$$

$$\text{where: } \alpha_i = \frac{w - t_i}{t_{i+k-1} - t_i}.$$

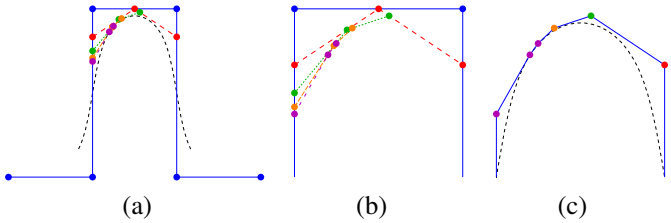

Figure 3: (a) A cubic B-spline curve (dashed black line), defined by the blue control polygon and knot vector $[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]$. The curve is refined four times between the fifth and sixth knot value at $5\frac{1}{2}$ (red), $5\frac{1}{4}$ (green), $5\frac{1}{8}$ (orange), and $5\frac{1}{16}$ (purple). (b) A close-up view of the refinement steps. In each case a new control point is inserted and two existing points are moved. (c) One possible presentation of the user-interface, with the finest refinement presented to the user.

Figure 3(a) and (b) shows an example of a cubic ($k = 4$) B-spline curve, with four new points introduced successively. For this cubic example, the introduction of a new control point requires that the two control points either side are moved to new locations (Equation 2). The resulting, refined, control polygon is shown in Figure 3(c).

A B-spline or NURBS patch is defined as a tensor product, where the basis functions in each direction are derived from separate knot vectors:

$$\mathbf{P}(s,t) = \sum_{i=1}^{m}\sum_{j=1}^{n} N_{i,k_s}(s)N_{j,k_t}(t)\mathbf{P}_{i,j}. \tag{3}$$

It is usual for the patch to have the same order (i.e., $k_s = k_t$) in both directions. Patches are thus defined by a quadrilateral grid of control points of size $m \times n$.

Control of finer detail in a patch can be achieved by introducing new knots, as in the curve case, but these propagate across the whole patch (Figure 1(c)). It is therefore impossible to introduce local control of fine detail in part of the patch without introducing unwanted control of fine detail elsewhere in the patch.

5. Outline of the method

Our desire is to introduce new control points in the user interface only in locations where the user wishes finer control. Our concept for achieving this is to have multiple layers of control points. Each layer is a tensor-product NURBS mesh. The difference between one layer and the next is the introduction of a single new knot in one of the principal directions, that is, in either the s or the t direction in Equation 3. When a new layer is created, links are formed from points in the previous layer to points in the new layer. These links determine the geometric positions of the new layer's control points from those in the layer above, using the simple relationships in Equation 2. Points in this structure are allocated one of three labels:

visible — A point that is available to the user to be manipulated and is therefore visible in the user interface.

hidden — A point that is not visible in the user interface but which is used in determining the final surface; its position is calculated internally from points in the layer above (Figure 2).

replaced — A point that is not visible in the user interface and plays no part in calculating the final surface; the blending function that it would have controlled is instead controlled by one or more *visible* points in lower layers.

When a layer is created, all its points are initially marked as *hidden* and then appropriate points are made *visible* in the user interface. For odd degrees (k even), these are the new point requested by the user and $(k-2)/2$ points on each side of the new point along the row (or column) on which the selected edge sits.

When a point's geometric position is changed, all of its dependent *hidden* points in the layer below are recalculated. Only one knot is introduced for any given layer, in either s or t direction; the recalculation therefore comprises only univariate calculations in that direction for each row of control points. Recalculation propagates down through the layers until it reaches the bottommost layer, and it is this layer of control points that is passed to the NURBS engine.

Switching a point's status from *hidden* to *visible* has implications for *visible* points in higher layers. A *visible* control point will be marked *replaced* when a matching control point in a lower layer becomes *visible* (see Section 6 for details). This is seen, for example, in the univariate case where introducing one new point leads to the replacement, in the visible control mesh, of both of the adjacent existing points (Figure 3(b–c)).

This concept can be extended to the introduction of arbitrarily many knots, with each new knot adding a new layer. Provided the newly-visible control points are separated sufficiently far from one another, this layering concept works perfectly (e.g., Figure 2). It also works for introducing a row of control points at the same knot location (e.g., Figure 2(c)) and for introducing a block of control points (e.g., Figure 2(e) and (f)) where points are refined first in one direction and then the other, creating two hidden layers.

However, the examples in Figure 2 are constructed carefully to avoid any challenging cases. Challenges occur when the user introduces a new control point near to existing *hidden* points. The question is: how best to handle the dependencies between the newly-desired *visible* points and the nearby *hidden* points. There are several possible alternative approaches, which are described and discussed in detail in Appendix A.

Our conclusion, from considering all these approaches, is that the only viable solution to these challenges, one which maintains the integrity of the mesh, is to ensure that any *hidden* point that becomes dependent on (i.e., would be calculated from) a *replaced* point is made *visible* to the user.

6. Algorithm

The data structure (Figure 4) comprises a set of layers, each of which is linked to the layer above (a coarser layer) and the layer below (a finer layer). Each layer is a valid tensor-product NURBS control mesh. Each control point, in each layer, is marked as one of *visible*, *hidden*, or *replaced*. Each control point is marked with a (s,t) co-ordinate corresponding to the position of the central knot in the support of its basis function. This constrains the algorithm to work only for odd degree (k even) because only odd degree B-splines have an odd number of knots in their support[1].

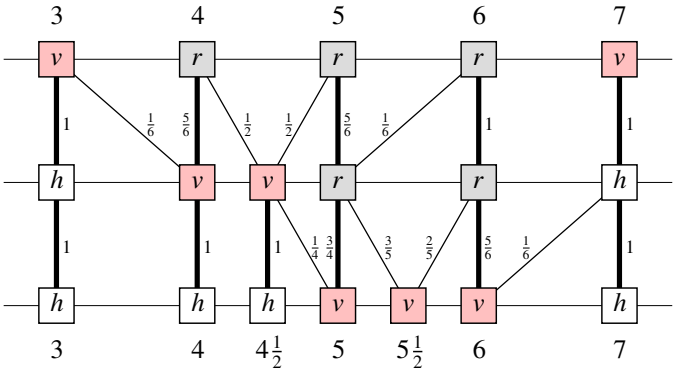

Figure 4: A one-dimensional example of the basic data structure. This example shows three layers of control points. The t value associated with each control point is shown at top and bottom of the diagram. The top row is the coarsest layer, defined by uniformly-spaced knot values; for example, the top-left point has a basis function defined by knots $[1,2,3,4,5]$. The central row shows the next layer, where the knot $4\frac{1}{2}$ has been inserted. The bottom row shows the third layer, where the knot $5\frac{1}{2}$ has been inserted. Each control point is marked as v (*visible*), r (*replaced*) or h (*hidden*). Thick connecting lines show parent-child relationships. Thin connecting lines show other relationships. Note that every point has a child (except in the lowest layer) but not every point has a parent. Weights on lines show how each *hidden* point is calculated from points in the layer above (and also how each of the other points was originally calculated from points in the layer above when the layer was created).

Each control point, in each layer, has either one or two weighted links to control points in the layer below. One of those links will be to a point that has the same (s,t) co-ordinate. We say there is a *parent–child* relationship between points of the same (s,t) co-ordinates in adjacent layers. For a given point, the one or two weights from the layer above, which always sum to one, provide the mechanism by which a *hidden* point's location is calculated from the location of the points in the layer above.

From the set of layers we can create the mesh that is presented to the user in the user-interface. It comprises all of the *visible* control points from all the layers, linked together with appropriate edges.

The assumed starting point is a single layer, layer 0, with all control points marked as *visible*. Because there is only a single layer at the start, the initial user-interface mesh is identical to the initial starting mesh.

We now define an algorithm (illustrated in Figures 6 and 5) which allows us to insert a single new control point on an edge

[1] The algorithm could be modified to support even degree B-splines, but odd degrees are sufficient to demonstrate the potential and the limitations of the proposed method.

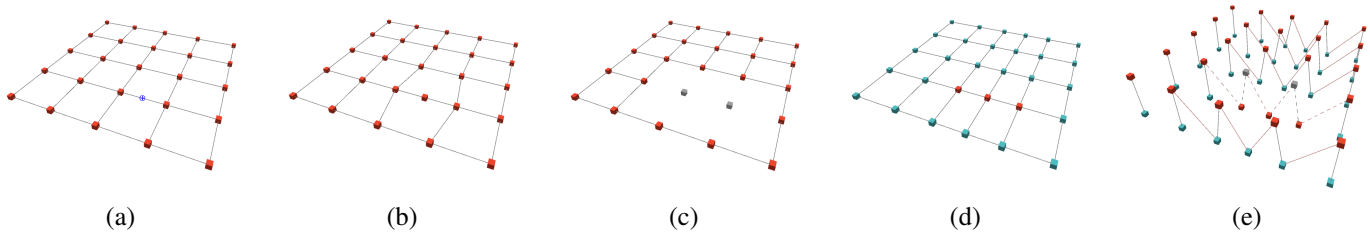

Figure 5: Screen-shots from our experimental system. (a) Selection of a point on an edge, at which a new control point is desired. (b) The user-interface view after insertion of the new control point. Notice that the points either side are moved as expected for an insertion in a cubic B-spline. (c) and (d) Layers 0 and 1 in the internal representation. (c) Layer 0 has two control points that are *replaced* (grey) and no longer used for manipulation. (d) Layer 1 has three control points that are *visible* (red) with the rest of the points being *hidden* (cyan) and calculated from points in Layer 0. (e) Explicit representation of the links between the two layers. Each *hidden* point in Layer 1 has a connection to one or two *visible* points in Layer 0. Note also that this view shows the connections that allowed calculation of the initial positions of the three *visible* points in Layer 1.

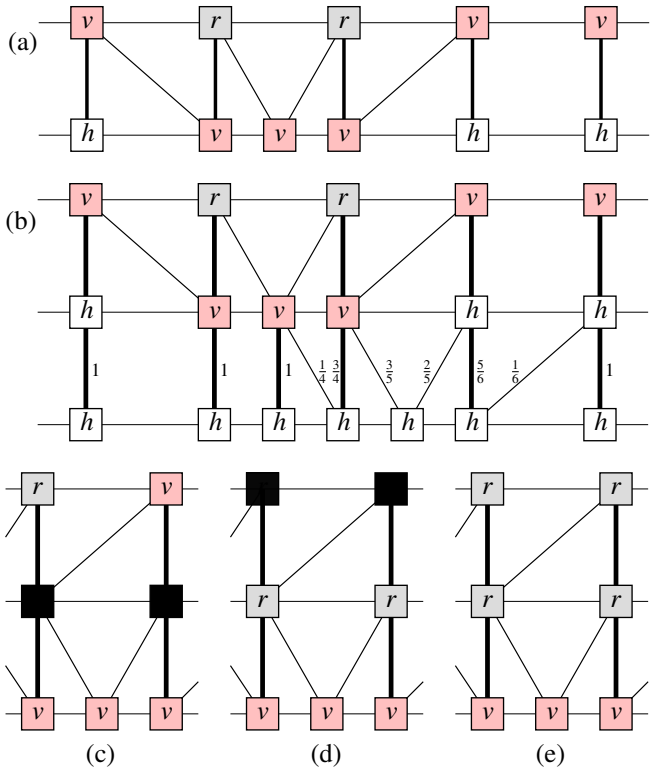

Figure 6: An example insertion. (a) Initial state. (b) Create a new layer (Step 3). (c) Mark the relevant points as *visible* (Step 4) and their parents as *to be replaced* (Step 5), shown here as a solid black square. (d) Recurse up to the next level (Steps 6, 5). (e) Final result.

in the mesh. Each point has a status, which may be *visible*, *hidden* or *replaced* and each point has a flag which allows it to be marked as *to be replaced* when necessary.

6.1. Algorithm for the insertion of a new control point

1. (Figure 5(a)) User requests that a new point be inserted on a specific edge in the mesh of visible points. The new knot value is halfway between the knot values associated with the control points at either end of that edge.
2. If there is already a layer that introduces that particular knot value, then the $k-1$ appropriate points in that layer are made *visible* and go to step 5
3. (Figures 5(e), 6(b)) Otherwise, a new layer is created with the newly-generated knot value. Links are formed to the control points in the previous layer, with weights for computing the control points in the new layer from the points in the previous layer (Equation 2). All points in the new layer have their geometric positions initialised by calculation from the points in the layer above. Mark as *hidden* all points in the new layer.
4. (Figures 5(d), 6(c)) Mark as *visible*, in the new layer, the point created on the specified edge and $(k-2)/2$ points either side of the new point. That is, mark as *visible* the $k-1$ points on that single row (or column) that are calculated from more than one point in the layer above.
5. (Figure 6(c–d)) For each point that is newly marked as *visible*, mark its parent as *to be replaced*.
6. (Figure 6(d–e), see Appendix A for detail) For each point that is marked as *to be replaced*, first check its status: if it is *hidden* then mark it as *replaced* and mark its parent as *to be replaced* and recurse on step 6; if it is *visible*, then it becomes *replaced* and its parent is unaffected; if it is already *replaced* then do nothing. Then look at the points that it contributes to. One will be its child, which will already have been dealt with. The other, if it exists, needs to be checked: if it is *hidden* then mark it *visible* and go to step 5; if it is *visible* or *replaced* then nothing needs to be done.
7. (Figure 5(b) and Section 6.2) Build a new control mesh for the user-interface based on all points that are marked as visible.

6.2. Algorithm for generating the user-interface view

The visible control mesh (VCM) for the user-interface is created by processing every visible point in the mesh and determining its connectivity to other visible points. Points in the VCM may be linked to points in other layers. Each control point has a permanent link to the control point in the layer above that it replaces, its "parent", if such a point exists. Each control point also has a permanent sets of links to its (up to four) neighbours in its own layer. These links are set when the layer is created and are never changed. In addition to these, each control point has a mutable set of links to its (up to four) neighbours in

the VCM. The VCM links must be regenerated (Step 7 in Section 6.1) whenever a new control point is inserted to the visible mesh.

1. Reset all VCM links to all be `null`.
2. For each layer, starting with the finest and working to the coarsest:
   ▹ For each *visible* control point, `p`, in the layer:
   ▹▹ For each of the four directions, d, of connectivity:
   ▹▹▹ If the VCM link from `p` in direction d is `null`, then set `q` to be the next point in that direction in that layer. While `q` is not *visible* and not `null`, set `q` to be its own parent. [The result is that `q` will be the first (and, if it exists, the only) *visible* point in the stack of parents of the point in direction d; if there is no *visible* point in that stack or there was no next point in direction d, then `q` is `null`.] Set `p`'s VCM link in direction d to point to `q`. If `q` is not `null` then set `q`'s VCM link in the opposite direction to point to `p`. [This makes a bi-directional link between the two points and means that points in coarser layers get the correct VCM links to points in finer layers.]

Drawing the VCM is straightforward: all points in the VCM are connected in a single graph, therefore it is only necessary to start with a single visible control point and draw it, and recursively draw the (up to) four points to which it is connected and the lines connecting them. Doing this naïvely would draw each point many times so a simple boolean value in each control point's data structure can be used to ensure that each point is drawn only once.

7. The generating system

Each layer, l, in the data structure comprises a complete tensor-product B-spline basis, V^l, which consists of a set of basis functions, N_i^l. The final surface is that produced by the lowest layer, using Equation 3. In addition to the bases for each layer, which are straightforward NURBS bases, it is necessary to consider the *generating system* associated with the VCM.

Each *visible* control point in the VCM has an associated *blending function*. These combine to make the generating system for the VCM. The generating system associated with the *visible* control points is most usefully compared with Giannelli et al.'s Truncated Hierarchical B-spline basis [6]. Indeed, it can be considered a variation on THB-splines.

Every blending function in our system is a weighted sum of NURBS basis functions from the lowest level. This is identical to the situation with THB-splines. However, THB-splines are presented in a different way by Giannelli et al.: they present them as basis functions at a higher level being truncated by subtracting basis functions from a lower level. The two views are equivalent mathematically, but, in contrast to THB-splines, our system is not based on a hierarchy of nested domains which govern refinement. Instead, our user interface is focused on handling refinement via (*visible*) control points.

We now demonstrate how we can represent the blending functions associated with the set of *visible* control points in a manner similar to that used for THB-splines.

Consider a rewriting of the bivariate B-spline definition, Equation 3, to remove unnecessary subscripts:

$$\mathbf{P}^l(s,t) = \sum_i N_i^l(s,t)\mathbf{P}_i^l,$$

where l is the level in the data structure and i ranges over all of the points in that level. If we remove explicit reference to the parameter space, (s,t), we can see clearly the relationship between one level and the next in the data structure:

$$\sum_i N_i^{l-1}\mathbf{P}_i^{l-1} = \sum_i N_i^l\mathbf{P}_i^l.$$

The way in which control point locations are calculated, Equation 2, can be stated in a single equation:

$$\mathbf{P}_i^l = (1-\alpha_i^l)\mathbf{P}_{i-1}^{l-1} + \alpha_i^l\mathbf{P}_i^{l-1}$$

where we have assumed that the index i indexes the rows and columns of the two-dimensional grid in an appropriate way, that is, it runs along each row (or column) in which a new knot is inserted before moving to the next. This allows us to show how NURBS basis functions, N_i^l, in one layer relate to basis functions, N_i^{l-1}, in the previous layer:

$$\begin{aligned}
\sum_i N_i^{l-1}\mathbf{P}_i^{l-1} &= \sum_i N_i^l\mathbf{P}_i^l \\
&= \sum_i N_i^l\Big((1-\alpha_i^l)\mathbf{P}_{i-1}^{l-1} + \alpha_i^l\mathbf{P}_i^{l-1}\Big) \\
&= \sum_i \Big(\alpha_i^l N_i^l + (1-\alpha_{i+1}^l)N_{i+1}^l\Big)\mathbf{P}_i^{l-1} \\
\Rightarrow N_i^{l-1} &= \alpha_i^l N_i^l + (1-\alpha_{i+1}^l)N_{i+1}^l.
\end{aligned}$$

We are now ready to demonstrate how to construct a generating system for the set of *visible* control points.

Each *hidden* or *visible* control point has a NURBS basis function, N_i^l, and a possibly-truncated blending function, T_i^l. For the bottom layer, V^n:

$$T_i^n = N_i^n.$$

Now, let $N_i^l \in V^l$ be a basis function associated with a *hidden* or *visible* control point:

$$N_i^l = \alpha_i^{l+1}N_i^{l+1} + (1-\alpha_{i+1}^{l+1})N_{i+1}^{l+1}. \tag{4}$$

The truncated versions depend on the status of the points $\mathbf{P}_i^{l+1}$ and $\mathbf{P}_{i+1}^{l+1}$ as follows:

$$\begin{aligned}
\text{both } \mathbf{P}_i^{l+1}, \mathbf{P}_{i+1}^{l+1} \textit{ hidden} \quad & T_i^l = \alpha_i^{l+1}T_i^{l+1} + (1-\alpha_{i+1}^{l+1})T_{i+1}^{l+1} \\
\text{only } \mathbf{P}_i^{l+1} \textit{ hidden} \quad & T_i^l = \alpha_i^{l+1}T_i^{l+1} \\
\text{only } \mathbf{P}_{i+1}^{l+1} \textit{ hidden} \quad & T_i^l = (1-\alpha_{i+1}^{l+1})T_{i+1}^{l+1}.
\end{aligned} \tag{5}$$

The first option corresponds to no truncation, the coarser blending function at level l is a weighted sum of two finer blending functions of level $l+1$ as in standard B-spline knot insertion. Basically, the newly created function 'does not see' the newly inserted knot.

The last two options correspond to truncation: the finer blending function from level $l+1$ is passed directly up the data

structure and 'replaces' the original coarser blending function of level l. In this case, the newly inserted knot 'remains visible' for the function.

Consequently, the T functions are either B-splines or combinations thereof from different levels, i.e., they are truncated B-splines.

The overall generating system, based solely on *visible* control points, creates the final surface:

$$\mathbf{P}(s,t) = \sum_{\mathbf{P}_i \in visible} T_i(s,t)\mathbf{P}_i.$$

To ensure partition of unity, the blending functions of the set of *visible* control points must truly encompass the set of NURBS basis functions of the lowest layer of the data structure. That is, every N_i^n must be incorporated into the T of *visible* points, with the contribution of each N_i^n summing to unity. Details of this, and of how we can check linear independence of the generating system, are in Appendix B.

Those familiar with the *blossom* (*polar forms*) formulation will be able to see an alternative, more compact, way of representing this mechanism in the one-dimensional case (e.g., Figure 4). However, the fact that the corresponding blending functions in our construction are not, in general, minimally supported B-splines but rather linear combinations thereof from different levels precludes the use of blossoms in the two-dimensional case. The blossom notation could be used for control points but not for evaluating the spline itself.

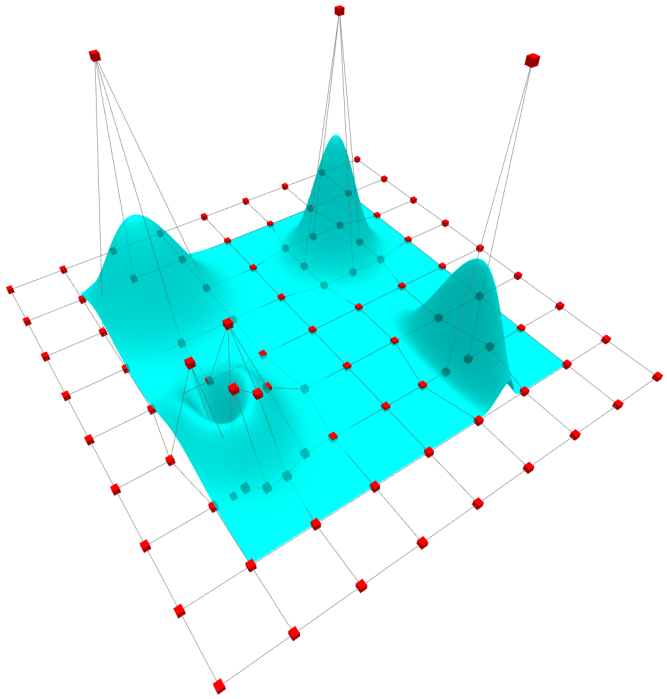

Figure 7: Four example features on a plane that is initialised with uniformly-space control points in a grid. Back left: a single original point raised above the zero plane. Back right: 2×2 original points are refined to 3×3 points, with the central point then raised to show a sharper peak than the one at back left. Front right: a single edge is refined three times and the central point then raised to show a peak that is narrow in the s direction but of the original width (i.e., similar to that at back left) in the t direction. Front left: a range of refinements to make a U-shaped ridge covering the same area as used by the single peak at back left.

8. Discussion

Figure 7 shows some simple examples in our experimental user interface. The method allows for the insertion of new control points on any visible edge. Once a point has been inserted it gives finer control of the local shape of the surface than would be possible with the original control points. The algorithm can be adapted to work for any method definable by knot insertion, including subdivision and NURBS of even degree.

Four optional features are worth discussing in more detail:

Simultaneous introduction of many new control points. Figure 2 shows the introduction of a row of control points (Figure 2(c)) and of a block of control points (Figure 2(e) and (f)). In these cases, the newly-inserted knot(s) suffice for all new points and so one or two hidden layers suffice to introduce many new control points. The user-interface can be designed to allow the user to specify insertion on a single edge (Figure 2(d)), or on several parallel edges (Figure 2(c)), or in a block (Figure 2(e) and (f)).

Insertion at arbitrary knot values. As written, the algorithm follows Gordon and Riesenfeld's original suggestion for B-spline refinement [16, §13], which is to place the new knot "midway (parametrically) between two of the previously existing knots." The advantage of this is that introducing a new control point between two knot lines where there has already been introduced a control point elsewhere in the mesh leads simply to a hidden control point being made visible and means that we do not have to create a new hidden layer nor do a new knot insertion. However, there is nothing in the method that prevents a knot being introduced at any value.

Higher degrees. Our implementation works for NURBS of arbitrary odd degree. However, the limitations of the method (see below) become increasingly obvious for higher degrees, because insertion of a single new control point has a wider influence across the mesh as degree increases.

Multiple NURBS patches. The algorithm can be applied to compositions of multiple NURBS patches. First, we need to ensure that adjacent patches are compatible, i.e., they are of the same degree across their shared edge and they share the knot vector along the edge. This can be always ensured by standard algorithms such as degree raising and knot insertion. Assuming that the two adjacent patches are compatible in this sense, there are still subtleties in what changes need to be propagated across their shared boundary. Naïvely one would expect that any knot insertion needs to propagate into the patches either side of the existing patch. However, because the patch does not change its geometry after knot insertion, this only needs to happen if there is a change in the visible control points on the actual edge between two patches.

An alternative is to merge two or more (compatible) NURBS patches into one NURBS surface, which can then be treated without any further modifications. However, this approach is

available only if the patches form a logical rectangular array when combined.

8.1. Limitations

Despite working, the method suffers from several limitations. The commonly occurring limitations are minor inconveniences, but the more rarely occurring ones could be considered a serious drawback for the user, because they cause a large number of unexpected (and therefore unwanted) control points to be generated from a single insertion. We discuss the limitations from the most commonly occurring, and least problematic, to the least commonly occurring.

New points may not appear exactly where the user expects. Figure A.14 gives a clear example of this. The user requests a new point on an edge that she sees as sloping slightly up from left to right (Figure A.14(a)). What she gets (Figure A.14(b) and (f)) is a new point on a horizontal edge slightly displaced from the horizontal line through the original (red) points. However, users of any system get used to its quirks. For example, normal cubic knot insertion causes two existing control points to apparently jump slightly away from the newly-introduced point (e.g., Figures 3 and 5(b)). Mathematically this is the right behaviour, but a novice user still needs to get used to it. Likewise, slight jumps in control point position are likely to be quickly accepted as a feature of the system, especially as the surface itself does not change when a new control point is inserted.

Unexpected control points may appear. In Figure A.14, the replacement of point ρ by point γ requires that we make point σ visible, leading to Figure A.14(f). This behaviour might surprise a user, though in this case a user might easily understand that the extra point appears because the newly-inserted point is adjacent to a previously-inserted point. More surprising would be such behaviour if it is caused by control points inserted on the far side of the mesh.

Lack of symmetry. Because previously-inserted points can affect later insertions there is a lack of symmetry in operations. Inserting two points in different orders can have different results. This is the case in point in Figure A.14(f), where the two points requested by the user are symmetric, but the result is not symmetric with respect to those two points. One result of the lack of symmetry is the possibility that the new control points will control different blending functions depending on the order in which the user inserts them. This might be considered problematic but recall that this is a user-orientated solution: the user requests the insertion of new points in a particular order, and adjusts the new points as they see appropriate. If they then insert a new point, it is with the full understanding of all previous insertions and they have to accept that the earlier insertions may affect the later ones.

Unusual-looking configurations may occur. There are pathological sequences of insertions that lead to corners (Figure 8(a)) and peninsulas (Figure 8(b)). Similar features appeared in the early versions of T-splines[2] but were omitted from the published version [10]. These unusual configurations are not errors; they are connected correctly in the user-interface grid, as shown in more detail in Appendix C. If they are deemed to be undesirable, they can be easily identified from the connectivity of the VCM and the unusual-looking connectivity "fixed" by forcing adjacenct control points in the same layer to become visible, creating a $\top$ or + connectivity. The disadvantage of this "fix" is that even more unexpected control points will become visible.

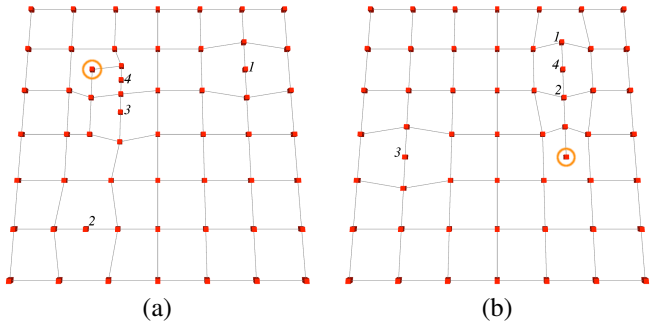

Figure 8: Unusual configurations (circled) that occur with particular sequences (labelled *1, 2, 3, 4*) of four insertions. (a) A two-connected corner. (b) A one-connected peninsula.

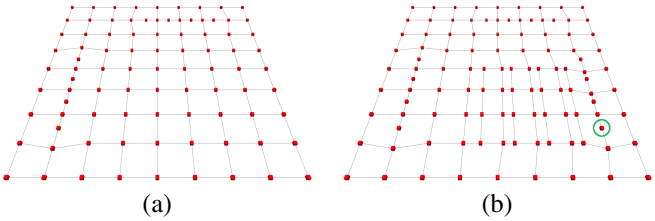

Figure 9: (a) Eight new control points have been introduced, the first four along the top of the grid and the others then at the left of the grid. (b) Inserting a single new point at right (circled in green) causes a cascade of 18 other new points to appear, both horizontally and vertically.

Cascades of unexpected control points may occur. In other pathological cases, previously inserted knots can cause a cascade of new control points to appear. That is, local insertion of a new control point has the effect of introducing a set of points across the mesh. Figure 9 shows such a pathological case. The original T-spline method [10] also suffered from unwanted cascade across the mesh [11, §4.3][12, §3.2.5], but more recent work [17] has demonstrated how T-splines can be refined without excessive propagation of control points. Our simpler mechanism does not admit such an elegant solution. As a user-centric method, the appropriate way to ameliorate the effect of any cascade is to make the user aware of the consequences of any insertion. A straightforward way to do this is, when the user hovers their cursor over a particular edge, to highlight all edges that would be affected if the user chose to refine that particular edge. A user who is aiming for a local refinement can then investigate which edge should be selected to introduce the best set of new control points.

[2] Tom Sederberg and Tom Hughes both independently confirmed that similar features were in the earliest versions of T-splines.

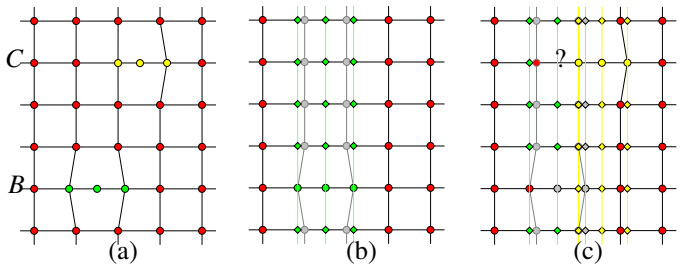

Figure A.10: (a) The user-interface presentation of a tensor-product NURBS patch (red) with two extra control points inserted, one on row B (green) and one on row C (yellow). In this cubic example, each newly-introduced point causes two other, existing, points to be replaced. Note: all figures in the Appendix show geometric space, not parameter space, with the red control points in a square grid before the insertions. (b) The layer corresponding to the point introduced on row B. Every row other than B requires three new points (shown in green) to be calculated from existing points. The replaced points are shown in grey. (c) The next layer down, corresponding to the point introduced on row C. There is a problem on row C.

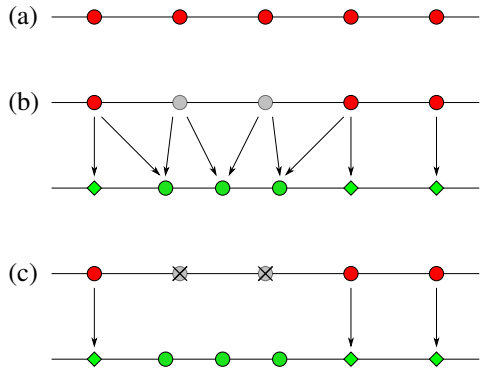

Figure A.11: (a) Row B of Figure A.10 before insertion of the new point. (b) The new point is inserted. Its position and the positions of the new points either side are calculated from the existing points (Equation 2). (c) The end result is that two *replaced* points are removed from the topmost layer and three points are made *visible* to the user from the next layer. All other points in that layer are calculated trivially from the corresponding point in the top layer.

9. Conclusion

We have shown that it is possible to provide the user with local refinement of the control mesh solely by modifying the user-interface. All that is required is a data structure for storing the layers and the straightforward algorithms for adding a new layer (Section 6.1) and for generating the user-interface view (Section 6.2). The limitations that we have highlighted (Section 8.1) mean that we do not expect our user-centric approach to compete with the more powerful mechanisms offered by T-splines, LRB-splines, and HB-splines. We have, however, shown that is is possible to use the standard NURBS implementation to provide locally refined behaviour to the user.

Acknowledgements

Kosinka was supported by the Engineering and Physical Sciences Research Council [EP/H030115/1]. We acknowledge Tao Yang, whose implemented our first test bed during his Masters project (2011–12), which provided insights into some of the potential solutions discussed in Appendix A. For useful discussions, we gratefully acknowledge Malcolm Sabin, Tom Hughes, and Oliver Deussen. We thank the organisers of the SMART 2014 workshop for allowing us to present the talk on which this paper is based.

Appendix A. Challenging cases when inserting new points

Challenging cases arise when the user requests a new *visible* point that is close to an existing *hidden* point. There are challenges for new points inserted on a row *parallel* to an existing insertion and challenges for new points inserted on a column *perpendicular* to an existing insertion on a row.

Appendix A.1. Challenges in parallel insertion

The first set of challenges that we consider is where we introduce new points on two *parallel* knot lines. Figure A.10 shows the context. In Figure A.10(a), we see the situation as presented in the user-interface. Two new control points are introduced on two parallel knot lines, B and C, in each case causing the two adjacent control points to be replaced by new points, in the same manner as happens in the univariate case. The knot lines, B and C, could be adjacent or separated by a small number of other knot lines (two in this example) or on completely opposite sides of the patch. In any case, the challenge is caused by the overlap between the newly-introduced *visible* points in the bottom layer, (c), and the previously-introduced *hidden* points in the previous layer, (b). Figure A.10(b) shows the introduction of new (green) points in the upper layer, related to the control point on row B. Figure A.10(c) shows the introduction of new (yellow) points in the lower layer, related to the control point on row C. Everything works well on all rows other than row C.

To get some intuition into the challenge, first consider row B (Figure A.11). The insertion of a new control point in this row is straightforward. The position of the new point and the points either side are calculated using the new knot value, the existing knot vector, and the positions of the control points. The three new control points are then made visible to the user in the user-interface, with the two *replaced* control points removed from the user-interface.

Consider now the subsequent insertion of a new control point on row C (Figure A.12). The insertion of this new layer (Figure A.12(b)) proceeds by direct analogy to the previous insertion (Figure A.11(b)). However, the removal of the *replaced* points from the user-interface (Figure A.12(c)) causes a problem: one of the points in the first hidden layer depends on one of the *replaced* points (ρ, dashed arrow) and that point is no longer available to the user. This insertion is therefore invalid, because the user has lost control of part of the generating system. This problem will arise in any situation where there is overlap of the extent of the newly-introduced points in the new

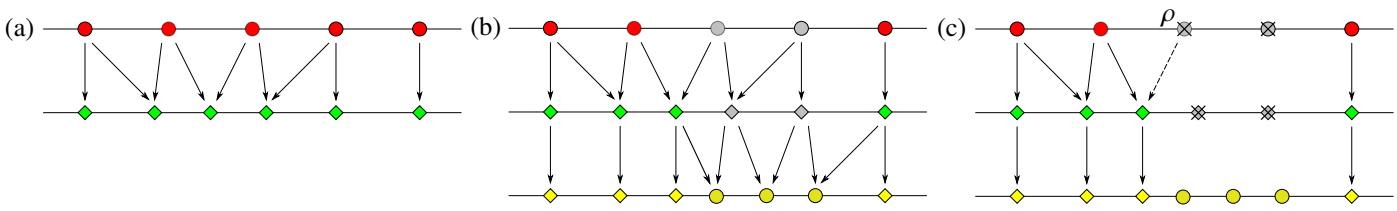

Figure A.12: (a) Row C of Figure A.10 before insertion of the new point. The top layer (red) is the original mesh and the layer below (green) is created by the introduction of a new control point on row B. On row C all points in the top layer are *visible* and all points in the layer below are *hidden*. (b) Introducing a new control point on row C creates a new layer (yellow). The position of the newly inserted point and the points either side are calculated from the existing points in the first hidden layer (Equation 2). (c) The end result is that two *replaced* points are removed from both the top layer and the middle layer and that three points are made *visible* from the bottom layer. All other points in the bottom layer are calculated from the corresponding point in the middle layer. The problem with this construction is that one of the *replaced* points, ρ, is needed to generate a point in the middle layer (dashed arrow) but that point ρ is no longer visible for the user to manipulate.

bottom layer with the extent of introduced points in any one of the layers above and it will happen even if the newly-inserted point is on the far side of the mesh, because the existing inserted knot spans the entire mesh.

Appendix A.2. Potential solutions in parallel insertion

We now consider ways to avoid the problem of hidden layers interfering in the way described above. We describe these for the parallel case here. In Appendix A.3, we consider the added challenges of insertion on knot lines that are perpendicular. There are three potential solutions for the parallel case, shown in Figure A.13.

Appendix A.2.1. Exchanging insertion order

Figure A.13(a) shows the hidden layer for row C being introduced before that for row B. This avoids the problem. Only row C needs to have the order swapped. This works for arbitrarily many rows and arbitrarily many insertions on any given row, provided all the insertions for each row are done for that row first. On each row, the hidden insertions related to all other rows can be done in any order subsequently. However, this neat solution does not generalise well to insertions on perpendicular knot lines (Appendix A.3).

Appendix A.2.2. Revealing hidden points

In this solution, we make visible any *hidden* point that depends on a *replaced* point. Figure A.13(b) shows the effect of making visible the affected hidden point, ω, in the first hidden layer. This is the point that was previously relying on a replaced control point. By making such points visible to the user, the problem is resolved. However, the means that introducing a new point on row C causes an extra control point to appear on that row. In general, the appearance of unexpected new points may be a mysterious, and therefore undesirable, side-effect for the user, especially if row C is distant from row B. However, this solution does generalise well to the case of insertions on perpendicular knot lines (Appendix A.3).

Appendix A.2.3. Not allowing replaced *control points*

A superficially-attractive solution is not to replace existing control points. All original control points stay in the user-visible mesh but each now controls a reduced blending function, with some of their original blending functions being controlled by the single newly-introduced control point. This is reminiscent of THB-splines [6], but the absence of nested domains means that it has drawbacks in our system. Figure A.13(c) shows how this plays out in our layer system. The original points all stay visible to the user, generating hidden control points as needed. The single newly-introduced control point, ν, is calculated as usual from Equation (2) and then made visible to the user for subsequent manipulation. This solution certainly works in the univariate case because in that case there are no problematic intervening hidden layers. However, the layer diagram in Figure A.13(c) reveals a problem in the bivariate case: the original control point labelled α, to the right of the new control point, ν, contributes to a point, β, in the first hidden layer, which then contributes to a point, γ, in the second hidden layer that is to the left of the new control point. Thus α, which should only influence the mesh to the right of ν, has influence to the left of ν and will continue to do so no matter how many further points are introduced between δ and α.

For the parallel case, we thus have two satisfactory insertion methods, exchanging layers (Appendix A.2.1) and revealing hidden control points (Appendix A.2.2), and two unsatisfactory methods, the original idea (Figure A.12) and the "no replacement" idea (Appendix A.2.3). Now we consider whether the two satisfactory methods work when points are inserted on perpendicular knot lines.

Appendix A.3. Challenges in perpendicular insertion

Figure 2 shows examples of insertion along perpendicular edges in which the method, as described, works without introducing any unexpected *visible* control points. Figure A.14, by contrast, shows the most challenging of the perpendicular insertion cases: insertion along adjacent perpendicular edges.

The challenge is that, again, a *replaced* point, ρ, is required for calculation of other, *hidden*, points. Let us consider our two satisfactory approaches from the parallel case.

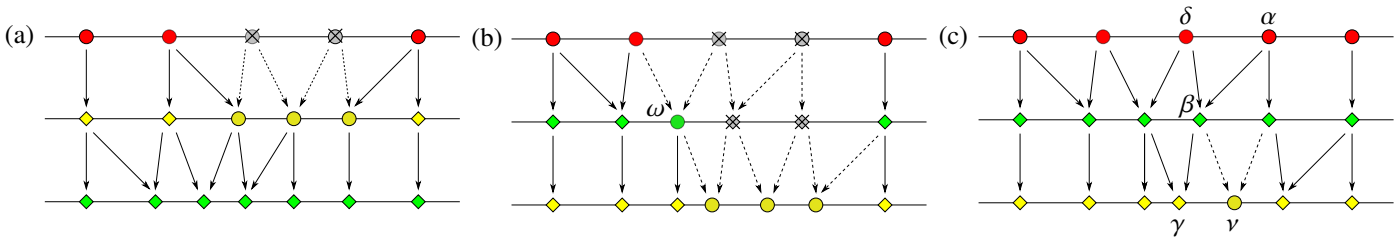

Figure A.13: Three potential solutions to the problem shown in Figure A.12(c). (a) Exchange the order in which hidden layers are introduced. (b) Reveal any problematic hidden points to the user. (c) Do not allow any original control points to be *replaced*.

While the method of exchanging layers would work in some perpendicular insertion configurations, it will not work in the most challenging case shown in Figure A.14. This is firstly because this case is perfectly symmetric. If we exchange the green and yellow insertions, then we get exactly the same problem. Secondly it is because the yellow insertion would need the context of red points that have already been *replaced*. The vertical offset of the new yellow points in Figure A.14(b) shows that the second insertion is affected by the first; they cannot be made independent and so they cannot be exchanged in order. Indeed, if one were to try to swap the order, one would find that two red control points required to do the yellow knot insertion had been *replaced* by green control points. If the user had not moved any of the green control points before requesting the new yellow point (arrow in Figure A.14(a)) then it would be plausible to exchange the layers but if the user has moved any of those three green points, any exchange of these layers becomes invalid.

The second satisfactory approach in the parallel case *does* generalise to the perpendicular case. This is where we reveal any hidden points that depend on *replaced* points. In the case of Figure A.14, the solution is straightforward: we make point σ visible to the user. This is the solution that maintains integrity and is incorporated in the algorithm (Section 6.1).

Appendix B. Features of the generating system

We consider how the algorithm ensures that the partition of unity of the initial generating system (which is, by definition, a NURBS basis) is maintained when a new layer is introduced, and we consider how we can check whether the generating system comprises linearly independent blending functions.

To ensure partition of unity, we must be sure that the set of generating functions of *visible* control points encompasses the set of NURBS basis functions of the lowest layer of the data structure. For this to be true, every N_i^n must be incorporated into the T of visible points, with the contribution of each N_i^n summing to unity. The iterative definition in Equation 5 shows that the T blending functions are passed up the layers until they hit a *visible* point, where they stop. Therefore, every *visible* point must have at least one *hidden* point as a child, that is, at least one of the three cases in Equation 5 holds for each *visible* point. Furthermore, no *replaced* point can have a *hidden* point as a child, otherwise part of a N_i^n basis function will not be included in the T blending function of any *visible* point. Finally, there may be no *hidden* points in the top layer, again because otherwise part of a N_i^n basis function will not be included in the blending function of any *visible* point.

Therefore, the highest level layer, V^0, must consist only of *visible* and *replaced* control points; there can be no *hidden* points in the top layer because such points require points above them that control them. The lowest level layer, V^n, must consist only of *visible* and *hidden* control points; there can be no *replaced* points in the bottom layer, because such points must be replaced by points in lower layers and no such points exist.

When the data structure is initialised, it comprises a single layer consisting entirely of *visible* points, meeting the conditions on both the topmost and bottommost layers. Its generating system is a standard tensor-product B-spline basis. Because of this, the blending functions attached to the initial set of *visible* points form a valid, linearly-independent, partition of unity.

To maintain a partition of unity, we must demonstrate that the algorithm in Section 6.1 ensures that, given a valid partition of unity amongst the *visible* control points, the result is a new valid partition of unity. The calculation of the α values using the standard method (Equation 3) ensures that the blending functions will be split correctly to maintain a partition of unity, so long as the algorithm ensures two things: (1) that no *visible* point has only *visible* points as its children and (2) that there is no direct connection between a *replaced* point in one layer and a *hidden* point in the next layer. In Step 3 of the algorithm a new layer is created comprising entirely *hidden* points. This does not affect the validity of the partition of unity amongst *visible* control points because it does not change the set of *visible* control points. Step 4 sets some of the points in the new layer to be *visible*, thereby invalidating the partition. Steps 5 and 6 adjust the status of other points to restore validity. The first part of Step 6 runs up the list of parents of a newly *visible* point and makes them all *replaced*, ensuring that no *visible* point has only *visible* points as its children. This ensures that all parts of a N_i^n that are allocated to a new *visible* point are removed from any higher-level points that are newly marked as *replaced*. However, that higher-level *replaced* point may have contributions from other N_i^n than those covered by the new *visible* point. Therefore, the second part of Step 6 looks at the other child of these newly *replaced* points to check whether it needs to be made *visible* in order to restore the partition of unity.

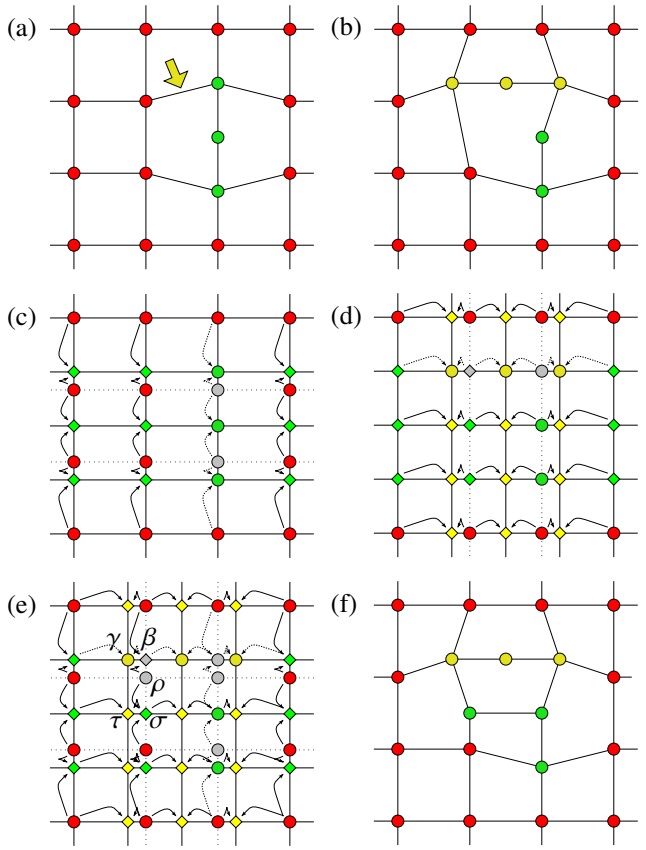

Figure A.14: The most challenging case of perpendicular insertion. (a) A point (green) has been inserted on a vertical edge. The user now wishes to insert a new point on an adjacent horizontal edge (yellow arrow). (b) Insertion could be reasonably expected to produce this configuration. Notice that the new point and the two adjacent points are offset vertically from the horizontal row of original (red) points because they lie on the hidden row of (green) points. (c) Creation of the green layer from the topmost layer. The arrows show which old points contribute to new points. The two grey points are *replaced* points. (d) Creation of the yellow layer from the green. (e) Combining (c) and (d) reveals the challenge: point γ replaces point β and the replacement propagates up the layers so that point ρ is also *replaced*. Unfortunately, point ρ is needed to create *hidden* points σ and τ. (f) A satisfactory solution is to make point σ *visible*, giving this VCM. N.B. There is an exaggeration by a factor of 1.5 in the offset of the introduced control lines from the original control lines in order to make this figure more comprehensible.

It might be thought that, given the above and the definitions in Equations 4 and 5, we should be able to apply, to our generating system, Giannelli et al.'s proofs [6] for THB-splines. This would allow us to demonstrate linear independence in addition to partition of unity. However, our construction is looser that the THB-spline construction. In particular, THB-splines rely on the concept of nested subdomains in the proof of linear independence. We do not have nested domains in the sense used by THB-splines, we have only nested spaces.

We can, however, demonstrate whether any particular configuration comprises linearly independent blending functions. We have $n+1$ levels $l = 0, \ldots, n$ and in each a set of visible control points, each with a blending function. For a particular level, l, let the set of visible control points be indexed by the index set $\mathcal{V}_l$. To prove linear independence, we need to show that:

$$\sum_{l=0}^{n} \sum_{i \in \mathcal{V}_l} c_i^l T_i^l = 0 \quad \Rightarrow \quad \forall c_i^l = 0. \tag{B.1}$$

Say we collect all the coefficients from Equation 5 and combine them to produce the same condition expressed in terms of N_i^n. Namely,

$$\sum_{l=0}^{n} \sum_{i \in \mathcal{V}_l} c_i^l T_i^l = \sum_i d_i N_i^n. \tag{B.2}$$

There exists a matrix M that computes the d_i from the c_i^l. Because the N_i^n are linearly independent, proving that M has full rank would show that the generating system forms a basis. This provides a straightforward algorithmic way of checking whether a given structure leads to a basis. However, it must be said that the proposed application of the method, *ab initio* design of surfaces, does not depend on linear independence of the generating system.

Appendix C. Pathological cases of insertion

In Section 8 two cases are presented where a sequence of four insertions leads to an interesting set of dependencies being revealed. Figure C.15 shows the layer structure for these cases, which demonstrates that, although the connectivity may look unusual in the user interface, it is correct.

References

[1] D. R. Forsey, R. H. Bartels, Hierarchical B-spline refinement, in: SIGGRAPH '88, ACM, 1988, pp. 205–212. doi:10.1145/54852.378512.

[2] G. Farin, Curves and Surfaces for CAGD: a practical guide, 5th Edition, Morgan Kaufmann, 2002.

[3] R. Kraft, Adaptive and linearly independent multilevel B-splines, in: A. Le Méhauté, C. Rabut, L. Schumaker (Eds.), Surface Fitting and Multiresolution Methods, Vanderbilt University Press, Nashville, 1997, pp. 209–218.

[4] A.-V. Vuong, C. Giannelli, B. Jüttler, B. Simeon, A hierarchical approach to adaptive local refinement in isogeometric analysis, Computer Methods in Applied Mechanics and Engineering 200 (49–52) (2011) 3554–3567. doi:10.1016/j.cma.2011.09.004.

[5] T. Hughes, J. Cottrell, Y. Bazilevs, Isogeometric analysis: CAD, finite elements, NURBS, exact geometry and mesh refinement, Computer Methods in Applied Mechanics and Engineering 194 (39-41) (2005) 4135–4195. doi:10.1016/j.cma.2004.10.008.

[6] C. Giannelli, B. Jüttler, H. Speleers, THB-splines: The truncated basis for hierarchical splines, Computer Aided Geometric Design 29 (7) (2012) 485–498. doi:10.1016/j.cagd.2012.03.025.

[7] J. Deng, F. Chen, Y. Feng, Dimensions of spline spaces over T-meshes, J. Comput. Appl. Math. 194 (2006) 267–283. doi:10.1016/j.cam.2005.07.009.

[8] J. Deng, F. Chen, X. Li, C. Hu, W. Tong, Z. Yang, Y. Feng, Polynomial splines over hierarchical T-meshes, Graphical Models 70 (4) (2008) 76–86. doi:10.1016/j.gmod.2008.03.001.

[9] T. Dokken, T. Lyche, K. F. Pettersen, Polynomial splines over locally refined box-partitions, Computer Aided Geometric Design 30 (3) (2013) 331–356. doi:10.1016/j.cagd.2012.12.005.

[10] T. W. Sederberg, J. Zheng, A. Bakenov, A. Nasri, T-splines and T-NURCCs, ACM Trans. Graph. 22 (2003) 477–484. doi:10.1145/882262.882295.

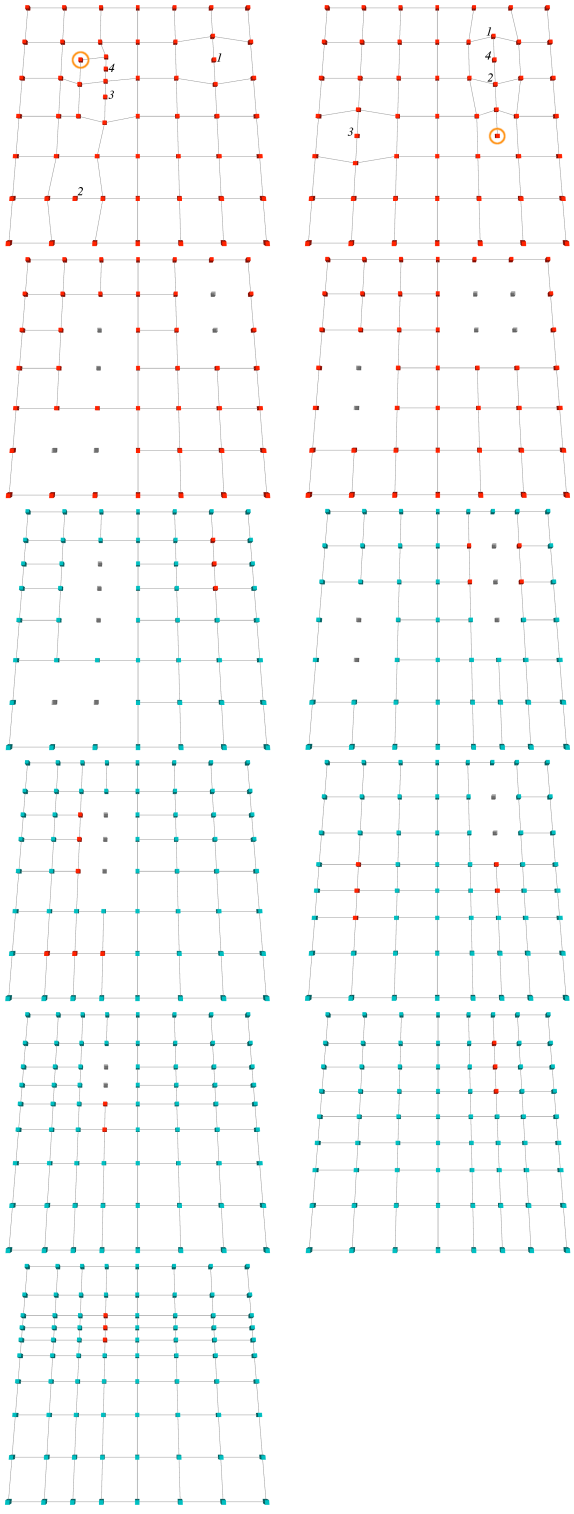

Figure C.15: Top: the user-interface view showing the sequence of point insertions (numbers) and the unusually-connected point that results (circled). Below: the layers in the system, showing *visible* control points in red and *replaced* control points in grey. This demonstrates that the unusual points are, in fact, correctly connected, with the *hidden* points (cyan) to which they are connected calculated from points in the layer above.

[11] T. W. Sederberg, D. L. Cardon, G. T. Finnigan, N. S. North, J. Zheng, T. Lyche, T-spline simplification and local refinement, ACM Trans. Graph. 23 (2004) 276–283. doi:10.1145/1015706.1015715.

[12] M. R. Dörfel, B. Jüttler, B. Simeon, Adaptive isogeometric analysis by local *h*-refinement with T-splines, Computer Methods in Applied Mechanics and Engineering 199 (5-8) (2010) 264–275. doi:http://dx.doi.org/10.1016/j.cma.2008.07.012.

[13] G. T. Finnigan, Arbitrary degree T-splines, Master's thesis, Brigham Young University, USA (2008).

[14] D. F. Rogers, J. A. Adams, Mathematical Elements for Computer Graphics, 2nd Edition, McGraw-Hill, 1990.

[15] W. Boehm, Inserting new knots into B-spline curves, Computer-Aided Design 12 (4) (1980) 199–201. doi:10.1016/0010-4485(80)90154-2.

[16] W. J. Gordon, R. F. Riesenfeld, B-spline curves and surfaces, Academic Press, New York, 1974, pp. 95–126.

[17] M. Scott, X. Li, T. Sederberg, T. Hughes, Local refinement of analysis-suitable T-splines, Computer Methods in Applied Mechanics and Engineering 213 (2012) 206–222.