In Journal of Graphics Tools, A.K. Peters, 4(4), pp. 37-50, 1999.

Incremental polygonization of deforming NURBS surfaces

Frederick W. B. Li Rynson W. H. Lau

Department of Computer Science, City University of Hong Kong, Hong Kong

{borbor, rynson}@cs.cityu.edu.hk

Abstract

Non-uniform rational B-splines (NURBS) is a powerful tool to model deformable objects. Their shapes can be easily modified by moving the control points. A common method to render these objects is by polygonization. However, the polygonization process is computationally very expensive. If the object deforms, we need to execute this process in every frame to reflect the geometric change of the object. This limitation makes real-time rendering of deforming objects very difficult. In this paper, we present an incremental method for polygonizing deforming objects modeled by NURBS surfaces. Some incremental techniques are introduced here to further improve the rendering performance of the method. They include an efficient mechanism for determining the deformation region when the surface deforms, an incremental crack prevention technique and an updating method for multiple control point movement.

1 Introduction

Deformable objects may be needed in many computer graphics applications. They are particularly useful in modeling clothes, facial expression, human and animal characters. NURBS surfaces [1, 2] are often used to model deformable objects because their shapes can be easily modified by moving the control points. A NURBS surface is a rational generalization of the tensor product non-rational B-spline surface defined as:

$$S(u,v) = \frac{\sum_{i=0}^{n}\sum_{j=0}^{m} w_{i,j} P_{i,j} R_{i,j}(u,v)}{\sum_{i=0}^{n}\sum_{j=0}^{m} w_{i,j} R_{i,j}(u,v)}$$

where $w_{i,j}$ are the weights, $P_{i,j}$ form a control net, and $R_{i,j}(u,v)$ are the basis functions.

To give the user a realistic perception and instant response, it is very important for an application to be able to render deformable objects at appropriate resolution interactively. However, this goal is difficult to achieve. Because majority of the hardware graphics accelerators can only handle polygons, polygonization becomes a standard way of rendering NURBS surfaces. There are a lot of polygonization methods proposed, such as [3, 4, 5]. Basically, these methods apply some tessellation algorithms to the defining equation of the surface to produce a polygon model for rendering. However, the polygonization process is computationally very expensive and must be executed in each frame to reflect the change of shape while the object is deforming. Real-time handling of a large number of deforming objects becomes very difficult.

In our earlier work [6], we described a technique to interactively render deforming NURBS surfaces. It is by incrementally updating a pre-generated polygon model of each deforming surface and refining its resolution without using the defining equation of the deforming surface. This paper improves on that technique in the following ways:

- an efficient incremental crack prevention method is introduced,
- a simple method for determining the deformation region (region of the surface affected by the movement of a control point) is suggested to limit the update of the polygon model to within the region only,
- a method is discussed that would reduce the update cost when more than one control point is moving simultaneously, and
- a detailed account on the implementation of the enhanced rendering method.

The rest of the paper is organized as follows. Section 2 describes our incremental polygonization method for deforming NURBS surfaces. Section 3 introduces an incremental crack prevention technique. Section 4 introduces an efficient mechanism for determining the deformation region. Section 5 investigates the update cost when multiple control points are changing simultaneously. Finally, section 6 presents some experimental results.

2 Polygonization of Deforming NURBS Surfaces

The basic idea of our method is that we maintain two data structures of the deforming surface, the surface model and a polygon model representing the surface model [6]. As the surface deforms, the polygon model is not regenerated through polygonization. Instead, it is incrementally updated to represent the deforming surface. Two techniques are fundamental to our method: *incremental polygon updating* and *resolution refinement*.

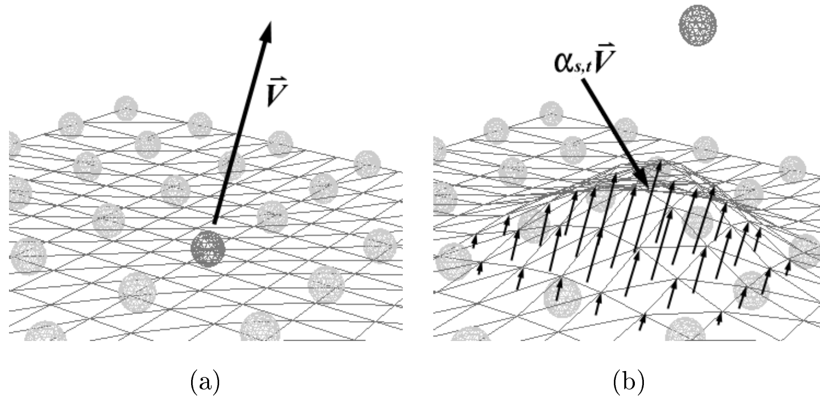
2.1 Incremental Polygon Model Updating

As the surface deforms at run-time, we incrementally update the existing polygon model to produce the deformed polygon model. To show how this technique works, we consider the polygonal representation of a surface obtained by evaluating the surface equation with some discrete parametric values. If a control point $P_{s,t}$ is moved to $\overline{P}_{s,t}$ with a displacement vector $\overrightarrow{V} = \overline{P}_{s,t} - P_{s,t}$, the incremental difference between the two polygonal representations of the surface before and after the control point movement is as follows:

$$\begin{aligned} \overline{S}(u,v) - S(u,v) &= \frac{(\overline{P}_{s,t} - P_{s,t}) w_{s,t} R_{s,t}(u,v)}{\sum_{i=0}^{n} \sum_{j=0}^{m} w_{i,j} R_{i,j}(u,v)} \\ &= \alpha_{s,t}(u,v) \overrightarrow{V} \end{aligned} \quad (1)$$

where $S(u,v)$ and $\overline{S}(u,v)$ are the polygon models of the surface before and after the control point movement, respectively. $w_{s,t}$ is the weight. $R_{s,t}(u,v)$ is the basis function, and $\alpha_{s,t}(u,v)$ is the deformation coefficient defined as follows:

$$\alpha_{s,t}(u,v) = \frac{w_{s,t} R_{s,t}(u,v)}{\sum_{i=0}^{n} \sum_{j=0}^{m} w_{i,j} R_{i,j}(u,v)} \quad (2)$$



(a) (b)

Figure 1: Incremental polygon model updating: (a) before deformation and (b) after deformation.

It is obvious that the deformation coefficient $\alpha_{s,t}(u, v)$ is constant for each particular set of (u, v) parameter values. Hence, if the resolution of the polygon model representing the surface remains unchanged before and after the deformation, we may precompute the deformation coefficients and update the polygon model incrementally by the deformation coefficients and the displacement vector as shown in Equation(1). This technique is very efficient since we need to perform only one addition and one multiplication on each affected vertex of the polygon model to produce the deformed one. In addition, its performance is independent of the complexity of the deforming surface.

In the implementation, the incremental polygon model updating is carried out in two stages, the *preprocessing* stage and the *run-time stage*. In the preprocessing stage, we tessellate the surface to obtain an initial polygon model. We also compute a set of deformation coefficient $\alpha_{s,t}(u, v)$ for each control point. As the surface deforms in run-time, the polygon model is incrementally updated with the set of deformation coefficients and the displacement vector of the moving control point. Figure 1(a) shows a surface deformation by moving a control point with displacement vector $\overrightarrow{V}$. Figure 1(b) shows the incremental updating of the affected polygon vertices.

2.2 Resolution Refinement

When a surface deforms, its curvature is also changed. If the curvature is increased or decreased by a large amount during the deformation process, the resolution of the corresponding polygon model may become too coarse or higher than necessary to represent the deformed surface, respectively. To overcome this problem, we describe a *resolution refinement* technique to refine the resolution of the polygon model and to generate the corresponding deformation coefficients incrementally according to the change in local curvature of the surface.

We first convert a NURBS surface into a set of Bézier patches using knot insertion [1]. Each Bézier patch is then subdivided into a polygon model by applying the de Casteljau subdivision formula [1] to the Bernstein polynomials in both u and v parametric directions. For example, in the u direction, we have:

$$P_i^r(u) = (1-u)\frac{w_i^{r-1}}{w_i^r}P_i^{r-1}(u) + u\frac{w_{i+1}^{r-1}}{w_i^r}P_{i+1}^{r-1}(u) \quad (3)$$

where $w_i^r(u) = (1-u)w_i^{r-1}(u) + uw_{i+1}^{r-1}(u)$ and $r = 1, \ldots, n$, $i = 0, \ldots, n-r$. $[\, w_i P_i \quad w_i \,]^{\mathrm{T}}$ are the homogeneous Bézier points with $P_i \in \mathbf{R^3}$, w_i are the weights and n is the degree of the surface. The other parametric direction has similar recursion.

If the surface deforms, the above equation becomes:

$$\overline{P}_i^r(u) = (1-u)\frac{w_i^{r-1}}{w_i^r}\overline{P}_i^{r-1}(u) + u\frac{w_{i+1}^{r-1}}{w_i^r}\overline{P}_{i+1}^{r-1}(u) \tag{4}$$

By subtracting Equation(3) from Equation(4), we obtain:

$$\overline{P}_i^r(u) - P_i^r(u) = (1-u)\frac{w_i^{r-1}}{w_i^r}(\overline{P}_i^{r-1}(u) - P_i^{r-1}(u)) + u\frac{w_{i+1}^{r-1}}{w_i^r}(\overline{P}_{i+1}^{r-1}(u) - P_{i+1}^{r-1}(u)) \tag{5}$$

We then substitute the deformation coefficients and the displacement vector into Equation(5) to obtain:

$$\alpha_i^r(u)\overrightarrow{V} = (1-u)\alpha_i^{r-1}(u)\overrightarrow{V} + u\alpha_{i+1}^{r-1}(u)\overrightarrow{V} \tag{6}$$

Finally, we have a de Casteljau formula after dividing both sides by the control point displacement vector $\overrightarrow{V}$:

$$\alpha_i^r(u) = (1-u)\alpha_i^{r-1}(u) + u\alpha_{i+1}^{r-1}(u) \tag{7}$$

for $r = 1, \ldots, n$, $i = 0, \ldots, n-r$. Equation(7) indicates that the deformation coefficients can be generated incrementally by the de Casteljau subdivision formula.

Hence, if the resolution of the polygon model needs to be increased, the new deformation coefficients can be calculated from adjacent deformation coefficients stored at existing vertices using the de Casteljau formula. The Horner's scheme may be chosen as an alternative in evaluating the de Casteljau formula to achieve a better performance. If the resolution of the polygon model needs to be decreased, we may delete some quad-tree nodes from the polygon model.

2.3 Implementation

The overall rendering method is divided into the preprocessing stage and the run-time stage. The following code summaries the operations performed during the two stages.

```
void preprocessingOperations() {
    array of bezierPatch = knotInsertion(NURBSSurface);
    for each bezierPatch[b] {
        polygonMesh[b] = deCasteljauSubdivision(bezierPatch[b]);
        for each vertex v in polygonMesh[b]
            deformCoeffSet[v] = calculateDeformCoeffSet(NURBSSurface, v);
        crackPrevention(polygonMesh[b]);
        bezierPatch[b].deformed = FALSE;
    }
}

void runtimeOperations() {
    for (;;) {
        if NURBSSurface deforms
            for each moving control point p
```

```
            for each bezierPatch[b]
                if inDeformRegion(bezierPatch[b], p) {
                    updatePolygonMesh(displacement vector of p, deformCoeffSet, polygonMesh[b]);
                    bezierPatch[b].deformed = TRUE;
                }

        for each bezierPatch[b]
            if (bezierPatch[b].deformed || viewing parameters change) {
                resolutionRefinement(polygonMesh[b]);
                crackPrevention(polygonMesh[b]);
                bezierPatch[b].deformed = FALSE;
            }
    }
}
```

During the preprocessing stage, we generate the initial polygon model of the NURBS surface and a set of deformation coefficients. During run-time, if the surface deforms, we would identify the affected Bézier patches and update the corresponding regions of the polygon model. Note that whether the surface deforms or not during a particular frame time, we may still need to perform a resolution refinement of the polygon model if some view parameters (such as the viewer's distance or orientation from the object) have changed. This is because when there is a change of orientation between the viewer and the object, for example, some part of the object may become nearer to the viewer while the rest may become farther away from the viewer. We therefore need to refine the resolution of the polygon model to optimize the rendering performance of the hardware graphics accelerator. This technique is referred to as view-dependent refinement or adaptive multi-resolution modeling [7, 8, 9, 10]. In the above code, *crackPrevention()* is used to detect and fix cracks, and *inDeformRegion()* is used to detect if a Bézier patch falls into the deformation region. We will describe these two operations in detail in sections 3 and 4.

3 Incremental Crack Prevention

Since the polygon model representing the deforming surface is adaptively subdivided, cracks may appear between neighboring polygons when they are at different resolution levels. To fix these cracks, we may employ the crack prevention methods proposed by Filip [11] or Barsky et al. [12]. These methods are simple to implement but computationally expensive. Basically, they traverse the polygon hierarchy starting from the root node. For each quad-tree node, they search the neighboring nodes and compare their subdivision levels. If the subdivision level of a neighboring node is lower than that of the current node, the mid-point and the end-points of the common edge are passed down to the child nodes. The vertices of these child nodes are then modified to fix the crack. This is done recursively until a sufficiently flat quad-tree node is reached.

To avoid identifying neighboring nodes and comparing subdivision levels for each quad-tree node during run-time, we propose an incremental crack prevention algorithm here. As the object moves near or further away from the viewer, the resolution of the polygon model may need to be adjusted to maintain the visual quality of the object. We mark the quad-tree nodes as SPLIT if they need to be subdivided, or as MERGE if their child nodes need to be merged, while we are performing the resolution refinement. Since cracks only occur in edges where the polygons at either side have changed resolution, we only need to check those nodes marked with SPLIT or MERGE. We have

identified all possible neighboring conditions of a SPLIT or MERGE node and the corresponding actions needed to be taken to fix the cracks. They are shown in Figure 2. We define two edge fixing functions, *adjustEdge()* and *restoreEdge()*. *adjustEdge()* flattens an edge by making the middle vertex of the edge collinear with its ending vertices, while *restoreEdge()* restores the middle vertex of an edge to the position before it was adjusted by *adjustEdge()*. For efficiency, a pre-adjusted vertex in $\mathbf{R^3}$ is associated with each vertex to store the location of the vertex prior to *adjustEdge()*. This pre-adjusted vertex will be updated accordingly if the object deforms. When the neighboring node is at a lower or the same subdivision level, *adjustEdge()* is called to update the corner

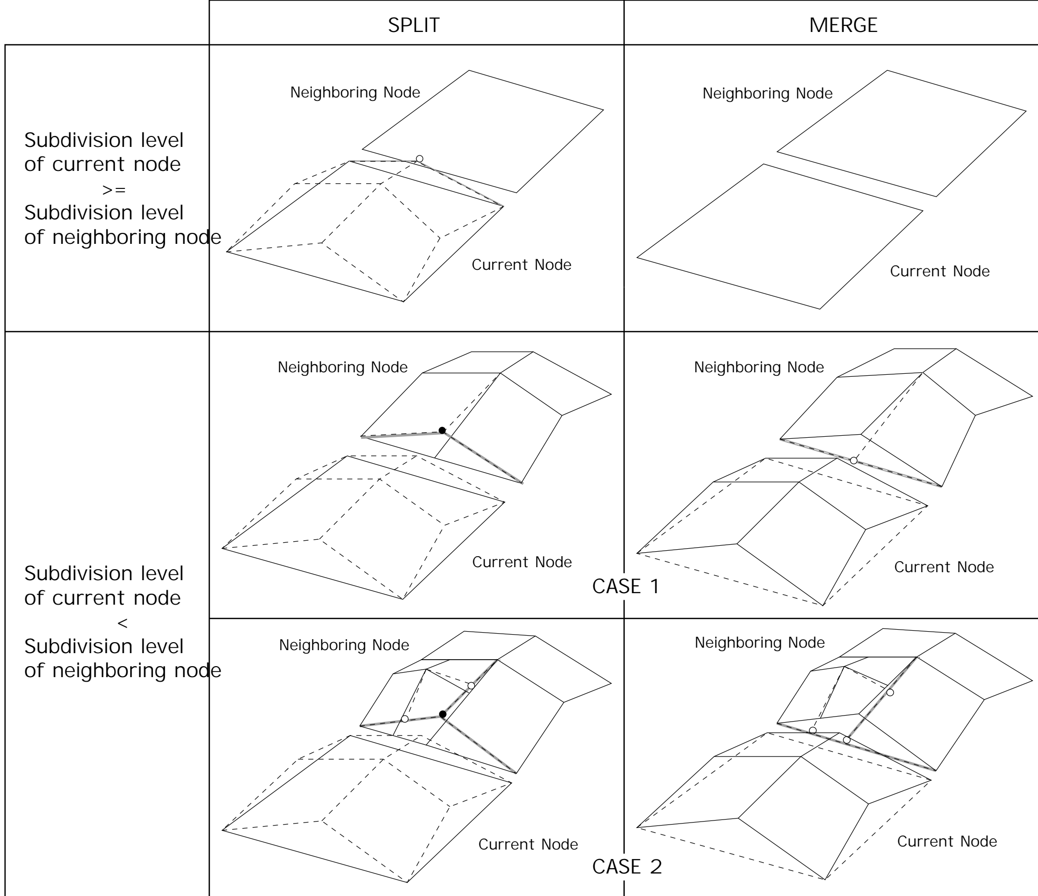


Figure 2: Crack prevention operations under different neighboring conditions.

vertices of the child nodes along the common edge by both the mid-point and end-points of that edge. When the neighboring node has a higher subdivision level, its polygon vertices must have previously been adjusted to prevent cracks. We therefore call *restoreEdge()* to restore the corner vertices of these polygons along the common edge to their original positions. The incremental crack prevention algorithm is shown as follows:

```
void incrementalCrackPrevention(quadTreeNode qnode) {
    if (qnode->isNotFlat())
        for (int node = 0; node < 4; node++)    // check its four child nodes
            incrementalCrackPrevention(qnode->child[node]);
    else
        if (qnode->isSPLIT() || qnode->isMERGE())
            for (int edge = 0; edge < 4; edge++) {    // check its four neighboring nodes
                neighborNode = searchNeighborNode(qnode, edge);
                neighborEdge = findCorrespondingEdge(neighborNode, edge);

                if (qnode->subdivisionLevel() >= neighborNode->subdivisionLevel())
                    if (qnode->isSPLIT())
                        adjustEdge(qnode, edge);
                else {
                    if (qnode->isSPLIT())
                        restoreEdge(neighborNode, neighborEdge);
                    else
                        adjustEdge(neighborNode, neighborEdge);

                    // fix common edge by recursively executing adjustEdge() on
                    // all child nodes of neighborNode along the edge
                    recursive_adjustEdge(neighborNode, commonEdge(neighborEdge));

                    // fix middle perpendicular edge by recursively executing
                    // adjustEdge() on all child nodes of neighborNode along the edge
                    recursive_adjustEdge(neighborNode, ppdEdge(neighborEdge));
                }
            }
}
```

This algorithm works well if the object does not deform. If the object deforms, the vertices of the quad-tree nodes will be updated. Hence, the modified vertices from *adjustEdge()* for crack prevention are no longer valid. In this situation, the incremental crack prevention algorithm will not work. To solve this problem, we introduce a *fast crack prevention algorithm*. We add a state variable called *edgeAdjusted* for each edge of a quad-tree node. We mark an edge as *edgeAdjusted* if it has been modified by *adjustEdge()*. To perform fast crack prevention, we traverse the polygon hierarchy and fix the edges of the quad-tree nodes marked as *edgeAdjusted* with *adjustEdge()*. This algorithm is executed prior to the incremental crack prevention algorithm. It is very efficient since it does not require the traversal of neighboring nodes and the comparison of their subdivision levels. The following code summarizes both the fast crack prevention algorithm and the whole crack prevention algorithm:

```
void fastCrackPrevention(quadTreeNode qnode) {
    if (qnode->isNotFlat()) {
        for (int edge = 0; edge < 4; edge++)    // check its four edges
            if (qnode->edgeAdjusted[edge])
                adjustEdge(qnode, edge);
        for (int node = 0; node < 4; node++)    // check its four child nodes
            fastCrackPrevention(qnode->child[node]);
    }
}

void crackPrevention(quadTreeNode qnode) {
    if NURBSSurface deforms
        fastCrackPrevention(qnode);
    incrementalCrackPrevention(qnode);
}
```

4 Deformation Region

The NURBS surface has a nice local modification property [2]. If the position of a control point $P_{i,j}$ is changed, only the shape of the surface within the parameter region $[u_i, u_{i+p+1}) \times [v_j, v_{j+q+1})$ is affected, where p and q are the degrees of the NURBS surface along u and v parameter directions, respectively. We refer to this region as the *deformation region*. Hence, if a NURBS surface deforms, the polygon model updating process only needs to be applied to this region to reduce the run-time computational cost. An intuitive way to implement this is to store the parameter range of each quad-tree node at the node. When the position of a control point is changed, we compare the parameter range of each quad-tree node with the deformation region and update all the quad-tree nodes which are inside the deformation region. However, this approach requires both extra memory to store the parameter range at each quad-tree node and CPU time to do the clipping at the affected nodes.

To improve the efficiency of the implementation, we perform the comparison on the Bézier patches generated through knot insertion. We notice that every Bézier patch has a well-defined parameter boundary. As an example, we consider a NURBS curve with knot vector $\{u_0, u_1, \cdots, u_8, u_9\} = \{0, 0, 0, 0, 1, 2, 3, 3, 3, 3\}$. The three Bézier segments generated after knot insertion will then have knot ranges $[u_0, u_4)$, $[u_4, u_5)$ and $[u_5, u_9)$. The NURBS surfaces also inherit this property. Therefore, we can store the parameter range in each Bézier patch rather than in each quad-tree node to reduce both computational and memory costs. The following code shows how we may check if a Bézier patch is within the deformation region:

```
bool inDeformRegion(bezierPatch b, controlPoint p) {
    int deformUmin = p->uIndex();
    int deformUmax = p->uIndex() + degreeU + 1;
    int deformVmin = p->vIndex();
    int deformVmax = p->vIndex() + degreeV + 1;
    bool isInDeform = TRUE;

    if ((b->knotUmax() < deformUmin) || (b->knotUmin() > deformUmax) ||
        (b->knotVmax() < deformVmin) || (b->knotVmin() > deformVmax))
        isInDeform = FALSE;

    return (isInDeform);
}
```

5 Handling of Multiple Control Point Movement

When deforming an object such as in virtual sculpting [13] or character animation, there is often more than one control point moving at the same time. With the method we have discussed so far, both the amount of memory for storing the deformation coefficients and the computational cost for updating the polygon model increase in proportional to the number of moving control points.

To solve this problem, we consider the deformation region introduced in the previous section. For a control point $P_{i,j}$ of a NURBS surface, it affects only the shape of the surface within the parameter region $[u_i, u_{i+p+1}) \times [v_j, v_{j+q+1})$, where p and q are the degrees of the surface along u and v parameter directions respectively. Consequently, the shape of a patch segment with the parameter range $[u_i, u_{i+1}) \times [v_j, v_{j+1})$ is controlled by the set of

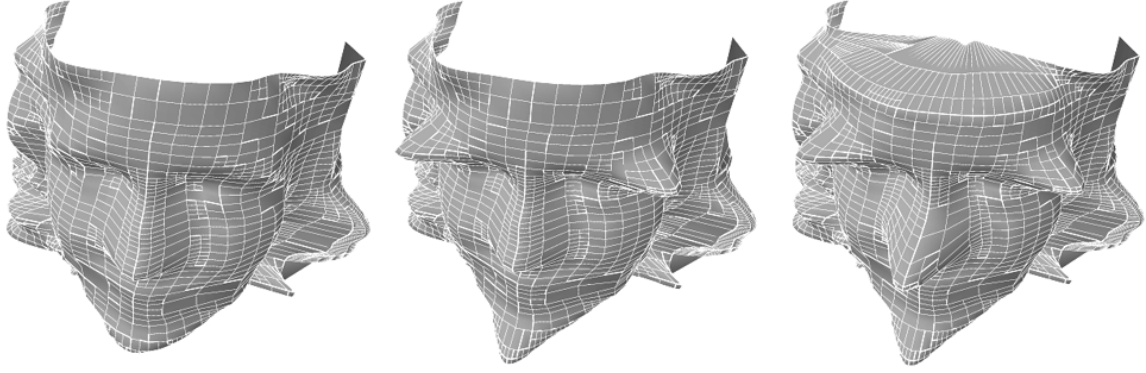


Figure 3: Model used in our experiments.

control points $P_{m,n}$ where $m = i - p, \ldots, i$ and $n = j - q, \ldots, j$. Hence, for each vertex in such a patch segment, we need to store only the set of deformation coefficient corresponding to the set of control points $P_{m,n}$. When the surface deforms, we need to perform at most $(p + 1) \times (q + 1)$ multiplications and additions to compute the final position of each vertex within this patch segment. In practice, we may obtain such patch segments by decomposing the NURBS surface using knot insertion.

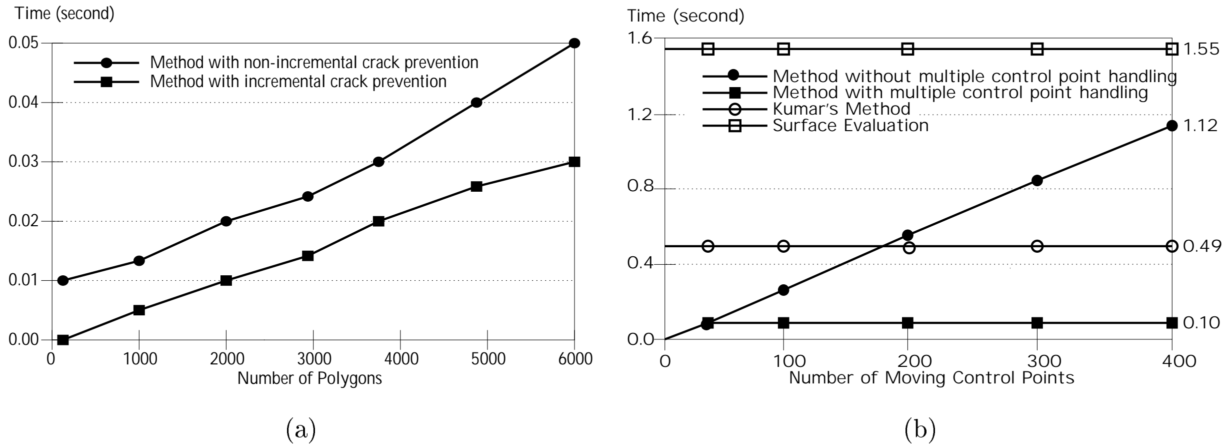
From the above discussion, we may conclude that both the computational and storage costs of multiple control point movement are bounded by the degree of the surface. Because surfaces of low degrees, say 3 or 4, are usually used to model objects, the computational cost of multiple control point movement is therefore very low.

6 Results and Discussions

We have implemented the new method in C++ with OpenGL. We tested its performance on a SGI Onyx2 with four 195MHz R10000 CPUs and an InfiniteReality^2 graphics accelerator. (Only one processor was activated during all our experiments.) We tested our method using the human face model shown in Figure 3, which has 1600 control points.

Figure 4(a) shows the run-time performance of our method when rendering deforming objects with an incremental crack prevention algorithm suggested in this paper and a non-incremental crack prevention algorithm suggested by Barsky's method [12]. In general, our method with the incremental crack prevention algorithm is roughly two times faster than with the non-incremental one. Because when incremental crack prevention is used, there is no need to identify neighboring nodes and compare subdivision levels for each quad-tree node.

Figure 4(b) shows the comparison of our method with and without multiple control point handling, the Kumar's method [14] and the surface evaluation method [15]. We tested the methods by moving different number of control points on the human face model simultaneously. The model is rendered with approximately 3600 polygons. Results show that the performance of our method with multiple control point handling is the highest of all methods compared and is independent of the number of moving control points. The performance of our method without multiple control point handling is inversely proportional to the increase in the number of moving control



(a) (b)

Figure 4: Performance comparison.

points. This is because the time spent on the incremental polygon model updating increases in proportional to the number of simultaneous moving control points. Results also show that our method is roughly 15 times faster than the surface evaluation method and 5 times faster than Kumar's method. This is due to the fact that both the surface evaluation method and Kumar's method need to spend extra processing time to perform tessellation from the defining equation of the surface to produce the deformed polygon model, while our method only performs incremental update on the current polygon model to produce the deformed one.

References

[1] G. Farin, *Curves and Surfaces for Computer Aided Geometric Design.* Academic Press, Fourth ed., 1997.

[2] L. Piegl and W. Tiller, *The NURBS Book.* Springer-Verlag, 1995.

[3] S. Abi-Ezzi and S. Subramaniam, "Fast Dynamic Tessellation of Trimmed NURBS Surfaces," *Proceedings of Eurographics '94*, pp. 108–126, 1994.

[4] S. Kumar, D. Manocha, H. Zhang, and K. Hoff, "Accelerated Walkthrough of Large Spline Models," *Proceedings of Symposium on Interactive 3D Graphics*, pp. 91–101, 1997.

[5] A. Rockwood, K. Heaton, and T. Davis, "Real-Time Rendering of Trimmed Surfaces," *Proceedings of ACM SIGGRAPH '89*, vol. 23, no. 3, pp. 107–117, 1989.

[6] F. Li, R. Lau, and M. Green, "Interactive Rendering of Deforming NURBS Surfaces," *Proceedings of Eurographics '97*, vol. 16, pp. 47–56, September 1997.

[7] H. Hoppe, "View-Dependent Refinement of Progressive Meshes," *Proceedings of ACM SIGGRAPH '97*, vol. 31, pp. 189–198, August 1997.

[8] R. Lau, D. To, and M. Green, "An Adaptive Multi-Resolution Modeling Technique Based on Viewing and Animation Parameters," *IEEE VRAIS'97*, pp. 20–27, 1997.

[9] D. To, R. Lau, and M. Green, "A Method for Progressive and Selective Transmission of Multi-Resolution Models," *To appear in the Proceedings of ACM Symposium on Virtual Reality Software and Technology*, December 1999.

[10] J. Xia and A. Varshney, "Dynamic View-Dependent Simplification for Polygonal Models," *Proceedings of IEEE Visualization '96*, pp. 327–334, October 1996.

[11] D. Filip, "Adaptive Subdivision Algorithms for a Set of Bézier Triangles," *Computer-Aided Design*, pp. 74–78, 1986.

[12] B. Barsky, T. DeRose, and M. Dippé, "An Adaptive Subdivision Method With Crack Prevention for Rendering Beta-spline Objects," Tech. Rep. UCB/CSD 87/348, Dept. of Computer Science, U.C. Berkeley, 1987.

[13] J. Wong, R. Lau, and L. Ma, "Virtual 3D Sculpturing with a Parametric Hand Surface," *Proceedings of Computer Graphics International '98*, pp. 178–186, 1998.

[14] S. Kumar, D. Manocha, and A. Lastra, "Interactive Display of Large-Scale NURBS Models," *Proceedings of Symposium on Interactive 3D Graphics*, pp. 51–58, 1995.

[15] C. de Boor, "On Calculating B-splines," *Journal of Approximation Theory*, vol. 6, pp. 50–62, July 1972.