The Visual Computer manuscript No.
(will be inserted by the editor)

Young In Yeo · Tianyun Ni · Ashish Myles · Vineet Goel · Jörg Peters

Parallel Smoothing of Quad Meshes

the date of receipt and acceptance should be inserted later

Abstract For use in real-time applications, we present a fast algorithm for converting a quad mesh to a smooth, piecewise polynomial surface on the Graphics Processing Unit (GPU). The surface has well-defined normals everywhere and closely mimics the shape of Catmull-Clark subdivision surfaces. It consists of bi-cubic splines wherever possible, and a new class of patches – *c-patches* – where a vertex has a valence different from 4. The algorithm fits well into parallel streams so that meshes with 12,000 input quads, of which 60% have one or more non-4-valent vertices, are converted, evaluated and rendered with 9×9 resolution per quad at 50 frames per second. The GPU computations are ordered so that evaluation avoids pixel dropout.

Keywords subdivision · GPU · smooth surface · quadrilateral mesh

1 Introduction and Contribution

Quad meshes, i.e. meshes consisting of quadrilateral facets, naturally model the flow of (parallel) feature lines and are therefore common in modeling for animation. Any polyhedral mesh can be converted into a quad mesh by one step of Catmull-Clark subdivision [3]. But preferably, a designer creates the mesh as a quad mesh so that no global refinement is necessary. *Smooth* surfaces are needed, for example, as the base for displacement mapping in the surface normal direction [8] (Fig 1).

Y. Yeo
University of Florida
E-mail: yiyeo@cise.ufl.edu

T. Ni
University of Florida

A. Myles
University of Florida

V. Goel
Advanced Micro Devices

J. Peters
University of Florida

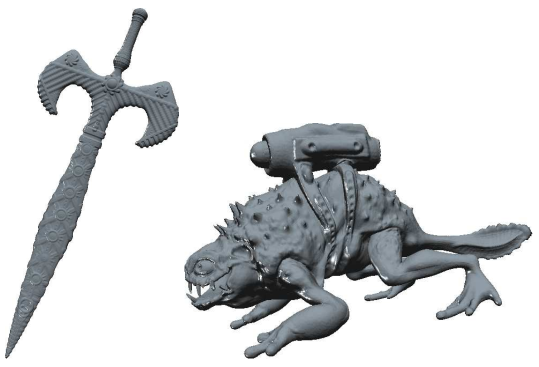
Fig. 1 GPU smoothed quad meshes with displacement mapping.

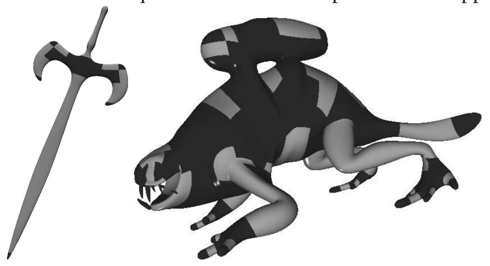
Fig. 2 Patches from ordinary quads (light) and extraordinary quads (dark).

For real-time applications such as gaming, interactive animation, simulation and morphing, it is convenient to offload smoothing and rendering to the Graphics Processing Unit (GPU). In particular, when morphing is implemented on the GPU, it is inefficient to send large data streams on a round trip to the CPU and back. Current and impending GPU configurations favor short explicit surface definitions, as derived below, over recursively defined surfaces.

For the following GPU-based surface construction, we distinguish two types of quads: ordinary and extraordinary. A quad is *ordinary* if all four vertices have 4 neighbors. Such a facet will be converted into a degree 3 by 3 patch in tensor-product Bernstein-Bézier (Bézier) form by the standard B-spline to Bézier conversion rules [4]. Therefore, any two adjacent patches derived from ordinary quads will join

C^2. The interesting aspect of this paper is the conversion of the *extraordinary* quads, i.e. quads having at least one, and possibly up to four, vertices of valence $n \neq 4$. We present a new algorithm for converting both types of quads on the fly so that

1. every ordinary quad is converted into a bicubic patch in tensor-product Bézier form, Figure 3(b);
2. every extraordinary quad is converted into a composite patch (short **c-patch**) with cubic boundary and defined by 24 coefficients, Figure 3(c);
3. the surface is by default smooth everywhere (Lemma 1);
4. the shape follows that of Catmull-Clark subdivision;
5. conversion and evaluation can be mapped to the GPU to render at very high frame rates (at least an order of magnitude faster than for example [2,19] on current hardware).

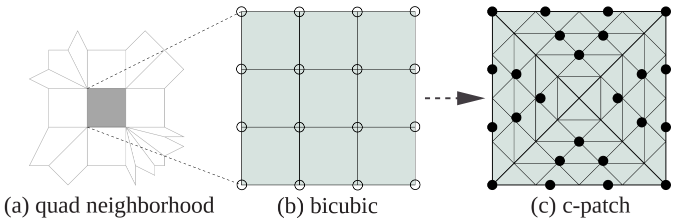

Fig. 3 (a) A quad neighborhood defining a surface piece. (b) A bicubic patch with 4×4 control points marked as $\circ$. This patch is the output if the quad is ordinary, and used to determine the shape of a (c) **c-patch** if the quad is extraordinary. A c-patch is defined by 4×6 control points displayed as $\bullet$. (For analysis, it can alternatively be represented as four C^1-connected triangular pieces of degree 4 with degree 3 outer boundaries identical to the bicubic patch boundaries.)

1.1 Some Alternative Mesh Smoothing Techniques on the GPU

A number of techniques exist to smooth out quad meshes. Catmull-Clark subdivision [3] is an accepted standard, but does not easily port to the GPU. Evaluation using Stam's approach [20] is too complex for large meshes on the GPU. The methods in [2,19,1] require separated quad meshes, i.e. quad meshes such that each quad has at most one point with valence $n \neq 4$. To turn quad meshes into separated quad meshes usually means applying at least one Catmull-Clark subdivision step on the CPU and four-fold data transfer to the GPU. In more detail, Shiue [19] implements recursive Catmull-Clark subdivision using several passes via the pixel shader, using textures for storage and spiral-enumerated mesh fragments. Bolz [1] tabulates the subdivision functions up to a given density and linearly combines them in the GPU. Bunnell [2] provides code for adaptive refinement. Even though this code was optimized for an earlier generation GPUs, this implementation adaptively renders the Frog (Figure 2) in real-time on current hardware (See Section 5 for a comparison with our approach). The main difference between our and Bunnell's implementation is that we decouple mesh conversion from surface evaluation and therefore do not have the primitive explosion before the second rendering pass. Moreover, we place conversion early in the pipeline so that the pixel shader is freed for additional tasks.

Three alternative smoothing strategies mimic Catmull-Clark subdivision by generating a finite number of bicubic patches. PCCM [14,15] generates NURBS output that could be rendered, for example by the GPU algorithm of Guthe et al. [6]. But this has not been implemented to our knowledge.

PN-quads [16] are a variant of the three-sided patches published in Vlachos et al. [22]. For each quad, one bicubic 'geometry patch' and one biquadratic 'normal patch' are generated. Adjacent geometry patches join C^0 along the edges and match the prescribed position P and normal N at each vertex. The separately computed normal patches also join continuously and interpolate the prescribed normals N at the vertices. Since the lighting is based on the continuous normal field defined by the normal patches, an impression of smoothness is conveyed; only the silhouette betrays the lack of smoothness in the actual geometry defined by the geometry patch. The shape of surfaces can be made more rounded by taking as input the limit points and normals of Catmull-Clark (PN-lim in Figure 19). The method of Loop and Schaefer [10] is very similar to PN-quads. It also generates one bicubic patch per quad following the shape of Catmull-Clark surfaces. Since these bicubic patches typically do not join smoothly, Loop and Schaefer compute two additional patches whose cross product approximates the normal of the bicubic patch. As pointed out in [22], these trompe l'oeils represent a simple solution when true smoothness is not needed. In a comparison to our method, we show in Section 5 that the lack of smoothness in [10] can result in visible artifacts.

The quincunx split of the quad by the c-patch reminds of the Zwart-Powell element [23,18], simplest subdivision [17] and 4-8 subdivision [21] due to the underlying box-spline directions.

2 The Conversion Algorithm

Here we give the algorithm for converting the quad mesh into coefficients that define a smooth surface of low degree. Analysis of the properties of this new surface type and the implementation of the algorithm on the GPU follow in the next sections. Essentially, the algorithm consists of computing new points near a vertex using Table 1, and, for each extraordinary quad, additional points according to Table 2 (see Figure 4). In Section 3, we will verify that these new points define a smooth surface and in Section 4, we show how the two stages naturally map to the vertex shader and geometry shader stage, respectively, of the current GPU pipeline.

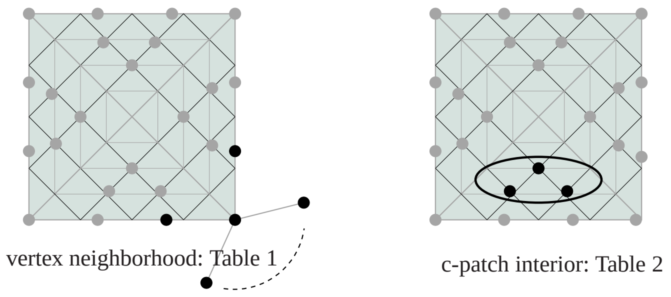

Fig. 4 Vertex neighborhoods with coefficients v^i and e^i_j and c-patch interiors with coefficients b^i_{211}, b^i_{121}, b^i_{112} .

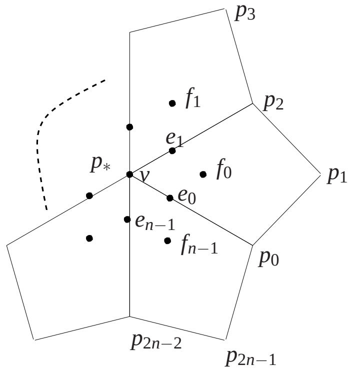

Fig. 5 Smoothing the vertex neighborhood according to Table 1. The center point p_*, its direct neighbors p_{2j} and diagonal neighbors p_{2j+1} form a vertex neighborhood.

f_j	$:=$	$(4p_* + 2p_{2j} + 2p_{2j+2} + p_{2j+1})/9$
e_j	$:=$	$(f_j + f_{j-1})/2$
v	$:=$	$\frac{1}{n(n+5)}\left(n^2 p_* + \sum_{j=0}^{n-1}\left(4p_{2j} + p_{2j+1}\right)\right)$
t_j	$:=$	$v + \frac{1}{n\sigma_n}\sum_{\ell=0}^{n-1}\cos\frac{2\pi(j-\ell)}{n}e_\ell,\ j = 0,1.$

Table 1 Computing control points v, e, f and t, the projection of e, at a vertex of valence n from the mesh points p_j of a vertex neighborhood; the subscripts are modulo $2n$. By default, $\sigma_n := \left(\mathsf{c}_n + 5 + \sqrt{(\mathsf{c}_n+9)(\mathsf{c}_n+1)}\right)/16$, the subdominant eigenvalue of Catmull-Clark subdivision.

2.1 Computing the Vertex Neighborhood

In the first stage, we focus on a vertex neighborhood. A *vertex neighborhood* consists of a mesh point p_* and mesh points $p_k, k = 0,\ldots,2n-1$ of all quads surrounding p_* (Figure 5). A vertex v computed according to Table 1 is the limit point of Catmull-Clark subdivision as explained, for example, in [7]. For $n = 4$, this choice is the limit of bicubic subdivision, i.e. B-spline evaluation. The rules for e_j and f_j are the standard rules for converting a uniform bicubic tensor-product B-spline to its Bernstein-Bézier representation of degree 3 by 3 [4]. The points t_j are a projection of e_j into a common tangent plane (see e.g. [5]). The default scale factor σ_n is the subdominant eigenvalue of Catmull-Clark subdivision. We note that for $n = 4$, $e_{j+2} = 2v - e_j$ and $\sigma_4 = 1/2$ so that the projection leaves the tangent control points invariant as $t_j = e_j$:

$$\text{for } n = 4, \quad t_j = v + \frac{2}{4}(e_j - e_{j+2}) = v + (e_j - v) = e_j. \quad (1)$$

2.2 Bi-cubic patches and c-patches

In the second stage, we gather vertex neighborhoods to construct patches on quads. Combining information from four vertex neighborhoods as shown in Figure 6, we can populate a tensor-product patch g of degree 3 by 3 in Bernstein-Bézier (Bézier) form [4]:

$$g(u,v) := \sum_{k=0}^{3}\sum_{\ell=0}^{3} g_{k\ell}\binom{3}{k}u^k(1-u)^{3-k}\binom{3}{\ell}v^\ell(1-v)^{3-\ell}.$$

The patch is defined by its 16 coefficients or *control points* $g_{k\ell}$. If the quad is ordinary, the formulas of Table 1 make this patch the Bézier representation of a bicubic spline in B-spline form. For example, in the notation of Figure 6, $(g_{k0})_{k=0,..3} = (v^0, t^0_0, t^1_1, v^1)$. If the quad is extraordinary, we

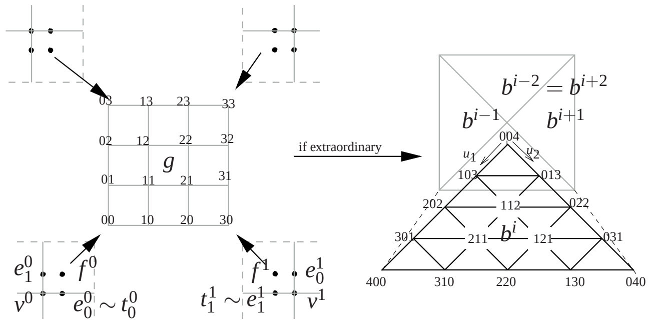

Fig. 6 Patch construction. On the left, the indices of the control points of g are shown. Four vertex neighborhoods with vertices v^i each contribute one sector to assemble the 4×4 coefficients of the Bézier patch g, for example $g_{00} = v^0$, $g_{10} = e^0_0$, $g_{11} = f^0$, $g_{30} = v^1$, $g_{31} = e^1_0$ (we use superscripts to indicate vertices; see also Figure 9). On the right, the same four sectors are used to determine a c-patch if the underlying quad is extraordinary. *Note that only a subset of the coefficients of the four triangular pieces b^i is actually computed to define the c-patch.* The full set of coefficients displayed here is only used to analyze the construction.

use the bicubic patch to outline the shape as we replace it by a c-patch (Figure 3(c)). A c-patch has the right degrees of freedom to cheaply and locally construct a smooth surface. **The c-patch** is defined by the 4×6 c-coefficients constructed in Tables 1 and 2:

$$v^i, t^i_0, t^i_1, b^i_{211}, b^i_{121}, b^i_{112},\ i = 0,1,2,3.$$

By construction, the c-patch and an adjacent tensor-product patch g have *identical boundary curves of degree 3* where they meet, an important consideration for preventing gaps in the final GPU implementation.

Alternatively, we can view one c-patch as the union of four polynomial patches b^i, $i = 0,1,2,3$ of *total degree* 4.

$$\boxed{\begin{aligned}
b^i_{211} &:= b^i_{310} + \tfrac{1+\mathrm{c}^i}{4}(t^{i+1}_1 - t^i_0) + \tfrac{1-\mathrm{c}^{i+1}}{8}(t^i_0 - v^i) \\
&\qquad + \tfrac{3}{4(\mathrm{s}^i+\mathrm{s}^{i+1})}(f^i - e^i_0) \\
b^i_{121} &:= b^i_{130} + \tfrac{1+\mathrm{c}^{i+1}}{4}(t^i_0 - t^{i+1}_1) + \tfrac{1-\mathrm{c}^i}{8}(t^{i+1}_1 - v^{i+1}) \\
&\qquad + \tfrac{3}{4(\mathrm{s}^i+\mathrm{s}^{i+1})}(f^{i+1} - e^{i+1}_1) \\
b^i_{112} &:= g_* + 3(b^i_{211} + b^i_{121} - b^{i+1}_{121} - b^{i-1}_{211})/16 \\
&\qquad + (b^{i+1}_{211} + b^{i-1}_{121} - b^{i+2}_{211} - b^{i-2}_{121})/16
\end{aligned}}$$

Table 2 Formulas for the 4×3 interior control points that, together with the vertex control points v^i and the tangent control points t^i_j, define a *c-patch*. See also Figures 9 and 10. Here $\mathrm{c}^i := \cos\frac{2\pi}{n_i}$, $\mathrm{s}^i := \sin\frac{2\pi}{n_i}$ and superscripts are modulo 4. By default, $g_* := (\sum_{i=0}^{3} v^i + 3(e^i_0 + e^i_1) + 9f^i)/64$, the central point of the ordinary patch.

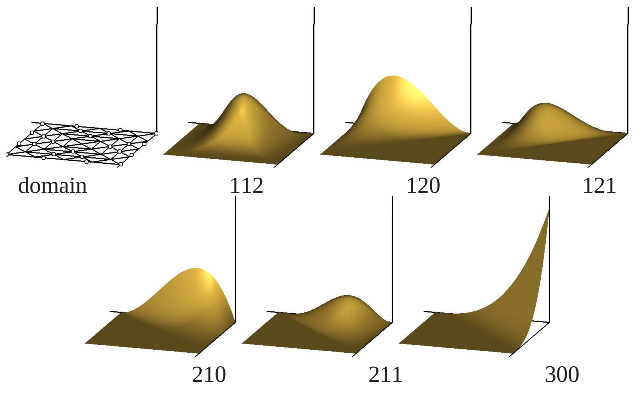

Fig. 7 The six basis functions of one sector of the c-patch.

A polynomial piece b^i of *total degree* 4 [4] has the Bézier form

$$b^i(u_1,u_2) := \sum_{\substack{k+\ell+m=4\\ k,\ell,m\ge 0}} b^i_{k\ell m} \frac{4!}{k!\ell!m!} u_1^k u_2^\ell (1-u_1-u_2)^m. \tag{2}$$

The 4×6 c-coefficients imply the interior control points of this representation (2) by C^1 continuity between the triangular pieces: for $j = 0,1,2,3$ and $i = 0,1,2,3$,

$$b^i_{3-j,0,1+j} = b^{i-1}_{0,3-j,1+j} := (b^i_{3-j,1,j} + b^{i-1}_{1,3-j,j})/2; \tag{3}$$

and the boundary control points $b^i_{k\ell 0}$ are implied by degree-raising [4]:

$$\begin{aligned} b^i_{400} &:= v^i, \quad b^i_{310} := (v^i + 3t^i_0)/4, \quad b^i_{220} := (t^i_0 + t^{i+1}_1)/2, \\ & b^i_{130} := (v^{i+1} + 3t^{i+1}_1)/4, \quad b^i_{040} := v^{i+1}. \end{aligned} \tag{4}$$

Basis functions corresponding to the 24 c-coefficients of the c-patch can be read off by setting one c-coefficient to one and all others to zero and then applying (3) and (4) to obtain the representation (2). Fig. 7 shows the six basis functions of one sector. Two pairs are symmetric.

2.3 Interior c-patch-coefficients

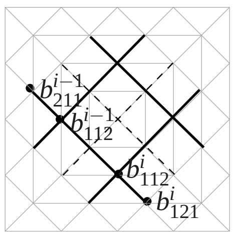

Fig. 8 Dark lines cover the control points involved in the C^2 constraints (5). The points on dashed lines are implied by averaging.

To derive the formulas for b^i_{211} and its symmetric counterpart b^i_{121} note that the formulas must guarantee a smooth transition between b^i and its neighbor patch on an adjacent quad, regardless whether the adjacent quad is ordinary or extraordinary. That is, the formulas are derived to satisfy *simultaneously* two types of smoothness constraints (see Section 3). By contrast, b^i_{112} is not pinned down by continuity constraints. We could choose each b^i_{112} arbitrarily without changing the formal smoothness of the resulting surface. However, we opt for increased smoothness at the center of the c-patch and additionally use the freedom to closely mimic the shape of Catmull-Clark subdivision surfaces, as we did earlier for vertices. First, we approximately satisfy four C^2 constraints across the diagonal boundaries at the central point b_{004} by enforcing

$$\begin{bmatrix} 1 & -1 & 0 & 0 \\ 0 & 1 & -1 & 0 \\ 0 & 0 & 1 & -1 \\ -1 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} b^0_{112} \\ b^1_{112} \\ b^2_{112} \\ b^3_{112} \end{bmatrix} = \frac{1}{2} \begin{bmatrix} b^0_{211} - b^1_{121} - q \\ b^1_{211} - b^2_{121} - q \\ b^2_{211} - b^3_{121} - q \\ b^3_{211} - b^0_{121} - q \end{bmatrix}, \tag{5}$$

where $q := \frac{1}{4}\sum_{i=0}^{3}(b^i_{211} - b^i_{121})$. The perturbation by q is necessary, since the coefficient matrix of the C^2 constraints is rank deficient. After perturbation, the system can be solved with the last equation implied by the first three. We add the constraint that the average of b^i_{112} matches $g_* := g(\frac{1}{2},\frac{1}{2})$, the center position of the bicubic patch. Now, we can solve for b^i_{112}, $i = 0,1,2,3$ to obtain the formula in Table 2.

3 Verifying Smoothness of the Surface

In this technical section we formally verify the following lemma. For the purpose of the proof, we view the c-patch in its equivalent representation (2) as four Bézier patches of total degree 4.

Lemma 1 *Two adjacent polynomial pieces a and b defined by the rules of Section 2 (Table 1, Table 2,* (3), (4)*) meet at least*

(i) C^2 *if a and b correspond to two ordinary quads.*
(ii) C^1 *if a and b are adjacent pieces of a c-patch;*
(iii) C^1 *if a and b correspond to two quads, exactly one of which is ordinary;*
(iv) *with tangent continuity if a and b correspond to two different extraordinary quads.*

Proof (i) If a and b are bicubic patches corresponding to ordinary quads, they are part of a bicubic spline with uniform knots and therefore meet C^2. (ii) If a and b are adjacent pieces of a c-patch then Equations (3) enforce C^1 continuity.

For the remaining cases, let b be a triangular piece. Let u the parameter corresponding to the quad edge between $b_{400} = v^0$, where $u = 0$ and the valence is n_0 and $b_{040} = v^1$ where $u = 1$ and the valence is n_1 (see Figures 9 for (iii) and 10 for case (iv)). By construction, the common boundary $b(u,0) = a(0,u)$ is a curve of degree 3 with Bézier control points (v^0, t_0^0, t_1^1, v^1) so that bicubic patches on ordinary quads and triangular patches on extraordinary quads match up exactly.

Denote by $\partial_1 b$ the partial derivative of b along its first parameter – i.e. along the common boundary – and by $\partial_2 b$ the partial derivative in its second variable. Since $b(u,0) = a(0,u)$, we have $\partial_1 b(u,0) = \partial_2 a(0,u)$. The partial derivative in the first variable of a is, similarly, $\partial_1 a$. We will verify that the following conditions implying tangent continuity hold:

if one quad is ordinary (case (iii)),

$$\partial_1 b(u,0) = 2\partial_2 b(u,0) + \partial_1 a(0,u); \tag{6}$$

if both quads are extraordinary (case (iv)),

$$\big((1-u)\lambda_0 + u\lambda_1\big)\partial_1 b(u,0) = \partial_2 b(u,0) + \partial_1 a(0,u), \tag{7}$$

$$\text{where } \lambda_0 := 1 + \mathrm{c}^0,\ \lambda_1 := 1 - \mathrm{c}^1, \text{ and } \mathrm{c}^i := \cos(\frac{2\pi}{n_i}).$$

Both equations, (6) and (7), equate vector-valued polynomials of degree 3 since we write $\partial_1 b(u,0)$ in degree-raised form. The equations hold if and only if all corresponding Bézier coefficients are equal on both sides. Off hand, this means checking four vector-valued equations for each of (6) and (7). However, in both cases, the setup is symmetric with respect to reversal of the direction in which the boundary $b(u,0)$ is traversed. That means, we need only check the first two equations (6') and (6") of (6) and the first two equations (7') and (7") of (7). We verify these equations by inserting the formulas of Tables 1 and 2.

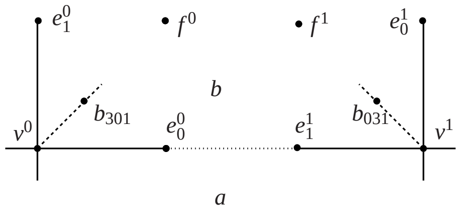

Fig. 9 C^1 transition between a triangular patch b (*top*) and a bicubic patch a (*bottom*).

To verify (6), the key observation is that $n_0 = n_1 = 4$ if one quad is ordinary. Hence $\mathrm{c}^0 = \mathrm{c}^1 = 0$ and $\mathrm{s}^0 = \mathrm{s}^1 = 1$ (cf. Table 2) and $t_j^i = e_j^i$. Therefore, for example (cf. Figure 9)

$$2\partial_2 b(0,0) = 2 \cdot 4(b_{301} - v^0) = 8\frac{3}{4}(\frac{e_0^0 + e_1^0}{2} - v^0)$$
$$= 3(e_0^0 + e_1^0) - 6v^0,$$

where the factor $\frac{3}{4}$ stems from raising the degree from 3 to 4; and the second Bézier coefficient of $\partial_1 b(u,0)$ (in degree-raised form) and of $2\partial_2 b(u,0)$ are respectively (cf. Figure 9)

$$3\frac{(e_0^0 - v^0) + 2(e_1^1 - e_0^0)}{3} \qquad \text{and}$$

$$2 \cdot 4(b_{211} - b_{310}) = 8(\frac{e_1^1 - e_0^0}{4} + \frac{e_0^0 - v^0}{8} + 3\frac{f^0 - e_0^0}{8}).$$

Then, comparing the first two Bézier coefficients of $\partial_1 b(u,0)$ and $2\partial_2 b(u,0) + \partial_1 a(0,u)$ yields equality and establishes C^1 continuity:

$$\underbrace{3(e_0^0 - v^0)}_{\partial_1 b(0,0)} = \underbrace{3(e_0^0 + e_1^0) - 6v^0}_{2\partial_2 b(0,0)} \underbrace{-3(e_1^0 - v^0)}_{\partial_1 a(0,0)} \tag{6'}$$

$$(e_0^0 - v^0) + 2(e_1^1 - e_0^0) = 2(e_1^1 - e_0^0) + (e_0^0 - v^0) + 3(f^0 - e_0^0)$$
$$- 3(f^0 - e_0^0). \tag{6''}$$

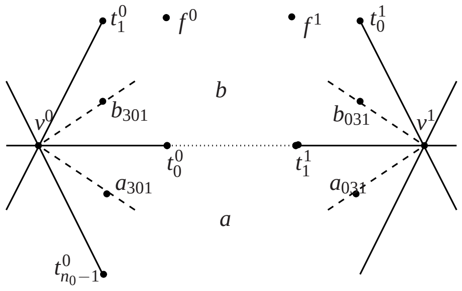

Fig. 10 G^1 transition between two triangular patches.

The equations for (7) are similar, except that we need to replace e_j by t_j and keep in mind that, by definition,

$$(t_{n_0-1}^0 - v^0) + (t_1^0 - v^0) = 2\mathrm{c}^0(t_0^0 - v^0).$$

Hence, for example,

$$\partial_2 b(0,0) + \partial_1 a(0,0) = 4(b_{301} - v^0 + a_{301} - v^0)$$
$$= \frac{3}{4} 4 \cdot 2\mathrm{c}^0(t_0^0 - v^0).$$

The first of the four coefficient equations of (7) then simplifies to

$$3(1 + \mathrm{c}^0)(t_0^0 - v^0) = 4(b_{301} + a_{301} - 2v^0)$$
$$= 3(\frac{t_1^0 + t_0^0}{2} - v^0 + \frac{t_1^{n_0-1} + t_0^0}{2} - v^0)$$
$$= 3\frac{1}{2}(2\mathrm{c}^0(t_0^0 - v^0) + 2(t_0^0 - v^0)). \tag{7'}$$

Noting that terms $(f_0 - e_0^0)/(8(\mathrm{s}^0 + \mathrm{s}^1))$ in the expansions of b_{211} and a_{211} cancel, the second coefficient equation is

$$6\lambda_0(t_1^1 - t_0^0) + 3\lambda_1(t_0^0 - v^0) = 12(b_{211} + a_{211} - 2b_{310})$$
$$= \frac{12 \cdot 2(1 + \mathrm{c}^0)}{4}(t_1^1 - t_0^0) + \frac{12 \cdot 2(1 - \mathrm{c}^1)}{8}(t_0^0 - v^0). \tag{7''}$$

It is easy to read off that the equalities hold. So the claim of smoothness is verified.

4 GPU Implementation

We implemented our scheme in DirectX 10 using the vertex shader to compute vertex neighborhoods according to Table 1 and the geometry shader primitive *triangle with adjacency* to accumulate the coefficients of the bicubic patch or compute a c-patch according to Table 2. We implemented conversion plus rendering in two variants: a 1-pass and a 2-pass scheme. Bicubic and c-patch are implemented in separate shaders.

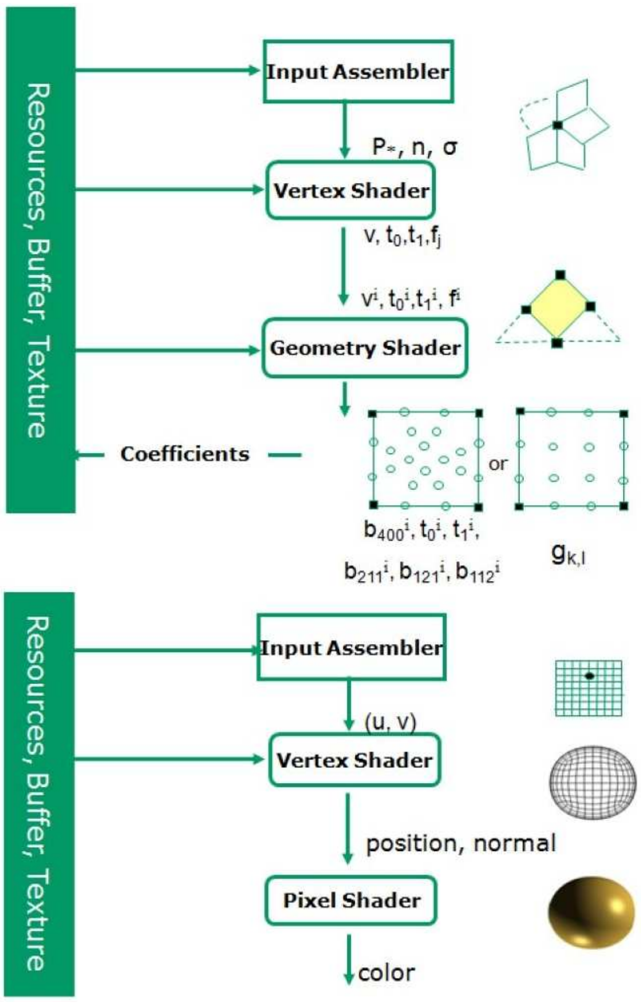

Fig. 11 2-pass implementation detailed in Table 3. The first pass converts, the second renders. Note that the geometry shader only computes at most 24 coefficients per patch and does not evaluate.

The *2-pass implementation* constructs the patches in the first pass using the vertex shader and the geometry shader and evaluates positions and normals in the second pass. Pass 1 streams out only the 4×6 coefficients of a c-patch. it does not stream out the $4\times\binom{4+2}{2}$ Bézier control points of the equivalent triangular pieces. The data amplification necessary to evaluate takes place by instancing a (u,v)-grid on the vertex shader in the *second pass*. That is, we *do not stream back large data sets after amplification*. Position and normal are computed on the (u,v) domain $[0..1]^2$ of the bicubic or of the c-patch (not on any triangular domains). Table 3 lists the input, output and the computations of each pipeline stage. Figure 11 illustrates this association of computations and resources. Overall, the 2-pass implementation has small stream-out, short geometry shader code and minimal amplification on the geometry shader (see Appendix).

In the *1-pass implementation*, the evaluation immediately follows conversion in the geometry shader, using the

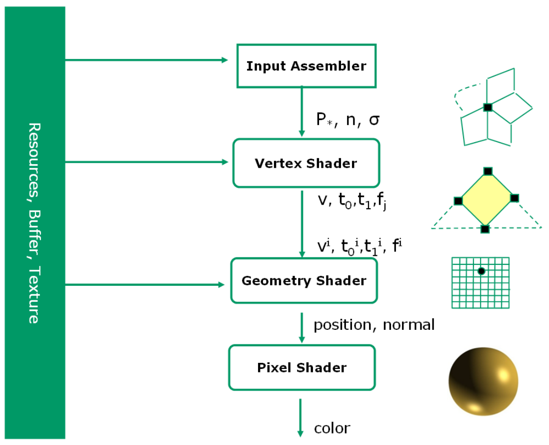

Fig. 12 At present, the 1-pass conversion-and-rendering must place patch assembly and evaluation on the geometry shader. This is not efficient.

Pass 1	Conversion
VS In	p_*,n,σ
VS	Use texture lookup to retrieve p_{2j},p_{2j+1}
	Compute v,e_j,f_j,t_0,t_1 (Table 1)
VS Out	$v,t_0,t_1,f_j,j=0..n-1$
GS In	$v^i,t_0^i,t_1^i,f^i,i=0..3$
GS	if ordinary quad
	assemble $g_{kl},k,l=0..3$ (Figure 6)
	else
	compute $b_{211}^i,b_{121}^i,b_{112}^i$ (Table 2)
GS Out	if ordinary quad, stream out $g_{kl},k,l=0..3$.
	else stream out $b_{400}^i,t_0^i,t_1^i,b_{211}^i,b_{121}^i,b_{112}^i$, $i=0..3$.
Pass 2	**Evaluating Position and Normal**
VS In	(u,v)
VS	if ordinary quad
	compute normal and position at (u,v)
	by the tensored de Casteljau's algorithm
	else
	Compute the remaining Bézier control points (3)
	Compute normal and position at (u,v)
	by de Casteljau's algorithm adjusted to c-patches.
VS Out	position, normal
PS In	position, normal
PS	compute color
PS Out	color

Table 3 2-Pass conversion: VS=vertex shader, GS=geometry shader, PS=pixel shader. VS Out of Pass 1 outputs n points f_j for one vertex (hence the subscript) and GS In of Pass 1 retrieves four points f^i, each generated by a different vertex of the quad (hence the superscript).

geometry shader's ability to amplify, i.e. output multiple point primitives for each facet (Figure 12). While a 1-pass implementation sounds more efficient than a 2-pass implementation, DX10 limits data amplification in the geometry shader so that the maximal evaluation density is 8×8 per quad. Moreover, maximal amplification in the geometry shader slows the performance. The performance difference between the two implementations is easily visible when comparing Tables 4 and 5, with the caveat that we did not spend much time optimizing the clearly slower 1-pass approach.

5 Results

We compiled and executed the implementation on the latest graphics cards of both major vendors under DirectX10 and tested the performance for several industry-sized models. Two surface models and models with displacement mapping are shown in Figure 2 and 1 respectively. Table 4 summarizes the performance of the 2-pass algorithm for different granularities of evaluation. The (rocket) Frog model, in particular, provides a challenge due to the large number of extraordinary patches.

Mesh	Frames per second			
(verts,quads, eqs)	$N = 5$	9	17	33
Sword (140,138, 38%)	965	965	965	703
Head (602,600, 100%)	637	557	376	165
Frog (1308,1292, 59%)	483	392	226	87

Table 4 Frames per second for some standard test meshes with each patch evaluated on a grid of size $N \times N$; eqs = percentage of extraordinary quads. Sword and Frog are shown in Figure 2, Head in Figure 12 of [12]. For the smallest object, Sword, at low resolution, rendering rather than evaluation is the bottleneck. The measurements were made on an NVidia GeForce 8800 GTX graphics card.

Mesh	Slower 1-pass implementation		
	$N = 2$	5	8
Sword	389	96	43
Head	108	34	15
Frog	44	10	4

Table 5 Performance of the 1-pass implementation.

The Frog Party shown in Figure 18 currently renders at 50 fps for uniform evaluation of nine frogs for N=9, i.e. on a 9×9 grid. That is, the implementation converts nine times 1292 coarse input quads, of which 59% are extraordinary, and renders nearly 1.5 million polygons 50 times per second. Additionally, our method scales well to higher tessellation levels, since the patch creation time does not increase for larger evaluation grids. On the same hardware, we measured Bunnell's efficient implementation (distribution accompanying [2]) featuring the single frog model, i.e. 1/9th of the work of the Frog Party, running at 44 fps with three subdivisions (equivalent to tessellation factor N=9). That is, GPU smoothing of quad meshes is an order of magnitude faster. Compared to [19], the speedup is even more dramatic. While the comparison is not among equals since both [19] and [2] implement recursive Catmull-Clark subdivision, it is nevertheless fair to observe that the speedup is at least partially due to our avoiding stream back after amplification (data explosion due to refinement).

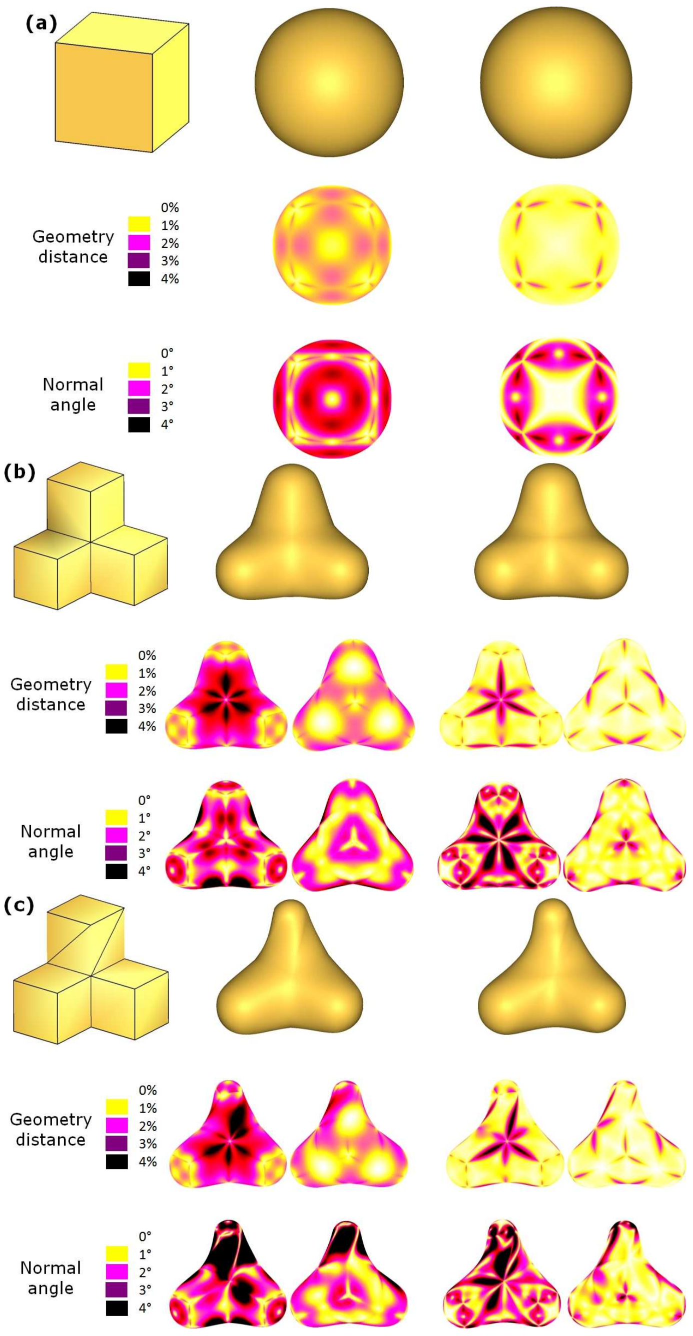

Fig. 13 Comparison to Catmull-Clark. Position (distance) and normal (angle) difference to the limit surface of Catmull-Clark subdivision for (*left*) the scheme in [10] and (*right*) the c-patch surface (see also Table 5). Lighter shading means better match. The number of compared samples are (a) 24578, (b) 73730, (c) 73730.

Figure 13,*right*, visualizes the approximation to a densely refined Catmull-Clark mesh. Both *geometric* distance, as percent of the local quad size, and *normal* distance, in degrees of variation, are measured. Large models and models with a large percentage of regular quads appear visually indistinguishable when rendered by subdivision or c-patch smoothing. We therefore chose small, predictable models with many extraordinary input quads (and without displacement). Table 5 quantifies and summarizes these distances. Since we have been asked to compare c-patch surfaces to the non-smooth approximation [10], Figure 13 and Table 5 juxtapose the measurements. The more subtle effect of not creating smooth surfaces is evident from Figure 14.

Average	Distance to Catmull-Clark		
	(a)	(b)	(c)
[10]			
Position	1.20	1.58	1.67
Normal	2.09	1.94	2.74
c-patch			
Position	0.70	0.77	0.80
Normal	1.48	1.64	1.77

Table 6 Average deviation from the Catmull-Clark limit surface in position (parametric distance scaled by local quad size) and normal (angle) for the examples in Fig. 13.

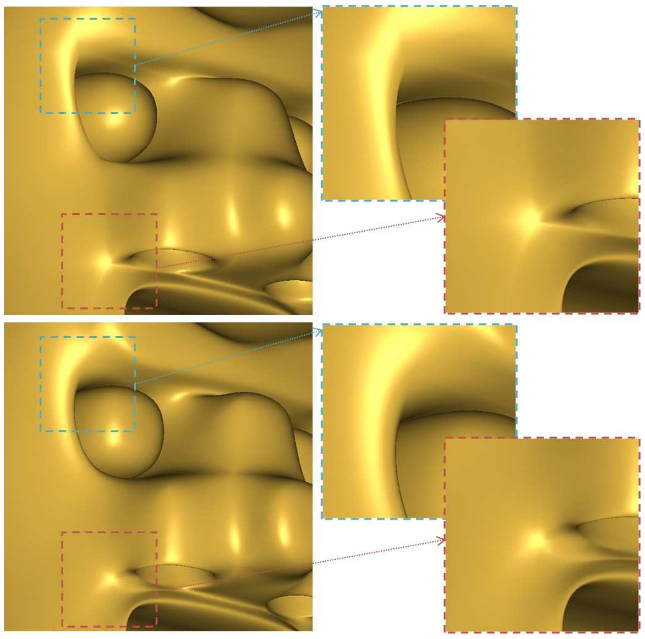

Fig. 14 C^0 **artifacts** (*top*) of [10] at the base of the nostril and arch of the eye (straight line). (*bottom*) $c-patch$ construction for comparison.

Despite the lower total degree and internal C^1 join, the visual appearance of c-patches is remarkably similar to that of bicubic patches. In particular, the close-up in Figure 17 illustrates our observation that c-patches do not create shape problems compared to a single bicubic patch. As is generally the recommendation for quad meshes, adjacent high-valent vertices in the input model should be avoided (see the sin terms in the denominator of the formulas of Table 2). The video [13] (see screen shots in Figures 17, 15, 16, 18) illustrates real time displacement and animation.

6 Discussion

Smoothing quad meshes on the GPU offers an alternative to highly refined facet representations transmitted to the GPU and is preferable for interactive graphics and integration with complex morphing and displacement. The separation into vertex and patch construction stages isolates the computation on arbitrary valences from the final patch construction, simplifying the vertex and geometry shaders. Moreover, the data transfer between passes in the 2-pass conversion is low since only 4×6 control points are intermittently generated.

Since we only compute and evaluate in terms of the 24 c-patch coefficients, the computation of the cubic boundaries shared by a bicubic and a c-patch is mathematically identical. An explicit 'if'-statement in the evaluation guarantees the exact same ordering of computations since boundary coefficients are only computed once, in the vertex shader, according to Table 1. That is, there is no pixel drop out or gaps in the rendered surface. The resulting surface is watertight.

We advertised a 2-pass scheme, since, as we argued, the DX10 geometry shader is not well suited for data amplification and evaluation after conversion. The 1-pass scheme outlined in Section 4 may become more valuable with the availability of a dedicated hardware tessellator [9]. Such a tessellator will make amplification more efficient and support watertight *adaptive tessellation* (which is why we only discussed uniform tessellation in Section 4). Such a hardware amplification will also benefit the 2-pass approach in that the (u,v) domain tessellation, fed into the second pass will be replaced by the amplification unit.

Acknowledgements This work benefited from CGAL's half-edge data structure, and used Bay Raitt's Frog and the ZBrush Sword model. This work was supported by NSF CCF-0728797.

References

1. J. Bolz and P. Schröder. Rapid evaluation of Catmull-Clark subdivision surfaces. In *Web3D '02: Proceeding of the seventh international conference on 3D Web technology*, pages 11–17, New York, NY, USA, 2002. ACM Press.
2. M. Bunnell. *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*, chapter 7. Adaptive Tessellation of Subdivision Surfaces with Displacement Mapping. Addison-Wesley, Reading, MA, 2005.
3. E. Catmull and J. Clark. Recursively generated B-spline surfaces on arbitrary topological meshes. *Computer Aided Design*, 10:350–355, 1978.
4. G. Farin. *Curves and Surfaces for Computer Aided Geometric Design: A Practical Guide*. Academic Press, 1990.
5. C. Gonzalez and J. Peters. Localized hierarchy surface splines. In S. S. J. Rossignac, editor, *ACM Symposium on Interactive 3D Graphics*, pages 7–15, 1999.

6. M. Guthe, A. Balázs, and R. Klein. GPU-based trimming and tessellation of NURBS and T-spline surfaces. *ACM Transactions on Graphics*, 24(3):1016–1023, 2005.
7. M. Halstead, M. Kass, and T. DeRose. Efficient, fair interpolation using Catmull-Clark surfaces. *Proceedings of SIGGRAPH 93*, pages 35–44, Aug 1993.
8. A. Lee, H. Moreton, and H. Hoppe. Displaced subdivision surfaces. In K. Akeley, editor, *Siggraph 2000, Computer Graphics Proceedings*, Annual Conference Series, pages 85–94. ACM Press / ACM SIGGRAPH / Addison Wesley Longman, 2000.
9. M. Lee. Next generation graphics programming on Xbox 360, 2006. http://download.microsoft.com/download/d/3/0/d30d58cd-87a2-41d5-bb53-baf560aa2373/next_generation_graphics_programming_on_xbox_360.ppt.
10. C. Loop and S. Schaefer. Approximating Catmull-Clark subdivision surfaces with bicubic patches. *ACM Transactions on Graphics*, 27(1):1–11, 2008.
11. T. Ni, Y. Yeo, A. Myles, V. Goel, and J. Peters. Smooth surfaces from 4-sided facets. Technical Report 2007-429, Dept CISE, University of Florida, 2007.
12. T. Ni, Y. Yeo, A. Myles, V. Goel, and J. Peters. GPU smoothing of quad meshes. In M. Spagnuolo, D. Cohen-Or, and X. Gu, editors, *IEEE International Conference on Shape Modeling and Applications, June 4 - 6, 2008, Stony Brook University, Stony Brook, New York, USA*, pages 3–10. ACM Press, 2008.
13. T. Ni, Y. Yeo, A. Myles, V. Goel, and J. Peters. GPU smoothing of quad meshes, 2008. http://www.cise.ufl.edu/research/SurfLab/08smi.
14. J. Peters. Patching Catmull-Clark meshes. In K. Akeley, editor, *Siggraph 2000, Computer Graphics Proceedings*, Annual Conference Series, pages 255–258. ACM Press / ACM SIGGRAPH / Addison Wesley Longman, 2000.
15. J. Peters. Modifications of PCCM. Technical Report 2001-001, Dept CISE, University of Florida, 2001.
16. J. Peters. PN-quads. Technical Report 2008-421, Dept CISE, University of Florida, 2008.
17. J. Peters and U. Reif. The simplest subdivision scheme for smoothing polyhedra. *ACM Trans. Graph.*, 16(4):420–431, 1997.
18. M. Powell. Piecewise quadratic surface fitting for contour plotting. In *Software for Numerical Mathematics*, pages 253–271. Academic Press, 1974.
19. L.-J. Shiue, I. Jones, and J. Peters. A realtime GPU subdivision kernel. *ACM Transactions on Graphics*, 24(3):1010–1015, 2005.
20. J. Stam. Exact evaluation of Catmull-Clark subdivision surfaces at arbitrary parameter values. In *SIGGRAPH*, pages 395–404, 1998.
21. L. Velho and D. Zorin. 4–8 subdivision. *Computer-Aided Geometric Design*, 18(5):397–427, 2001. Special Issue on Subdivision Techniques.
22. A. Vlachos, J. Peters, C. Boyd, and J. L. Mitchell. Curved PN triangles. In *2001, Symposium on Interactive 3D Graphics*, Bi-Annual Conference Series, pages 159–166. ACM Press, 2001.
23. P. Zwart. Multivariate splines with nondegenerate partitions. *SIAM Journal on Numerical Analysis*, 10(4):665–673, 1973.

Appendix: Shader code

The HLSL code will be posted at the authors' web site. The code below has been edited for readability.

```
struct InputMeshVertex {
    uint n       : BLENDINDICES0; // valence
    uint index   : BLENDINDICES1;}; // 1-ring start
struct Sector {               // one c-patch sector
    float4           b112;
    float4       b211, b121;
    float4 b300, b210, b120; };
typedef Sector cPatch[4]; // one c-patch
struct VertexOutput {
    uint n                 : BLENDINDICES1;
    float4 v               : SV_POSITION;
    float4 t0              : TANGENT0;
    float4 t1              : TANGENT0;
    float4 f[MAX_VALENCE]  : POSITION0;
};


// Vertex Shader
VertexOutput VertexBasedExtyPatchConstruction(
   InputMeshVertex input,
   uint vID : SV_VertexID )
{
    VertexOutput vout;
    float4 direct[MAX], // direct neighbors
           diag[MAX];   // diagonal neighbors
    float4 vLocation;   // vertex position

    // position (from vertex texture cache)
    vLocation = float4( gVertexLocation.Load(
       int3(vID, gAnimationFrame,0)).xyz,1);

    // Set vertex to Catmull-Clark limit
    uint n      = input.n;
    uint index  = input.index;
    vout.v      = vLocation*n*n;
    [unroll]
    for (uint i = 0; i < n; ++i) {
        float ftmp  = gRingIndex.Load(
           int2(index+i*2  , 0));
        float4 vtmp = gVertexLocation.Load(
           int3((uint)ftmp, gAnimationFrame,0));
        direct[i] = float4(vtmp.xyz, 1);
        ftmp  = gRingIndex.Load(
           int3(index+i*2+1, 0));
        vtmp = gVertexLocation.Load(
           int3((uint)ftmp, gAnimationFrame,0));
        diag[i] = float4(vtmp.xyz, 1);
        vout.v += 4.0*direct[i] + diag[i];
    }
    vout.v /= (n*(n+5));

    // Face points
    [unroll]
    for (uint i = 0; i < n; ++i) {
        uint im = (i+n-1) % n;
        vout.f[i] = (4.0/9.0)*vout.v
           + (2.0/9.0)*direct[im]
           + (1.0/9.0)*(direct[i]+diag[im]);
    }

    // Two tangents; cCos(n,i)=cos((2*PI/n)*i
    vout.t0 = float4(0.0, 0.0, 0.0, 0.0);
    vout.t1 = float4(0.0, 0.0, 0.0, 0.0);
    for (uint i = 0; i < n; ++i) {
       uint ip = (i + 1) % n;
       float4 e = 0.5*(vout.f[i]+vout.f[ip]);
       vout.t0 += cCos(n,i)*e;
       vout.t1 += cSin(n,i)*e;
    }
    const float c = cCos(n,1);
    const float sigma =
       (c+5+sqrt((c+9)*(c+1)))/16;
    vout.t0 /= n * sigma;
    vout.t1 /= n * sigma;
    vout.n = input.n;
    return vout;
}
```

```
// Geometry Shader
[maxvertexcount(24)]
void FacetBasedExtyPatchConstruction(
   triangleadj VertexOutput input[6],
   inout PointStream<GS_OUTPUT> Stream,
   uint pID : SV_PrimitiveID )
{
    cPatch pat; // c-patch coefficients

    // Load index offsets (packed in uint)
    uint rot_packed = gOffsetData.Load(
       int3(pID, 0, 0));
    uint rot_off[4];
    [unroll]  // 4 bits encode each rotation
    for (uint i = 0; i < 4; ++i)
       rot_off[i] = (rot_packed >> (4*i)) & 0xF;

    // Compute b300, b210 and b120
    [unroll]
    for (uint k = 0; k < 4; ++k) {
        const uint km = (k+4-1) % 4;
        const uint n = input[k].n;
        const uint off  = rot_off[k];
        const uint offm = (off+(n-1))%n;
        pat[k ].b300 = input[k].v;
        pat[k ].b210 = input[k].v
           + input[k].t0*cCos(n,offm)
           + input[k].t1*cSin(n,offm);
        pat[km].b120 = input[k].v
           + input[k].t0*cCos(n,off )
           + input[k].t1*cSin(n,off );
    }

    // Compute b211, b121 for each sector
       (removed - see Table 2)

    // Compute b112 for each sector
    [unroll]
    for (uint k = 0; k < 4; ++k) {
        const uint km  = (k+3) % 4;
        const uint km2 = (k+2) % 4;
        const uint kp  = (k+1) % 4;
        const uint kp2 = (k+2) % 4;
        pat[k].b112 = b004
            + (3.0/16.0) * (
               pat[k ].b211 + pat[k ].b121
             - pat[kp ].b121 - pat[km ].b211)
            + (1.0/16.0) * (
               pat[kp].b211 + pat[km].b121
             - pat[kp2].b211 - pat[km2].b121);
    }

    // Stream out the c-patch control points
       (removed -- straightforward)
}
```

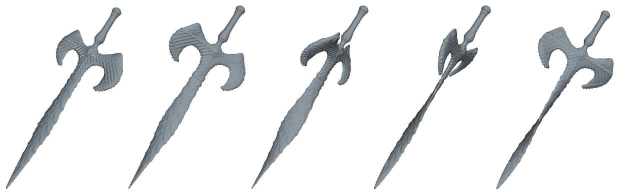
Fig. 15 Real time displacement on the twisting Sword model. See [13].

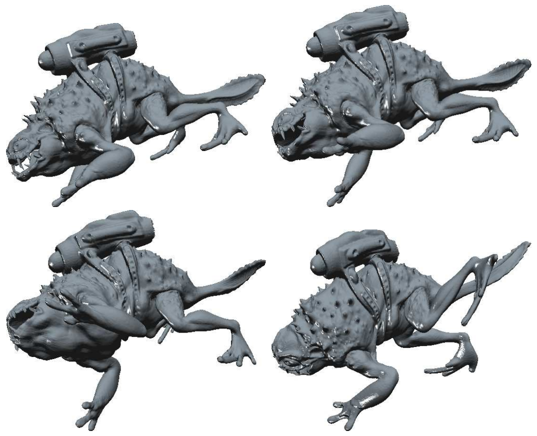
Fig. 16 Real time displacement on the twisting Frog model [13].

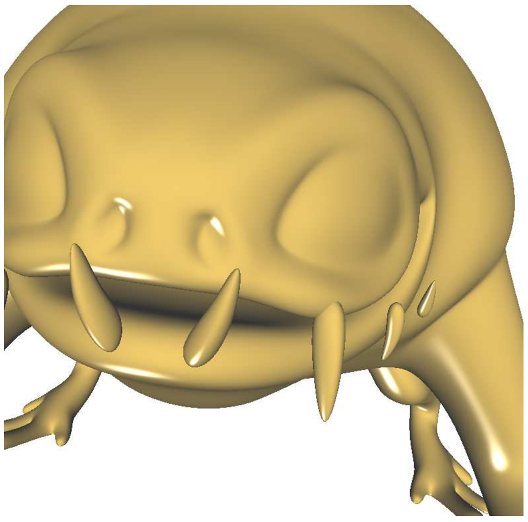
Fig. 17 Close-up of the Frog.

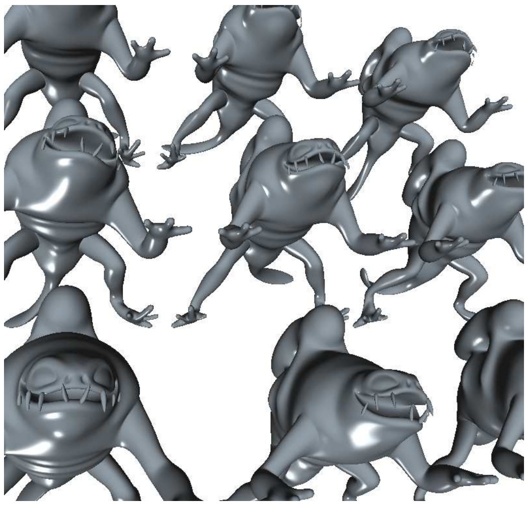
Fig. 18 Asynchronous animation of nine Frogs [13].

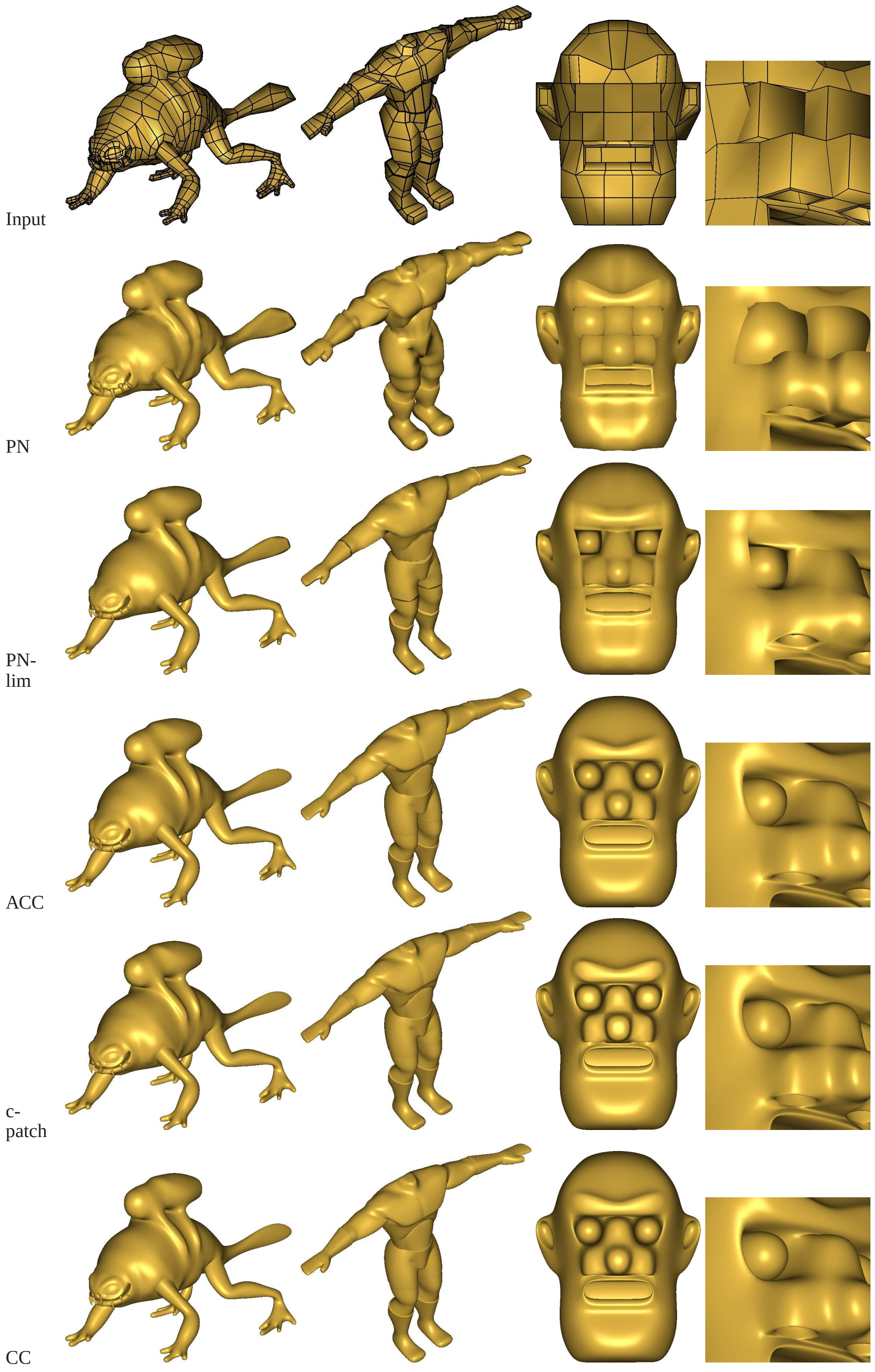

Fig. 19 Comparison. (Input) quad mesh, (PN)-quad and (PN-lim) PN-Quad using Catmull-Clark limit points and normals [16], (ACC) [10] (c-patch) this paper, (CC) Catmull-Clark subdivision.