Efficient Pixel-accurate Rendering of Animated Curved Surfaces

Young In Yeo, Sagar Bhandare, Jörg Peters

University of Florida, USA,
jorg@cise.ufl.edu

Abstract. To efficiently animate and render large models consisting of bi-cubic patches in real time, we split the rendering into pose-dependent, view-dependent (Compute-Shader supported) and pure rendering passes. This split avoids recomputation of curved patches from control structures and minimizes overhead due to data transfer – and it integrates nicely with a technique to determine a near-minimal tessellation of the patches while guaranteeing sub-pixel accuracy. Our DX11 implementation generates and accurately renders 141,000 animated bi-cubic patches of a scene in the movie 'Elephant's Dream' at more than 300 frames per second on a 1440×900 screen using one GTX 580 card.

1 Introduction

Curved, smooth, piecewise polynomial surfaces have become standard in high end, movie-quality animation. Subdivision surfaces [1,2], spline (NURBS) surfaces or Bézier patch-based surfaces are chosen over polygonal, polyhedral, or faceted-based representations both for aesthetic reasons and for their ability to represent models more compactly. In particular, curved surfaces yield more life-like transitions and silhouettes and, in principle, support arbitrary levels of resolution without exhibiting polyhedral artifacts (see Fig. 1). But while curved surfaces are commonly used in cinematic production and geometric design, they are not commonly used for interactive viewing. Animation artists and designers typically work off faceted models at a given resolution and have to call special off-line rendering routines to inspect the true outcome of their work. At the other end of the spectrum, game designers opt for coarsely-faceted models, made more acceptable by careful texturing, to achieve real-time rendering with limited resources under competing computational demands, e.g. computing game physics. In an attempt to narrow the gap, a number of mesh-to-surface conversion algorithms have been developed in the past years that run efficiently on the GPU (see Section 2). But so far their rendering has depended on screen projection heuristics without guarantees of accuracy.

The present paper explains how to render, at interactive rates, and on high-resolution screens, a substantial number of animated curved surfaces free of perceptible polyhedral artifacts, parametric distortion and pixel dropout. The paper leverages and extends the authors' approach [3] for efficiently determining the near-minimal tessellation density required for *pixel-accurate rendering* (see Section 2). Determining the near-minimal tessellation density requires, depending on the model, between 1% and 5% extra work. However, by avoiding overtessellation, pixel-accurate rendering is often faster than rendering based on heuristics (see Fig. 2, middle and right). Specifically, the paper shows

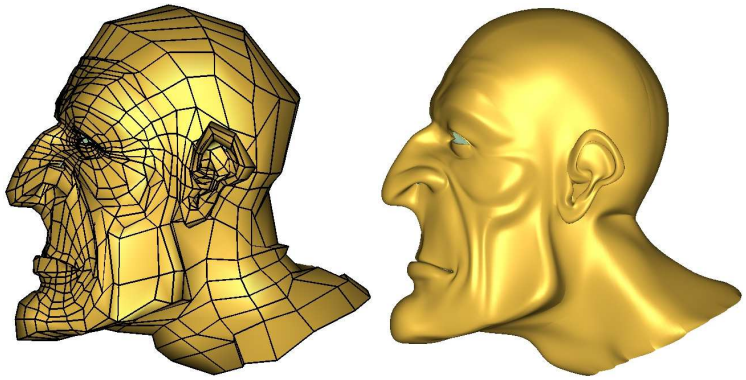

Fig. 1. Faceted versus smooth: Proog's head.

how to integrate the approach into animation the animation pipeline to make it interactive. Rather than repeatedly sending large control nets from the CPU to the GPU for rendering, we load the base mesh(es) once and we apply morph-target and skeletal-animation transformations to the characters' mesh model on the GPU and convert it into a curved surface. We then use the natural partition of animated sequences into pose-dependent, view-dependent and pure rendering frames to compute both the animation and the pixel-accurate patch-tessellation in a combination of two, one or no Compute Shaders preceding each standard rendering pass on the GPU.

As proof of concept, we animated and rendered 141K patches of a scene of the open-source movie Elephants Dream. In 2006, each frame of the movie required 10 minutes of CPU time at full-HD resolution [4]. We can now render the higher-order surfaces and textures (leaving out post effects) on the GPU at more than 300 frames per second Fig. 15 thanks to parallelism and new algorithms that take advantage of this parallelism. To wit, doubling processor speed every year since 2006 would reduce the time per frame only to ca 10 seconds per frame, three orders of magnitude slower.

Overview. In Section 2 we review the definition of pixel-accurate rendering of curved surfaces, animation basics and the conversion of faceted to smooth curved surfaces. Section 3 presents the idea and formulas for enforcing pixel-accurate rendering. Section 4 presents the algorithm and an efficient implementation, including pseudo-code, of pixel-accurate rendering of animated curved surfaces. In Section 5 we analyze the implementation's performance and discuss trade-offs and alternative choices. We also compare to a similar widely-available DX11 sample program.

2 Background

To efficiently pixel-accurately render the surfaces of Elephants Dream on the GPU, our proposed animation framework has to near-optimally set the tessellation factor for Bézier patches after replicating linear skeletal animation, relative shape-key animation (morph targets), and mesh-to-surface conversion on the GPU.

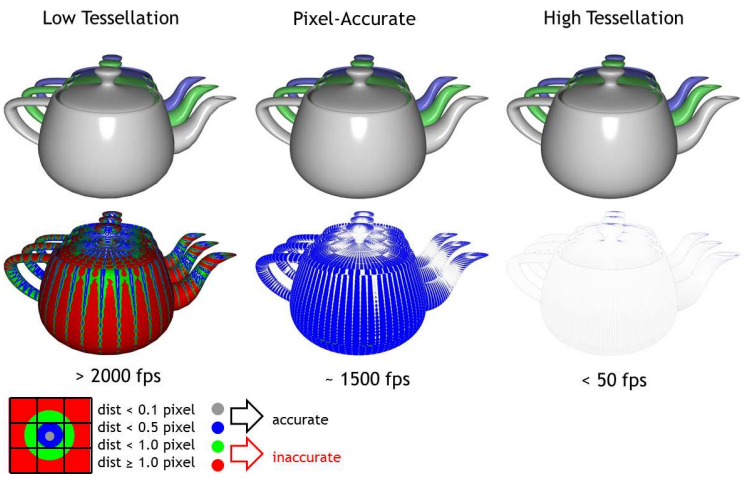

Fig. 2. Balancing pixel-accuracy and rendering speed. No red or green colors should be visible in the lower row if the tessellation is sufficiently fine for pixel-accuracy. The red and the green spots indicate a parametric distortion of more than 1/2 pixel (cf. the color coding *lower left*). Additional objects are analyzed in Fig. 12 of [3].

Tessellation and Pixel-accuracy. A key challenge when working with a curved surface is to set the density of evaluation so that the surface triangulation is a good proxy of the smooth surface. The density, or tessellation, has to be sufficiently high to prevent polyhedral artifacts and parametric distortions, and sufficiently low to support fast rendering. In modern graphics pipelines, the level of detail can be prescribed by setting the tessellation factor(s) τ of each patch $\mathbf{p} : (u, v) \in U \to \mathbb{R}^3$ of the curved surface. In 3D movie animation, it is common practice to over-tessellate and shade a very high number of fragments. Real-time animation cannot afford this since each fragment is evaluated, rasterized and shaded. Since the camera is free to zoom in or out of the scene, fixed level of tessellation results in faceted display or overtessellation. This disqualifies approaches that require setting τ a priori. Popular screen-based heuristics based on measuring edge-length or estimating flatness (see e.g. [5], [6, Sec 7]) do not come with guarantees or require an a priori undetermined number of passes to recursively split patches and verify that the measure falls below a desired tolerance.

Pixel-accurate rendering, Fig. 2, middle, determines the tessellation density (just) fine enough to guarantee correct visibility, prevent parametric distortion or pixel-dropout. Pixel-accuracy has two components: covering (depth) accuracy and parametric (distortion) accuracy [3, Section 3]. *Covering accuracy* requires that each pixel's output value be controlled by one or more unoccluded pieces of patches whose projection overlaps it sufficiently and *parametric accuracy* requires that for each pixel the following holds (cf.Fig. 3). Let $\left[\begin{smallmatrix}\mathtt{x}\\ \mathtt{y}\end{smallmatrix}\right]$ be the pixel's center, $\mathbf{p} : \mathbb{R}^2 \to \mathbb{R}^3$ a surface patch and (u, v) a parameter pair. Then the surface point $\mathbf{p}(u, v) \in \mathbb{R}^3$ must project into the pixel:

$$\|P(\mathbf{p}(u, v)) - \left[\begin{smallmatrix}\mathtt{x}\\ \mathtt{y}\end{smallmatrix}\right]\|_\infty < 0.5. \tag{1}$$

Inequality (1) prevents *parametric distortion*: if $P(\mathbf{p}(u,v))$ lies outside the pixel asso-

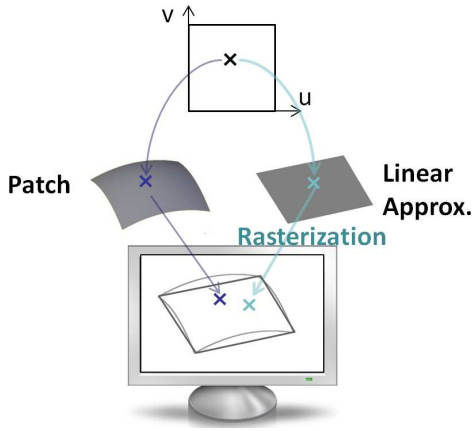

Fig. 3. Triangulation and projection distorting the image of a curved surface. Pixel-accurate rendering guarantees that the distortion is at below pixel level.

ciated with parameters (u,v) then the wrong texture, normal or displacement is computed causing artifacts incompatible with accurate rendering. Parametric inaccuracy is color-encoded in Fig. 2: lack of accuracy is shown in red and green. Predictably, too coarse a tessellation yields a high frame rate and too fine a tessellation slows down rendering. The largely grey coloring of the teapot in Fig. 3, row two, under pixel-accurate rendering, indicating a distortion just below the pixel threshold, is therefore highly desirable. Work similar to [3], but based on the bounds in [7] includes [8] and most recently [9].

Skeletal animation. The most common technique for character animation, used by the artists of Elephants Dream, is linear blend skinning, also known as linear vertex blending or skeletal subspace deformation [10]. Here a character is defined by a template, a faceted model, called skin. The models animation or deformation is defined by a time-varying set of rigid transformations, called bones, that are organized into a tree structure, called skeleton. Any vertex position in a linear blend skin is expressed as a linear combination of the vertex transformed by each bone's coordinate system: at time t_i, a convex combination ω_k of bone transformations $\mathbf{R}_k$ is applied to each skin vertex initial position $\mathbf{v}(0)$:

$$\mathbf{v}(t) = \big(\sum_k \omega_k \mathbf{R}_k(t)\big)\mathbf{v}(0), \quad \sum_k \omega_k = 1. \tag{2}$$

The weights ω_k are assigned by the artist. Section 4 provides pseudo-code.

Since this direct linear combination of rotation matrices generically does not yield a valid rotation, a number of improvements have been suggested [11,12]. In particular dual quaternions [12] are sufficiently simple to have been implemented in Blender. Our

framework is agnostic to the choice of animation since its implementation as a Compute Shader allows alternative animation techniques to be substituted such as deformation of the mesh points with respect to control cages (see e.g. [13,14,15,16]). However, since the artists of Elephants Dream used linear blend skinning, and compensated for its shortcomings, our real-time rendering applies linear, skeletal animation.

(a) without shape keys (b) with shape keys

Fig. 4. Emo's mouth opened with **shape keys**.

Shape Keys. For more nuanced, say facial expressions, Elephants Dream, and hence our implementation, additionally applies shape keys, also known as morph targets or blend shapes. Shape keys average between morph targets representing standard poses (see e.g. [17] for a detailed explanation.)

Mesh-to-Surface Conversion. In recent years, a number of algorithms have been developed to use polyhedral meshes as control nets of curved surfaces and efficiently evaluate these curved surfaces on the GPU. Such algorithms include conversions to piecewise polynomial and rational representation [18,19,20,21] as well as subdivision [22,23,6,24]. Our framework is agnostic to the choice of conversion algorithm. To be able to compare our GPU implementation to a widely accessible implementation, we chose Approximate Catmull-Clark (ACC) [20]: optimized shader code of ACC animation, SubD11, is distributed with MicroSoft DX11 [25]. The output of ACC is one bi-cubic patch patch for each face of the (refined) control mesh (plus a pair of tangent patches to improve the impression of smoothness as in [26]). Note that parametric accuracy is not concerned with whether ACC provides a good approximation to subdivision surfaces, an issue of independent interest (cf. [27,28]).

3 Computing near-minimal accurate tessellation levels

The two main ingredients that make pixel-accurate rendering efficient are avoiding recursion and triangulating as coarsely as possible while guaranteeing pixel-accuracy (see Fig. 5). This section explains how to address both challenges by computing a near-minimal tessellation factor τ in a single step according to the approach in [3]. The tessellation factor is computed with the help of slefe-boxes [29]. Bilinear interpolants of these slefe-boxes, called slefe-tiles, sandwich the curved surface and the triangulation

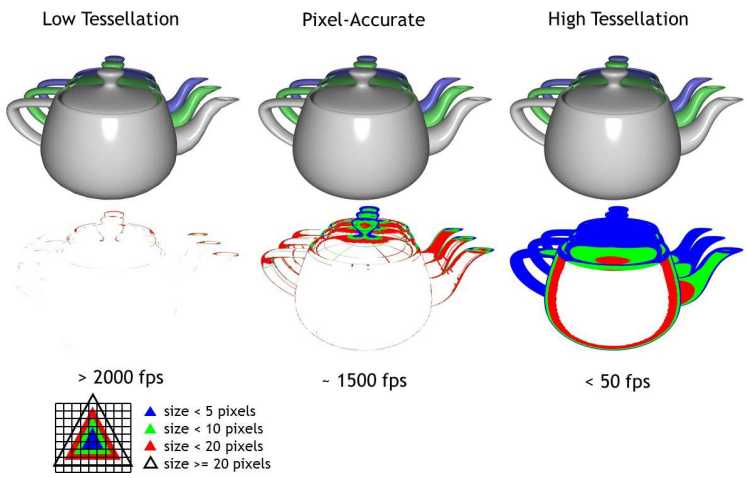

Fig. 5. Optimal tessellation of curved surfaces. Fewer, hence bigger triangles improve efficiency. (Note the different use of color-coding from Fig. 2).

as illustrated in Fig. 6. Such slefe-boxes are not traditional bounding boxes enclosing a patch. Rather the maximal width of slefe-boxes gives an upper bound on the variance of the exact curved surface from triangulation. This reflects the goal: to partition the domain sufficiently finely so that the variance and hence the 'width' of the all projected slefe-boxes and therefore of the slefe-tiles falls below a prescribed tolerance, e.g. half the size of a pixel.

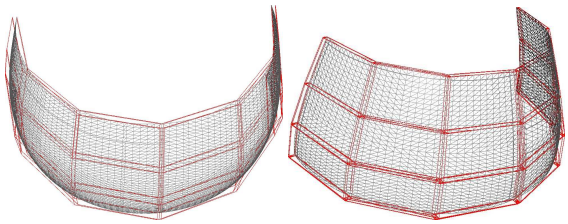

Fig. 6. The bi-linear interpolants to groups of four slefe-boxes define **slefe-tiles** that locally enclose the surface. Note that the tiles, while useful of collision, are **never explicitly computed** for the pixel-accurate rendering.

Since knot insertion stably converts NURBS patches of degree (d_1, d_2) to tensor-product patches in Bézier-form (*glMap2* in OpenGL) with coefficients $c_{ij} \in \mathbb{R}^3$ and

basis functions b_j^d,

$$p(u,v) := \sum_{i=0}^{d_1} \sum_{j=0}^{d_2} c_{ij} b_j^{d_2}(v) b_i^{d_1}(u), \qquad (u,v) \in [0..1]^2, \tag{3}$$

and since subdivision surfaces can be treated as nested rings of such patches, we focus on tensor-product Bézier patches. (Knot insertion can be a pre-processing step or done on the fly on the GPU. Rational patches are rarely used in animation; if needed, for strictly positive weights, bounds in homogeneous space plus standard estimates of interval arithmetic do the trick.) Moreover, slefe-boxes for patches in tensor-product form can be derived from bounds in one variable and the computations for building slefe-boxes are separate in each x, y and z coordinate. We can therefore simplify the discussion in the next subsection to one univariate polynomial piece p in Bézier-form with coefficients $c_j \in \mathbb{R}$ and parameter $u \in [0\,..\,1]$:

$$p : \mathbb{R} \to \mathbb{R}, u \mapsto p(u) := \sum_{j=0}^{d} c_j b_j^d(u), \quad b_j^d := \binom{d}{j} (1-u)^{d-j} u^j.$$

Subdividable Linear Efficient Function Enclosures, abbreviated as **slefes**, tightly sandwich non-linear functions p, such as polynomials, splines and subdivision surfaces, between simpler, piecewise linear, lower and upper functions, $\underline{p}$ and $\overline{p}$:

$$\underline{p} \le p \le \overline{p},$$

[30,31,32,33,29,34,35]. Specifically, in one variable, [30] shows that (cf. Fig. 7, *left*)

$$\begin{aligned} p(t) \le \overline{p}(t) := \ell(t) &+ \sum_{j=1}^{d-1} \max\{0, \nabla_j^2 p\}\, \overline{\mathsf{a}_j^d}^m(t) \\ &+ \sum_{j=1}^{d-1} \min\{0, \nabla_j^2 p\}\, \underline{\mathsf{a}_j^d}_m(t). \end{aligned} \tag{4}$$

with the matching lower bound $\underline{p}$ obtained by exchanging $\min$ and $\max$ operators. Here

$$\mathsf{a}_j^d, \quad j = 1, \ldots, d-1,$$

are polynomials that span the space of polynomials of degree d minus the linear functions $\ell(t)$; $\overline{\mathsf{a}_j^d}^m$ is an m-piece upper and $\underline{\mathsf{a}_j^d}_m$ an m-piece lower bound on a_j^d; and $\nabla_j^2 p := c_{j-1} - 2c_j + c_{j+1}$ is a second difference of the control points. If p is a linear function, upper and lower bounds agree. The tightness of the bounds is important since loose bounds result in over-tessellation. Fig. 7b shows an example from [3], where the min-max or AABB bound is looser by an order of magnitude than the slefe-width $\mathsf{w} := \max_{t\in[0..1]} \overline{p}(t) - \underline{p}(t)$.

Being piecewise linear, the bounding functions $\overline{\mathsf{a}_j^d}^m$ and $\underline{\mathsf{a}_j^d}^m$ in (4) are defined by their values at the uniformly-spaced break points. These values can be pre-computed.

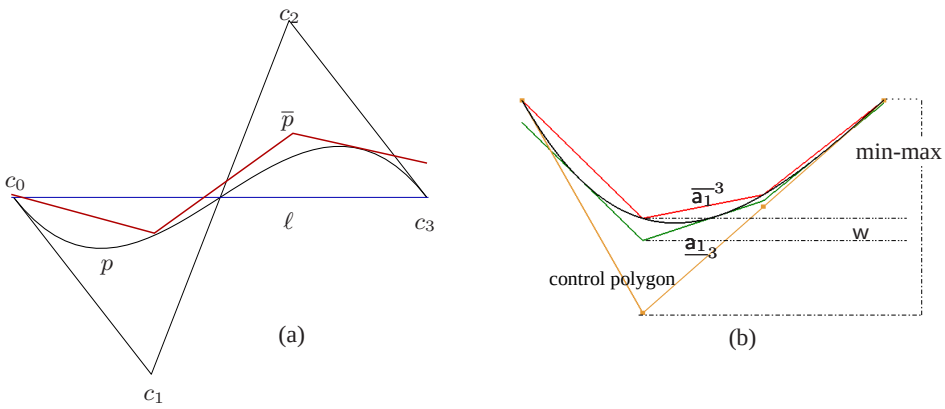

Fig. 7. The **slefe-construction** from [29]. (a) The function $p(t) := -b_1^3(t) + b_2^3(t)$ and its upper bound $\overline{p}$. (b) The lower bound $\underline{\mathsf{a}_1}_3$ and the upper bound $\overline{\mathsf{a}_1}^3$ tightly sandwiching the function $\mathsf{a}_1 := -\frac{2}{3}b_1^3(t) - \frac{1}{3}b_2^3(t)$, using $m = 3$ segments. Table 1 shows $\mathsf{w} = \max_{[0..1]} \overline{p} - \underline{p}$ to be < 0.07. The corresponding number for [7] (not illustrated) is $\frac{6}{8} = 0.75$ and for the min-max-bound $\frac{2}{3}$.

$t =$	0	1/3	2/3	1
$\overline{\mathsf{a}_1}^3$	0	-.370370..	-.296296..	0
$\underline{\mathsf{a}_1}_3$	-.069521..	-.439891..	-.315351..	-.008732..

Table 1. Values at **breakpoints of a** $m = 3$**-piece slefe**. This table and the tables for higher degree can be downloaded [36]. Similar slefe-tables exist for splines with uniform knots [30].

Since, for $d = 3$, i.e. cubic functions, $\mathsf{a}_2^3(1-t) = \mathsf{a}_1^3(t)$, Table 1 lists all numbers needed to compute Fig. 7, e.g., for $t = 1/3$, the upper and lower breakpoint values $-.370370..$ and $-.439891...$ Moreover, by tensoring, the 8 numbers suffice to compute all bounds required for ACC patches: the tensor-product patch (3) can be bounded by computing the upper values $\tilde{c}_{ij}, i = 0, \ldots, d_1$ (for each $j = 0, \ldots, m_2$) of the 1-variable slefe in the v direction and then treat the values as control points when computing the upper slefe in the u direction:

$$p(u,v) \le \sum_{i=0}^{d_1} \sum_{j=0}^{m_2} \tilde{c}_{ij} b_j^1(v) b_i^{d_1}(u) \le \sum_{j=0}^{m_2} \sum_{i=0}^{m_1} \bar{c}_{ij} b_i^1(u) b_j^1(v).$$

Ensuring pixel-accuracy The slefes just discussed are for functions, i.e. one coordinate of the image. Since we want to control the variance of the surface patches from their triangulation we now consider a patch $\mathbf{p} : \mathbb{R}^2 \to \mathbb{R}^3$ with three coordinates bounded by bilinear interpolants to upper and lower values at the grid points (u_i, v_j), $i, j \in \{0, 1, \ldots, m\}$. For each (u_i, v_j), abbreviating $\overline{\mathbf{p}_{ij}} := \overline{\mathbf{p}(u_i, v_j)}$, $\underline{\mathbf{p}_{ij}} := \underline{\mathbf{p}(u_i, v_j)}$, a

slefe-box is defined as

$$\underline{\overline{\mathbf{p}}}(u_i, v_j) := \frac{\overline{\underline{\mathbf{p}}_{ij}} + \underline{\overline{\mathbf{p}}_{ij}}}{2} + [-\frac{1}{2}..\frac{1}{2}]^3(\overline{\underline{\mathbf{p}}_{ij}} - \underline{\overline{\mathbf{p}}_{ij}}), \tag{5}$$

where $[-\frac{1}{2}..\frac{1}{2}]^3$ is the $\mathbf{0}$-centered unit cube. That is, the slefe-box is an axis-aligned box in $\mathbb{R}^3$ (see red boxes in Fig. 8) centered at the average of upper and lower values.

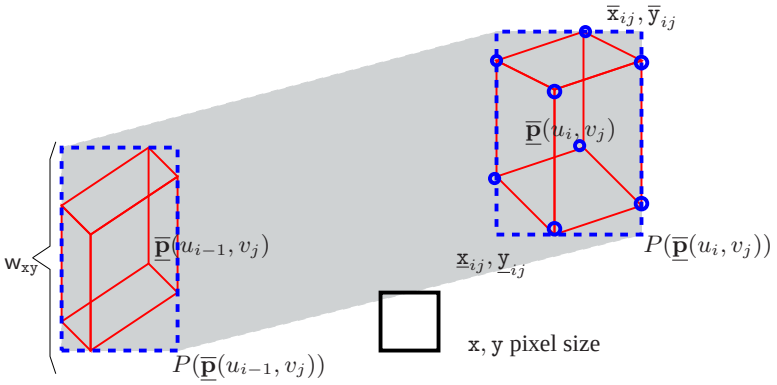

Fig. 8. Projected slefe-boxes. The projected slefe-boxes (red) are enclosed by axis-aligned rectangles (blue, dashed) whose linear interpolant (grey area) encloses the image (here of $\mathbf{p}([u_{i-1}..u_i], v_j)$). The (square-root of the) maximal edge-length of the dashed rectangles, in pixel size, determines the tessellation factor $\tau_{\mathbf{p}}$.

To measure parametric accuracy, we define the minimal screen-coordinate-aligned rectangle that encloses the screen projection $\left[\begin{smallmatrix}\mathtt{x}\\ \mathtt{y}\end{smallmatrix}\right] := P(\underline{\overline{\mathbf{p}}}(u_i, v_j))$ of to the slefe-box with index i, j (see the blue dashed rectangles in Fig. 8):

$$q_{ij} := [\underline{\mathtt{x}}_{ij}..\overline{\mathtt{x}}_{ij}] \times [\underline{\mathtt{y}}_{ij}..\overline{\mathtt{y}}_{ij}] \supsetneq P(\underline{\overline{\mathbf{p}}}(u_i, v_j)). \tag{6}$$

The maximal edge length over all q_{ij} is the parametric *width* $\mathsf{w}_{\mathtt{xy}}$. This width is a close upper bound on the variance from linearity in the parameterization since the width of the projected boxes dominates the width of the slefe-tiles – that therefore need not be computed. The width shrinks to zero when the parameterization becomes linear.

We want to determine the tessellation factor $\tau_{\mathtt{xy}} \in \mathbb{R}$ so that $\mathsf{w}_{\mathtt{xy}} < 1$. Let $\mathsf{w}_m(\mathbf{p})$ be the width of the projection of patch $\mathbf{p}$ measured for a slefe with m pieces and k_m a constant between 1.5 and 1, depending only on m. Since partitioning the u-domain into $1/h$ segments, and re-representing the function over the smaller interval before re-applying the bound, scales the maximal second difference down quadratically to h^2 its original size (cf. Fig. 9), partitioning both the u- and the v-domain into

$$\tau_{\mathtt{xy}}(m, \mathbf{p}) := k_m \sqrt{\mathsf{w}_m(\mathbf{p})} \tag{7}$$

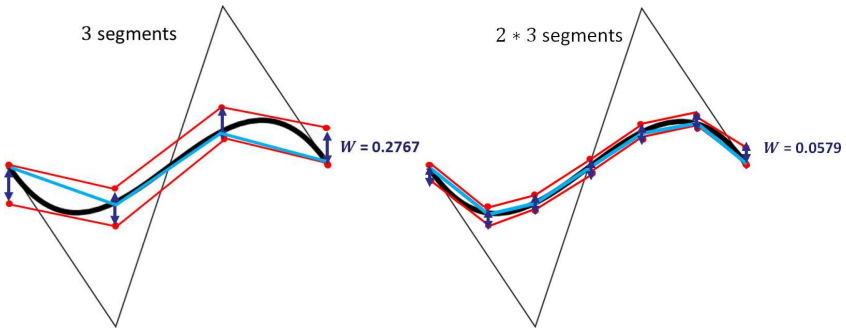

Fig. 9. Shrinkage of the width for a curve segment under subdivision. *black:* cubic curve, control polygon, *blue:* piecewise linear interpolant, *red:* slefe,

many pieces, confines the parameter distortion to below one unit (cf. Fig. 10) Analogously, the width $\mathsf{w}_{\mathbf{z}}(m, \mathbf{p})$ of the depth component $\mathbf{z}$ of the projection measures depth of the slefe-tiles and therefore trustworthiness of the z-buffer test for covering accuracy.

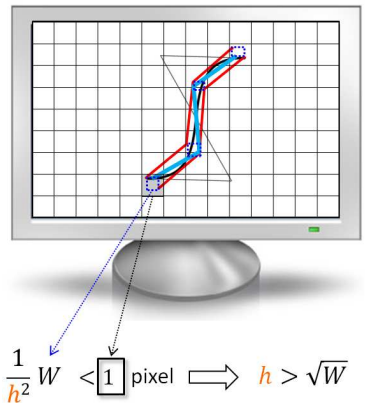

Fig. 10. Shrinkage of slefe under h-fold subdivision.

To guarantee that any error due to linearization is below pixel size and the depth threshold $\mathrm{tol}_{\mathbf{z}}$, we compute the width for low m, say $m = 2$ or 3, and then apply (7) to obtain a safe tessellation factor of

$$\tau_{\mathbf{p}} := \max\{\tau_{\mathrm{xy}}(m, \mathbf{p}), k_m \sqrt{\mathsf{w}_{\mathbf{z}}(m, \mathbf{p})}/\mathrm{tol}_{\mathbf{z}}\}. \tag{8}$$

Fig. 5 shows that the resulting triangles are, as hoped for, typically much larger than pixels and experiments confirm that (8) determines a near-minimal $\tau_{\mathbf{p}}$ in the sense that, for typical models, already a 10% decrease in $\tau_{\mathbf{p}}$ leads to pixel inaccuracy.

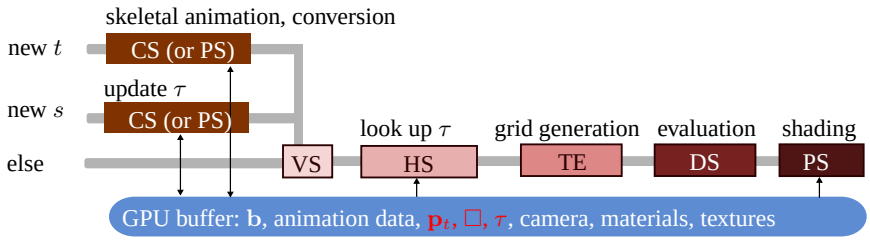

Fig. 11. Mapping of animation and conversion to curved surfaces to the DX11 graphics pipeline. CS=Compute Shader, VS=Vertex Shader, HS= Hull Shader, TE=Tessellation Engine, DS= Domain Shader, PS=Pixel Shader.

4 Algorithm and Implementation

The main costs, that our algorithm for rendering animated curved surfaces seeks to minimize, are the conversion of the mesh to the surface patch coefficients and rendering the patches with pixel accuracy. For details of the implementation of pixel accuracy as a Compute Shader pre-pass, we refer to [3]. The key to minimizing the conversion cost is to restrict conversion to pose changes of the animated character. The key to efficient pixel-accurate rendering is to integrate the control of the variance of the curved patch geometry from its triangulation, as just explained in Section 3, with the conversion to minimize overhead. Specifically, we split the work as follows.

- For every *pose* (geometry or mesh connectivity) change, re-compute the control mesh, all affected patches and slefe-boxes.
- For every *view* change, measure the width w_{xy} of the boxes' screen projections and their depth variance w_z.
- Determine the tessellation factor τ according to (8), i.e. a low as possible while still guaranteeing pixel-accuracy to make best use of the efficient rasterization stage on the GPU.

Pose and view change. To minimize conversion and τ computation cost, our implementation calls either two, one or no Compute Shader passes followed by a standard DX11 rendering pass. This is illustrated in Fig. 12 and the details are as follows.

(a) If the scene does not change in view or pose then the stored animated curved surface at time step t, $\mathbf{p}_t$, is rendered with the existing tessellation factors.

(b) For each view change at time step s that is not an animation step, the modelview transformations are applied to the saved $\mathbf{p}_t$ and the tessellation factors τ are updated to guarantee pixel-accuracy for the new viewpoint. Then (a) is executed.

(c) For each pose change (animation step t), the coefficients of the animated curved surface $\mathbf{p}_t$ are computed by executing the animation and conversion steps. The coefficients of $\mathbf{p}_t$ are stored in the GPU buffer. Then the slefe-boxes are re-computed and stored and the same computations are executed as in (b).

Throughout, only modified patches are updated.

Mapping to GPU Shader Code. In modern graphics APIs the triangulation density is set by up to six tessellation factors per surface patch. The two interior tessellation factors are set to τ, while the other four tessellation factors, corresponding to the boundaries, are set to the maximum of the interior factors of the patches sharing the boundary. This coordination in the Compute Shader pass guarantees a consistent triangulation by avoiding mismatch along boundaries between differently tessellated patches.

The pseudo-code of the Compute Shaders is given below. Detailed pseudo-code of pixel-accurate slefe-estimates is presented in Section 6 of [3]. The rendering pass is standard DX11 rendering.

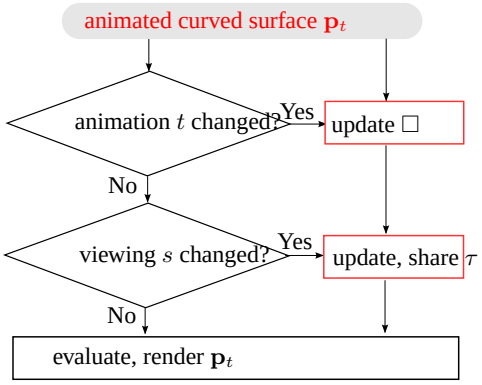

Fig. 12. Updating slefe-boxes □ and the tessellation factors τ is only required when the input mesh is animated or the view is changed.

The data flow outlined in Fig. 12 is made concrete by the following pseudocode. The mapping of the pseudocode to the DX11 graphics pipeline is shown in Fig. 11. Recall that each bi-cubic patch has $4 \times 4 = 16$ coefficients.

```
  function MAIN(t, s)
    if new t then COMPUTE_SHADER_POSE_CHANGE(t)
    end if
    if new s then COMPUTE_SHADER_VIEW_CHANGE(s)
    end if
  end function

[shared_mem cpts[16]]
[num_threads 16]
  function COMPUTE_SHADER_POSE_CHANGE(t)
    vtx_id ← thread_id + (patch_id * 16)
    SHAPE_KEY(vtx_id, t)
    SKELETAL_ANIMATION(vtx_id, t)
    CONVERT_TO_ACC(vtx_id)
```

```
    end function

[shared_mem width[16]]
[num_threads 16]
    function COMPUTE_SHADER_VIEW_CHANGE(s)
        vtx_id ← thread_id + (patch_id * 16)
        width[thread_id] ← project_slefe(vtx_id)
        synchronize_threads()
        if thread_id = 1 then
            TF ← pick_max_width(width)
            save_to_gpu(TF_buffer, patch_id, TF)
        end if
    end function

    function SHAPE_KEY(vtx_id, t)
        base_sk ← get_base_shape_key(vtx_id)
        shaped_vtx ← (0, 0, 0)
        for sk in shape_keys[vtx_id] do
            sk_wt ← get_shape_key_wt(vtx_id, sk, t)
            shaped_vtx += sk_wt * (sk.v[vtx_id] − base_sk.v[vtx_id])
        end for
        rest_vtx[vtx_id] ← shaped_vtx + base_sk.v[vtx_id]
    end function

    function SKELETAL_ANIMATION(vtx_id, t)
        tot_wt ← sum_influence_weights(vtx_id)
        final_mat ← zero_matrix(4, 4)
        for bone_i in influencing_bones[vtx_id] do
            posed_bone_mat ← pose_mat(bone_i, t)
            rest_bone_mat_inv ← rest_mat_inv(bone_i)
            bone_wt ← get_bone_wt(vtx_id, bone_i)/tot_wt
            final_mat += (rest_bone_mat_inv * posed_bone_mat * bone_wt)
        end for
        posed_vtx[vtx_id] ← rest_vtx[vtx_id] * final_mat
    end function

    function CONVERT_TO_ACC(vtx_id)
        cpts[thread_id] ← (0, 0, 0)
        for i ← 0 to stencil_size[vtx_id] do
            stencil_vtx ← posed_vtx[stencil_lookup[vtx_id, i]]
            cpts[thread_id]+=stencil_wt[vtx_id, i] * stencil_vtx
        end for
        normalize(cpts[thread_id])
        save_to_gpu(cpt_buffer, vtx_id, cpts[thread_id])
        synchronize_threads()
        slefe ← update_slefe(thread_id)
        save_to_gpu(slefe_buffer, vtx_id, slefe)
    end function
```

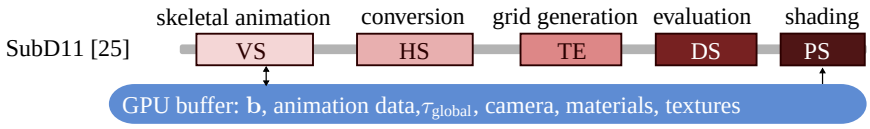

Fig. 13. DX11 SubD11 implementation [25]. CS=Compute Shader, VS=Vertex Shader, HS= Hull Shader, TE=Tessellation Engine, DS= Domain Shader, PS=Pixel Shader.

GPU processing	% of total
skeletal animation + conversion	58
slefe bounds	4
pose change total	62
view change	6
rendering pass	32

Table 2. Distribution of work per frame among pose, view and rendering. Pose change dominates.

5 Discussion and Comparison

Performance. Table 2 shows the work distribution of a rendering cycle. The pose change consists of mesh animation and conversion plus recomputation of slefe-box vertices. The pose change dominates the work, but the recomputation of the slefe bounds accounts for less than 4%. The slefe bounds and their projection make up ca 10% of the overall work. According to measurements in Section 7 of [3], the bounds are within 12% of the optimal for widely-used, representative test examples in computer graphics (the tessellation factor in the implementation of [3] was inadvertently scaled by $\sqrt{2}$). Given that tight bounds reduce work when accurate rendering is required, it is not surprising that 10% computational overhead buys a considerable speedup compared to the overtessellation of conservatively-applied heuristics.

We used an NVidia GeForce GTX 580 graphics card with Intel Core 2 Quad CPU Q9450 at 2.66GHz with 4GB memory to render the geometry of the movie Elephants Dream. Elephants Dream is a 10-minute-long animated movie whose source is open. In 2006 it was reported to have taken 125 days to render, consuming up to 2.8GB of memory for each frame in Full-HD resolution (1920×1080) [4]. That is, each frame took on the order of 10 minutes to render. Since the Elephants Dream character meshes of Proog and Elmo contain triangles, but ACC requires a quadrilateral input mesh, we applied the standard cure of one step of Catmull-Clark subdivision yielding 140,964 curved surface patches for Proog and Emo together. In our implementation, we replicated Elephants Dream except that we did not apply post effects so as to isolate the effect of improved patch rendering. The 141K textured bi-cubic ACC patches render at over 300 frames per second (fps) with full pixel-accuracy. (We also used a variant of ACC that avoids the increase in patches and rendered 32K quads and 350 triangles at 380 fps when animating every frame and 1100 fps when animating at 33 frames per

second.) For comparison, the SubD11 demo scene in Fig. 14 has 4K quadrilaterals and its frame rate varies with the user-set tessellation factor TF (see upper left of Fig. 14) between 250 fps at the coarsest level $TF = 1$ and 23 fps at TF$= 64$. For a detailed analysis of how model size, screen size, etc. affect pixel-accurate rendering see [3].

Memory usage and data transfer. By placing the animation and the conversion from the quad mesh to the Bézier patches onto the GPU, the approach is memory efficient and minimizes data transfer cost. For example, one frame in the Proog and Emo scene has up to 0.25 million bi-cubic Bézier patches requiring 206.5 MB of GPU memory. Traditional CPU-based animation would transfer this amount of data to the graphics card at every frame. In our approach, for the same scene, just once at startup, the static mesh of 4MB plus 9MB of shape key data are transferred; also the skeletal animation data per frame (45kB for 684 'bones') and the 289 shape keys (1kB) are packed into GPU buffers at startup. Moreover, the near-optimal *ephemeral triangulation* via the tessellation engine saves space and transfer cost compared to massive, 'pre-baked' triangulations.

Relation to Micro-polygonization. An established alternative for high-quality rendering, used in 3D movie animation, is micro-polygonization. *Micro-polygonization* owes its prominence to the Reyes rendering framework [37]. Since canonical implementations of micro-polygonization are recursive (cf. [5]), micro-polygonization is harder to integrate with current graphics pipelines [38] and leads to multiple passes as refinement and testing are interleaved. Even on multiple GPUs, there is a trade-off between real-time performance and rendering quality [39] (RenderAnts). Micro-polygonization aims to tessellate the domain U of a patch into (u, v) triangles so that the *size* of the screen projection of their image triangles is less than half a pixel. By contrast, pixel-accurate rendering aims at minimally partitioning the patches, just enough so that the difference, under projection, between the triangulated surface and the true non-linear surface is less than half a pixel: pixel-accurate rendering forces the *variance*, between the displayed triangulated surface and the exact screen image, to below the visible pixel threshold.

Comparison with the DX11 ACC SubD11 distribution. Our implementation is similar to that of SubD11 [25]: both implement skeletal animation and apply mesh conversion by accessing a 1-ring neighborhood of each quadrilateral. However, our implementation uses a sequence of Compute Shaders to animate and convert while SubD11 uses the Vertex Shader and the Hull Shader. See Fig. 11 for the execution pipeline of our algorithm and compare to that of SubD11, Fig. 13.

Since SubD11 executes in a single pass it appears to be more efficient. However, the Vertex Shader (VS) animation and Hull Shader (HS) conversion that perform the bulk of the work in SubD11need to be synchronized by the index buffer mechanism to prevent conversion before every vertex of a surface patch is animated; and SubD11 does not support interactive adaptive tessellation (without cracks) and must re-execute animation and conversion steps even when no view or pose change occur.

In our approach the main work, apart from rendering, is executed in the Compute Shader (CS). This automatically provides the necessary synchronization and allows coordination for interactive GPU-based *adaptive* tessellation without cracks. Using the Compute Shader also allows saving partial work in the GPU buffer (the animated surface $\mathbf{p}_t$ and the tessellation factors τ) and thereby reduces data transfer and commu-

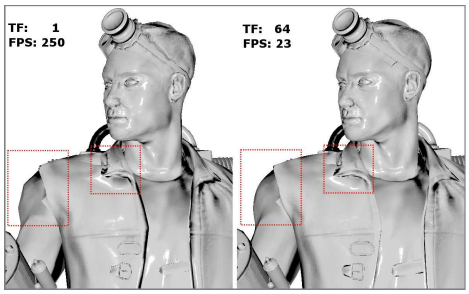

Fig. 14. DX11 SubD11 model from [25] consisting of 3,749 ACC patches (plus 150,108 flat triangles). The screen is captured at 1440x900 resolution. Setting $TF = 1$ results in polyhedral artifacts, at the shoulder and neck, while setting it high to remove these artifacts, decreases the frames per second by an order of magnitude.

nicates edge tessellation factors for adaptive rendering without mismatch. Executing only the appropriate type of the CS avoids re-computation, and guarantees sub-pixel accuracy. The end of the next section compares timings. A further advantage of using the Compute Shader is that it allows an indexed list rather than a fixed-size array when accessing neighbors. The Hull Shader limitation on primitives in SubD11 constrains the vertex valence, i.e. the number of points that can be accessed to construct the ACC patches. This matters for Proog and Emo models which contain 256 vertices of valence 32.

Compute Shader vs. Pixel Shader. We explored executing animation and τ-computation in a Pixel Shader (PS) pass. For large data sets, our CS implementation was clearly more efficient (see Table 3; Note that the CS has less overhead than a extra pass.). This can partly be attributed to higher parallelism: we can use 16 threads per patch in the CS as opposed to one per patch on the PS. (We could use 16 pixels in the PS, but would then have to synchronize to be able to compute τ). We also tried to use the Hull Shader (HS). But not only is the HS computationally less efficient on current hardware, but the HS also can not provide the necessary communication of adaptive tessellation factors to neighbor patches. The rightmost column VS* of Table 3 shows that just executing the animation in the Vertex Shader is already slower than executing animation and conversion in the CS. This explains why our code is considerably faster than SubD11, even though our code guarantees sub-pixel accuracy while SubD11 does not.

Anim Updates/Sec	CS	PS	VS*
33	311	184	253
every frame	130	53	75

Table 3. Performance in frames per second when placing animation and computation of τ onto the **CS or PS** or* just the animation onto the VS.

6 Conclusion

To optimally leverage the approach to pixel-accurate rendering of [3] to skeleton-based animation, we partitioned the work for pixel-accurate rendering into stages that match animation-dependent transformations and view-time dependent camera motions. This allocation is as natural as it is practically powerful: it allows us to combine interactive animation with high-quality rendering of curved surfaces. For gaming and animation it is crucial to spend minimal effort in redrawing static images since many other operations, say physics simulations, compete for compute resources. Also, in the game setting, the user often pauses to react to new information – so there is not continuous animation. The result is accurate for the given bi-cubic patches – distortion is below half a pixel, i.e. the error is not visible; it is efficient – there is no recursion and triangles are of maximal size; the adaptation is automatic – there is no need for manually setting the level of detail; and our implementation is fast, rendering 141k patches at more than 300 frames per second.

We tested the framework by rendering scenes of the movie Elephants Dream at $10\times$ real-time, leaving enough slack for larger data sets, complex pixel shaders and the artists' other work. Since the final pass is a generic DX11 rendering pass, it is fully compatible with displacement mapping (not used in Elephants Dream) and post effects. (We are not claiming pixel-accurate displacement, since this notion is not well-defined: displacement maps prescribe discrete height textures that require interpretation.) The rendering speed can provide high visual quality under interactive response. This may be useful for interactive CAD/CAM design in that the user no longer has to guess a suitable level of triangulation.

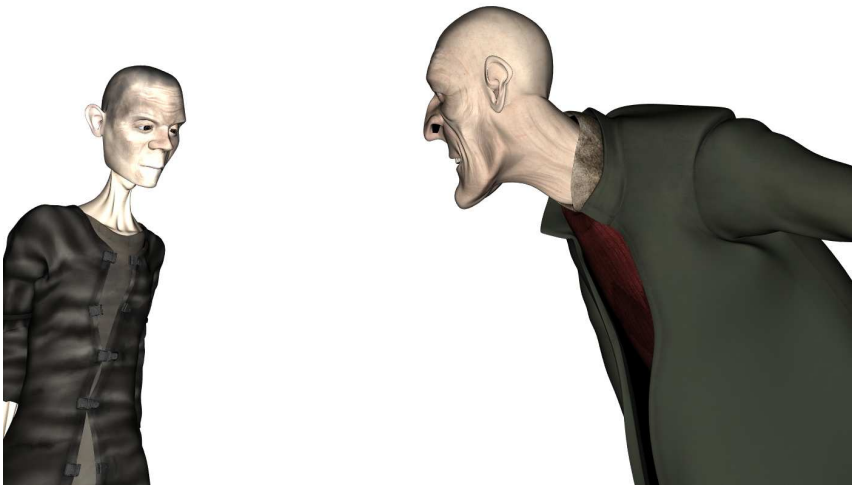

Fig. 15. Proog and Emo scene rendered in 7 seconds by Blender on a Intel Core 2 Duo CPU at 2.1GHz with 3GB memory; and in 3×10^{-3} seconds by our GPU algorithm.

Acknowledgements

This work was supported in part by NSF Grant CCF-1117695. We thank the contributors to Elephants Dream for creating this wonderful resource and the creators of SubD11 to provide source code and model. Georg Umlauf's insightful question after the conference presentation prompted the inclusion of the constant k_m in the paragraph following (6).

References

1. T. DeRose, M. Kass, T. Truong, Subdivision surfaces in character animation, in: Proceedings of the 25th annual conference on Computer graphics and interactive techniques, SIGGRAPH '98, ACM, New York, NY, USA, 1998, pp. 85–94.
2. J. Peters, U. Reif, Subdivision Surfaces, Vol. 3 of Geometry and Computing, Springer-Verlag, New York, 2008.
3. Y. I. Yeo, L. Bin, J. Peters, Efficient pixel-accurate rendering of curved surfaces, in: Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games, I3D '12, ACM, New York, NY, USA, 2012, pp. 165–174. doi:10.1145/2159616.2159644. URL http://doi.acm.org/10.1145/2159616.2159644
4. Blender, Foundation, Elephants dream, http://orange.blender.org (2006).
5. M. Fisher, K. Fatahalian, S. Boulos, K. Akeley, W. R. Mark, P. Hanrahan, DiagSplit: parallel, crack-free, adaptive tessellation for micropolygon rendering, ACM Transactions on Graphics 28 (5) (2009) 1–8.
6. M. Nießner, C. T. Loop, M. Meyer, T. DeRose, Feature-adaptive GPU rendering of Catmull-Clark subdivision surfaces, ACM Trans. Graph 31 (1) (2012) 6.
7. D. Filip, R. Magedson, R. Markot, Surface algorithms using bounds on derivatives, Computer Aided Geometric Design 3 (4) (1986) 295–311.
8. M. Guthe, A. Balázs, R. Klein, GPU-based trimming and tessellation of NURBS and T-Spline surfaces, ACM Transactions on Graphics 24 (3) (2005) 1016–1023.
9. J. Hjelmervik, Hardware based visualization of b-spline surfaces, presentation, Eighth International Conference on Mathematical Methods for Curves and Surfaces Oslo, June 28 July 3, 2012.
10. N. Magnenat-Thalmann, R. Laperrière, D. Thalmann, Joint–dependent local deformations for hand animation and object grasping, in: Graphics Interface '88, 1988, pp. 26–33.
11. F. Cordier, N. Magnenat-Thalmann, A data-driven approach for real-time clothes simulation, Computer Graphics Forum 24 (2) (2005) 173–183.
12. L. Kavan, S. Collins, J. Zára, C. O'Sullivan, Geometric skinning with approximate dual quaternion blending, ACM Trans. Graph. 27 (2008) 105:1–105:23.
13. T. Ju, S. Schaefer, J. D. Warren, Mean value coordinates for closed triangular meshes, ACM Trans. Graph 24 (3) (2005) 561–566.
14. K. Zhou, X. Huang, W. Xu, B. Guo, H.-Y. Shum, Direct manipulation of subdivision surfaces on GPUs, ACM Trans. Graph 26 (3).
15. P. Joshi, M. Meyer, T. DeRose, B. Green, T. Sanocki, Harmonic coordinates for character articulation, ACM Trans. Graph 26 (3) (2007) 71.
16. Y. Lipman, D. Levin, D. Cohen-Or, Green Coordinates, ACM Transactions on Graphics 27 (3) (2008) 78:1–.
17. Blender, Foundation, Shape keys, http://wiki.blender.org/index.php/Doc:2.4/Manual/Animation/Techs/Shape/Shape_Keys.

18. A. Myles, T. Ni, J. Peters, Fast parallel construction of smooth surfaces from meshes with tri/quad/pent facets, Computer Graphics Forum 27 (5) (2008) 1365–1372.
19. Y. I. Yeo, T. Ni, A. Myles, V. Goel, J. Peters, Parallel smoothing of quad meshes, The Visual Computer 25 (8) (2009) 757–769.
20. C. T. Loop, S. Schaefer, Approximating Catmull-Clark subdivision surfaces with bicubic patches, ACM Trans. Graph 27 (1).
21. C. Loop, S. Schaefer, T. Ni, I. Castano, Approximating subdivision surfaces with Gregory patches for hardware tessellation, ACM Trans. Graph. 28 (2009) 151:1–151:9.
22. J. Bolz, P. Schröder, Rapid evaluation of Catmull-Clark subdivision surfaces, in: Web3D '02: Proceeding of the seventh international conference on 3D Web technology, ACM Press, New York, NY, USA, 2002, pp. 11–17.
23. M. Bunnell, GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation, Addison-Wesley, Reading, MA, 2005, Ch. 7. Adaptive Tessellation of Subdivision Surfaces with Displacement Mapping.
24. M. Nießner, C. T. Loop, G. Greiner, Efficient evaluation of semi-smooth creases in catmull-clark subdivision surfaces, 2012, p. 4.
25. MicroSoft, Subd11 sample (direct3d11), `http://preview.library.microsoft.com/en-us/library/ee416576` (November 2008).
26. A. Vlachos, J. Peters, C. Boyd, J. L. Mitchell, Curved PN triangles, in: 2001, Symposium on Interactive 3D Graphics, Bi-Annual Conference Series, ACM Press, 2001, pp. 159–166.
27. I. Boier-Martin, D. Zorin, Differentiable parameterization of Catmull-Clark subdivision surfaces, in: R. Scopigno, D. Zorin (Eds.), Symp. on Geom. Proc., Eurographics Assoc., Nice, France, 2004, pp. 159–168.
28. L. He, C. Loop, S. Schaefer, Improving the parameterization of approximate subdivision surfaces, in: C. Bregler, P. Sander, M. Wimmer (Eds.), Pacific Graphics, 2012, pp. xx–xx.
29. J. Peters, Mid-structures of subdividable linear efficient function enclosures linking curved and linear geometry, in: M. Lucian, M. Neamtu (Eds.), Proceedings of SIAM conference, Seattle, Nov 2003, Nashboro, 2004.
30. D. Lutterkort, Envelopes of nonlinear geometry, Ph.D. thesis, Purdue University (Aug. 2000).
31. D. Lutterkort, J. Peters, Tight linear bounds on the distance between a spline and its B-spline control polygon, Numerische Mathematik 89 (2001) 735–748.
32. D. Lutterkort, J. Peters, Optimized refinable enclosures of multivariate polynomial pieces, Computer Aided Geometric Design 18 (9) (2002) 851–863.
33. J. Peters, X. Wu, On the optimality of piecewise linear max-norm enclosures based on slefes, in: L. L. Schumaker (Ed.), Proc. Curves and Surfaces, St Malo 2002, Vanderbilt Press, 2003.
34. X. Wu, J. Peters, Interference detection for subdivision surfaces, Computer Graphics Forum, Eurographics 2004 23 (3) (2004) 577–585.
35. X. Wu, J. Peters, An accurate error measure for adaptive subdivision surfaces, in: Proceedings of The International Conference on Shape Modeling and Applications 2005, 2005, pp. 51–57.
36. X. Wu, J. Peters, Sublime (subdividable linear maximum-norm enclosure) package, `http://surflab.cise.ufl.edu/SubLiME.tar.gz`, accessed Jan 2011 (2002).
37. R. L. Cook, L. Carpenter, E. Catmull, The Reyes image rendering architecture, in: M. C. Stone (Ed.), Computer Graphics (SIGGRAPH '87 Proceedings), 1987, pp. 95–102.
38. K. Fatahalian, S. Boulos, J. Hegarty, K. Akeley, W. R. Mark, H. Moreton, P. Hanrahan, Reducing shading on GPUs using quad-fragment merging, ACM Trans. Graphics, 29(3), 2010 (Proc. ACM SIGGRAPH 2010) 29 (4) (2010) 67:1–8.
39. K. Zhou, Q. Hou, Z. Ren, M. Gong, X. Sun, B. Guo, Renderants: interactive Reyes rendering on GPUs, ACM Trans. Graph 28 (5).