Efficient Pixel-Accurate Rendering of Curved Surfaces

Young In Yeo*
University of Florida

Lihan Bin†
Advanced Micro Devices

Jörg Peters‡
University of Florida

Abstract

A curved or higher-order surface, such as spline patch or a Bézier patch, is rendered pixel-accurate if it displays neither polyhedral artifacts nor parametric distortion. This paper shows how to set the evaluation density for a patch just finely enough so that parametric surfaces render pixel-accurate in the standard graphics pipeline. The approach uses tight estimates, not of the size under screen-projection, but of the *variance* under screen projection between the exact surface and its triangulation. An implementation, using the GPU tessellation engine, runs at interactive rates comparable to standard rendering.

1 Introduction

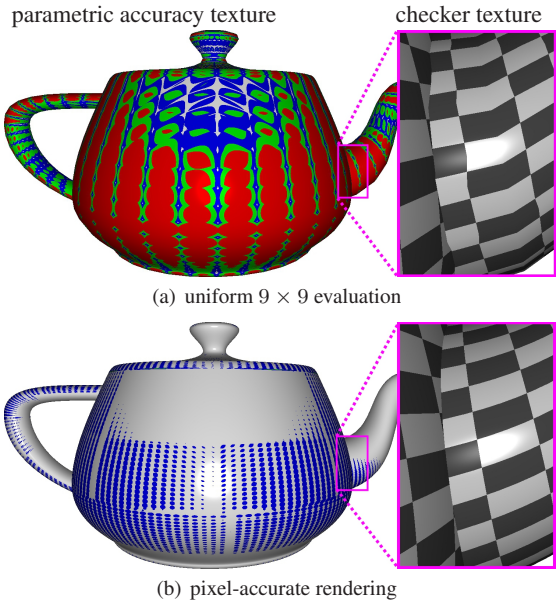
When a model uses curved surfaces, the designer is typically called upon to set one or several levels of tessellation for display. But in an interactive setting, where the camera is free to zoom in or out of the scene, no fixed set of levels can avoid faceted display or overtessellation. While a number of smart heuristics are being used to adjust the tessellation density, based on viewing and surface properties, it is desirable to guarantee accurate rendering (and avoid overtessellation). To deliver such guarantees, this paper makes precise the notion of pixel-accurate rendering: pixels are to be controlled by one or more non-occluded patches projecting to it (covering accuracy) and have correct associated domain parameters (parametric accuracy – cf. Fig. 1a).

Deriving an algorithm for safe yet efficient tessellation starts with the observation that the existing graphics pipeline is efficient and accurate for polyhedral surfaces. Therefore the evaluation density (tessellation factor) needs only guarantee that the *variance* between the image of the triangulation of the surface and the true surface is not perceptible on a pixel-based screen. To be most efficient for the existing graphics pipeline, the evaluation density should be *minimal* – just so that the error stays below the visible (pixel) threshold.

Guaranteeing sub-pixel variance represents a different approach to accurate rendering than micro-polygonization. Consider a uniform, flat bicubic patch, orthogonal to the viewing direction and covering the whole screen. Micro-polygonization recursively splits such a patch ten times in each direction to bound the pixel-*extent* of the projection. A variance-based approach need not split the patch at all to yield pixel-accuracy.

Overview After the literature review, Section 3 defines pixel-accuracy. Introducing a mechanism for setting the tessellation density requires showing on one hand that the setting guarantees accuracy and, on the other, that it is not overly conservative. Section 4 explains known tight enclosures for splines and Section 5 their novel use (9) that ensures pixel-accuracy without ever computing the enclosures. Section 6 specifies an algorithm and its DX11 implementation to measure performance on realistic data sets (Section 7). Section 8 lays out implementation trade-offs. An accompanying video shows non-instanced 'killeroos' (100×3042 patches) pixel-accurately rendered at 150 frames-per-second and 100 non-instanced 'monster frogs' (100×1292 patches) at 310 fps.

*e-mail: yiyeo@cise.ufl.edu
†e-mail: Lihan.Bin@amd.com
‡e-mail: jorg@cise.ufl.edu



(a) uniform 9×9 evaluation

(b) pixel-accurate rendering

Figure 1: *Visualizing* **parametric (in)accuracy**, *one of two criteria for judging the rendering of curved surfaces. Green and red pixels are inaccurate as follows. Each pixel has an associated u, v parameter and patch* $\mathbf{p}$. *The pixel is colored [grey, blue, green] if the distance of the pixel center to the screen projection of* $\mathbf{p}(u, v)$ *is at most* $[0.1, 0.5, 1]$ *pixel widths. If it is red, the distance is greater than 1. The enlargement shows the effect of parametric inaccuracy on a checkered texture. The parametric distortion in (a) causes incorrect horizontal kinks and vertical oscillations.*

2 Related Literature

The variance-based approach to pixel-accurate rendering requires tight bounds on the deviation from linearity (rather than the size of the projected image) and a mechanism to predict the decrease of the deviation when increasing the tessellation density. Such exact prediction allows us to avoid recursive splitting-and-reassessment of patch bounds.

LoD and Bounding Constructs. Setting tessellation density is related to level-of detail schemes for polyhedral models (see e.g. the survey [Xia et al. 1997]). It faces the same trade-off of bounding construct efficiency vs. tightness (cf. Fig. 2) and has to consider the cost of initialization, modification (e.g. under rotation or deformation) and testing. Treatment of 'popping artifacts' and choice of simplification operator [Hoppe 1996; Gross et al. 1995] however

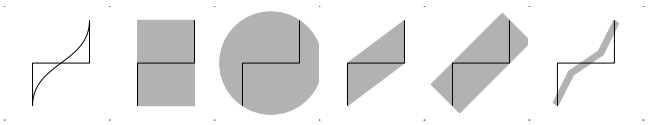


Figure 2: Bounding constructs *for a cubic curve with a step-like control polygon. Here less grey means better! Curve and control polygon, min-max or axis-aligned bounding box (AABB), bounding disk, convex hull (here equal to an 8-dop), oriented bounding box,* $m = 3$*-piece* slefe.

is automated by the continuous refinement in *hardware tessellation* [Drone et al. 2008]. Higher-order surfaces such as NURBS (B-spline), Bézier patches and subdivision surfaces, can be enclosed by their polyhedral control net (see e.g. [Farin 1988]). The well-known estimates of Filip et al. [Filip et al. 1986] (see also [Sheng and Hirsch 1992; Tookey and Cripps 1997; Guthe et al. 2005]) bound the difference of a piecewise linear interpolant at the domain corners to a smooth function piece p in terms of a sum of all mixed second order derivatives $\partial_1^i \partial_2^j p$, $i + j = 2$. However, already for a simple example such as Fig. 3, this bound is 10 times larger than the more specialized slefe bounding structure defined in [Lutterkort 2000; Peters 2004]. We will use the estimates provided by slefes in a novel way in Section 5, without ever constructing slefes.

Micropolygonization. In the split-and-dice phase of the Reyes architecture [Cook et al. 1987] higher-order surfaces are recursively partitioned and tested until they qualify for uniform partition (dicing) into micro-polygons. Micro-polygons are expected to project to no more than a quarter pixel. Split-and-dice presents a challenge for parallel execution, both because of work load imbalance and an *a priori* unknown depth of recursion. A number of recent publications have focused on micro-polygonization on the GPU using either the min-max AABB bound or an image space edge-length heuristic to estimate the extent of the projection. In *Real-time Reyes-style adaptive surface subdivision*, Patney and Owens [Patney and Owens 2008] ported the split-and-dice stage on the GPU using CUDA. A *Data-Parallel Rasterization of Micro-polygons* [Fatahalian et al. 2009] improves micro-polygon rasterization and [Fisher et al. 2009] improves split-and-dicing with the Diagsplit. Figure 10 of [Fisher et al. 2009] gives a comparison of the accuracy of splitting heuristics. [Eisenacher and Loop 2010] report micro-polygon generation at 30-50% of the speed of standard rasterization. *RenderAnts* by Kun Zhou et al. [Zhou et al. 2009] implements the complete Reyes-pipeline based on a special GPGPU programming language, called BSGP (see also [Tzeng et al. 2010]). Load-balancing within various stages addresses some of the challenges of split-and-dice. For supersampling and anti-aliasing, [Fatahalian et al. 2010] point to the problem of many tiny fragments that challenge memory access and SIMD efficiency and suggest improving shader efficiency by merging micro-polygons.

3 Pixel-accuracy

According to [Cook et al. 1987] the goal of micro-polygonization is to generate a base representation that is natural in the sense that uv-axis-aligned grids of micro-polygons map to surface pieces for which no inverse perspective calculation is necessary. We make this goal precise by distinguishing two components of pixel-accuracy: covering accuracy and parametric accuracy. Below, a pixel, the smallest screen unit that can be controlled, is defined as a half-open square $\left[\begin{smallmatrix}\mathrm{x}\\ \mathrm{y}\end{smallmatrix}\right] + \frac{1}{2}[-1,1)^2$ centered at $\left[\begin{smallmatrix}\mathrm{x}\\ \mathrm{y}\end{smallmatrix}\right] \in \mathbb{N}^2$, $P : \mathbb{R}^3 \to \mathbb{R}^2$ is the projection to the screen and $\| \begin{smallmatrix} a \\ b \end{smallmatrix} \|_\infty := \max\{|a|, |b|\}$.

Covering accuracy requires that each pixel's output value be controlled by one or more unoccluded pieces of patches whose projection overlaps it sufficiently. – Here 'sufficiently' qualifies overlap with respect to alternative (anti-aliasing) sampling strategies, 'control' means that the pixel shader can take the unoccluded piece into consideration when determining the output value (color), and 'one or more' accounts both for semi-transparency and for multiple pieces' projections partially overlapping the pixel. Inaccurate covering due to triangulation and rasterization leads to non-smooth silhouettes (showing occluded background or controlling output without overlap), false intersection lines, and incorrect depth-ordering (where an originally curved and front-most fragment looses in the depth-buffer comparison). In addition to incorrect display, covering inaccuracy can result in noise or pixel dropout.

Parametric accuracy requires that for all pixels the following holds. Let $\left[\begin{smallmatrix}\mathrm{x}\\ \mathrm{y}\end{smallmatrix}\right]$ be the pixel's center and let $\mathbf{p}, u, v$ be an associated surface $\mathbf{p} : \mathbb{R}^2 \to \mathbb{R}^3$ and (u, v) parameter pair. Then the surface point $\mathbf{p}(u, v) \in \mathbb{R}^3$ must project into the pixel:

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

– Parametric accuracy makes precise the criterion stated in Reyes, that uniform traversal of the domain results in a near-uniform traversal of the screen-projected image. Even when the pixel is covered by the proper surface piece, too coarse a triangulation can lead to *parametric distortion*, i.e. $P(\mathbf{p}(u, v))$ lies outside the pixel associated with parameters (u, v). Since (u, v) are used for texture look-up, to evaluate surface properties such as normals and to determine displacement and procedural shaders this can cause a multitude of artifacts incompatible with accurate rendering. Parametric inaccuracy is color-encoded in Fig. 1a. The resulting texture distortion is shown in the enlargement.

Standard processing in the graphics pipeline can be pixel-accurate. When the screen projection of a spline patch is sufficiently uniform in the parameters and its depth varies sufficiently little. Fig. 12 shows how pieces much larger than a micro-polygon can be rasterized without loss of pixel-accuracy.

4 Patch representation, function enclosure

To take advantage of the existing, efficient graphics pipeline, we want to determine the coarsest partition that guarantees pixel-accuracy when rasterizing. Unlike micro-polygonization, the goal is to control the *variance of the projection* between exact and triangulated surface, not the size of the projection. This section explains slefes whose estimates are used in a novel way in Section 5 to ensure pixel-accuracy.

Parametric representation The common higher-order surface representations are Bézier patches (*glMap2* in OpenGL), NURBS patches (*gluNurbsSurface* in OpenGL) and subdivision surfaces. Subdivision surfaces are splines with singularities [Peters and Reif 2008] and can be treated as nested rings of spline patches. NURBS are easily converted to polynomial pieces in tensor-product Bézier-form (5), by the well-known stable technique called 'knot insertion'. Below we therefore focus on Bézier patches. Analogous estimates apply directly to NURBS and subdivision surfaces.

A polynomial p of degree d in the variable u and with coefficients $c_j \in \mathbb{R}$ has the Bézier-form

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

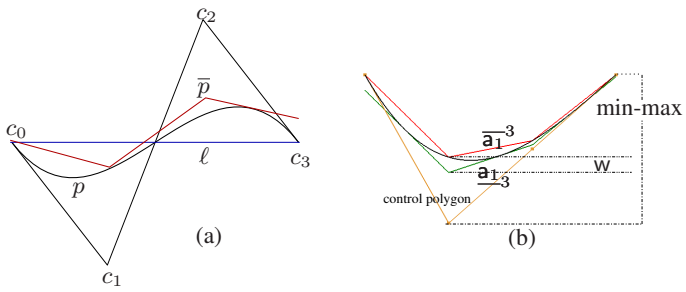


Figure 3: *The* slefe**-construction** *from [Peters 2004]. (a) The function $p(t) := -b_1^3(t) + b_2^3(t)$ and its upper bound $\overline{p}$. (b) The lower bound $\underline{\mathsf{a}_1}_3$ and the upper bound $\overline{\mathsf{a}_1}^3$ tightly sandwiching the function $\mathsf{a}_1 := -\frac{2}{3}b_1^3(t) - \frac{1}{3}b_2^3(t)$, using $m = 3$ segments. Table 1 shows $\mathsf{w} = \max_{[0..1]} \overline{p} - \underline{p}$ to be < 0.07. The corresponding number for [Filip et al. 1986] (not illustrated) is $\frac{6}{8} = 0.75$ and for the min-max-bound $\frac{2}{3}$.*

Subdividable Linear Efficient Function Enclosures (short: slefes, pronounced sleeves) [Lutterkort 2000; Peters 2004] tightly sandwich non-linear functions, such as polynomials, splines and subdivision surfaces, between simpler, piecewise linear, upper and lower functions:

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

Specifically, [Lutterkort 2000] shows that (cf. Fig. 3, *left*)

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

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 $\nabla_j^2 p := -d(c_{j-1} - 2c_j + c_{j+1})$ 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. 3b shows the min-max or AABB bound to be looser by an order of magnitude than the width $\mathsf{w} := \max_{t\in[0..1]} \overline{p}(t) - \underline{p}(t)$ of slefes. The next paragraph shows that slefes are inexpensive to compute.

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

Table 1: *Values at breakpoints of a $m = 3$-piece* slefe.

Pre-computation Being piecewise linear the bounding functions $\overline{\mathsf{a}_j^d}^m$ and $\underline{\mathsf{a}_j^d}_m$ in (2) are defined by their values at the uniformly spaced break points. These values can be pre-computed. Table 1 lists all numbers needed to compute Fig. 3, e.g. the values $-.370370..$ and $-.439891..$ at $t = 1/3$. This table and the tables for higher degree can be downloaded [Wu and Peters 2002]. Due to symmetry, for example $\mathsf{a}_2^3(1-t) = \mathsf{a}_1^3(t)$, only one half of the $d-1$ tables are needed, i.e. just 1 for $d = 3$.

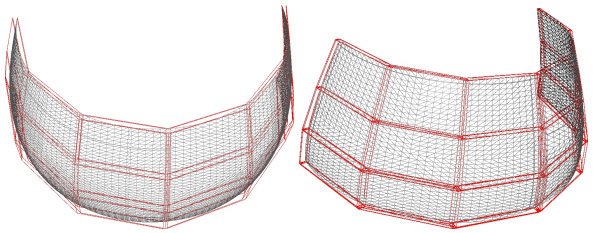


Figure 4: slefe**-tiles** *formed by four* slefe*-boxes locally enclose the surface (but are never explicitly computed).*

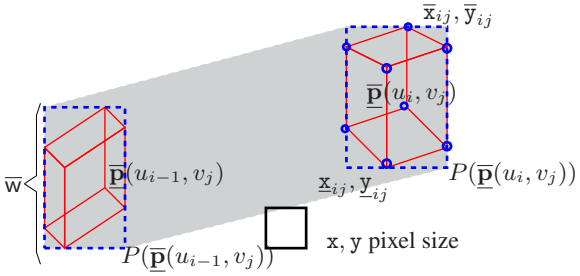


Figure 5: Projected slefe**-boxes.** *The projected* slefe*-boxes (red) are enclosed by axis-aligned rectangles (blue, dashed) whose convex hull (grey area) encloses the image (here of $\mathbf{p}([u_{i-1}..u_i], v_j)$). The (square-root of the) maximal edge-length of the rectangles, in pixel size, determines the tessellation factor $\tau_{\mathbf{p}}$.*

Bound Improvement under Refinement Consider u_i equally spaced in U and let

$$\mathsf{w}(m, U, p) := \max_{i=0,\ldots,m} \overline{p}(u_i) - \underline{p}(u_i). \tag{3}$$

Restricting the U-domain from $[0..1]$ to $[u, u+h]$, $h << 1$, reduces the maximal second difference

$$|\nabla^2 p| := \max_{i=1..d-1} |\nabla^2 c_i| \in \mathbb{R} \quad \text{to} \quad h^2 |\nabla^2 p|.$$

Comparing with (2) and Definition (3), we see that

$$\mathsf{w}(m, [u, u+h], p) \le h^2\, \mathsf{w}(m, [0, 1], p). \tag{4}$$

That is, partitioning the u-domain into $1/h$ segments, and re-representing the function over the smaller interval before applying the bound, scales w by h^2.

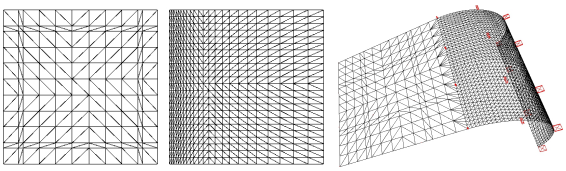
Tensor-product Surface patches are polynomial surface pieces in tensor-product Bézier-form of degree d_1, d_2 in the variables $(u, v) \in [0..1]^2$,

$$p(u, v) := \sum_{i=0}^{d_1} \sum_{j=0}^{d_2} c_{ij} b_j^{d_2}(v) b_i^{d_1}(u). \tag{5}$$

Appendix 10 shows how to use bounds in one variable to bound tensor-product patches.

5 Estimates ensuring pixel-accuracy: Bounding the Variance of Surfaces from their Triangulations

The slefes discussed above are for functions, i.e. one coordinate of the image. Now we consider a patch $\mathbf{p} : \mathbb{R}^2 \to \mathbb{R}^3$ with three



(a) parametric accuracy (b) covering accuracy

Figure 6: Adaptation of $\tau_{\mathbf{p}}$ to parametrization and geometry. *(a) Both patches are flat. The left patch is uniformly parametrized, the right non-uniformly with higher density for low x values. To guarantee parametric accuracy, variance-based tessellation $\tau_{\mathbf{p}}$ is higher for the right patch. (b) The left patch is flat, the right curves. To guarantee covering accuracy, variance-based tessellation is higher for the patch on the right where some* slefe*-boxes are shown in red.*

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), $\overline{\mathbf{p}_{ij}} := \overline{\mathbf{p}(u_i, v_j)}$ and $\underline{\mathbf{p}_{ij}} := \underline{\mathbf{p}(u_i, v_j)}$, we have a slefe-*box*

$$\overline{\underline{\mathbf{p}}}(u_i, v_j) := \frac{\overline{\mathbf{p}_{ij}} + \underline{\mathbf{p}_{ij}}}{2} + [-\frac{1}{2}..\frac{1}{2}]^3(\overline{\mathbf{p}_{ij}} - \underline{\mathbf{p}_{ij}}). \qquad (6)$$

Here $[-\frac{1}{2}..\frac{1}{2}]^3$ is the $\mathbf{0}$-centered unit cube. The slefe-box is an axis-aligned box in $\mathbb{R}^3$ (see red boxes in Fig. 5 and 6b) centered at the average of upper and lower values. Unlike exact surface points, which yield at best an estimate of the true surface in their immediate neighborhood, slefe-*tiles* spanned by four neighboring boxes $\overline{\underline{\mathbf{p}}}(u_i, v_j), i \in \{k, k+1\}, j \in \{\ell, \ell+1\}$ (cf. Fig. 4) tightly enclose the patch restricted to the domain rectangle with corners (u_i, v_j), $i \in \{k, k+1\}, j \in \{\ell, \ell+1\}$. The slefe-tiles are similar, but typically tighter than the convex hull of the control points (cf. Fig. 2). The tightness of the slefe-tiles implied by the size of the slefe-boxes will be crucial since looser bounds will force us to tessellate the surface more finely. However, the slefe-tiles in Fig. 4 are for exposition only. We will never compute slefe-tiles, as one might do for collision detection, because we are only interested in the difference between the screen projection of the nonlinear patch and its linear approximation.

Width To measure parametric accuracy, we define the minimal screen-coordinate-aligned rectangle that encloses the screen projection $\left[\begin{smallmatrix}\mathtt{x}\\ \mathtt{y}\end{smallmatrix}\right] := P(\overline{\underline{\mathbf{p}}}(u_i, v_j))$ of to the slefe-box with index i, j (see the blue dashed rectangles in Fig. 5):

$$q_{ij} := [\underline{\mathtt{x}}_{ij}..\overline{\mathtt{x}}_{ij}] \times [\underline{\mathtt{y}}_{ij}..\overline{\mathtt{y}}_{ij}] \supsetneq P(\overline{\underline{\mathbf{p}}}(u_i, v_j)). \qquad (7)$$

The maximal edge length over all q_{ij} is the *width*:

$$\overline{\mathtt{w}}(m, U, \mathbf{p}) := \max_{\substack{i=0,\ldots,m\\ j=0,\ldots,m}} \max\{\overline{\mathtt{x}}_{ij} - \underline{\mathtt{x}}_{ij}, \overline{\mathtt{y}}_{ij} - \underline{\mathtt{y}}_{ij}\}. \qquad (8)$$

The first argument of the width is the tessellation density m, the second the patch domain, e.g. $U := [0..1]^2$ for Bézier patches, and the third the patch. The width is a close upper bound on the variance from linearity in the parameterization. The width shrinks to zero when the parameterization becomes linear.

Decrease of Width with Denser Tessellation: Predicting Sufficient Tessellation To guarantee parametric accuracy, we want to uniformly partition the u, v-domain so that the width is below pixel-size. For some small m, say $m = 3$, we estimate the width $\overline{\mathtt{w}}$ as in (8). To reduce the width below one pixel, we need to determine the tessellation factor $\overline{\tau} \in \mathbb{R}$ so that $\overline{\mathtt{w}}(\overline{\tau}, [0..1], \mathbf{p}) < 1$. We apply (4) to see that if each piece of the m-times partitioned domain were further partitioned $1/h := \sqrt{\overline{\mathtt{w}}(m, [0..1], \mathbf{p})}$ times then the predicted width of the resulting piecemeal enclosures (which we never compute!) guarantees that the variance of the projection between exact and triangulated surface differs by less than one pixel. In the definition of $\overline{\mathtt{w}}(m, [0..1], \mathbf{p})$ we have to take the maximum of the screen-coordinates $\mathtt{x}$ and $\mathtt{y}$ since generically neither coordinate depends on just one of u and v. (If the GPU knew more about the patch at this stage of the graphics pipeline we might do better: for example if the patch was known to model a cylinder and the u parameter maps to a straight line on the cylinder, then the u-direction need not be partitioned at all. Of course, if the cylinder had parameter lines at 45 degrees to the axis, unequal tessellation and GPU-level intelligence would yield no savings.) And, since the contributions of partitioning the u- and the v-domain are mixed in the tensor-product, we generically need to partition the u- and the v-direction equally, by the same factor.

Then, for any initial choice $m > 0$, partitioning both the u- and the v-domain each into

$$\overline{\tau}_{\mathtt{xy}}(m, \mathbf{p}) := m\sqrt{\overline{\mathtt{w}}(m, [0..1], \mathbf{p})} \qquad (9)$$

many pieces, confines the parameter distortion to at most one unit. Analogously, the width $\mathtt{w}_{\mathtt{z}}(m, [0, 1], \mathbf{p})$ of the depth component $\mathtt{z}$ of the projection measures flatness of the slefe-tiles and therefore trustworthiness of the z-buffer test for covering accuracy. To enforce the depth tolerance $\mathtt{tol}_{\mathtt{z}}$, we partition into $\overline{\tau}_{\mathtt{z}}(m, \mathbf{p}) := m\sqrt{\mathtt{w}_{\mathtt{z}}(m, [0, 1], \mathbf{p})}/\mathtt{tol}_{\mathtt{z}}$ pieces.

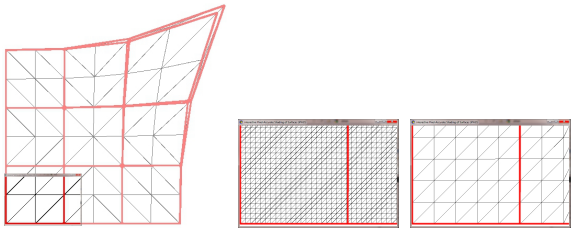
Setting the tessellation factor To guarantee that any error due to linearization is below pixel and depth thresholds, we compute the width for low m, say $m = 2$ or 3, and then apply (9) to obtain a safe tessellation factor of

$$\tau_{\mathbf{p}} := \max\{\overline{\tau}_{\mathtt{xy}}(m, \mathbf{p}), \overline{\tau}_{\mathtt{z}}(m, \mathbf{p})\}. \qquad (10)$$

Fig. 12 illustrates that, as expected, the resulting pieces are typically much larger than micro-polygons.

6 Algorithm and Implementation

We can now guarantee pixel-accuracy by computing the tessellation factor $\tau_{\mathbf{p}}$ and sampling the surface accordingly.



(a) patch beyond thescreen (lower left) (b) without Q-test (c) with Q-test

Figure 7: $\tau_{\mathbf{p}}$ depends only on Q_{ij} that overlap the screen. *(a) One patch with only its two lower left Q_{ij} (of 9) overlapping the screen (which is delineated by grey task and side bars.) (b) Tessellation density based on the whole patch. (c) Tessellation density based on the visible patch: 1/16 of (b) since the upper-right, high-curvature* slefe*-boxes do not influence the width.*

Computing and Sharing Tessellation Factors Every time the surface changes, e.g. due to animation, we need to re-compute the slefe-boxes $\overline{\mathbf{p}}(u_i, v_j)$ by (6). Every time the view changes, we need to re-compute the projections q_{ij} of the slefe-boxes by (7). We take the opportunity to also compute the $m \times m$ minimal screen-coordinate-aligned $\mathbf{xy}$-*rectangles* Q_{ij} that each enclose the screen projection of a slefe-tile:

$$Q_{ij} := [\min\{q_{kl}\}..\max\{q_{kl}\}] \subsetneq \mathbb{R}^2, \quad {\scriptstyle k\in\{i,i+1\},l\in\{j,j+1\}}.$$

This yields, for little cost, the Q-test, a cheap test for overlap with the screen. Only those $\mathbf{xy}$-rectangles that overlap the screen participate in determining the width (8). The possible reduction in $\tau_\mathbf{p}$ when only a small piece of the patch overlaps the screen is illustrated in Fig. 7 (see also the video). In particular, if no Q_{ij} overlaps the screen, $\overline{\mathbf{w}} = 0$ and hence $\tau_\mathbf{p} = 0$. To guarantee water-tightness along boundaries, $\tau_\mathbf{p}$ is communicated to all edge-adjacent patches.

Pixel-Accurate Rendering in DX11 The Tessellation Engine available with DX11 allows the specification of both interior and edge tessellation factors. To guarantee water-tightness, the Hull Shader compares, for each edge of its patch $\mathbf{p}$, the interior tessellation factor $\tau_\mathbf{p}$ with the factor $\tau_\mathbf{q}$ of its edge-neighbor. The edge-tessellation factor $\tau_{\mathbf{p},\mathbf{q}}$ for $\mathbf{p}$ on the boundary between patches $\mathbf{p}$ and $\mathbf{q}$ is then, by default, set to $\tau_{\mathbf{p},\mathbf{q}} := \max\{\tau_\mathbf{p}, \tau_\mathbf{q}\}$. Only if $\tau_\mathbf{p} = 0$ is the edge factor $\tau_{\mathbf{p},\mathbf{q}}$ set to zero. A patch outside the viewing frustum therefore has minimal cost. Note that the default correctly sets $\tau_{\mathbf{q},\mathbf{p}} = \tau_\mathbf{q}$ for $\tau_\mathbf{p} = 0$ and $\tau_\mathbf{q} > 0$.

The Domain Shader evaluates the spline pieces at the (u_i, v_j) coordinates generated by the Tessellation Engine according to the factors set by the Hull Shader. Using bitwise commutative operations, the evaluation exactly matches along the patch boundary (see e.g. [Castano 2008]). The resulting triangles, possibly augmented by (u_i, v_j) and the patch and object ids for more complex shaders, are sent through the Rasterizer and the z-buffer to be rendered.

DX11 implementation Fig. 8 shows the variance-based approach mapped to the DX11 graphics pipeline. We abbreviate: CS=Compute Shader, HS = Hull Shader, TE = Tessellation Engine, DS = Domain Shader, PS = Pixel Shader. We name the algorithm *interactive pixel-accurate shading of surfaces*, iPASS for short (since the natural abbreviation of PIXel-Accurate Rendering is already taken).

The iPASS Algorithm

Input: Patches $\mathbf{p}$ with coefficients $\mathbf{c}_{ij} \in \mathbb{R}^3$, slefeTable(s)
Output: Pixel-accurate rendering of $\mathbf{p}$.

Whenever triggered: Compute and distribute $\tau_\mathbf{p}$

CS (per patch $\mathbf{p}$) If $\mathbf{p}$ changed, compute its slefe-boxes $\overline{\mathbf{p}}(u_i, v_j)$. If the view changed, compute q_{ij} by (7), the $\mathbf{xy}$-rectangles Q_{ij} and $\tau_\mathbf{p}$ according to (9) using the Q-test. Place the value into the edge-slot of the edge-adjacent patches. Patches outside the viewing frustum receive $\tau_\mathbf{p} = 0$.

Pixel-Accurate Rendering Pass

HS (per patch $\mathbf{p}$) Set the interior tessellation factor to $\tau_\mathbf{p}$ and the edge factor to $\tau_{\mathbf{p},\mathbf{q}} := \max\{\tau_\mathbf{p}, \tau_\mathbf{q}\}$. If $\tau_\mathbf{p} = 0$, re-set the edge factor to $\tau_{\mathbf{p},\mathbf{q}} = 0$.

TE Generate the (u, v) parameters according to the factors.

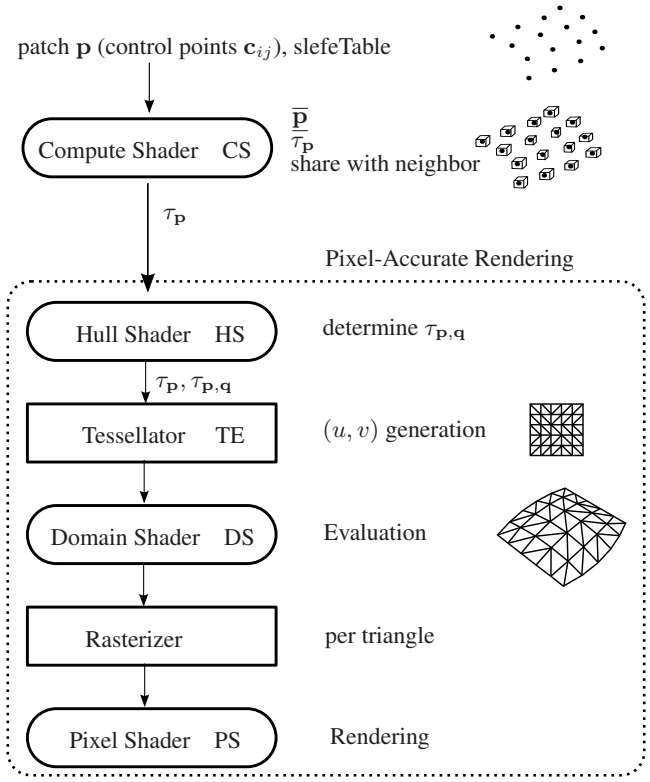


Figure 8: *DX11 passes used by iPASS.*

DS (per parameter pair (u, v)) Evaluate $\mathbf{p}$ at (u, v).

PS (per pixel) Apply shaders.

END

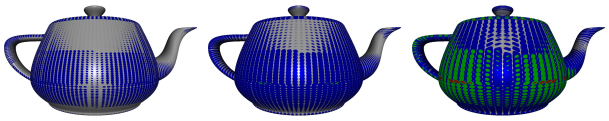
7 Performance

We want to check that iPASS delivers pixel-accuracy efficiently for generic models consisting of bi-cubic patches, such as the four models in Fig. 12.

Size and Distribution of Pixel-accurate Triangles The right panels of Fig. 12 show the triangle-size distribution for the models. The bar on the far right summarizes this distribution. Since any triangle-size distribution must vary depending on the view, zoom and the layout of the patches, we normalized each image to cover 38% of the screen. White indicates large triangles exceeding 20 pixels. Bounding the variance from the linear approximation evidently results in triangles whose projection is typically much larger than pixel-size. We observe that, generically, the number of triangles projecting to fewer than 5 pixels is low and, due to their small size, their overall percentage of screen coverage is still lower. Triangles of micro-polygon size, i.e. covering half a pixel or less, are rare ($< 1\%$) and cluster around the silhouettes.

Tightness of $\tau_\mathbf{p}$ The left panels of Fig. 12 confirm pixel-accuracy for the models. From the arguments in Section 5, it is clear that we can artificially construct patches and views so that the bounds are optimal and the iPASS choice of $\tau_\mathbf{p}$ is minimal. To estimate how tight the slefe-estimates are generically, we scaled the $\tau_\mathbf{p}$

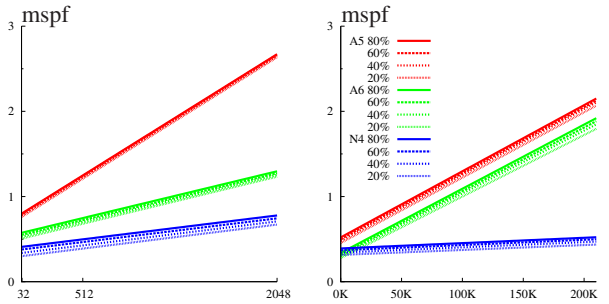
of the four test models by numbers less than 1.0 to artificially lower their tessellation density. We then analyzed pixel-accuracy by comparing for each pixel and its associated (u, v), the pixel position to $P(\mathbf{p}(u, v))$. Measurement and prediction of $\tau_\mathbf{p}$ are therefore unrelated computations, as they should be. We found $0.75\tau_\mathbf{p}$ still safe for most patches, but already $0.5\tau_\mathbf{p}$ unsafe for large areas (cf. Fig. 9). For the test cases, therefore, the computed $\tau_\mathbf{p}$ is close to minimal.



(a) $\tau_\mathbf{p}$ (b) $0.75\tau_\mathbf{p}$ (c) $0.5\tau_\mathbf{p}$

Figure 9: Minimality of $\tau_\mathbf{p}$. *Error (green, red) when reducing the tessellation factor.*

Performance scales linearly Our currently most efficient implementation (cf. the accompanying video) animates 100 monster frogs (129K patches) at 310 fps on a 1440×900 screen without instancing. With the same setup, 100 killeroos morphing into man (304K patches) render at 150 fps. To measure worst-case performance, we forced recomputation of all slefe-boxes and projections at every time step. We measured and analyzed performance on three architectures: **N4** a NVidia GeForce GTX 480 with Intel Core 2 Quad CPU Q9450 at 2.66GHz with 4GB memory, **A5** an ATI Radeon HD 5870 with Intel Core 2 Quad CPU Q6600 at 2.40GHz with 3GB memory, and **A6** an AMD Radeon HD 6970 with Intel Core 2 Quad CPU Q9450 at 2.66GHz with 4GB memory.



(a) mspf *vs* $n_\mathbf{p}$ (b) mspf *vs* $n_\triangle$

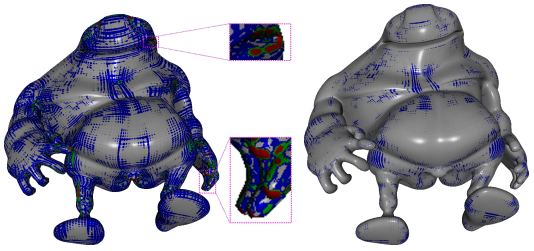
Figure 10: Linear Scaling. *(a) mspf vs number of patches $n_\mathbf{p}$ on configurations N4 (blue), A5 (red), A6 (green) for varying screen cover, keeping $n_\triangle$ and screen coverage $n_\square$ constant for each line at 20%, 40%, 60% or 80%. (b) mspf vs number of triangles $n_\triangle$ keeping $n_\mathbf{p}$ and $n_\square$ constant.*

As Fig. 10 shows, for a fixed architecture, screen size and display model, run-time performance scales linearly as for standard rendering. The cost in milliseconds per frame **mspf** is

$$\text{mspf} = k + c_0 n_\mathbf{p} + c_1 n_\triangle + c_2 n_\square, \tag{11}$$

where k is the constant overhead of the passes, $n_\mathbf{p}$ the input size represented by the number of patches, $n_\triangle$ the size of intermediate computations represented by the number of triangles generated by the tessellation engine and $n_\square$ the output size represented by the number of pixels that require non-trivial shading. The percentage of the screen covered by these pixels is called *cover*. To verify the formula, we kept two of the three parameters in (11) constant while varying the third. In Fig. 10a, the number of triangles $n_\triangle$ and shaded pixels $n_\square$ (cover of 20%, 40%, 60%, 80%) are constant for each measured graph. We vary the number of patches $n_\mathbf{p}$ by splitting the original patches into four while halving the tessellation factor to preserve the number of triangles. In Fig. 10b, the number of patches $n_\mathbf{p}$ and the percentage of shaded pixels $n_\square$ are constant. We vary the number of triangles $n_\triangle$ by choosing the tessellation factor to be $\mu\tau_\mathbf{p}$ with μ varying from 1.5 to 0.25 by steps of 0.25, and measure also for $\mu = 0.1$. The graphs have a linear regression, with standard errors below one percent.

Comparison to a screen bound heuristic Recall that we guarantee that the *variance* between rasterized and exact patch-piece is less than half a pixel in `x` and `y`. The micro-polygon criterion aims at enforcing accuracy by generating pieces of *size* less than half a pixel in `x` and `y`. Since safe recursive split-and-dice requires an unknown number of multiple passes, it is a priori slower on the GPU than our single compute shader pass. To mimic micro-polygonization in a single pass, we set the tessellation factor to a multiple $\frac{1}{\mu}$ of the maximal edge-length of the AABB bounding box of the patch projection, aiming at a polygon size of μ^2 pixels. Setting $\mu = 1, 2, 4$ yields massive overtessellation and highly reduced fps. Reducing the tessellation factor further by setting $\mu = 8$ (typically considered sufficient for uniform 16×16 dicing), is still slower than iPASS and results in the inaccuracies shown in Fig. 11a.



(a) Screen bound heuristic (b) Pixel-accurate

Figure 11: Screen bound Heuristic *results in parametric inaccuracy shown in red.*

Compute Shader vs Pixel Shader We experimented with replacing the compute shader (CS) with a pixel shader (PS) pass. Triggering a PS pass is indeed faster for our four models, the more so the fewer the patches. When rendering more patches, e.g. 10 killeroos, however, the advantage reverses in favor of the CS. The higher constant cost of CS-initialization amortizes better with more work that, in our implementation, is better parallelized on the CS. We are in the process of testing whether additional patch culling in the CS (see e.g. [Loop et al. 2011]) pays off.

8 Discussion

Choice of $m = 3$ for $\tau_\mathbf{p}$ The algorithm is pixel-accurate for any initial choice m of slefe-segments. But there is a trade-off between the tightness of the estimate and the cost of computing the initial, low-m bound. Comparing the performance for $m = 2, 3, 4$ over the range of models, we found that $m = 3$ maximizes fps for all models.

No pixel-dropout Since the boundary tessellation factors of adjacent patches agree, there are no T-joints. Boundary points are

computed by the univariate de Casteljau algorithm, i.e. repeated pairwise averaging that is implemented as a bitwise commutative operation. Therefore boundary points are computed bitwise consistently for a patch and its neighbor.

High Zoom and Partial Patches High zoom means fewer patches to tessellate since $\tau_{\mathbf{p}} = 0$ for patches outside the viewing frustum. Also, when a smooth patch fills the screen under high zoom, its local depth variation can be expected to be low since the tangent plane is locally a good approximation. Since patch segments that project sufficiently far outside the screen do not contribute to $\tau_{\mathbf{p}}$, zoomed-in terrain or water surfaces can have low tessellation (cf. Fig. 7). Finally, at 1440×900 resolution $\tau_{max} = 64$ splits a screen-filling patch into quads of size $2^4 \times 2^4$. Together, this explains why we have not encountered that the maximal tessellation factor of 64 set by graphics cards was exceeded. If the maximum is exceeded, our preferred solution is lazy binary subdivision, replacing the patch by four subpatches in the work-queue and maintaining a link to restore the original patch when the subpatch $\tau_{\mathbf{p}}$ s add to less than 64.

Compatibility with shaders Integrating shaders with the Rendering Pass is straightforward. We verified this for defocus, motion blur, MSAA, transparency, pixel-accurate trimming and more.

9 Conclusion

We defined pixel-accuracy as covering accuracy plus parametric accuracy to display curved shape correctly and prevent parametric distortion. Of the four level-of-detail criteria considered in the classical survey [Xia et al. 1997], silhouettes are correctly captured by covering accuracy, increase due to gradient is captured by parametric accuracy, the length of the screen-space projection is not relevant and visibility culling is performed when $\tau = 0$.

To practically enforce pixel-accuracy, in particular within the existing graphics pipeline, we showed how to use the contraction of tight enclosures of polynomial pieces when reducing the domain, to predict a tessellation density that guarantees that the variance between an exact surface piece and its triangulation drops below the pixel threshold. Due to equation (9), the underlying slefes are never constructed, but only differences of a *few* slefe-based screen-projected points, to establish and predict variance. This reflects the change of objective compared to the literature: the shift from controlling size to controlling *variance* (between the projection of the exact surface and its triangulation).

Finally, we demonstrated an implementation under DX11 that can render 300K patches at 150 frames per second, a speed comparable to inaccurate, moderately dense, fixed-level tessellation.

Acknowledgements This work was supported in part by grant NSF CCF-1117695. We gratefully acknowledge Bay Raitt's "Monster Frog" and "Big Guy". The Killeroo model is courtesy of Headus Pty Ltd (http://www.headus.com/au).

References

CASTANO, I., 2008. Tessellation of displaced subdivision surfaces in dx11. http://origin-developer.nvidia.com/object/gamefest-2008-subdiv.html. Gamefest 2008.

COOK, R. L., CARPENTER, L., AND CATMULL, E. 1987. The Reyes image rendering architecture. In *Computer Graphics (SIGGRAPH '87 Proceedings)*, M. C. Stone, Ed., 95–102.

DRONE, S., LEE, M., AND ONEPPO, M., 2008. Direct3d 11 tessellation. http://www.microsoft.com/download/en/details.aspx?id=23111. Gamefest 2008.

EISENACHER, C., AND LOOP, C. 2010. Data-parallel micropolygon rasterization. In *Eurographics 2010 Annex: Short Papers*, S. Seipel and H. Lensch, Eds.

FARIN, G. 1988. *Curves and Surfaces for Computer Aided Geometric Design: A Practical Guide*. Acad. Press.

FATAHALIAN, K., LUONG, E., BOULOS, S., AKELEY, K., MARK, W. R., AND HANRAHAN, P. 2009. Data-parallel rasterization of micropolygons with defocus and motion blur. In *HPG '09: Proc High Performance Graphics 2009*, ACM, New York, NY, USA, 59–68.

FATAHALIAN, K., BOULOS, S., HEGARTY, J., AKELEY, K., MARK, W. R., MORETON, H., AND HANRAHAN, P. 2010. Reducing shading on GPUs using quad-fragment merging. In *ACM Trans. Graphics, 29(3), 2010 (Proc. ACM SIGGRAPH 2010)*, vol. 29, 67:1–8.

FILIP, D., MAGEDSON, R., AND MARKOT, R. 1986. Surface algorithms using bounds on derivatives. *Computer Aided Geometric Design 3*, 4, 295–311.

FISHER, M., FATAHALIAN, K., BOULOS, S., AKELEY, K., MARK, W. R., AND HANRAHAN, P. 2009. DiagSplit: parallel, crack-free, adaptive tessellation for micropolygon rendering. *ACM Transactions on Graphics 28*, 5 (Dec.), 1–8.

GROSS, M., GATTI, R., AND STAADT, O. 1995. Fast multiresolution surface meshing. *Proceedings of Visualization'95*, 135–142.

GUTHE, M., BALÁZS, A., AND KLEIN, R. 2005. GPU-based trimming and tessellation of NURBS and T-Spline surfaces. *ACM Transactions on Graphics 24*, 3 (July), 1016–1023.

HOPPE, H. 1996. Progressive meshes. *Proceedings of SIGGRAPH '96*, 99–108.

LOOP, C., NIESSNER, M., AND EISENACHER, C. 2011. Effective back-patch culling for hardware tessellation. In *Proceedings of Vision, Modeling and Visualization*.

LUTTERKORT, D. 2000. *Envelopes of Nonlinear Geometry*. PhD thesis, Purdue University.

PATNEY, A., AND OWENS, J. D. 2008. Real-time Reyes-style adaptive surface subdivision. *ACM Trans. Graph 27*, 5, 143.

PETERS, J., AND REIF, U. 2008. *Subdivision Surfaces*, vol. 3 of *Geometry and Computing*. Springer-Verlag, New York.

PETERS, J. 2004. Mid-structures of subdividable linear efficient function enclosures linking curved and linear geometry. In *Proceedings of SIAM conference, Seattle, Nov 2003*, Nashboro, M. Lucian and M. Neamtu, Eds.

SHENG, X., AND HIRSCH, B. E. 1992. Triangulation of trimmed surfaces in parametric space. *Computer-Aided Design 24*, 8, 437–444.

TOOKEY, R., AND CRIPPS, R. 1997. Improved surface bounds based on derivatives. *Computer Aided Geometric Design 14*, 8, 787–791.

TZENG, S., PATNEY, A., AND OWENS, J. D. 2010. Task management for irregular-parallel workloads on the GPU. In *High Perf. Gr.*, ACM, J. Hensley and et al., Eds., 29–37.

WU, X., AND PETERS, J., 2002. Sublime (subdividable linear maximum-norm enclosure) package. `http://surflab.cise.ufl.edu/SubLiME.tar.gz`. Accessed Jan 2011.

XIA, J. C., EL-SANA, J., AND VARSHNEY, A. 1997. Adaptive real-time level-of-detail-based rendering for polygonal models. *IEEE Trans. Vis. Comput. Graph 3*, 2, 171–183.

ZHOU, K., HOU, Q., REN, Z., GONG, M., SUN, X., AND GUO, B. 2009. Renderants: interactive Reyes rendering on GPUs. *ACM Trans. Graph 28*, 5.

10 Appendix: Tensor-product Bounds

The tensor-product patch (5) 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:

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

Tensored slefes avoid the need to store and access the much larger table of (possibly tighter) pre-computed bounds in two variables.

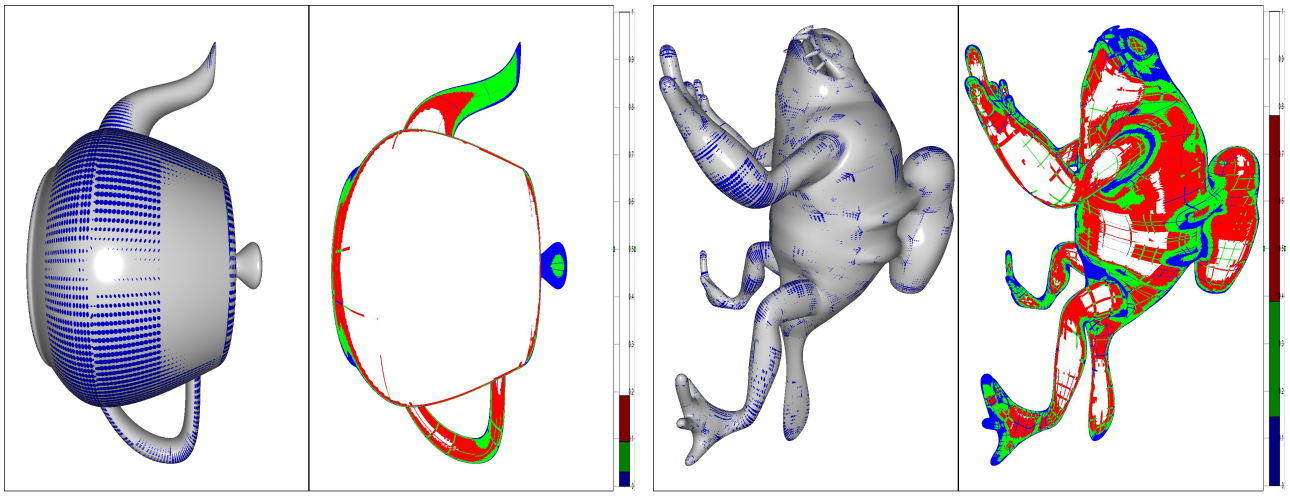
Let $\overline{\mathsf{slefe}}(\mathbf{c}, m)$ be the routine that returns the $m+1$ values of the upper slefe. We compute

$$\begin{aligned} &\text{for } i = 0, \ldots, \ d_1, \\ &\quad [\tilde{c}_{i0}, \tilde{c}_{i1}, \ldots, \tilde{c}_{im_2}] := \overline{\mathsf{slefe}}([c_{i0}, c_{i1}, \ldots, c_{id_2}], m_2) \end{aligned}$$

and obtain the values $\bar{c}_{ij}$, $i = 0, \ldots, m_1$, $j = 0, \ldots, m_2$ of the upper bound on the tensor-product from a second application

$$\begin{aligned} &\text{for } j = 0, \ldots, m_2, \\ &\quad [\bar{c}_{0j}, \bar{c}_{1j}, \ldots, \bar{c}_{m_1 j}] := \overline{\mathsf{slefe}}([\tilde{c}_{0j}, \tilde{c}_{1j}, \ldots, \tilde{c}_{d_2,j}], m_1). \end{aligned}$$

Lower values are computed analogously.



(a) teapot (b) frog

(c) big guy (d) killeroo

Figure 12: Pixel-accuracy and triangle distribution *for 38% screen cover. left panels of (a),(b),(c),(d): blue and grey indicate pixel-accuracy. Grey color indicates for the pixel's* u, v *that* $\mathbf{p}(u, v)$ *projects to less than 0.1 pixel sizes away from the pixel center. Blue means less than 0.5. right panels: the colors indicate triangle size as* $blue < 5$, $green < 10$, $red < 20$, $white >= 20$ *pixels. The bar on the right summarizes the triangle size of the non-background pixels. Grid patterns stem from the tessellation engine.*