Eurographics Symposium on Rendering (2026)
I. Gkioulekas and A. Jarabo (Editors)

GPU View-Adaptive Crack-Free Subdivision of Bézier Surfaces

Seyedmasih Tabaei[*, 1] Bastian Kuth[*, 1, 2] Max Oberberger[2] Quirin Meyer[1]

[*]Joint first authors
[1]Coburg University of Applied Sciences and Arts, Germany [2]AMD, Germany

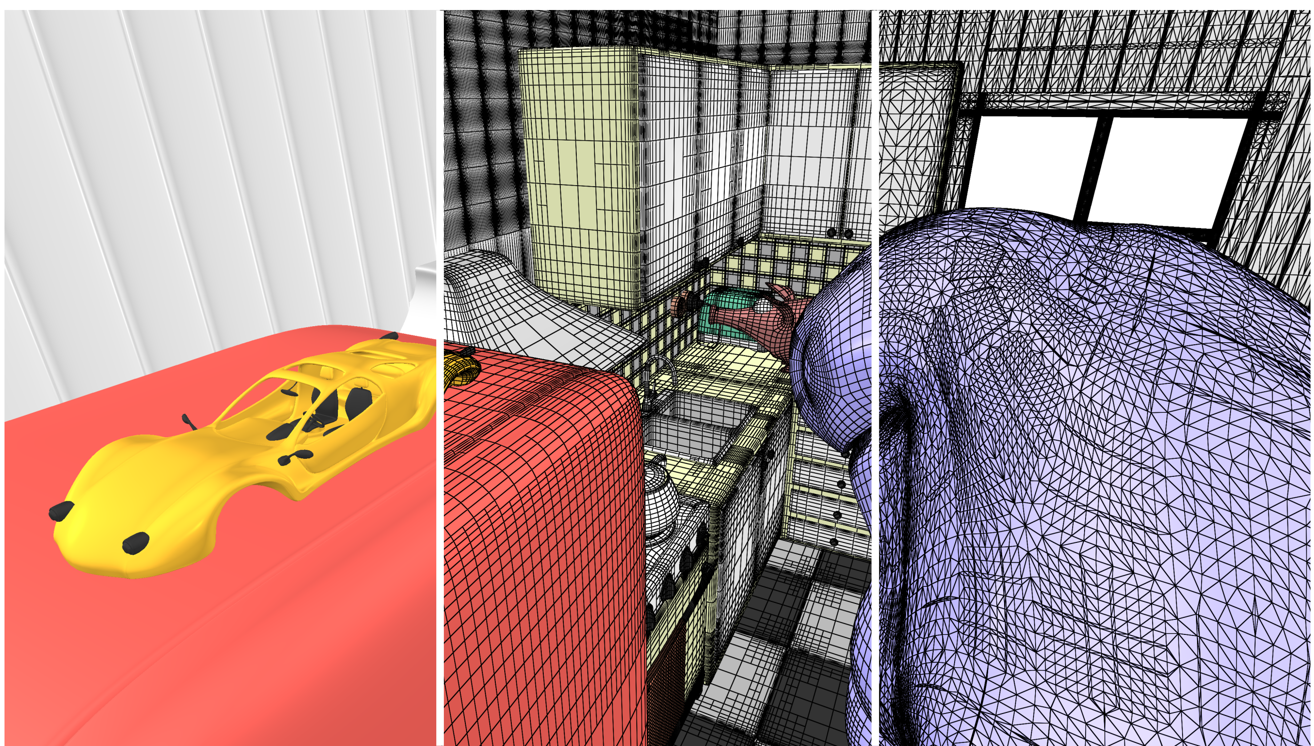


Figure 1: We recursively subdivide bicubic Bézier patches (left) with our novel view-adaptive and crack-free method directly on the GPU (center). We implement our method in GPU work graphs, outperforming an equivalent execute-indirect implementation in terms of speed and memory usage. By producing fewer triangles, our method can even render faster than hardware tessellation (right).

Abstract

We present a recursive GPU algorithm for view-adaptive and crack-free Bézier subdivision. We contribute an improved memory-efficient sub-patch representation, a numerically stable and fast reconstruction of subdivided Bézier patches, a novel controllable subdivision criterion, a communication-free crack-closing technique, and carefully designed computations for producing watertight triangulations. Our GPU work graphs implementation can outperform both hardware tessellation and an implementation based on execute-indirect, while using orders of magnitude less memory. Our results demonstrate that recursive split problems map efficiently to GPU work graphs, suggesting broader applicability.

CCS Concepts

• *Computing methodologies* → *Graphics processors; Rasterization; Parametric curve and surface models; Massively parallel algorithms;*

© 2026 The Author(s).
Proceedings published by Eurographics - The European Association for Computer Graphics.
This is an open access article under the terms of the Creative Commons Attribution License, which permits use, distribution and reproduction in any medium, provided the original work is properly cited.

1. Introduction

Bicubic Bézier surfaces are widely used in computer graphics. Their formulation extends naturally to higher-order or rational variants, which are as expressive as B-Splines or non-uniform rational B-splines (NURBS). Except at extraordinary regions, a Catmull-Clark subdivision (CCS) surface coincides with a Bézier surface. A graphics processing unit (GPU) usually renders bicubic Bézier surfaces by approximating them with triangles obtained from subdivision [PO08] or hardware tessellation [LS08]. Since recursive subdivision has been infeasible on GPUs, systems have instead relied on iterative breadth-first subdivision. However, state of the art GPU breadth-first subdivision methods either are not view-adaptive [MWS*20, DV21, KOCM23], or do not produce crack-free and watertight triangulations [PO08, EML09]. Furthermore, their implementations require multiple GPU dispatches with expensive synchronization and high global-memory demand. With memory capacity and bandwidth growing more slowly than compute capabilities, this can increasingly become a bottleneck. While hardware tessellation directly addresses the bandwidth issue, it tends to over-triangulate the surface. Subdivision better adapts the triangulation density to surface features.

To overcome the disadvantages of GPU compute breadth-first subdivision, we make the following contributions:

- We propose an improved **sparse representation for sub-patches** to save on bandwidth.
- We reconstruct Bézier points of a sub-patch from this representation with a new **numerically stable and fast reconstruction**.
- Through our **subdivision criterion** based on a controllable geometric and parametric error, our method is view-adaptive.
- We **close cracks** between patches of different subdivision levels with a novel method that requires no synchronization.
- We carefully design all computations to obtain a **watertight** tessellation of the original Bézier surface.

While our method can be implemented using execute-indirect, we show the benefits of using GPU work graphs [Mic25]. With work graphs, we are able to formulate GPU subdivision *recursively*, requiring only a single dispatch and no explicit synchronization. For our test scene, our work graphs implementation outperforms a conventional execute-indirect implementation in speed while drastically reducing memory requirements. Both of our implementations can outperform hardware tessellation at similar visual errors.

2. Background

Before covering related work in Sec. 2.2, we summarize computer-aided geomtric design (CAGD) concepts relevant to our work in Sec. 2.1. Unless stated otherwise, we build upon Farin's textbook [Far02]. Especially the concepts of *blossoming* and *multi-affinity* are crucial for our contributions.

2.1. Bézier Curves and Surfaces

A *cubic Bézier curve* is a function that maps $u \in \Delta$ to $\mathbb{R}^3$:

$$\vec{C}(u) = \sum_{i=0}^{3} B_i^{\Delta}(u) \cdot \vec{p}_i, \quad u \in \Delta = [a,b] \subset \mathbb{R}$$

with the *Bézier points* $\vec{p}_i \in \mathbb{R}^3$. Δ is the *parameter domain* and

$$B_i^{\Delta}(u) = \binom{3}{i} \cdot (1-\alpha(u))^{3-i} \cdot \alpha(u)^i, \quad \alpha(u) = \frac{u-a}{b-a}$$

are the *cubic* BERNSTEIN *polynomials*.

The *blossom* [Ram89] of a cubic Bézier curve is a function $\vec{c}(u_1,u_2,u_3) \to \mathbb{R}^3$ and has the following properties [Sei93]:

- *Bézier-Point Property*: $\vec{c}$ computes the Bézier points:
$$\begin{aligned} \vec{p}_0 &= \vec{c}(a,a,a), \quad \vec{p}_1 = \vec{c}(a,a,b), \\ \vec{p}_2 &= \vec{c}(a,b,b), \quad \vec{p}_3 = \vec{c}(b,b,b). \end{aligned} \tag{1}$$
- *Diagonality*: If the three parameters are equal, i.e., $u = u_1 = u_2 = u_3$, we evaluate the curve, i.e., $\vec{c}(u,u,u) = \vec{C}(u)$.
- *Symmetry*: $\vec{c}(u_1,u_2,u_3) = \vec{c}(\pi(u_1,u_2,u_3))$ where $\pi(u_1,u_2,u_3)$ is any permutation of the triple (u_1,u_2,u_3). Due to this, Eq. (1) is usually written compactly as $\vec{p}_i = \vec{c}\left(a^{\langle 3-i \rangle}, b^{\langle i \rangle}\right)$, where the superscript $\langle i \rangle$ indicates the multiplicity i of the parameter.
- *Multi-Affinity*: We can linearly blend one parameter with $s \in \mathbb{R}$ either before or after evaluating $\vec{c}$:
$$\vec{c}((1-s)\,u_l + s\,u_r, *) = (1-s)\,\vec{c}(u_l,*) + s\,\vec{c}(u_r,*), \tag{2}$$
where $*$ indicates that the other parameters remain constant. Due to symmetry, we can interpolate any parameter in Eq. (2).

Subdivision With blossoms, we can express subdivision. When subdividing a *base curve* defined over $[a,b]$ at $\bar{u}$, we get a *left sub-curve* over $[a,\bar{u}]$ and a *right sub-curve* over $[\bar{u},b]$. The sub-curves map to the same points as the base-curve, except from different domains. To compute Bézier-points of the left sub-curve, we use the Bézier-point property to determine the blossom of $\vec{p}_i^{(l)}$:

$$\begin{aligned} \vec{p}_0^{(l)} &= \vec{c}(a,a,a), \quad \vec{p}_1^{(l)} = \vec{c}(a,a,\bar{u}), \\ \vec{p}_2^{(l)} &= \vec{c}(a,\bar{u},\bar{u}), \quad \vec{p}_3^{(l)} = \vec{c}(\bar{u},\bar{u},\bar{u}). \end{aligned}$$

Multi-affinity gets unknown blossoms from known ones, e.g.,

$$\begin{aligned} \vec{p}_1^{(l)} &= \vec{c}(a,a,\bar{u}) = \vec{c}(a,a,(1-\alpha(\bar{u})\cdot a) + \alpha(\bar{u})\cdot b) \\ &= (1-\alpha(\bar{u}))\cdot \underbrace{\vec{c}(a,a,a)}_{\vec{p}_0} + \alpha(\bar{u})\cdot \underbrace{\vec{c}(a,a,b)}_{\vec{p}_1}, \end{aligned}$$

which leads to the DE CASTELJAU subdivision algorithm.

With a tensor product, we obtain a *bicubic tensor product Bézier surface* – or *patch* for short. It has 16 Bézier points $\vec{p}_{i,j} \in \mathbb{R}^3, 0 \le i,j \le 3$ over the 2D parameter domain $[a,b] \times [c,d] = \Delta_u \times \Delta_v$:

$$\vec{S}(u,v) = \sum_{j=0}^{3} B_j^{\Delta_v}(v) \cdot \underbrace{\left(\sum_{i=0}^{3} B_i^{\Delta_u}(u) \cdot \vec{p}_{i,j} \right)}_{\vec{c}_j(u)}, \quad (u,v) \in \Delta_u \times \Delta_v.$$

This is also known as the "curve-of-curves" approach. Thus, algorithms from curves naturally elevate to surfaces. Consider evaluation at $(\bar{u},\bar{v})$. We first evaluate four inner Bézier curves $\vec{c}_j(\bar{u})$. This provides four Bézier points for the outer curve in $\bar{v}$ which we evaluate at $\bar{u}$ to obtain the point on the surface.

The same holds for subdivision: From a *base-patch*, we first subdivide its inner four curves at $\bar{u}$. This gives us four left and four right

© 2026 The Author(s).
Proceedings published by Eurographics - The European Association for Computer Graphics.

curves forming two new patches. We subdivide all curves of the resulting patches at $\bar{v}$. That gives us in total four *sub-patches* over the domains $[a,\bar{u}]\times[c,\bar{v}]$, $[a,\bar{u}]\times[\bar{v},d]$, $[\bar{u},b]\times[c,\bar{v}]$, and $[\bar{u},b]\times[\bar{v},d]$.

A *tensor product blossom* of a bicubic Bézier surface is a function $\vec{s}(u_1,u_2,u_3\,|\,v_1,v_2,v_3)\to\mathbb{R}^3$. Similar to Farin [Far02], we use the bar $|$ to better distinguish the two main parameter directions. The tensor product blossom has the following properties:

- *Bézier-Point Property*: $\vec{p}_{i,j}=\vec{s}\left(a^{\langle 3-i\rangle},b^{\langle i\rangle}\,\middle|\,c^{\langle 3-j\rangle},d^{\langle j\rangle}\right)$.
- *Diagonality*: $\vec{s}\left(u^{\langle 3\rangle}\,\middle|\,v^{\langle 3\rangle}\right)=\vec{S}(u,v)$.
- *Symmetry* holds in each parameter direction:

$$\vec{s}(u_1,u_2,u_3\,|\,v_1,v_2,v_3)=\vec{s}(\pi(u_1,u_2,u_3)\,|\,\pi(v_1,v_2,v_3))\,.$$

- *Multi-Affinity* generalizes to bilinear blending in $s,t\in\mathbb{R}$:

$$\begin{aligned}&\vec{s}((1-s)\cdot u_l+s\cdot u_r,*\,|\,(1-t)\cdot v_l+t\cdot v_r,*)=\\&(1-s)\cdot(1-t)\cdot\vec{s}(u_l,*\,|\,v_l,*)+s\cdot t\cdot\vec{s}(u_r,*\,|\,v_r,*)+\\&(1-s)\cdot t\cdot\vec{s}(u_l,*\,|\,v_r,*)+s\cdot(1-t)\cdot\vec{s}(u_r,*\,|\,v_l,*)\,.\end{aligned}\qquad(3)$$

Multi-affinity together with symmetry are crucial to our crack-free subdivision reconstruction of Sec. 3.3.2. They let us compute any blossom, and thus the Bézier points of a subdivided patch.

2.2. Related Work

Catmull [Cat74] pioneered subdivision for bicubic Bézier patch rendering. His system recursively subdivides patches until each one fits within a single pixel. Patney and Owens [PO08] replaced this recursive depth-first strategy with a breadth-first approach, providing the first data-parallel GPU implementation. However, their method does not eliminate cracks. Eisenacher et al. [EML09] speed up this method and hide cracks, but missing pixels can still occur. Both approaches [PO08, EML09] are bandwidth-intensive. Our sparse sub-patch representation builds upon previous work [EL10, Eis11], but we contribute a smaller memory footprint and add crack-fixing capabilities.

Schwarz and Stamminger [SS09] render bicubic Bézier patches using a GPU software implementation similar to hardware tessellation [AB06, NKF*16]. Tessellation can close cracks through edge tessellation patterns [Sch09]. Using hardware tessellation, Loop and Schaefer approximate CCS subdivision surfaces with bicubic patches and lower-degree tangent patches for shading [LS08]. Strugar [Str09] uses a quad tree to render terrain and fixes cracks by morphing vertices. Unlike our approach, Strugar does not support transitions between quad tree levels that vary more than one level.

Guthe et al. [GBK05] present a GPU system for NURBS rendering. ETER [XLC*23] renders NURBS and hides cracks using an algorithm based on conservative rasterization and a visibility buffer. They report speed-ups when replacing tessellation with tensor cores and mesh shaders. WATER [ZLCL25] improves upon ETER by integrating evaluation more tightly into the mesh shader and reducing the number of rendered triangles. PaRas [WC25] reports performance improvements over ETER using numerical root-finding for surface evaluation. Unterguggenberger et al. [ULW*25] render single parametric surfaces using compute-shader subdivision. Their system renders through hardware tessellation or point-based compute-shader rasterization. However, their approach requires multiple dispatches for subdivision and evaluation.

We use GPU work graphs [Mic25], which provide an efficient replacement for execute-indirect [KOM25]. They have been used for procedural generation [KOF*24, KOF*25], image denoising [KWW*24], and sparse linear algebra [WET*25]. Work graph *mesh nodes* can directly submit geometry to the rasterizer.

3. GPU Rendering of Bicubic Bézier Surfaces

The input to our algorithm is a set of bicubic *base-patches* and a pixel-space tolerance ε. Each base-patch has 16 Bézier points with three floats each. Starting from a base-patch, we

1. determine in Sec. 3.2 whether a patch requires subdivision.
2. If so, we perform a 1:4 split at its parameter domain center. The four *sub-patches* are then recursively processed by step 1. We describe our subdivision reconstruction algorithm in Sec. 3.3.
3. If the patch is ready to draw, we use our crack-free rendering method of Sec. 3.4.

First, we describe our memory efficient sub-patch representation.

3.1. Sub-Patch Representation using Quad Tree Coordinates

For a sub-patch, we do not explicitly store the subdivided Bézier points. Similar to Eisenacher [Eis11], we encode the sub-patch parameter domain $[a,b]\times[c,d]$ and reconstruct the subdivided Bézier points in Sec. 3.3.

The base-patch parameter domain is $[0,1]\times[0,1]$. Every sub-patch is a node in a full quad-tree. At quad-tree depth l, we have at most 4^l nodes, where $l=0$ is the base-patch or the coarsest depth.

In a *quad node*, we store its base-patch id, its subdivision depth l, and the 2D node index (m,n) in u and v directions relative to l. This gives us the sub-patch domain boundaries $[a,b]\times[c,d]$:

$$a=m\cdot 2^{-l},\; b=(m+1)\cdot 2^{-l},\; c=n\cdot 2^{-l},\; d=(n+1)\cdot 2^{-l}$$

When subdividing a node (l,m,n), we compute child node parameters as $(l+1,2m+x,2n+y)$, with $x,y\in\{0,1\}$.

Approaches storing sub-patch Bézier points [PO08, EML09], require 192 bytes assuming a 3D float per point. Our implementation supports $l_{\max}=10$ subdivision depths. Therefore, our quad node requires $\lceil\log_2 l_{\max}\rceil=4$ bits to store the depth l and $l_{\max}$ bits for each m,n. We improve Eisenacher's data structure [Eis11] in two ways. First, Eisenacher's approach uses 16 bytes per sub-patch, suggesting the use of four floating-point values. We propose a more compact representation for quad tree nodes that reduces the memory requirements by more than half. Second, our data structure contains crack fixing information described in Sec. 3.4. Together with the base-patch id, all information fits into 8 bytes. This greatly reduces the memory-bandwidth pressure over Eisenacher's approach.

© 2026 The Author(s).

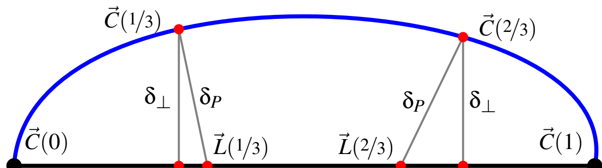


Figure 2: Edge Subdivision Metrics. To test whether an edge curve $\vec{C}$ (blue) requires subdivision, we use the orthogonal distances $\delta_\perp$ to the line $\vec{L}$ (black) approximating the curve. For accurate parameterization, we use the parametric distance δ_P.

3.2. Subdivision Criterion

For rendering, we approximate the patch by two triangles spanning the four patch corners. To test if this is sufficient, we perform a two-tier test. For simplicity, we implicitly apply a parameter map from the sub-patch domain $[a,b] \times [c,d]$ to $[0,1] \times [0,1]$.

At the *edge tier*, we consider the four boundary curves of a patch. For each boundary curve $\vec{C}$, we evaluate the points $\vec{C}(1/3)$ and $\vec{C}(2/3)$. We compute the *orthogonal distances* $\delta_\perp$ of those points from the line through the curve's end points $\vec{L}(u) = (1-u)\vec{C}(0) + u\vec{C}(1)$ as shown in Fig. 2. If any of those two distances is greater than a threshold, the patch requires further subdivision.

Otherwise, all boundary curves are sufficiently flat for rendering and we enter the *inner tier* of our test. There, we evaluate the patch at the inner points $\vec{S}(1/3,1/3)$, $\vec{S}(2/3,1/3)$, $\vec{S}(1/3,2/3)$, and $\vec{S}(2/3,2/3)$. If the orthogonal distances of any inner point to its closest triangles is greater than a threshold, the patch requires further subdivision.

Fig. 3 shows that orthogonal distances control the geometric error and are sufficient for accurate silhouettes, but can cause shading artifacts. To alleviate them, we use the *parametric distances* δ_P of Fig. 2, i.e., the distance between $\vec{L}(u)$ and $\vec{C}(u)$ at same parameter locations $u \in \{1/3, 2/3\}$. The results of Fig. 3c show better texturing but more patches. To trade patch count against quality, we linearly interpolate both metrics with λ into a unified *subdivision criterion*.

As thresholds for *view-adaptive rendering*, we project the pixel-space error ε into the camera-space coordinate system of the edge midpoints (edge tier) or the patch centers (inner tier). The criterion provides a strict upper bound on the screen-space error at the sampled locations; we approximate the full error over the patch by these samples.

3.3. Subdivision through Reconstruction

Here, we show how we use a quad node to reconstruct the sub-patch over $[a,b] \times [c,d]$ from the base patch. The curve-of-curves approaches described in Sec. 3.3.1 seem obvious, but can cause cracks when combined with our crack-fixing method of Sec. 3.4. Therefore, we propose to use the algorithm of Sec. 3.3.2.

3.3.1. Curve Reconstruction

From a base-curve over the interval $[0,1]$, we want to compute the BÉZIER-points of the curve over $[a,b]$. One way of doing that is to subdivide the interval at b. Then, we subdivide the resulting left

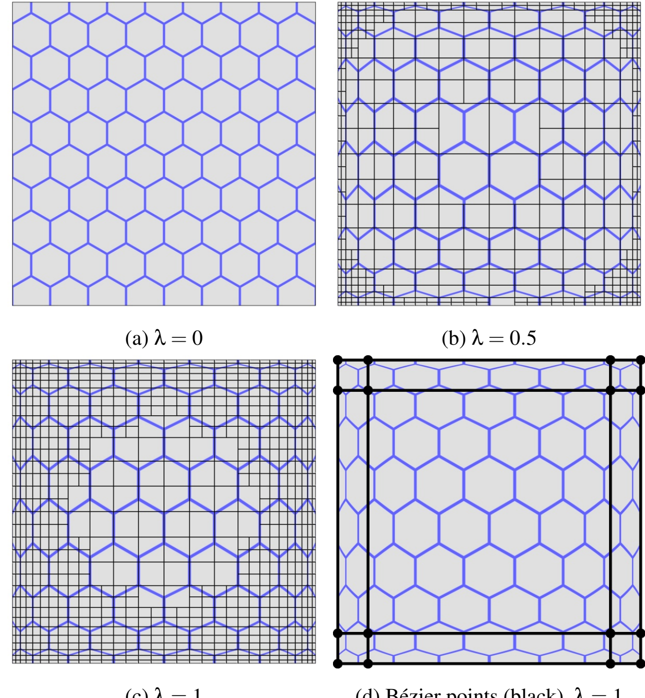


Figure 3: Metric Difference. In an extreme example, a perfectly flat patch, where $\delta_\perp = 0$, can be rendered without subdivision (a). However, the control Bézier points may still distort the parametric coordinates (d). Interpolating $\lambda > 0$ forces subdivision (b, c).

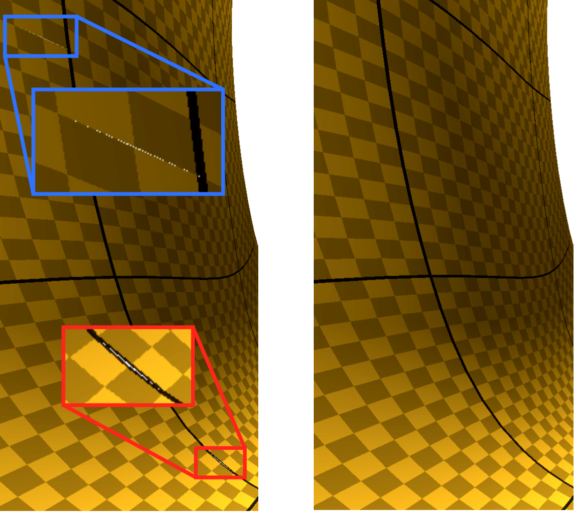
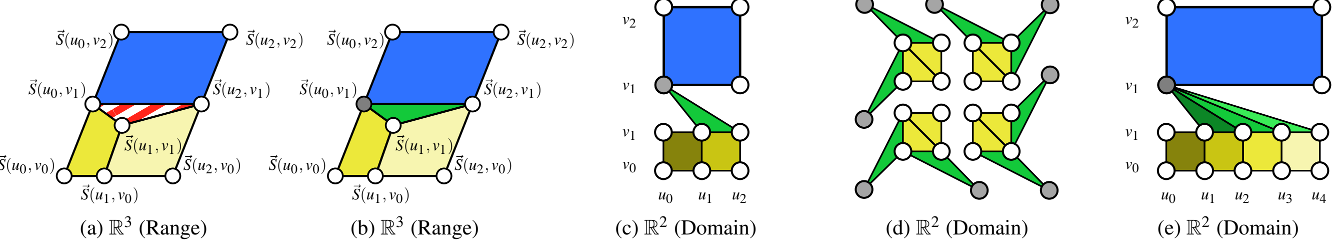


Figure 4: Crack Artifacts. Cracks – highlighted with boxes – occur during curve-based sub-patch reconstruction. Tensor product blossoms (right) are crack free. Black curves mark base-patch borders.

© 2026 The Author(s).



(a) $\mathbb{R}^3$ (Range) (b) $\mathbb{R}^3$ (Range) (c) $\mathbb{R}^2$ (Domain) (d) $\mathbb{R}^2$ (Domain) (e) $\mathbb{R}^2$ (Domain)

Figure 5: Wedges. (a) If a patch (blue) has neighbors of finer depth (yellow), cracks (red-white striped area) can occur. (b) We add a *wedge* (green) to fill the crack, with the grey point indicating the *wedge pivot*. (c) Shows the same in parameter domain. Note v_1 is depicted twice to better visualize the triangle. (d) For each of the four sub-patches originating from one parent, only two of four edges can have wedges, thus a patch is approximated with two triangles (yellow) and up to two wedges (green). (e) *Wedges* also fill cracks across higher depth differences.

curve over $[0,b]$ at a/b. Note that a/b is the relative location of a with respect to $[0,b]$. That subdivision's right curve yields the desired Bézier points over $[a,b]$. Consider a neighboring curve over $[b,c]$, which shares the boundary point at b with the curve over $[a,b]$. Its reconstruction subdivides $[0,1]$ at c and then the resulting left curve at b/c. On the $[a,b]$ side, b is an exact subdivision point; on the $[b,c]$ side, the same point is computed via the floating-point division b/c, causing a discontinuity. When applied to surfaces, we get unwanted cracks, as highlighted in blue in Fig. 4.

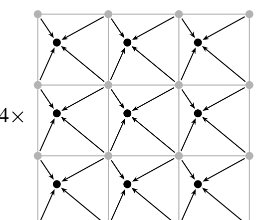
We are able to preserve continuity at b, when computing blossoms $\vec{c}(a,a,a)$, $\vec{c}(a,a,b)$, $\vec{c}(a,b,b)$, and $\vec{c}(b,b,b)$ with the multi-affinity property of Eq. (2). However, cracks still occur at base-patch edges and corners as shown in red in Fig. 4. This is because floating-point operations are not associative and we cannot always guarantee a consistent operation order between base-patches.

3.3.2. Tensor Product Blossom Reconstruction

For crack-free reconstruction, we compute the 16 Bézier points of the sub-patch over $[a,b] \times [c,d]$ from the tensor product blossoms $\vec{s}(a,a,a|c,c,c), \vec{s}(a,a,b|c,c,c), \ldots, \vec{s}(b,b,b|d,d,d)$. With multi-affinity of Eq. (3), we evaluate the blossoms from the base-patch Bézier points $\vec{p}_{i,j}$ with successive bilinear interpolation. In the following code, `float3 P[i][j]` is the base-patch Bézier point $\vec{p}_{i,j}$ and also serves as output for the corresponding sub-patch Bézier point:

```
float2 U = {a, b}, V = {c, d};
float3 D[2][2][3][3], E[2][2][2][2];
```

```
for(int l = 0; l < 2; ++l)
for(int k = 0; k < 2; ++k)
for(int j = 0; j < 3; ++j)
for(int i = 0; i < 3; ++i)
 D[l][k][j][i] = bilerp(
  U[k], V[l],
  P[j+0][i+0],
  P[j+0][i+1],
  P[j+1][i+0],
  P[j+1][i+1]);
```




```
for(int l = 0; l < 2; ++l)
for(int k = 0; k < 2; ++k)
for(int j = 0; j < 2; ++j)
for(int i = 0; i < 2; ++i)
 E[l][k][j][i] = bilerp(
  U[k], V[l],
  D[l][k][j+0][i+0],
  D[l][k][j+0][i+1],
  D[l][k][j+1][i+0],
  D[l][k][j+1][i+1]);
for(int l = 0; l < 2; ++l)
for(int k = 0; k < 2; ++k)
for(int j = 0; j < 2; ++j)
for(int i = 0; i < 2; ++i)
 P[2*l+k][2*j+i] = bilerp(
  U[i], V[k],
  E[l][j][0][0],
  E[l][j][0][1],
  E[l][j][1][0],
  E[l][j][1][1]);
```

Our cross-like operation order and the pair-wise parenthesis settings during bilinear interpolation `bilerp` guarantee that neighboring patches and sub-patches evaluate to the exact same floating-point position for same floating-point parameters `u, v`. This is key to crack-freeness:

```
float3 bilerp(float u0, float v0,
 float3 p00, float3 p10,
 float3 p01, float3 p11){
  float u1 = 1 - u0, v1 = 1 - v0;
  float3 P00 = p00 * (u1 * v1);
  float3 P10 = p10 * (u0 * v1);
  float3 P01 = p01 * (u1 * v0);
  float3 P11 = p11 * (u0 * v0);
  return (P00+P11) + (P10+P01);
}
```

We use this evaluation instead of those from Sec. 3.3.1, since it is equally fast but crack-free, as shown in the right of Fig. 4.

3.4. Crack-free Quad Node Rendering with Wedges

Once a quad node is marked for rendering, we approximate it with two triangles spanned by the four corner points of its associated patch. If neighboring quad nodes are of different depth, unwanted cracks occur as in Fig. 5a. We close each crack with an extra triangle, called *wedge*, as shown in green in Fig. 5b and c. A quad node's edge receives a wedge only if this edge had already satisfied the subdivision metric from Sec. 3.2 at a coarser depth than the

© 2026 The Author(s).
Proceedings published by Eurographics - The European Association for Computer Graphics.

quad node. Consequently, quad nodes at finer depths are responsible for closing cracks to coarser depths.

To achieve this, we require the depth at which a quad node's edge was considered sufficiently flat when rendering the quad node, see edge-tier of Sec. 3.2. For a quad node at depth l, this is trivial for the two edges that were created when subdividing the parent node, as these newly created edges must have depth l or finer. For the two other edges, this information (l_0, l_1) is passed down from the parent node, requiring another 4 bits each. Thus, which edges are passed down depends on whether the quad node is a lower-left, upper-left, lower-right, or upper-right node with respect to its parent, see Fig. 5d. We determine this with the least significant bits of the quad node's 2D node index (m,n). Since two edges are trivially crack-free, only up to two wedges are required per quad node.

The *wedge pivot* is the third point that forms a wedge together with the edge, marked as gray in Fig. 5. Its domain coordinates are given by rounding (m,n) to the coarser edge depth l_0 or l_1. Which of the two, and the rounding direction are again dependent on where the quad node is with respect to its parent. If the rounding operation does not change the parametric location, thus would produce a degenerate triangle, the wedge can be omitted. This can be seen in Fig. 5c and e, where rounding u_0 to the parent edge does not change the parametric location. As a wedge triangle is degenerate in the parametric domain of its patch, it cannot overlap with non-wedge triangles of the patch, filling cracks without artifacts.

Note that our crack fixing logic does not require any communication or synchronization between patches. Instead, we have to pay special attention and carefully arrange floating-point operations of the algorithms of Sec. 3.2 and Sec. 3.3.2 to produce identical results on edges of neighboring patches. For example, at the edge-tier, we do not evaluate $\vec{C}(2/3)$ directly, but instead reverse the control point order and evaluate that curve at $1/3$. This is analogous to hardware tessellation, where adjacent patches independently compute matching outer tessellation factors and then fill any resulting cracks with their own geometry, without inter-patch communication.

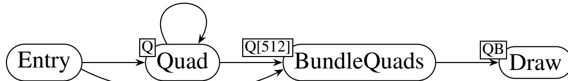
To render a quad node and its wedges, we employ a mesh shader. Here, one mesh shader group writes the geometry of up to 32 quad nodes to the rasterizer. As a quad node with both wedges has 6 vertices and 4 triangles, one mesh shader group outputs 192 vertices and 128 triangles at most. We sample a sub-patch using five curve evaluations. A curve $\vec{C}(u)$ is evaluated by

```
float3 C(float u0, float3[4] p) {
  float u1 = 1 - u0;
  float4 B(u1 * u1 * u1,
       3 * u1 * u0 * u1,
       3 * u0 * u1 * u0,
         u0 * u0 * u0);
  return ((B.x * p[0]) + (B.y * p[1])) +
         ((B.z * p[2]) + (B.w * p[3]));
}
```

where `p[i]` is the control point $\vec{p}_i$ and `u0` is u. Similar to B-Spline evaluation [NLMD12], parenthesis, operation order, and compiler-flags for precise floating-point arithmetic assure watertightness, provided that adjacent base-patches share bitwise-identical boundary control points along their common edges.

3.5. GPU Work Graphs Implementation

We implement our system[1] using Direct3D 12 *GPU work graphs* [Mic25]. A *work graph* is a directed acyclic graph whose vertices are (nodes). Each node is an HLSL compute shader with associated launch metadata. Nodes communicate by sending and receiving [records], which parameterize the work performed by downstream nodes. Although the graph is acyclic, work graphs support *trivial recursion*, allowing a node to enqueue [records] that target itself. A special type of leaf node, the *mesh node*, enables direct rasterization from within a work graph. Mesh nodes encapsulate a pipeline state object containing a mesh shader and a pixel shader. The following work graph subdivides and renders Bézier models:



The (Entry) node processes each base-patch with one thread each. A thread reads the Bézier points of its assigned base-patch and evaluates the metric from Sec. 3.2 to decide whether subdivision is required. If so, it emits four [Q] records, as indicated by the arrow. [Q] occupies 8 bytes for the sub-patch information (l,m,n) of Sec. 3.1 and for the crack-fixing information (l_0,l_1) of Sec. 3.4, and is filled accordingly. Note that GPU work graphs limit the output-record size in thread launch mode to 128 bytes, so explicitly writing out 192 bytes of Bézier points for each sub-patch would be infeasible.

The (Quad) node receives four [Q] records. This node uses the quad node information in [Q] to reconstruct the sub-patch as described in Sec. 3.3.2, evaluates the metric, and, if necessary, creates and sends four subdivided [Q] records to itself.

If (Quad) or (Entry) decide for no subdivision, they pass their input [Q] to (BundleQuads). As an optimization [KOF*25], (BundleQuads) collects up to 512 (Quad)s and batches them into a single record [QB]. Finally, the mesh shader of Sec. 3.4 of the mesh node (Draw) outputs the triangles to the raster stage. (Quad) and (Entry) may also cull a patch if its bounding box lies outside the view frustum. Sub-patches marked ready-for-rendering are back-face culled before bundled. For lighting computations, we compute the normal vector in the pixel shader by evaluating the partial derivatives of the base-patch.

4. Results and Discussion

We evaluate our implementation on an AMD Radeon RX 7900 XTX GPU, driver version 25.30.17.01, at a 3840×2160 resolution. We use the custom kitchen scene of Fig. 1 and 7 containing a Utah teapot as a native Bézier model together with typical converted CCS assets. By combining multiple common test assets in a single scene, we better reflect real-world usage and enable a meaningful evaluation of culling, adaptivity, and overall system behavior. We approximated each extraordinary CCS patch with a single bicubic Bézier patch [LS08]. The scene contains 35 940 base-patches.

[1] https://github.com/CoburgGraphicsLab/AdaptiveBezierSubdivision

© 2026 The Author(s).
Proceedings published by Eurographics - The European Association for Computer Graphics.

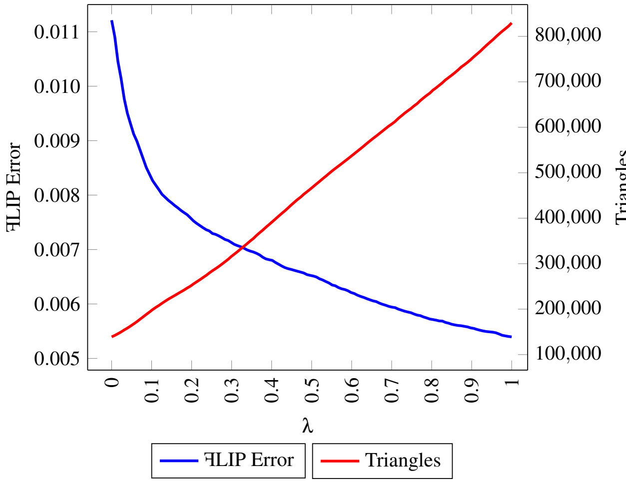


Figure 6: Changes in ꟻLIP error and triangle count with varying λ. For the first camera perspective in Tab. 1, we use a fixed $\varepsilon = 1$. Note that a larger λ shifts the sensitivity of our error metric to the parametrization quality required for accurate texture mapping. As intended, more triangles are generated, and the ꟻLIP error relative to the ground truth decreases.

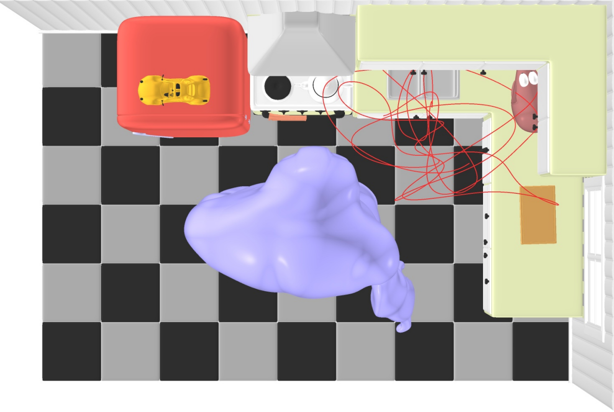


Figure 7: Test Scene with Standard Models. We render with a camera path animation indicated by the red thread.

We found that setting $\lambda = 0.5$ of our subdivision criterion produces consistently similar errors. Fig. 6 shows how changing λ influences the number of generated triangles and image quality.

4.1. Comparison

We compare our method against *hardware tessellation*. To determine the view-adaptive tessellation factors from ε, we use the uniform metric described by Buchenau and Guthe [BG21] and set them in the hull shader. Patches outside the view frustum are culled by setting all tessellation factors to zero. The domain shader evaluates the 3D points for each Bézier patch at the sample locations provided by the hardware tessellator. The pixel shader is identical to the one used in our work-graphs implementation.

We also implemented our subdivision method using *execute-indirect* [Mic25]. Each recursion step is implemented as a separate compute-shader dispatch, with barriers synchronizing in-between. We perform breadth-first subdivision: input patches are read from an input buffer and subdivided patches are written to an output buffer. After each subdivision step, the buffer semantics are swapped. Quad nodes that are ready for rendering are written to a dedicated final buffer. Atomic counters track the next free output location in each buffer. One additional dispatch is used between steps to reset counters and compute thread group count of the next indirect dispatch. For rendering, we use the same mesh and pixel shaders as in our work-graph implementation.

Since our subdivision variants, i.e., the work-graphs and execute-indirect version, implement the same algorithm, they produce almost identical images. Due to the compiler producing different code for both variants, we observe occasional differences during back-face culling of sub-patches that are ready for rendering. However, hardware tessellation produces different images. For fair comparison, we render ground-truth images by tessellating all patches with all tessellation factors set to 64. Then, we compute the ꟻLIP error [ANA21] from the ground-truth against the tessellation and our subdivision variants at various pixel errors ε as shown in Tab. 1.

4.2. Performance

We gather median frame-time and triangle count across an image sequence along the camera path of Fig. 7. Table 2 compares performance metrics of our GPU work graphs, execute-indirect, and hardware tessellation implementation. At a comparable image quality, i.e., $\lambda = 0.5$, we produce images faster than tessellation for $\varepsilon \geq 1$; for $\varepsilon = 0.5$ tessellation is slightly faster. However, for the same geometric error bound, i.e., $\lambda = 0$, our subdivision methods are significantly faster. For all configurations subdivision produces significantly fewer triangles than hardware tessellation. Therefore, our subdivision variants outperform hardware tessellation for frames in which culling is effective or recursion terminates early due to sufficiently planar sub-patches. This effect is amplified with work graphs because it can adjust to dynamic work loads, whereas execute indirect must adhere to a pessimistic and static dispatch plan.

4.3. Memory Requirements

As hardware tessellation is an opaque feature, we are not able to assess the amount of internally required memory. For our execute-indirect variant, we need to allocate two buffers for sub-patch-storage and one final buffer, all prepared for the worst-case scenario to avoid hazardous buffer overflows. Consequently, we limit our execute-indirect variant to a maximum depth 6, requiring $8\,\mathrm{B} \cdot 35940 \cdot \left(4^4 + 4^5 + 4^6\right) = 1.44\,\mathrm{GiB}$. The work graphs runtime requires the application to allocate a moderate amount of *backing memory*. The work-graphs runtime computes the backing memory size such that a given work graph is guaranteed to execute even under worst case assumptions. Our implementation requires 26.39 MiB, while supporting a maximum depth of 10. Therefore, execute-indirect requires orders of magnitude more memory than our work graphs implementation.

© 2026 The Author(s).
Proceedings published by Eurographics - The European Association for Computer Graphics.

ε							
0.5	Tessellation	0.00381	0.00293	0.00224	0.00512	0.00449	0.00510
	Subdivision	0.00391	0.00287	0.00239	0.00408	0.00467	0.00503
1	Tessellation	0.00613	0.00465	0.00408	0.00839	0.00720	0.00887
	Subdivision	0.00652	0.00466	0.00420	0.00695	0.00755	0.00831
2	Tessellation	0.01027	0.00767	0.00687	0.01272	0.01143	0.01441
	Subdivision	0.01019	0.00809	0.00666	0.01177	0.01323	0.01445

Table 1: Quality Evaluation. At six different perspectives and for various pixel errors ε, we compute ꟻLIP errors of our **tessellation** variant and our **subdivision** variants against the ground truth. Setting $\lambda = 0.5$ produces similar errors. Underlined numbers show the better error.

			$\lambda = 0.5$		$\lambda = 0$	
		Tessellation	Execute-Indirect	Work Graphs	Execute-Indirect	Work Graphs
$\varepsilon = 0.5$	Render Time	0.477 ms	0.499 ms	0.490 ms	0.372 ms	0.331 ms
	Triangles (%Wedges)	661 556	324 222 (22.8 %)	324 222 (22.8 %)	67 937 (23.3 %)	67 856 (23.3 %)
$\varepsilon = 1$	Render Time	0.446 ms	0.437 ms	0.412 ms	0.359 ms	0.313 ms
	Triangles (%Wedges)	356 267	156 717 (22.6 %)	156 717 (22.6 %)	33 674 (22.0 %)	33 653 (22.0 %)
$\varepsilon = 2$	Render Time	0.426 ms	0.393 ms	0.358 ms	0.342 ms	0.303 ms
	Triangles (%Wedges)	207 857	77 883 (22.8 %)	77 886 (22.8 %)	15 863 (18.9 %)	15 863 (18.9 %)

Table 2: Benchmark Results. We measure median render time and triangle counts on a 2000 frame long camera path animation from Fig. 7 for different pixel-space errors ε. Our methods produce a similar visual error to tessellation at $\lambda = 0.5$ and the same geometric error at $\lambda = 0$. For our subdivision methods, we also provide the ratio of wedge triangles for crack-fixing to the total amount.

4.4. Crack-freeness

We can close cracks when our reconstruction method is invariant under parameter domain reordering. To evaluate that, we reconstruct the same sub-patch using all possible parameter reorderings and verify that they produce the same reconstructed sub-patch. We perform the test separately and directly on the GPU and mark patches failing that test. While the reconstruction of Sec. 3.3.2 passes the test, all other reconstruction methods fail it.

4.5. Implementation Effort

Being specifically designed for patch rendering, we found hardware tessellation the easiest of all variants to realize. Our work-graphs implementation was significantly easier to implement than its execute-indirect counterpart. While both variants share the same core-functionality, execute-indirect uses three buffers and four kernels, iterates over multiple dispatches, requires explicit synchronization and atomic counters. Our work-graphs implementation requires a single dispatch on the host side with reusable setup code.

5. Conclusion and Future Work

We introduced a memory-efficient patch representation, a subdivision reconstruction method, a subdivision criterion, and a crack-closing technique, in a numerically-stable high-performance bicubic Bézier renderer. Next, we want to extend it to CCS surfaces. We observed that a 1:4 split problem solved with work graphs can outperform existing approaches in case the amount of work is unknown upfront. This can serve as a model for similar recursive problems such as level-of-detail or bounding volume hierarchy algorithms and applications beyond graphics.

Acknowledgments

Keenan Crane, Bay Raitt, and OpenSubdiv kindly donated the models Pig, Big Guy, and Car to academia.

References

[AB06] ANDREWS J., BAKER N.: Xbox 360 system architecture. *IEEE Micro 26*, 2 (Mar. 2006), 25–37. 3

[ANA21] ANDERSSON P., NILSSON J., AKENINE-MÖLLER T.: Visualizing and communicating errors in rendered images. In *Ray Tracing Gems II*, Marrs A., Shirley P., Wald I., (Eds.). 2021, ch. 19, pp. 301–320. 7

[BG21] BUCHENAU C., GUTHE M.: Real-time curvature-aware reparametrization and tessellation of Bézier surfaces. In *Vision, Modeling, and Visualization*. The Eurographics Association, Dresden, 2021, pp. 47–54. 7

© 2026 The Author(s).
Proceedings published by Eurographics - The European Association for Computer Graphics.

[Cat74] CATMULL E. E.: *A subdivision algorithm for computer display of curved surfaces*. PhD thesis, University of Utah, 1974. 3

[DV21] DUPUY J., VANHOEY K.: A halfedge refinement rule for parallel Catmull-Clark subdivision. *Computer Graphics Forum 40*, 8 (2021), 57–70. 2

[Eis11] EISENACHER C.: *Tessellation and Texture Synthesis: Advanced Methods for Interactive Content Creation in Movies*. PhD thesis, Friedrich-Alexander Universität Erlangen-Nürnberg, 2011. 3

[EL10] EISENACHER C., LOOP C. T.: Data-parallel micropolygon rasterization. In *Eurographics (Short Papers)* (2010), pp. 53–56. 3

[EML09] EISENACHER C., MEYER Q., LOOP C.: Real-time view-dependent rendering of parametric surfaces. In *Proceedings of the symposium on Interactive 3D graphics and games* (2009), pp. 137–143. 2, 3

[Far02] FARIN G. E.: *Curves and Surfaces for CAGD: A Practical Guide*, 5th ed. Morgan Kaufmann series in computer graphics and geometric modeling. Morgan Kaufmann, San Francisco, CA, 2002. 2, 3

[GBK05] GUTHE M., BALÁZS A., KLEIN R.: GPU-based trimming and tessellation of NURBS and T-spline surfaces. In *ACM SIGGRAPH 2005 Papers* (New York, NY, USA, 2005), SIGGRAPH '05, ACM, pp. 1016–1023. 3

[KOCM23] KUTH B., OBERBERGER M., CHAJDAS M., MEYER Q.: Edge-friend: Fast and deterministic Catmull-Clark subdivision surfaces. *Computer Graphics Forum 42*, 8 (2023). 2

[KOF*24] KUTH B., OBERBERGER M., FABER C., BAUMEISTER D., CHAJDAS M., MEYER Q.: Real-time procedural generation with GPU work graphs. *Proc. ACM Comput. Graph. Interact. Tech. 7*, 3 (Aug. 2024), 47:1–47:16. 3

[KOF*25] KUTH B., OBERBERGER M., FABER C., PFEIFER P., TABAEI S., BAUMEISTER D., MEYER Q.: *Real-Time GPU Tree Generation*. The Eurographics Association, 2025. ISSN: 2079-8687. 3, 6

[KOM25] KUTH B., OBERBERGER M., MEYER Q.: GPU work graphs. In *SIGGRAPH Courses* (New York, NY, USA, 2025), SIGGRAPH Courses '25, ACM. 3

[KWW*24] KAWALA F., WITTMANN M., WILDGRUBE F., TROJAHN P., BAUMEISTER D., MEMBARTH R.: Work graphs based denoising for real-time ray tracing. Poster, 2024. Presented at High Performance Graphics 2024. 3

[LS08] LOOP C., SCHAEFER S.: Approximating Catmull-Clark subdivision surfaces with bicubic patches. *ACM Trans. Graph. 27*, 1 (Mar. 2008). 2, 3, 6

[Mic25] MICROSOFT: *DirectX-Specs*, 2025. URL: `https://microsoft.github.io/DirectX-Specs/`. 2, 3, 6, 7

[MWS*20] MLAKAR D., WINTER M., STADLBAUER P., SEIDEL H.-P., STEINBERGER M., ZAYER R.: Subdivision-specialized linear algebra kernels for static and dynamic mesh connectivity on the GPU. *Computer Graphics Forum 39*, 2 (2020), 335–349. 2

[NKF*16] NIESSNER M., KEINERT B., FISHER M., STAMMINGER M., LOOP C., SCHÄFER H.: Real-time rendering techniques with hardware tessellation. *Computer Graphics Forum 35*, 1 (2016), 113–137. 3

[NLMD12] NIESSNER M., LOOP C., MEYER M., DEROSE T.: Feature-adaptive GPU rendering of Catmull-Clark subdivision surfaces. *ACM Trans. Graph. 31*, 1 (Feb. 2012). 6

[PO08] PATNEY A., OWENS J. D.: Real-time Reyes-style adaptive surface subdivision. In *ACM SIGGRAPH Asia 2008 Papers* (New York, NY, USA, 2008), SIGGRAPH Asia '08, ACM. 2, 3

[Ram89] RAMSHAW L.: Blossoms are polar forms. *Computer Aided Geometric Design 6*, 4 (1989), 323–358. 2

[Sch09] SCHWARZ M.: *Soft Shadows, Curved Surfaces and Perceptual Sensitivity: Advanced Methods for Improving Realism in Real-time Rendering*. PhD thesis, University of Erlangen-Nuremberg, 2009. 3

[Sei93] SEIDEL H.-P.: An introduction to polar forms. *IEEE Comput. Graph. Appl. 13*, 1 (Jan. 1993), 38–46. 2

[SS09] SCHWARZ M., STAMMINGER M.: Fast GPU-based adaptive tessellation with CUDA. *Computer Graphics Forum 28*, 2 (2009), 365–374. 3

[Str09] STRUGAR F.: Continuous Distance-Dependent Level of Detail for Rendering Heightmaps. *Journal of Graphics, GPU, and Game Tools 14*, 4 (2009), 57–74. 3

[ULW*25] UNTERGUGGENBERGER J., LIPP L., WIMMER M., STEINBERGER M., KERBL B., SCHÜTZ M.: Real-time rendering methods with adaptive levels of detail for fast rendering of parametric objects on modern GPUs. *IEEE Transactions on Visualization and Computer Graphics* (2025), 1–12. 3

[WC25] WANG K., CHEN R.: PaRas: A rasterizer for large-scale parametric surfaces. In *SIGGRAPH Conference Papers* (New York, NY, USA, 2025), SIGGRAPH Conference Papers '25, ACM. 3

[WET*25] WILDGRUBE F., EHRETT P., TROJAHN P., MEMBARTH R., BECKMANN B., BAUMEISTER D., CHAJDAS M.: GPUs all grown-up: Fully device-driven SpMV using GPU work graphs. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture* (New York, NY, USA, 2025), ISCA '25, ACM, pp. 1777–1791. 3

[XLC*23] XIONG R., LU Y., CHEN C., ZHU J., ZENG Y., LIU L.: ETER: Elastic tessellation for real-time pixel-accurate rendering of large-scale NURBS models. *ACM Trans. Graph. 42*, 4 (July 2023). 3

[ZLCL25] ZENG Y., LU Y., CHEN C., LIU L.: WATER: Watertight tessellation for real-time pixel-accurate rendering of large-scale surfaces. *ACM Trans. Graph. 44*, 6 (Dec. 2025). 3

© 2026 The Author(s).