DiagSplit: Parallel, Crack-free, Adaptive Tessellation for Micropolygon Rendering

Matthew Fisher* Kayvon Fatahalian* Solomon Boulos* Kurt Akeley† William R. Mark‡ Pat Hanrahan*

Abstract

We present DIAGSPLIT, a parallel algorithm for adaptively tessellating displaced parametric surfaces into high-quality, crack-free micropolygon meshes. DIAGSPLIT modifies the split-dice tessellation algorithm to allow splits along non-isoparametric directions in the surface's parametric domain, and uses a dicing scheme that supports unique tessellation factors for each subpatch edge. Edge tessellation factors are computed using only information local to subpatch edges. These modifications allow all subpatches generated by DIAGSPLIT to be processed independently without introducing T-junctions or mesh cracks and without incurring the tessellation overhead of binary dicing. We demonstrate that DIAGSPLIT produces output that is better (in terms of image quality and number of micropolygons produced) than existing parallel tessellation schemes, and as good as highly adaptive split-dice implementations that are less amenable to parallelization.

Keywords: tessellation, micropolygons, real-time rendering

1 Introduction

The desire for high realism in computer graphics has led to very complex geometric models. Detailed artistry and 3D data acquisition produce surfaces that contain millimeter-scale features. These surfaces would require millions of polygons to be represented accurately, but can be represented compactly by using analytic descriptions (such as parametric or subdivision surfaces) in conjunction with procedural or texture-mapped displacement to capture intricate detail.

Many graphics systems require surfaces to be tessellated into polygon meshes for rendering. Sampling surfaces at any fixed resolution produces a static tessellation that, depending on the view, may contain either too few polygons (resulting in visual artifacts) or too many polygons (yielding low performance). Adaptive tessellation is required to produce a mesh that yields high visual quality, but low rendering cost, regardless of variation in surface detail or camera location.

Our work seeks to enable high-quality tessellation in future interactive systems. In this paper, we design an algorithm, DIAGSPLIT, for generating view-dependent tessellations of displaced parametric surfaces that is suitable for high-throughput, parallel processing. DIAGSPLIT creates meshes that do not contain cracks or T-junctions, and it adapts well even when surface position depends heavily on displacement.

DIAGSPLIT does not strive to generate the minimum number of (potentially large) polygons required to accurately represent a surface; instead, its output is a micropolygon mesh (a mesh containing polygons of less than a pixel in area). Micropolygons are a common rendering primitive in offline rendering because they faithfully represent high-complexity surfaces and support high-quality object-space shading. We seek to evolve the real-time graphics pipeline to support efficient micropolygon generation and have designed DIAGSPLIT specifically to integrate tightly with future pipeline implementations.

*Stanford University:
{mdfisher, kayvonf, boulos, hanrahan}@graphics.stanford.edu
†Microsoft Research: kakeley@microsoft.com
‡Intel Corporation: william.r.mark@intel.com

		Adaptive	Parallel	Crack-free
NOSPLIT	(D3D11)	[limited]	✓	✓
BINSPLIT	(REYES)	[overtess]	✓	✓
ISOSPLIT	(REYES)	✓		✓
DIAGSPLIT	(this paper)	✓	✓	✓

Table 1: *Comparison of tessellation algorithms: popular schemes choose different quality and efficiency trade-offs. The algorithm presented in this paper,* DIAGSPLIT, *performs adaptive crack-free tessellation while remaining parallel.*

2 Background

Adaptive tessellation in offline and real-time systems has been studied extensively. Many approaches to adaptive tessellation aim to approximate the surface using as few polygons as possible. These schemes produce large polygons for flat regions of surfaces [Moule and McCool 2002; Eisenacher et al. 2009]. They do not seek to produce micropolygons.

Unlike modern GPUs, micropolygon renderers compute surface appearance at mesh vertices, rather than fragments. This presents unique challenges for tessellation because surface sampling must be sufficiently fine to capture geometric and appearance detail. A tessellation should sample the surface uniformly in screen space, producing at least one micropolygon per pixel (this rate is adjustable depending on need). Undersampling results in geometric artifacts due to piecewise linear interpolation of surface x-y position (e.g. silhouette errors) and depth (e.g. visibility errors). Interpolation of vertex appearance creates shading artifacts (faceted shading) when the surface is undersampled. The performance penalty of overtessellation in a micropolygon pipeline is severe. In addition to increasing surface evaluation and rasterization work, overtessellation results in extra shading as well. Applications that perform expensive shading computations will have performance that is strongly correlated with the number of micropolygons in the resulting tessellation.

Adaptive tessellation must also avoid frame-to-frame discontinuities ("popping") as tessellation changes in response to object or camera movement, and must avoid producing mesh cracks. Micropolygons are sufficiently small that popping artifacts are rarely visible, however, avoiding cracks is difficult in a parallel implementation. We wish to independently process many regions of the surface simultaneously, yet the resulting tessellations must align properly at region boundaries to avoid cracks. As shown in Table 1, the most widely used tessellation schemes achieve some, but not all, of our algorithm goals (adaptive, crack-free, and parallel).

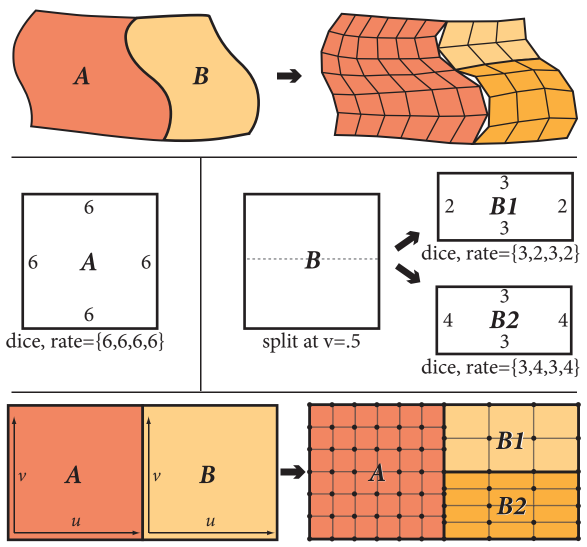


Figure 1: *Adaptive tessellation can produce cracks along patch boundaries. Top: Tessellation of surface patches A and B into micropolygons. Cracks are visible along the A-B boundary. Middle: Patch A is diced directly. Patch B is split into subpatches B1 and B2. Bottom: Partitioning of A and B's parametric domains by Split-Dice. Tessellations of A and B sample the surface at different domain points (black dots) along the shared edge, resulting in cracks.*

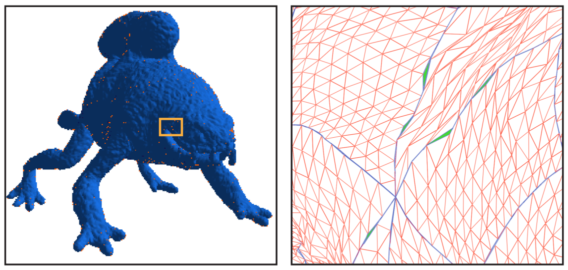


Figure 2: *A tessellated surface with cracks. Left: Cracks in the surface reveal the back side of the model, which shows up as orange points (backfacing geometry is rendered in orange). Right: Zoomed view of the tessellation. Patch boundaries are shown in blue; backfacing geometry (seen through cracks) is green.*

2.1 Split-Dice

Split-Dice is a powerful algorithm used by renderers such as Pixar's Reyes to tessellate surfaces based on the Lane-Carpenter algorithm [Lane et al. 1980]. It serves as the framework for the algorithms we will compare in this paper. The first phase, Split, recursively subdivides patches to create smaller subpatches. Split allows the tessellation to adapt to variations in the projection of the surface to screen coordinates, as the final tessellations of subpatches are permitted to have different densities of polygons across the surface. Splitting terminates when adaptivity is no longer needed; that is, when uniform parametric tessellation of each subpatch is estimated to produce micropolygons that are all approximately a pre-specified area in screen space. Then, the dice phase uniformly tessellates subpatches generated by Split into micropolygon meshes. The advantage of using both Split and Dice is that Split provides reasonable adaptivity to varying surface complexity, while Dice retains the efficiency of uniform tessellation at a subpatch granularity.

The decision to split or dice a patch is made by estimating the variation in the surface's screen-space derivative with respect to each parametric direction. Surfaces whose derivative varies significantly across the patch are not suitable for uniform tessellation and should be split. Surfaces whose rate is approximately constant across the patch can be diced using a number of micropolygons determined by this constant. Computing these derivatives can be done analytically for many surface types, such as Bezier patches [Blinn 1978; Catmull 1974; Clark 1979], although this computation does not take displacement mapping into account.

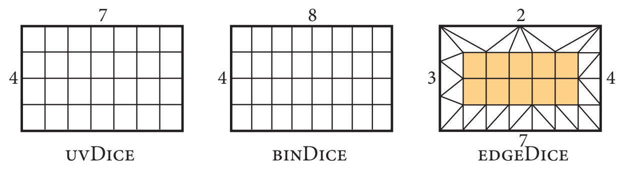


Figure 3: *Comparison of dicing methods: The Reyes dicer (*UVDICE*) accepts tessellation factors for the patch's u and v parametric directions. It produces a uniform mesh tessellation according to these factors. Binary UV dice only tessellates each edge using a power-of-two number of segments. The D3D11 tessellator (*EDGEDICE*) can accommodate per edge factors. Each edge of the patch is tessellated uniformly with a number of segments equal to the provided factor. The* EDGEDICE *scheme "stitches" a uniformly tessellated interior mesh to segments along the edges.*

The split phase is important because uniform parametric tessellation of base primitives does not always yield a good surface tessellation. Patches with poorly distributed control points, varying curvature, or which undergo perspective foreshortening suffer from overtessellation, undertessellation, or poor distribution of polygon size if the input patches are diced directly. To avoid these problems, developers working in environments without Split are encouraged to carefully author content that is suitable for uniform tessellation. This incurs a human cost and likely does not scale to environments relying heavily on user-generated content.

The execution of Dice lends itself to a data-parallel implementation because the position and attributes for each vertex in the output mesh can be evaluated in parallel. In contrast, Split presents two challenges to a high-performance implementation. First, it performs unbounded data amplification, potentially generating a large number (e.g., thousands) of subpatches from a single base primitive. Second, it complicates crack avoidance by dynamically introducing additional boundaries between subpatches, not just base primitives. Care must be taken to make the final mesh crack-free along these new edges.

To understand how cracks occur, consider adjacent surface patches A and B in Figure 1 (top). For each patch, Split will either partition the patch or compute tessellation factors for dicing. Patch A is not split, and is diced uniformly using six micropolygons along each edge. Patch B is split at its midpoint in the v parametric direction creating subpatches B1 and B2, which dice their span of the A-B edge using two and four micropolygons respectively. Patch A's tessellation along this edge does not match that of Patch B (Figure 1-bottom). Since the sampling of the curved surface is different, surface cracks appear in the micropolygon mesh shown in the top-right of the figure. Figure 2 shows a tessellation that produces cracks on a real surface.

DIAGSPLIT combines ideas from both the Reyes and D3D11 systems; we briefly describe the merits of these two approaches in the context of the Split-Dice framework.

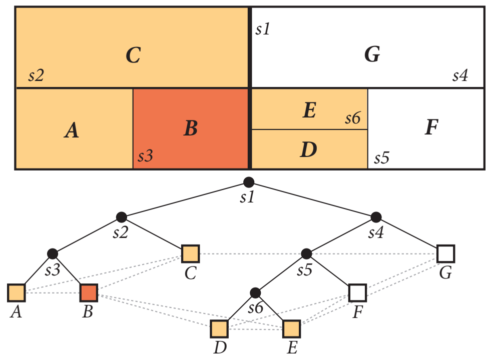


Figure 4: *Patch dependencies in* IsoSplit*: The tree of six splitting decisions ($s1$-$s6$) used to partition a base primitive into seven subpatches ($A \rightarrow G$). The split tree structure encodes subpatch adjacency information. For example, the state of patches A, C, D, and E must be maintained in memory to stitch patch B to its neighbors. Dependencies between subpatches are shown as gray dotted lines.*

2.2 Reyes Tessellation

Reyes [Cook et al. 2008] is a direct implementation of the Split-Dice algorithm. It divides each base primitive into subpatches by partitioning along isoparametric directions. In this paper, we refer to isoparametric splitting as IsoSplit. The dice phase tessellates each subpatch emitted by Split into a grid of micropolygons by uniformly tessellating the subpatch in each parametric direction. We refer to this implementation of dicing as uvDice (Figure 3-left). The decision to split or dice, and, if dicing, the number of segments to tessellate in each direction, are determined by computing derivatives across the subpatch interior.

Many approaches have been taken to produce a crack-free mesh using Reyes. Some Reyes implementations (e.g., Apodaca and Gritz [2000] and Foster [2009]) fix cracks by leveraging information from adjacent subpatches to stitch together meshes once final vertex positions are known. Adjacency information can be encoded by explicit pointers or by maintaining structures representing splitting decisions as shown in Figure 4. Interior nodes of this tree correspond to splitting decisions. Leaf nodes correspond to diced subpatches. In this example, fixing cracks along the edges of subpatch B requires access to vertex information from four neighboring subpatches (A,C,D,E). Notice that this scheme introduces dependencies between subpatches as indicated by the dotted lines. Subpatch B cannot be completed until vertex positions of subpatches A,C,D,E are known. Conversely, subpatches A,C,D,E must be maintained by the system until subpatch B is complete. These dependencies make it difficult to stream subpatch data through the graphics system (subpatch edge information must be retained in memory until the entire base primitive is complete), bloating working sets and preventing large-scale parallelism. For this reason, we do not consider crack fixing by maintaining adjacency information a viable solution for real-time graphics systems.

Other Reyes implementations avoid cracks without mesh adjacency information. Instead, they constrain subpatch tessellations to power-of-two rates (called binary dicing, see binDice, Figure 3) and constrain splitting to ensure adjacent subpatches agree on the rate at which their shared edge is diced (Apodaca and Gritz [2000] refers to this as pasting). We use BinSplit to refer to a splitting scheme that relies on binary dicing (Table 1, second row). While binary dicing is attractive due to its simplicity, it unfortunately results in poor tessellations. For example, if a patch is optimally diced with 12 segments in one parametric direction, it will have to either be diced with 8 segments (undertessellation) or 16 segments (overtessellation). In our renderer, we observe an approximately 2x increase in tessellation polygon count when enabling binary dicing and rounding tessellations up to the nearest power of two to avoid undertessellation.

2.3 D3D11 Tessellation

Tessellation functionality featured in the D3D11 pipeline [Microsoft 2009] does not provide the ability to split base primitives but provides three pipeline stages that together perform a flexible implementation of Dice (we call this scheme NoSplit, Table 1, first row). The D3D11 Hull stage emits a surface parameterized on either a quadrilateral or triangular domain and surface tessellation factors along each domain edge. Then, a fixed-function Tessellate stage generates a mesh with vertices at domain points (u, v) determined by the tessellation factors [Moreton 2001]. Last, the Domain shader stage evaluates the surface's position and custom vertex attributes at each point, yielding a renderable mesh.

In contrast to uvDice, which utilizes only two independent tessellation factors (one for each parametric direction), the D3D11 tessellator generates a triangle mesh from four independent factors (one for each domain edge). We call this more flexible dicing strategy edgeDice (Figure 3-right). In edgeDice, both the edges and the interior of the patch are uniformly tessellated in the parametric domain. The number of interior segments in a given parametric direction is taken to be the maximum of the two opposing edge factors, scaled by an interior tessellation scale parameter S between 0 and 1 (although different scaling factors are allowed in each parametric direction, we will make use of only isotropic scaling). Triangles along the edge of the tessellation stitch the uniform interior to edge segments.

D3D11 tessellation is designed for real-time performance. Each base primitive is processed independently, enabling parallelism. edgeDice is implemented efficiently in fixed-function hardware and surface evaluation at mesh vertices is data-parallel. However, the performance benefits of this design are tempered by three notable constraints.

First, as stated above, the scheme lacks the adaptability of split (the contribution of this paper overcomes this limitation).

Second, D3D11 tessellation requires that primitives support arbitrary parametric evaluation. This enables surface evaluation to be expressed as a domain shader program operating on a single parametric location, but the convenience of this abstraction prevents the use of efficient forward differencing schemes that rely on uniform spacing of domain points [Lien et al. 1987]. While many subdivision surfaces can be evaluated directly [Stam 1998], these techniques often require one to two levels of subdivision to meet the conditions for direct evaluation, resulting in overtessellation. Furthermore, direct evaluation of patches with extraordinary vertices is computationally expensive in comparison to a regular bicubic patch. Approximation schemes [Loop and Schaefer 2008] can be used to produce a very good approximation to the underlying subdivision surface with a set of Bezier patches. Recent work has extended this approach to include surfaces with creases and corners [Kovacs et al. 2009], and we feel research will continue to expand the set of surfaces with an efficient, parametric evaluation.

Third, independent primitive processing requires special care by application developers to prevent cracks. The parametric location of vertices generated by D3D11 tessellation is determined entirely by the edge factors computed by the Hull Stage. Thus, to ensure a crack-free boundary between two primitives, Hull stage processing of each primitive must produce an identical tessellation factor for the shared edge. Further, domain shader execution on parametric points along edges must always yield the same surface position, regardless of the patch the vertex belongs to. These properties are non-trivial for a shader author to guarantee, especially in the presence of texture-based displacement; when adjacent base primitives have different displacement texture coordinates (e.g. on texture atlas seams), they may not evaluate consistently along the shared edge. Some approaches used to solve this problem are to use seamless texture atlases [Purnomo et al. 2004] or to avoid UV-atlas assignment altogether [Burley and Lacewell 2008].

3 Algorithm

In this section we describe DIAGSPLIT, a variant of the Split-Dice algorithm designed for real-time pipelines. DIAGSPLIT tessellates parametric surfaces into micropolygon meshes. It adapts well to surface complexity, does not rely on inter-subpatch mesh stitching to eliminate cracks, and does not incur the overtessellation of binary dicing.

Following the NOSPLIT and BINSPLIT algorithms described in Section 2, DIAGSPLIT determines surface tessellation along subpatch edges using only properties of the edge. Subsequently, DIAGSPLIT determines surface tessellation of the subpatch interior using the Split-Dice algorithm. The interior tessellation is guaranteed to match the previously-determined edge behavior, ensuring that there are no T-junctions or cracks. DIAGSPLIT meets this requirement via two significant modifications to Reyes Split-Dice:

- DIAGSPLIT is permitted to split subpatches along non-isoparametric directions (hence the name DIAGSPLIT). Non-isoparametric splits occur *only* when necessary to prevent cracks.
- DIAGSPLIT requires EDGEDICE dicing to stitch tessellations of subpatch interiors to the tessellation required along edges.

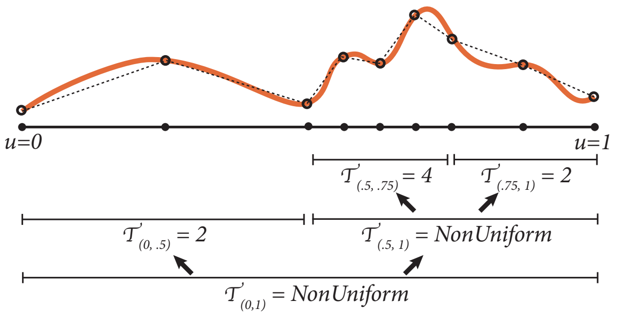
In addition to these changes, DIAGSPLIT also considers the final, displaced position of the surface when computing edge tessellation factors and sets EDGEDICE's interior tessellation scale parameter S to produce micropolygons that closely approximate a user-specified target area.

3.1 DiagSplit

Recall that in NOSPLIT the D3D11 Hull shader computes uniform tessellation factors for all patch edges. In DIAGSPLIT, we define the tessellation along an edge using the function $\mathcal{T}$. Given an edge, $\mathcal{T}$ designates that the edge can be uniformly diced using t segments, or that non-uniform tessellation along the edge is necessary. If non-uniform tessellation is required, the edge is partitioned at its parametric midpoint, and $\mathcal{T}$ is used to determine the tessellation along each partition. This process, applied recursively, fully determines a piecewise-uniform adaptive tessellation along the original edge (Figure 5).

DIAGSPLIT ensures that the tessellation of the interior of the patch generated by the Split-Dice process conforms to the behavior along edges dictated by $\mathcal{T}$. This guarantee holds for edges of base patches and also holds recursively for edges introduced by splits.

DIAGSPLIT's behavior is simple when $\mathcal{T}$ dictates that tessellation along all edges of a subpatch is uniform. The subpatch is diced using EDGEDICE according to the four edge tessellation factors. If at least one subpatch edge requires non-uniform tessellation, the subpatch cannot be diced and must be split. Consider the case where the tessellation must be non-uniform along both edges in the u parametric direction (Figure 6-top). In accordance with the edge behavior described above, DIAGSPLIT will split the subpatch along the line connecting the parametric midpoint of each edge. Notice that this behavior is equivalent to that of ISOSPLIT.



u=0
u=1
$\mathcal{T}'_{(.5,.75)} = 4$
$\mathcal{T}'_{(.75,1)} = 2$
$\mathcal{T}'_{(0,.5)} = 2$
$\mathcal{T}'_{(.5,1)} = NonUniform$
$\mathcal{T}'_{(0,1)} = NonUniform$

Figure 5: DIAGSPLIT *produces tessellations that are piecewise-uniform along edges. In this example, tessellation along an edge over the 0-1 parametric domain is determined by $\mathcal{T}$. $\mathcal{T}$ computes an integer tessellator factor when the surface should be tessellated uniformly along an edge. Otherwise,* DIAGSPLIT *will perform a split dividing the edge at its parametric midpoint and $\mathcal{T}$ is used to set the tessellations along the two partitions.*

When only one edge in the u parametric direction forces non-uniform tessellation (Figure 6-middle, bottom), DIAGSPLIT will split the subpatch along the line between the parametric midpoint of the non-uniform bottom edge and *some point* in the uniform tessellation along the top edge. Our implementation chooses the point closest to the parametric midpoint. If the edge tessellation factor for the top edge is odd, this split occurs along a non-isoparametric line (a diagonal in parametric space). As a result, DIAGSPLIT can generate subpatches with non-isoparametric quadrilateral domains. When splitting is required along both parametric directions, DIAGSPLIT is free to choose which split to perform first or it can implement a split that directly produces four subpatches.

Following the behavior described above, DIAGSPLIT generates tessellations that connect adjacent subpatch interior regions (uniformly tessellated) using two rows of triangles [Rockwood et al. 1989]. Each subpatch interior is independently stitched to its own edges by EDGEDICE. Adjacent patches stitch to the same segments along their shared edge, so no stitching across subpatches is required to prevent cracks.

Pseudocode for a recursive implementation of DIAGSPLIT operating on quadrilateral domain subpatches is given in Figure 7. The function *DiagSplit* carries out the splitting procedure given a subpatch defined by its four parametric corners (*SubPatch*) and tessellation factors (*EdgeFactors*) for all edges (note that edge tessellation factors can take on the special value *NonUniform*). The subroutine *PartitionEdge* is used to compute split points and tessellation factors for new subedges when splits occur. The provided implementation easily extends to triangle domains where each triangle subpatch is split into triangular subpatches. Although not shown in the example, in the rare condition that an edge with a tessellation factor of one is partitioned in a split, we produce a triangular child subpatch and proceed with triangle-domain tessellation.

We call the reader's attention to a detail of the pseudocode that was not discussed in the description of the DIAGSPLIT algorithm above. When partitioning an edge that is assigned a uniform tessellation factor (the "else" clause of *PartitionEdge*), our implementation derives tessellation factors for subedges without additional calls to $\mathcal{T}$. This ensures that subpatch tessellations together contain exactly t uniform segments as required. Calling $\mathcal{T}$ to determine the tessellation along the subedges is incorrect in this case, as there is no

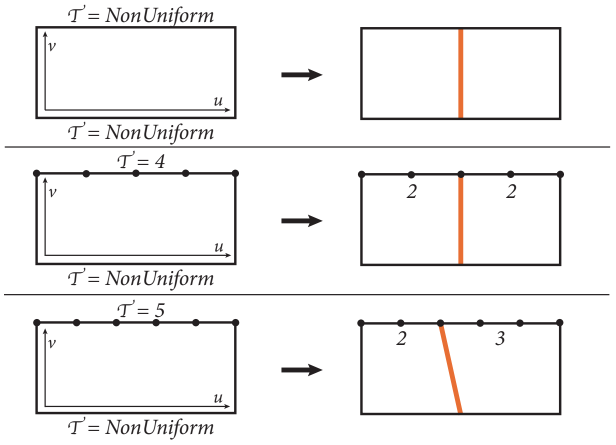


Figure 6: DIAGSPLIT *produces tessellations that adhere to edge behavior defined by $\mathcal{T}$. Top: When tessellation along both edges in the same parametric direction is non-uniform,* DIAGSPLIT *splits the subpatch along the line through the midpoint of both edges. Middle, Bottom: When only one of the edges in a parametric direction requires a split, the split runs through the vertex on the opposite edge closest to the edge midpoint. Subedges of the uniform edge are constrained to ensure agreement with the tessellation factor dictated by $\mathcal{T}$. Bottom: Partitioning a uniform edge requiring an odd number of tessellation segments requires a non-isoparametric split.*

guarantee it would yield edge factors that sum to the value t.

D3D11's EDGEDICE implementation positions mesh vertices within the $[0,1]^2$ domain. Although DIAGSPLIT produces subpatches spanning non-isoparametric quadrilateral domains, EDGEDICE can be used unchanged. We map EDGEDICE's output in the unit square to the base patch's parametric domain using bilinear interpolation. The surface is then evaluated directly at these parametric locations.

It is often useful to compute subpatch bounding boxes during rendering. Unfortunately, the non-isoparametric domains generated by DIAGSPLIT complicate computation of these bounds. For example, the bounding box of a bicubic Bezier patch is efficiently computed from the patch's control cage. However, the non-isoparametric regions of a bicubic base patch do not constitute bicubic patches, precluding this efficient implementation. We compute conservative subpatch object-space bounds by computing a bounding box of the subpatch in parametric space, and then bounding the bicubic patch that corresponds to this isoparametric region.

3.2 Edge Tessellation Factors

DIAGSPLIT's tessellation quality depends heavily on the implementation of $\mathcal{T}$. $\mathcal{T}$ should be cheap to compute and accurately estimate the number of segments needed to tessellate an edge well. We define a good tessellation to be one where no segment exceeds a maximum-specified screen space length L.

Both analytic and sampling-based methods have been used to compute tessellation factors. For example, it is possible to compute the surface derivatives of a Bezier patch directly from the control points. These derivatives can then be used to compute an upper bound on the tessellation factor of an edge [Clark 1979; Rockwood et al. 1989]. Large variation in surface derivative along an edge indicates that the edge is non-uniform and needs to be split. One disadvantage of this analytic approach is that it does not take the displaced positions of the surface into account. Also, it is more complicated to apply to non-isoparametric subpatch edges because

```
$DiagSplit(SubPatch = \{P_{00}, P_{10}, P_{11}, P_{01}\},$
            $EdgeFactors = \{t_{v=0}, t_{u=1}, t_{v=1}, t_{u=0}\})$
  if $t_{v=0}$ or $t_{v=1} = NonUniform$
    $\{P_{v=0}, t^a_{v=0}, t^b_{v=0}\} \leftarrow PartitionEdge(P_{00}, P_{10}, t_{v=0})$
    $\{P_{v=1}, t^a_{v=1}, t^b_{v=1}\} \leftarrow PartitionEdge(P_{01}, P_{11}, t_{v=1})$
    $t_{split} \leftarrow \mathcal{T}(P_{v=0}, P_{v=1})$
    $Split(\{P_{00}, P_{v=0}, P_{v=1}, P_{01}\}, \{t^a_{v=0}, t_{split}, t^a_{v=1}, t_{u=0}\})$
    $Split(\{P_{v=0}, P_{10}, P_{11}, P_{v=1}\}, \{t^b_{v=0}, t_{u=1}, t^b_{v=1}, t_{split}\})$
  else if $t_{u=0}$ or $t_{u=1} = NonUniform$, ...
  else dispatch ($SubPatch$, $EdgeFactors$) to tessellator
```

```
$PartitionEdge(P_{\text{start}}, P_{\text{end}}, EdgeFactor = t)$
  if $t = NonUniform$
    $P \leftarrow (P_{\text{start}} + P_{\text{end}})/2$
    $t_0 \leftarrow \mathcal{T}(P_{\text{start}}, P), t_1 \leftarrow \mathcal{T}(P, P_{\text{end}})$
  else
    Choose vertex index $I = Floor(t/2)$ as split vertex
    $P \leftarrow$ Parametric coordinates of vertex $I$
    $t_0 \leftarrow I, t_1 \leftarrow t - I$
  return $\{P, t_0, t_1\}$
```

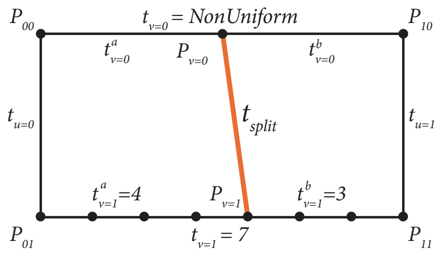


Figure 7: DIAGSPLIT*'s splitting algorithm: The function DiagSplit takes four parametric coordinates that define an input patch and emits subpatches that, when diced, contain micropolygons with near-uniform area.*

```
$\mathcal{T}(P_{\text{start}}, P_{\text{end}})$
  for $i = 0$ to $N - 1$
    $P_i = P_{\text{start}} + (i/(N-1)) * (P_{\text{end}} - P_{\text{start}})$
    $L_i = ToScreen(P_i) - ToScreen(P_{i-1})$
  $t_{\min} = \left\lceil (\sum_{i=1}^{N-1} L_i)/R \right\rceil$
  $t_{\max} = \lceil (N * \max_i(L_i)/R) \rceil$
  if $t_{\max} - t_{\min} \geq SplitThreshold$
    return $NonUniform$
  else return $t_{\max}$
```

Figure 8: *Implementation of $\mathcal{T}$: $\mathcal{T}$ determines whether a parametric edge has significant screen-space variation and requires splitting, or should be tessellated using t segments of equal parametric length, with a goal of producing vertices spaced apart by R pixels. It makes this decision by sampling the surface N times and looking at the screen-space length of the $N - 1$ edges that are formed.*

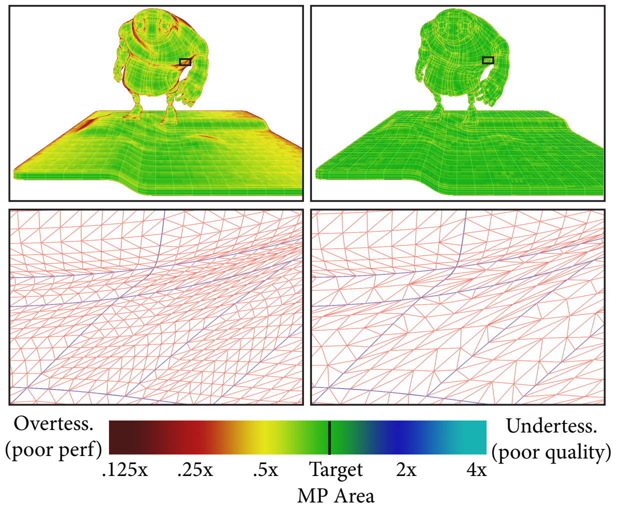


Figure 9: *Normalizing triangle areas using the interior tessellation scale parameter S. Left (2.1M triangles): Rendering* BIGGUY *using S=1 results in many triangles below the target area. Right (1.6M triangles): Adaptively modifying S results in a more uniform triangle area distribution. In both tessellations, blue lines denote subpatch boundaries.*

these edges are not necessarily Bezier curves. Analytic techniques can be applied to non-isoparametric edges by computing an isoparametric bounding box around an edge, then computing derivative bounds over this region. This gives an exact upper bound, albeit at higher computational cost compared to what is required for isoparametric edges.

Our implementation of $\mathcal{T}$ supports any surface type and does not rely on analytic techniques. Instead, we coarsely sample the screen space position of the surface at N uniformly spaced points along the edge. We treat the edge as a piecewise-linear curve consisting of $N-1$ segments connecting the points. The sum of all segment lengths constitutes a lower bound on the edge's actual length and is used to compute t_{min}, a lower bound for the edge tessellation factor. We also estimate a tessellation factor upper bound, t_{max}, based upon the longest of the $N-1$ segments. When the difference between t_{min} and t_{max} exceeds a specified threshold, the edge is determined to require adaptive tessellation. Otherwise, $\mathcal{T}$ returns t_{max} as the tessellation factor for the edge. Figure 8 summarizes this implementation of $\mathcal{T}$.

To ensure that the mesh is crack-free, $\mathcal{T}$ must always return the same result for both subpatches that share an edge. Like the D3D11 pipeline, we can achieve independent processing of subpatches by using only edge information that is available to both subpatches. In particular, $\mathcal{T}$ cannot use information such as numerical derivatives on the patch interior. Differences in numerical precision due to different subpatch orientations can also lead to inconsistent $\mathcal{T}$ evaluations. Ordering control points along an edge by their world space position can be used to ensure consistent numerical evaluation.

3.3 Subpatch Interior Scaling

$\mathcal{T}$ estimates the tessellation rate of an edge, using only edge information. No information about the interior of the patch is used. As a result, no guarantees are made about the quality of the tessellation inside the patch. A reasonable tessellation goal is to produce tessellations with a number of triangles that is proportional to the screen-space coverage of the subpatch. Bounding maximum edge length in the micropolygon mesh guarantees surface geometry and shading are sampled at least as frequently as this length, but enforcing this constraint causes oversampling of subpatches with poor aspect ratio (such as those along object silhouettes, see Figure 9-left). For this reason, we instead choose to prioritize generating triangles that closely match a desired target area.

To achieve a more uniform distribution of triangle areas, we adjust subpatch interior tessellations by modifying EDGEDICE's interior scale parameter $S \in [0,1]$. S isotropically scales the interior tessellation in each parametric direction. Varying S controls the number of triangles in the final tessellation and, correspondingly, the area of these triangles.

S is computed by estimating the subpatch's screen-space area. Prior to dicing, we evaluate the surface at a 3x3 uniform grid of points (these points define four quads) in the subpatch and conservatively estimate the subpatch's area, A_{patch}, as four times the area of the largest of these quads. To approximate the target micropolygon area of A_{tri} pixels, we set S so that the subpatch's diced mesh contains $A_{\text{patch}}/A_{\text{tri}}$ triangles.

Given edge tessellation factors a, b, c, d (let M_{u} and M_{v} be the maximum factors in the u and v parametric directions) the number of triangles contained in a quadrilateral patch is given by:

$$\begin{aligned} N_{\text{tris}} &= 2((SM_{\text{u}}-2)(SM_{\text{v}}-2)+ \\ &\quad (SM_{\text{u}}-2)+(SM_{\text{v}}-2))+a+b+c+d \end{aligned}$$

Given a desired triangle count, the interior tessellation scaling is determined by solving this equation for S.

Our method makes three assumptions: that a 3x3 uniform grid is a good estimate of the final subpatch area, that triangles in a diced subpatch all have the same area (splitting makes this a safe assumption), and that there is no integer rounding applied to the interior tessellation factor after scaling by S. Nevertheless, the technique behaves very well in practice. In Figure 9, observe that vertex positions on subpatch boundaries are not changed by area scaling. Even using area scaling, some triangles remain very small; this tends to occur at subpatches on object silhouettes, which have near-zero area, so any tessellation of the patch interior will generate triangles with area smaller than A_{tri} since we must produce at least enough triangles to triangulate the vertices along the patch edges.

4 Evaluation

Direct comparison of recent work in the field of parallel, adaptive tessellation is challenging because of wide variation in research goals. Some studies have focused primarily on efficient GPU implementations (with little regard to the quality of tessellation output), while others have focused on algorithms for adapting to surface detail. Further, some techniques intend to approximate surfaces with large polygons when possible while others target micropolygons. We choose to evaluate DIAGSPLIT by comparing its quality and performance against three schemes that, in our opinion, best characterize the space of alternative methods. For high quality, we wish to produce tessellations that represent surfaces accurately and are suitable for per-vertex shading calculations. To meet both requirements, we seek to produce triangle micropolygons that are approximately 0.5 pixels in area. For high performance, DIAGSPLIT must avoid overtessellation and the cost of splitting must be kept low.

Our implementation of DIAGSPLIT uses the splitting algorithm described in Section 3 for performing splits, EDGEDICE with interior area scaling for dicing, and an implementation of $\mathcal{T}$ that samples the (potentially displaced) surface at four points along subpatch edges (N=4). By construction, DIAGSPLIT permits a

DiagSplit Execution Summary

	Base Patches	MP Area (pixels) Avg.	MP Area (pixels) Max	Max Split Depth Avg.	Max Split Depth Max	Verts/Subpatch Avg.	Overhead Surface Evals (% of Total)
LONGPLANE	1	.48	.60	20.0	20	160.4	9
BIGGUY	2,570	.45	1.27	2.0	6	134.3	12
MONSTERFROG	1,292	.41	3.84	2.4	10	117.7	14
BUMPY	6	.43	2.85	13.0	13	145.1	11
ZONEPLATE	1	.25	14.24	79.0	79	52.2	32
COLUMNS	13,044	.40	1.38	1.7	10	89.4	20
ZINKIA	151,651	.45	1.23	0.1	13	66.4	25

Table 2: *Execution statistics from tessellating scenes in Figure 10 using* DIAGSPLIT. DIAGSPLIT *tessellations contain triangles that on average are within 20% of the target micropolygon area (0.5 pixels).*

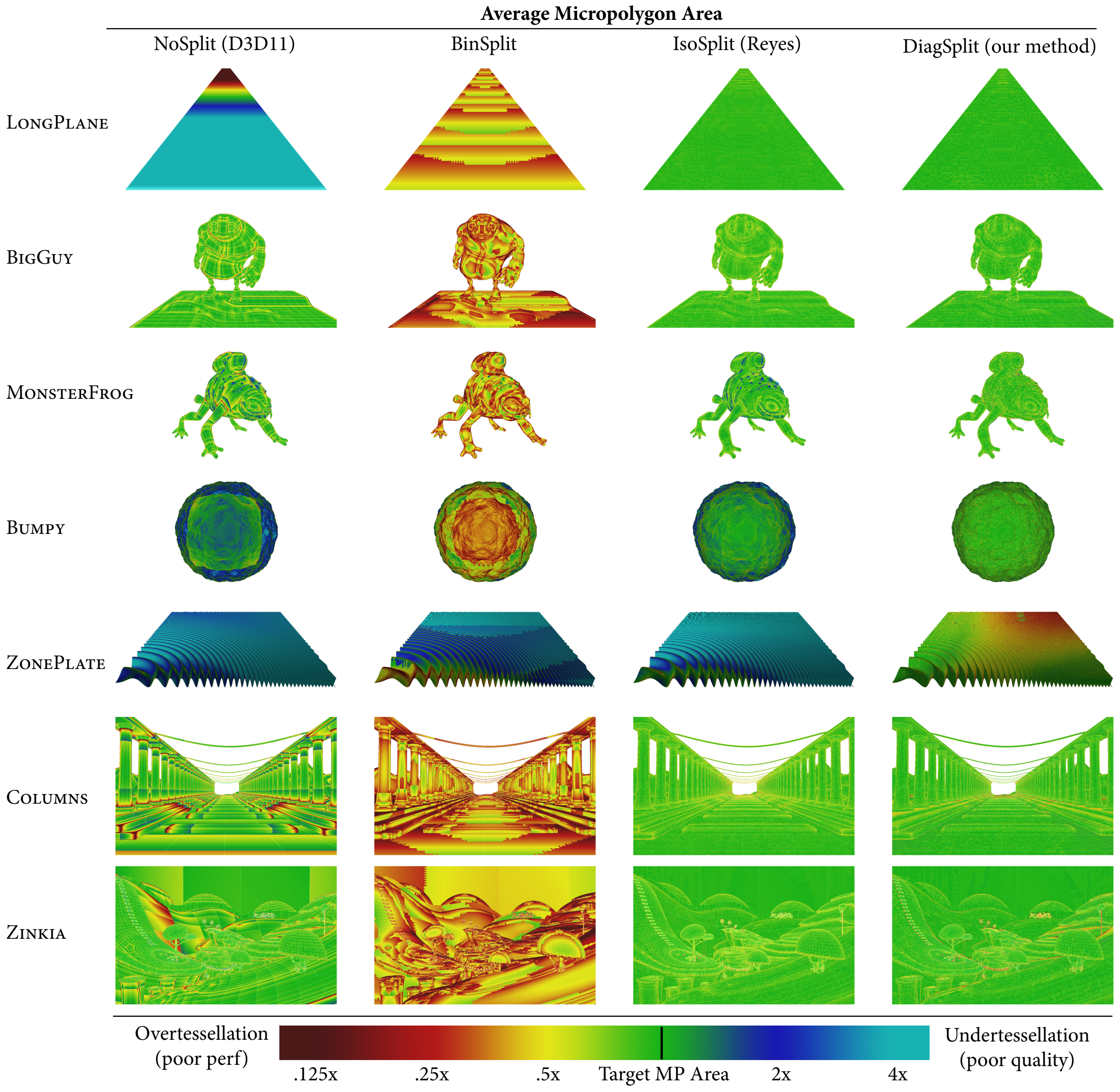


Figure 10: *We seek tessellations containing micropolygons of approximately 0.5 pixels in area. In the images above, green pixels are covered by micropolygons very close to the target size. Red and blue pixels indicate overtessellation and undertessellation respectively. For all scenes,* DIAGSPLIT *generates a tessellation that is as good as or better than alternative approaches (Zinkia scene © Zinkia Entertainment, S.A.).*

parallel implementation. We compare this configuration of DIAGSPLIT against the following alternatives.

NOSPLIT: This configuration is parallelizable and mimics the behavior of the D3D11 pipeline as well as the CudaTess system by [Schwarz and Stamminger 2009]. It does not perform splits, uses EDGEDICE dicing with area scaling, and uses the same edge-based $\mathcal{T}$ as DIAGSPLIT.

BINSPLIT: This configuration performs isoparametric splitting with binary UVDICE dicing. Dicing factors are computed using $\mathcal{T}$, but are rounded up to the nearest power of two. Binary dicing allows for a simple, crack-free, parallel implementation. The algorithms described in Eisenacher et al. [2009] and Patney et al. [2009], if configured to output micropolygons, produce tessellations similar to BINSPLIT. Patney and Owens [2008] also implement BINSPLIT but do not prevent cracks.

ISOSPLIT: This configuration mimics an advanced Reyes implementation and performs isoparametric splitting and UVDICE dicing. ISOSPLIT evaluates the surface at a 4x4 grid of points spanning the subpatch to make splitting decisions and determine dicing factors. Post-tessellation stitching removes cracks, but requires complex data structures that preclude efficient parallel implementation.

We evaluate the four algorithms using the seven examples shown in Figure 10 rendered at 1728×1080 resolution. Scene geometry is modeled using Catmull-Clark subdivision surfaces that are directly evaluated using the Loop-Schaefer approximation scheme [2008]. These scenes test different aspects of tessellation:

- LONGPLANE contains a single patch undergoing severe perspective foreshortening. Adaptive tessellation is required to avoid overtessellation.
- BIGGUY and MONSTERFROG are simple character tests (MONSTERFROG features textured displacement).
- BUMPY and ZONEPLATE use high-frequency procedural displacement to create surface detail (subdivision base cages are a cube and plane respectively).
- COLUMNS and ZINKIA are full scenes containing large variation in base-primitive size.

4.1 Algorithm Comparisons

Figure 10 visualizes the quality and performance of all algorithms by coloring images according to the average area of micropolygons overlapping each pixel. Properly tessellated areas containing 0.5 pixel area micropolygons are green, areas overtessellated by at least 4x are red (poor performance), and areas undertessellated by more than 4x are light blue (poor quality). Since previous algorithms do not consider displacement mapping during splitting, we first compare algorithm performance on non-displacement mapped scenes.

DIAGSPLIT, NOSPLIT, and BINSPLIT each provide parallelizable, crack-free tessellation solutions. Of these three algorithms, only DIAGSPLIT consistently produces good tessellations. While DIAGSPLIT meets the target area very well (most regions of the DIAGSPLIT images are green), NOSPLIT generates areas of overtessellation and undertessellation. The problem is acute in the pathological case of LONGPLANE, where a single patch undergoes significant perspective foreshortening. NOSPLIT's uniform tessellation of LONGPLANE is too coarse near the viewer and too fine at a distance. Removing undertessellation by tessellating conservatively (so that the near region of LONGPLANE becomes green) increases overtessellation afar and results in over 8.2x as many vertices as DIAGSPLIT. Conservative NOSPLIT tessellations of other scenes contain 1.4x (BIGGUY) to 2.2x (COLUMNS) more vertices. BINSPLIT is adaptive but rounds dicing factors to powers of two and lacks DIAGSPLIT's interior area scaling capabilities (unlike EDGEDICE, UVDICE cannot modify its interior tessellation independently from tessellation at subpatch edges). As a result, BINSPLIT overtessellates severely. We measure that BINSPLIT generates between 2.1x (ZINKIA) to 3.0x (COLUMNS) more vertices than DIAGSPLIT. When content is authored well for uniform tessellation (BIGGUY, MONSTERFROG, COLUMNS), BINSPLIT's tessellations contain more micropolygons than NOSPLIT's. In addition, because BINSPLIT-tessellation factors jump between powers of two, it produces tessellations that are less locally uniform than those of the other three algorithms. Modifying BINSPLIT to round edge factors to the nearest power of two (rather than rounding up) reduces overtessellation, but introduces large regions of undertessellation.

DIAGSPLIT is designed for subpatch-parallel execution while ISOSPLIT is not. However, because ISOSPLIT does not rely on preserving agreement along edges to prevent cracks, it is able to make splits and set tessellation rates using information from subpatch interiors as well as edges. Further, ISOSPLIT does not need to determine subpatch tessellation factors until a decision to dice is made (it requires no edge constraints). Despite retaining more flexibility, we find that in tests where displacement is not present, ISOSPLIT and DIAGSPLIT tessellations approximate the desired micropolygon area equally well.

When displacement is present, DIAGSPLIT produces better tessellations than ISOSPLIT because it accounts for the displaced position of the surface (not just the subdivision limit surface) in $\mathcal{T}$. The disparity in quality is notable when displacement is large. For example, ISOSPLIT (as well as NOSPLIT and BINSPLIT) consistently undertessellates ZONEPLATE because splitting decisions assume the surface is a flat plane. In offline rendering, resolving this problem requires manually increasing tessellation amounts for such objects. This is labor intensive and produces overtessellation in areas of the object that do not contain significant displacement. Accounting for displacement during tessellation requires no user input and yields a better approximation to the final surface. We have verified that modifying ISOSPLIT to account for displacement yields tessellations of approximately the same quality as DIAGSPLIT's. Accounting for displacement in NOSPLIT has little benefit because the scheme is incapable of adapting to the increased surface detail.

4.2 Edge Sampling

Our implementation of DIAGSPLIT uniformly samples surface position along an edge to compute tessellation factors. We empirically determined that four samples per edge (N=4) yields a good balance between cost to compute $\mathcal{T}$ and overall tessellation quality. Figure 11 visualizes tessellations that result from a range of edge sampling rates.

Even when surfaces contain no high frequency detail, approximating an edge as a line formed by its endpoints (N=2) results in poor quality at silhouettes. When an edge crosses over a silhouette, it is common for the projected position of its endpoints to fall nearly on top of each other, triggering undertessellation. This sampling error is troublesome because it negates a major advantage of the micropolygon representation: high quality silhouettes.

In practice we find that N=3 often suffices for smooth, undisplaced surfaces and that our choice of N=4 yields good results for most displacements. Of course, any uniform sampling of the edge is prone to aliasing, which is clearly present in the ZONEPLATE scene (surface detail becomes very high frequency in the top-right corner of the plane). However, with the exception of pathological cases such as ZONEPLATE, we observe little difference between values

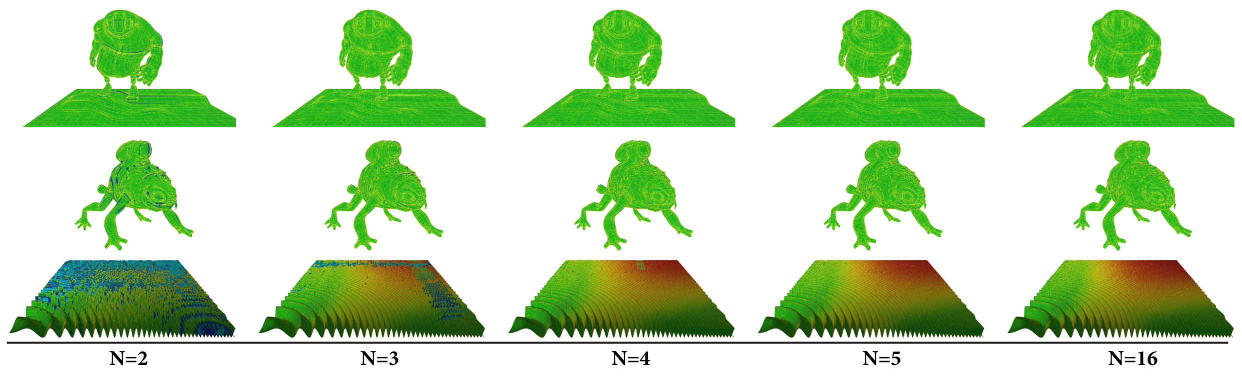


Figure 11: DIAGSPLIT *tessellation quality improves as the number of surface samples used by $\mathcal{T}$ increases. Errors due to sparse edge sampling are particularly apparent in* ZONEPLATE*, which contains a surface with high frequency detail. Our implementation of* DIAGSPLIT *uses four samples to determine the tessellation factor for an edge (N=4).*

of N greater than five. We have not studied the impact of employing irregular or adaptive sampling techniques in $\mathcal{T}$; the advantage of adaptivity is largely achieved through splitting and it is important that the cost of $\mathcal{T}$ remain as small as possible to keep the overhead of producing the tessellation low.

4.3 DiagSplit Characteristics

Table 2 quantifies DIAGSPLIT's behavior on our workloads. The table corroborates the result illustrated by Figure 10; on all scenes except ZONEPLATE, DIAGSPLIT generates triangles that are, on average, within 20% of the 0.5 pixel target area. Micropolygons of this size do not approximate the high frequency detail of ZONEPLATE well, so DIAGSPLIT tessellates more finely to yield a close approximation to the surface.

Table 2 also provides insight into the overhead of adaptivity. The cost of determining where DIAGSPLIT places tessellation vertices is dominated by point sampling surface position. These evaluations, which occur in $\mathcal{T}$ and to compute interior scale parameters prior to dicing, add to the fundamental cost of evaluating surface position and shading interpolants at final micropolygon vertex locations. When subpatches are reasonably large, less than 14% of all surface evaluations constitute overhead of computing the split-dice tessellation. This fraction grows to 20 and 25% in COLUMNS and ZINKIA, which contain many small base patches (these scenes have the smallest number of vertices per diced subpatch). However, small base patches are rarely split (see split depth in Table 2), so DIAGSPLIT incurs essentially the same "overhead" to compute edge factors as NOSPLIT. In practice (Section 4.4, Figure 12) we measure splitting overhead to be a larger fraction of total tessellation time than these counts indicate because our implementation more aggressively optimizes surface evaluation at final mesh vertices.

4.4 Performance Evaluation

We parallelized DIAGSPLIT on a multi-core CPU to better understand its performance characteristics. Our implementation leverages locality inherent in split-dice by processing split subpatches in depth-first order. We represent subpatches compactly as 52 byte records (four parametric coordinates, four tessellation factor constraints, and a pointer to the base patch) so supporting a 20 element stack requires less than 1 KB of storage (Table 2 shows this is more than sufficient for most scenes). Thus, our implementation runs almost entirely out of local data caches and does not exhibit the large memory footprint or high-bandwidth limitations of breath-first tessellation [Patney and Owens 2008; Eisenacher et al. 2009].

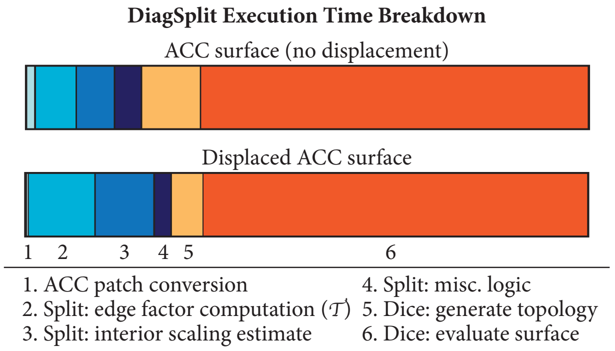


Figure 12: *Relative cost of major* DIAGSPLIT *operations when tessellating* MONSTERFROG *with and without displacement. Dicing dominates execution time of computing a* DIAGSPLIT *tessellation.*

We have vectorized key sections of the code (namely surface evaluation), resulting in a fast implementation. A *single core* of a 3 GHz Intel Core i7 processor tessellates Approximate Catmull-Clark (ACC) surfaces at 20.8M triangle micropolygons per second. This timing includes the cost of converting the base mesh to ACC patches, as well as evaluation of surface normals, tangents, and texture coordinates for shading. Performance increases to over 41.3M MPs/sec (per core) if only surface position is required (e.g., when generating shadow maps). When evaluating displaced surfaces our implementation sustains 10.9M MPs/sec (the cost of accessing texture data is expensive in our CPU implementation). Figure 12 displays the cost of major DIAGSPLIT operations when tessellating MONSTERFROG with and without displacement. Although dicing is more heavily optimized in our implementation, it still dominates execution time. On wide vector architectures the relative cost of split may increase as it is more challenging to vectorize $\mathcal{T}$ and area scaling estimates than tune surface evaluation at mesh vertices.

We parallelize DIAGSPLIT by having multiple cores cooperate to perform tessellation work using a shared work queue. The granularity with which work is dispatched to cores is a parameter of the system (multiple patches to multiple subpatches at a time). This simple implementation scales well out to eight processors (6.2x for non-displaced surfaces, 6.7x for displaced ones). On an eight core Intel Core i7 system, we tessellate ACC surfaces at a rate of 129.9M MPs/sec (73.1M when displaced). We stress that this performance is obtained while also producing a very high quality and low micropolygon count tessellation.

In addition to the implementation discussed above, we have integrated DiagSplit into a parallel, sort-first rasterization pipeline running on multi-core CPUs. Sort-first renderers scale by operating on screen tiles in parallel. Our renderer processes different subpatches of the same base patch simultaneously when the primitive covers multiple screen tiles. Preliminary end-to-end studies indicate that generating DiagSplit tessellations of displaced ACC surfaces constitutes only 30% of total rendering time when micropolygons are shaded using a simple Phong model (single texture lookup) and rasterized to a 16x multi-sampled framebuffer. We expect more realistic shading computations and sophisticated rasterization techniques (such as motion blur and defocus) to dwarf the cost of micropolygon generation in a real-time system. The overhead of adaptivity when computing a tessellation is more than justified because avoiding overtessellation decreases the amount of work performed by subsequent, more costly, rendering operations.

5 Discussion

DiagSplit adapts the Split-Dice algorithm to perform non-isoparametric cuts and use a dicing scheme that supports different tessellation factors for each subpatch edge. These modifications allow DiagSplit to retain the adaptivity and crack-free quality of advanced Reyes implementations while processing subpatches in parallel. DiagSplit generates tessellations containing fewer and more uniform micropolygons than existing parallel methods.

DiagSplit's advantages significantly outweigh the increase in cost of computing subpatch bounding boxes, edge tessellation factors, and numerical derivatives of vertex shading quantities that results from non-isoparametric splits. Also, given new methods for representing subdivision surfaces using parametric approximations, we do not view DiagSplit's limitation to parametrically-evaluable surfaces as severe for real-time rendering.

We believe DiagSplit can be integrated tightly into future real-time rendering pipelines and that it will be possible to generate micropolygon representations of complex scenes in real-time in the near future. Still, many additional aspects of graphics pipeline implementations must be re-tuned to efficiently render micropolygon workloads. In conjunction with DiagSplit, we have conducted a detailed evaluation of micropolygon rasterization [Fatahalian et al. 2009], but the challenges of culling and shading micropolygons and efficiently parallelizing an end-to-end micropolygon rendering pipeline still require further study.

Acknowledgments

Support for this research was provided by the Intel Foundation Ph.D. Fellowship Program, an Intel Larrabee Research Grant, the National Science Foundation Graduate Research Fellowship Program, and the Fannie and John Hertz Foundation. Zinkia scene © Zinkia Entertainment, S.A.

References

Apodaca, A. A., and Gritz, L. 2000. *Advanced RenderMan: Creating CGI for Motion Pictures*. Morgan Kaufmann.

Blinn, J. F. 1978. *Computer display of curved surfaces.* PhD thesis, The University of Utah.

Burley, B., and Lacewell, D. 2008. Ptex: Per-face texture mapping for production rendering. In *Computer Graphics Forum*, vol. 27, Blackwell Publishing Ltd, 1155–1164.

Catmull, E. E. 1974. *A subdivision algorithm for computer display of curved surfaces.* PhD thesis, The University of Utah.

Clark, J. H. 1979. A fast scan-line algorithm for rendering parametric surfaces. In *Computer Graphics (Proceedings of ACM SIGGRAPH '79)*, ACM, 174.

Cook, R., Carpenter, L., and Catmull, E. 2008. The Reyes image rendering architecture. In *Computer Graphics (Proceedings of ACM SIGGRAPH '87)*, vol. 27, 1–11.

Eisenacher, C., Meyer, Q., and Loop, C. 2009. Real-time view-dependent rendering of parametric surfaces. In *I3D '09: Proceedings of the 2009 Symposium on Interactive 3D Graphics and Games*, ACM, 137–143.

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: Proceedings of the Conference on High Performance Graphics 2009*, ACM, 59–68.

Foster, C., 2009. Aqsis renderer. `http://aqsis.org/`.

Kovacs, D., Mitchell, J., Drone, S., and Zorin, D. 2009. Real-time creased approximate subdivision surfaces. In *I3D '09: Proceedings of the 2009 symposium on Interactive 3D graphics and games*, ACM, 155–160.

Lane, J. M., Carpenter, L. C., Whitted, T., and Blinn, J. F. 1980. Scan line methods for displaying parametrically defined surfaces. *Communications of the ACM 23*, 1, 23–34.

Lien, S., Shantz, M., and Pratt, V. 1987. Adaptive forward differencing for rendering curves and surfaces. *Computer Graphics (Proceedings of ACM SIGGRAPH '87) 21*, 4, 111–118.

Loop, C., and Schaefer, S. 2008. Approximating Catmull-Clark subdivision surfaces with bicubic patches. In *ACM Transactions on Graphics*, vol. 27, 1–11.

Microsoft, 2009. DirectX 11 SDK: August 2009. `msdn.microsoft.com/en-us/directx/`.

Moreton, H. 2001. Watertight tessellation using forward differencing. In *Proceedings of the Eurographics Workshop on Graphics Hardware*, ACM, 25–32.

Moule, K., and McCool, M. 2002. Efficient bounded adaptive tessellation of displacement maps. In *Graphics Interface*, 171–180.

Patney, A., and Owens, J. D. 2008. Real-time Reyes-style adaptive surface subdivision. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH Asia) 27*, 5.

Patney, A., Ebeida, M. S., and Owens, J. D. 2009. Parallel view-dependent tessellation of Catmull-Clark subdivision surfaces. In *Proceedings of High Performance Graphics 2009*, 99–108.

Purnomo, B., Cohen, J. D., and Kumar, S. 2004. Seamless texture atlases. In *SGP '04: Proceedings of the 2004 Eurographics Symposium on Geometry Processing*, ACM, 65–74.

Rockwood, A. P., Heaton, K., and Davis, T. 1989. Real-time rendering of trimmed surfaces. In *Computer Graphics (Proceedings of SIGGRAPH '89)*, 107–116.

Schwarz, M., and Stamminger, M. 2009. Fast GPU-based adaptive tessellation with CUDA. In *Computer Graphics Forum*, vol. 28, Blackwell Publishing Ltd, 365–374.

Stam, J. 1998. Exact evaluation of Catmull-Clark subdivision surfaces at arbitrary parameter values. In *Proceedings of ACM SIGGRAPH '98*, ACM, 395–404.