Pacific Graphics 2014
J. Keyser, Y. J. Kim, and P. Wonka
(Guest Editors)

Volume 33 (2014), Number 7

Fractional Reyes-Style Adaptive Tessellation for Continuous Level of Detail

G. Liktor[1] and M. Pan[2] and C. Dachsbacher[1]

[1]Karlsruhe Institute of Technology
[2]Crytek GmbH

Abstract

In this paper we present a fractional parametric splitting scheme for Reyes-style adaptive tessellation. Our parallel algorithm generates crack-free tessellation from a parametric surface, which is also free of sudden temporal changes under animation. Continuous level of detail is not addressed by existing Reyes-style methods, since these aim to produce subpixel-sized micropolygons, where topology changes are no longer noticeable. Using our method, rendering pipelines that use larger triangles, thus sensitive to geometric popping, may also benefit from the quality of the split-dice tessellation stages of Reyes. We demonstrate results on a real-time GPU implementation, going beyond the limited quality and resolution of the hardware tessellation unit. In contrast to previous split-dice methods, our split stage is compatible with the fractional hardware tessellation scheme that has been designed for continuous level of detail.

Categories and Subject Descriptors (according to ACM CCS): I.3.1 [Hardware Architecture]: Parallel processing— I.3.5 [Computational Geometry and Object Modeling]: Curve, surface, solid and object representations—

1. Introduction

Complex geometry is often represented in compact analytic or procedural forms, offering several benefits compared to polygonal models during rendering. Most importantly, such surfaces can be tessellated dynamically to provide the necessary level of detail (LoD) for a given view, optimizing the input of visibility computation and shading. Surfaces represented this way typically also need less storage as the triangulation happens on-the-fly, often after applying animation on the coarse level. Under animation, dynamic tessellation can cause sudden topology changes and may lead to a noticeable "popping" of the geometry.

Modern GPUs feature a dedicated tessellation unit for the evaluation of parametric patches and curves. By using fractional tessellation [Mor01], which smoothly interpolates between different tessellation resolutions, it can avoid popping artifacts. However, the uniform nature of the algorithm often leads to over- or undertessellation in screen space, and the hardware implementation has a limited resolution.

To improve existing real-time tessellation methods, we were inspired by the off-line Reyes pipeline which tessellates surfaces into subpixel-sized *micropolygons* [CCC87]. Reyes achieves its higher quality by first recursively subdividing patches (*split*) into subpatches that can be safely tessellated uniformly (*dice*). It is, however, not trivial to adopt *split-dice* in real-time applications: existing methods do not guarantee any temporal tessellation coherence as micropolygons hide topology changes due to their scale.

In this paper we modify the *split* stage so that it retains continuous LoD in a Reyes-style tessellation framework, and works as a seamless extension of the fractional tessellation implemented in modern GPUs. Although our method does not have any resolution limitations, we do not address micropolygon rendering (where fractional tessellation is by definition suboptimal). Instead, the primary goal of our work is to improve the visual quality of a real-time, rasterization-based pipeline, for cases where the resolution and quality of the fixed function tessellator is not sufficient. We make the following contributions:

- **A parallel, crack-free split stage** for quadrilateral and triangular parametric patches.
- **Fractional splitting**, a method that smoothly introduces lower level subpatches. To our knowledge, this is the first formulation of *split-dice* that provides continuous LoD and works with fractional hardware tessellation.
- **Iterative refinement for subpatch edge factor computation** which also ensures a gradual adjustment of tessellation quality without temporal artifacts.

2. Background and Related Work

We first provide an overview of view-dependent tessellation of parametric surfaces, focusing on algorithms that were designed for massively parallel architectures. A primary challenge of such methods is how to preserve the water-tightness of a surface, as it is inefficient to maintain global topology information during processing.

2.1. Reyes

One of the first adaptive tessellation methods is the Lane-Carpenter algorithm [LCWB80], which recursively subdivides patches until all edges satisfy a view-dependent flatness criteria. Pixar's Reyes architecture [CCC87] refined this approach to a two-level adaptive tessellation framework, referred to as *split-dice*: the *split* phase subdivides patches until they can be uniformly tessellated. The resulting subpatches are *diced* into regular micropolygon grids. The combination of adaptive subdivision on a subpatch granularity and the performance of uniform tessellation is one major benefit of Reyes. The first parallel GPU implementation of *split-dice* was presented by Patney and Owens [PO08]. They solved the problem of unbounded geometry amplification by replacing the recursive split stage with an iterative breadth-first processing. As a limitation, they used a constant dicing resolution and did not address surface cracks. We discuss crack-free Reyes-style methods in Sec. 2.3.

2.2. Hardware Tessellation

Moreton developed a hardware-friendly evaluation of parametric patches over quadratic and triangular domains [Mor01]. Fig. 1 provides an overview of the tessellation patterns generated by this method. While patches are tessellated uniformly in their interior, which allows a fast hardware implementation, independent *edge factors* can be chosen for each side, therefore creating watertight connections between patches of different resolutions. Moreton also defined a variant of the method that uses fractional tessellation factors to seamlessly morph between subsequent integer resolutions. New vertices are first generated at the location of integer vertices, then slide out as the fractional component of the tessellation factor increases.

Since all patch edges can be evaluated in both parametric directions, the fractional pattern needs to be symmetric, creating new vertices in pairs. Due to this symmetry a fractional edge might have more segments than an integer edge with the same tessellation factor. The hardware implements two kinds of schemes: *fractional odd*, and *fractional even*. This means that the tessellation matches the integer factors at odd or even numbers, respectively. For example, in Fig. 1 an edge of factor 4.0 has 5 segments. The overhead of this suboptimal behavior amortizes as the tessellation factor increases. Furthermore, the moving vertices might lead to *swimming artifacts* by sampling different parametric locations over time. This problem is particularly important in

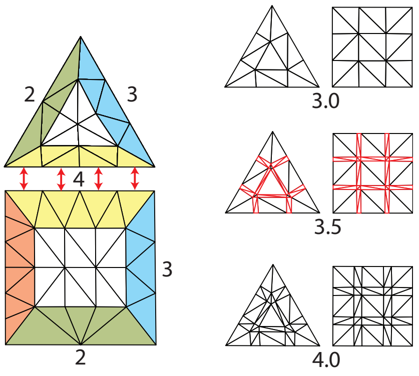


Figure 1: *Hardware tessellation for quad and triangular domains [Mor01].* ***Left:*** *patches are tessellated uniformly in their interior (white), but arbitrary factors can be assigned to edges, creating a transitional triangulation (colored) and allows a watertight match between patches of different interior factors.* ***Right:*** *fractional tessellation allows smooth transitioning between integer levels; fractional triangles are highlighted with red (fractional odd pattern).*

the presence of displacement mapping, but it can be alleviated by selecting proper displacement LoD based on the tessellation density [NL13].

The main limitation of this technique is that vertices are evaluated regularly inside each patch, yielding suboptimal quality if a patch projects non-uniformly into screen space. Munkberg et al. [MHAM08] proposed a non-uniform warping of the parametric domain. This adapts the tessellation pattern to perspective foreshortening, but still lacks the refinement level of split-dice.

2.3. Addressing Surface Cracks

As parametric patches are tessellated independently, they need special care to preserve continuity across borders. Avoiding cracks becomes intricate with *split-dice*: there will be transitions between different tessellation levels where only one of the adjacent patches are split on the same edge. The tessellation of the lower-level subpatches might not be consistent with the original edge (Fig. 2).

Stitching Pixar's Photorealistic Renderman connects subpatches of different split levels with a strip of additional micropolygons [AG99]. This algorithm requires adjacency information of the generated subpatches, which is normally derived from a patch dependency tree. Unfortunately, this is incompatible with a massively parallel pipeline.

Binary dicing Another way to solve the problem is avoiding T-junctions. It is possible to limit the tessellation of edges to power-of-two rates, which ensures that there is always a vertex where an adjacent split potentially occurs. This simple strategy suffers from poor tessellation quality: the nearest power-of-two number might greatly differ from the optimal value.

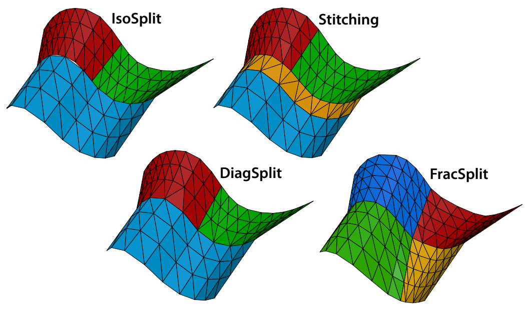


Figure 2: *Cracks may occur when adjacent patches make different splitting decisions.* ***IsoSplit*** *splits at the parametric edge center along an isocurve. Renderman uses* ***Stitching*** *to connect patches of different subdivision levels.* ***DiagSplit*** *matches split locations to tessellation vertices.* ***FracSplit*** *splits edges on both sides, but one patch eventually becomes triangular as the split factor reaches zero.*

DiagSplit Fisher et al. [FFB*09] combine split-dice with the edge-based integer tessellation of Moreton [Mor01] to prevent cracks instead of eliminating them. Their *DiagSplit* algorithm evaluates a τ tessellation metric over each edge, depending exclusively on the edge properties and thus ensuring consistency across adjacent patches. τ can also return a value indicating that the edge is non-uniform, and needs to be split. They avoid cracks by connecting the non-uniform edge midpoint to a vertex on the uniformly diced edge. This may lead to non-isoparametric (diagonal) splits, hence the name of the method.

None of the above methods addresses popping artifacts, which limits their application to micropolygon rendering. Furthermore, *DiagSplit* is not compatible with Moreton's fractional tessellation method (see Fig. 3).

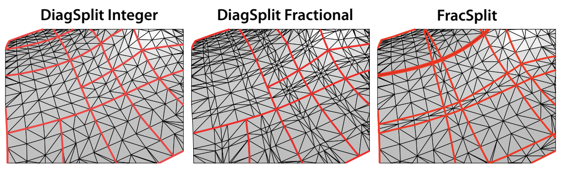


Figure 3: *DiagSplit only works with integer tessellation (**left**). The symmetric nature of fractional tessellation prevents the matching of vertices on the original edge (**middle**). Our method splits edges on both patches, ensuring that tessellation always matches (**right**).*

3. The Fractional Split-Dice Algorithm

In this section we describe our modified Reyes-style tessellation pipeline, which provides better adaptation than uniform tessellation, but does not necessarily refine the surface to the sub-pixel level (see Table 1 for a comparison against previous work). Our algorithm, which we refer to as *FracSplit*, is built around the following principles:

	Tess. Quality	Continuous LoD	Crack-free
HW. Tessellation [Mor01]	●○○○	✓	✓
DiagSplit [FFB*09]	●●●●	🚫	✓
FracSplit (this paper)	●●●○	✓	✓

Table 1: *Our method seeks to create a bridge between high-quality micropolygon tessellation, which is not popping-free, and fractional hardware tessellation.*

1. **Purely edge-based splitting.** Decisions about splitting are based on the edge properties only. If an edge is ever split, then it will happen on both adjacent patches.
2. **Fractional split factors.** We make "soft decisions" about edge splitting, by computing a per-edge split factor. This evaluates to 0 if the edge can be diced or to a fractional value if the edge needs to be split. By reaching the value 1.0, we get an equivalent result to median split.
3. **Fractional tessellation.** *FracSplit* is compatible with fractional tessellation as implemented on current GPUs, which we use for the dicing stage.

Fractional splitting is the key component of continuous LoD: we never split a parametric patch instantaneously in the middle, which may lead to visual popping. Instead, we use a fractional value to express the "split-affinity" of an edge, and interpolate the parametric split between one corner of the patch and the edge midpoint.

Let us briefly illustrate our concept on a bicubic Bézier patch. We can approximate an edge with its 4 control points (Fig. 4, left). The split stage may subdivide the control

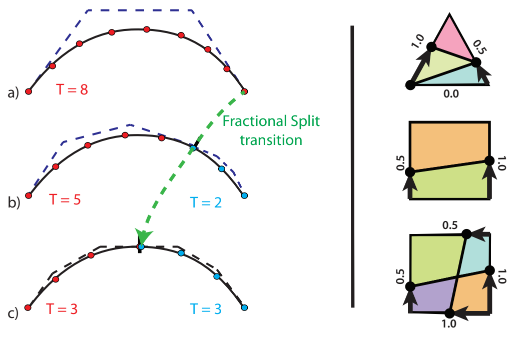


Figure 4: ***Left:*** *fractional splitting on a cubic Bézier curve, 4 control points* ***(a)****. Median split creates popping by the change of topology and reevaluation of edge factors* ***(c)****. Our method gradually introduces a new subdomain from an existing corner, so the refinement does not result in popping* ***(b)****.* ***Right:*** *the extension of this concept to the 2D domain gives us a method to generate fractional subpatches.*

cage (using the de Casteljau algorithm), and get a more accurate bound. Existing *split-dice* algorithms would perform the subdivision in the middle, which can lead to popping: besides the split location, the edge factors of the new subpatches might differ. *FracSplit* gradually introduces the splitting location from one corner of an edge thus allowing the reevaluation of the tessellation. We can now extend this concept to planar coordinates and get the complete splitting algorithm. The fractional split factors define sliding split locations on each edge. Fig. 4 shows possible subpatches that the algorithm may generate.

3.1. Parallel Splitting

Algorithm 1 provides a parallel breadth-first implementation of our fractional split stage (shown for quad-patches). The method `FracSplit` is called during every iteration in parallel, and processes one subpatch from a working array, defined by its four corners in the parametric domain of the base patch (`Subdomain`). `EdgeCode` is a bit-mask encoding the edge directions relative to the patch winding order (e.g. edges in clockwise direction are marked with 1). If the patch is visible (`BoundNCull`), we compute fractional split and tessellation factors for every edge. To simplify the code, we only show splitting in the u parametric dimension. Completed patches are added to the `DiceQueue` for fractional tessellation, while split patches are processed further in the next iteration from the `WorkQueue`.

Algorithm 1 Pseudocode of the fractional split stage

```
FracSplit(Subdomain{$uv_{00}, uv_{01}, uv_{10}, uv_{11}$}, EdgeCode){

    if( BoundNCull( Subdomain ) ) return;

    $\{s,t\} \leftarrow$ GetSplitAndTessFactors( Subdomain );
    needUSplit = $\max(s_{v=0}, s_{v=1}) > 0$;
    if( needUSplit ) {
        $e_{v=0}$ = SplitEdge($uv_{00}, uv_{10}, s$, EdgeCode$_{v=0}$);
        $e_{v=1}$ = SplitEdge($uv_{01}, uv_{11}, s$, EdgeCode$_{v=1}$);

        WorkQueue $\leftarrow$ {{$uv_{00}, e_{v=0}, e_{v=1}, uv_{01}$}, EdgeCode};
        WorkQueue $\leftarrow$ {{$e_{v=0}, uv_{10}, uv_{11}, e_{v=1}$}, EdgeCode};
    } else {
        DiceQueue $\leftarrow$ {Subdomain, t};
    }
}

SplitEdge($uv_0, uv_1, splitFactor, EdgeDir$){
    $uv_{root}$ = lerp($uv_0, uv_1, EdgeDir$);
    $uv_{mid} = 0.5(uv_0 + uv_1)$;
    return lerp($uv_{root}, uv_{mid}, splitFactor$);
}
```

3.2. Avoiding Cracks

We ensure that our method avoids cracks on two levels: in contrast to previous split-dice methods, *FracSplit* avoids T-junctions among the subpatch edges after splitting (see principle 1 of Sec. 3). Shared edges are then diced consistently using the fractional tessellation of Moreton [Mor01].

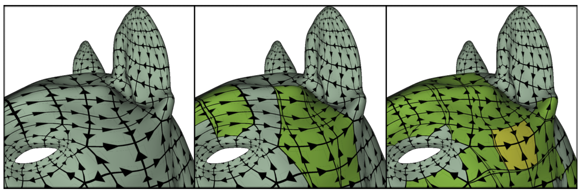


Figure 5: *In a preprocessing step we assign consistent directions to patch edges. During rendering we propagate these to subpatch levels. Fractional splits always occur at the edge roots first, ensuring temporal coherence.*

Consistent edge directions As already introduced in Fig. 4, we rely on directed edges to interpolate split locations between corners and edge midpoints. While we could have opted for a symmetric splitting pattern, the advantage of our decision is that it keeps the number of generated subpatches identical to median split (e.g. a symmetric pattern might generate 9 subpatches from a quad patch if all edges have a fractional split factor). The challenging part of our method is ensuring that edge directions match across adjacent patches to avoid T-junctions. Furthermore, opposite edges of a quad patch need to share their directions, otherwise splitting would cause popping on the diagonal of the patch (and poor tessellation quality).

We have addressed the problem of directing edges with a simple algorithm: we traverse all quad-strips of the surface and assign the same direction to all edges which are shared by two quads in the strip. A strip traversal terminates if a previously visited edge, a border, or a triangle face is reached. Fig. 5 visualizes edge directions that were generated this way. Edges of triangle patches can have an arbitrary orientation (Fig. 6).

The edge traversal step and the additional storage needed for the direction data is a disadvantage of this approach, however, the former only needs to be executed once (e.g. on authoring from a DCC tool), and the memory equirement is essentially one bit per edge. The edge directions of subpatches can be easily determined in parallel during the splitting step, thus require no global information.

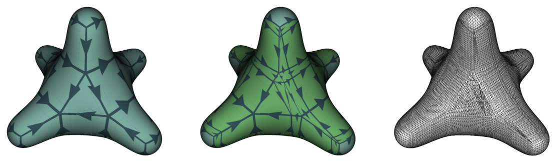


Figure 6: *Edge directions and fractional splitting for a simple combination of triangular and quad patches, with the corresponding tessellation pattern.*

4. Implementation

We have implemented our algorithm in the Direct3D 11 pipeline, and also evaluated it inside a commercial game engine. We augmented the hardware tessellation unit with a computational *FracSplit* stage prior to the rasterization. Our algorithm works with any kind of parametrization over quad or triangle domains, but in this paper we focus on the rendering of Catmull-Clark subdivision surfaces [CC78]. To convert these to parametric patches we use the Gregory patch approximation method of Loop et al. [LSNC09].

4.1. Rendering Pipeline with Fractional Splitting

During a *FracSplit* iteration, we use 16 threads to collaborate on the processing of a subpatch. This matches the number of control points of a bicubic Bézier patch, which we also use to approximate Gregory patches (this is precise enough for a split decision). Triangular patches are approximated with cubic Bézier triangles, requiring only 10 active threads. This design allows us to efficiently parallelize the processing of subpatches without synchronization, as 16 (index-aligned) threads always operate in lockstep on current GPUs.

Culling We have implemented back-patch and frustum culling for Bézier patches (`BoundNCull` in Algorithm 1). The base patches are first trimmed to the parametric bounding box of the subpatch. The frustum culling can be then evaluated using a fast parallel reduction. For back-patch culling we use the approximate cone-of-normals method from Sederberg et al. [SM88], which is able to analytically bound normals of Bézier patches. When displacement is present, there is no known method which would efficiently bound patch normals.

Data queues Once a subpatch passed the culling test, we compute tessellation and split factors for each edge (Section 4.2). If a split occurs, we add its children to a work queue for further processing. Patney and Owens [PO08] provided a lockless algorithm for breadth-first subdivision using parallel stream compaction. However, their technique requires redundant memory allocation for the sparse output buffer and the bandwidth-overhead of compaction is significant. Instead we use *Append buffers* (provided by Direct3D 11) to generate a compact queue directly with atomics.

Asynchronous processing The dynamically changing number of subpatches poses another important implementation challenge. We need to determine the number of *FracSplit* threads and the draw call parameters for the rasterization step, but it is essential to avoid CPU-GPU synchronization between the iterations. For rasterization, Direct3D 11 provides indirect draw calls, where attributes can be loaded from a GPU buffer. For the computational split stage, however, we have found that the *persistent thread* model worked the best, where a constant number of threads consumes elements from a shared input queue. The CPU controls only the number of iterations (by dispatching kernels), which is currently a user-defined value.

4.2. Tessellation and Split Factors

The tessellation metric τ (`GetSplitAndTessFactors` in Algorithm 1) in our implementation is based on approximating the length of cubic Bézier curves using their control cage (see Fig. 4). If significant displacement is present, we switch to the sparse parametric sampling of the final surface at 4 locations per edge. In both cases we approximate the final result by the sum of tessellation factors of three linear segments. For each segment e_i we compute the tessellation factors using the following formula:

$$\tau_i = \frac{\|\mathbf{a}-\mathbf{b}\|}{0.5(\mathbf{a}.z+\mathbf{b}.z)}\text{c\_TessFactor},$$

where τ_i essentially approximates the projected diameter of a sphere drawn around the segment. c_TessFactor is a global constant computed from the desired triangle size, screen resolution and camera field of view. This concept, illustrated in Fig. 7 is well-accepted in real-time rendering, and slightly overtessellates silhouette edges for better displacement mapping [Can11].

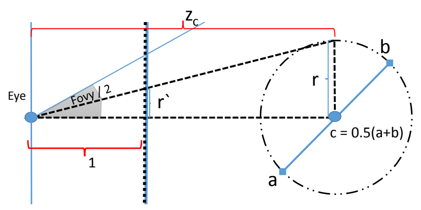


Figure 7: *We compute linear edge factors by estimating the projected diameter of the edge circumsphere.*

For the split factor s we use a very simple threshold formula: $s = \text{clamp}(\frac{\tau - \text{c\_SplitLimit}}{\text{c\_SplitLimit}}, 0, 1)$. If s exceeds the threshold, we split the edge. In our results, we set this value to 16.

5. Results and Discussion

In this section we evaluate the tessellation generated by our algorithm and compare it to existing methods: hardware tessellation (*HWTess*) and our implementation *DiagSplit* using the same hardware-based parallel framework we introduced in Section 4. In summary, we show that we are able to tessellate with no practical resolution limits and in most cases with better quality than hardware tessellation (Sec. 5.1). We also analyze the performance aspects of our method to understand its benefits and limitations (Sec. 5.2).

5.1. Quality

We seek to improve the quality of *HWTess* by addressing two main factors. First, we demonstrate that *FracSplit*, like other split-dice methods, does not have resolution limits. Second, *HWTess* tessellates large patches uniformly in object space, which often yields highly irregular triangle sizes in screen

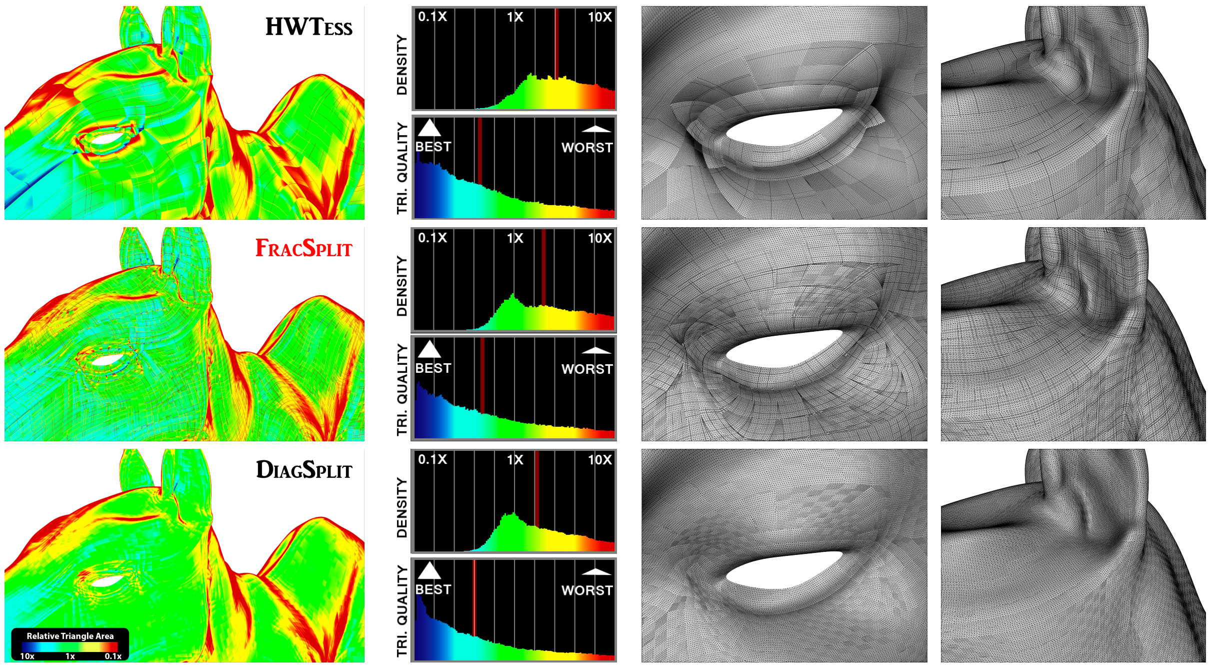


Figure 8: *We compare the tessellation accuracy of our method to* ***HWTess*** *and* ***DiagSplit****.* ***Left column:*** *screen space triangle area relative to the desired rate (red:* 10*x overtessellation, blue:* 10*x undertessellation).* ***Middle left:*** *Histograms of triangle area and quality distributions. The quality is measured as the ratio of incircle and circumcircle radius, relative to an equilateral triangle.* ***Right:*** *in these closeup wireframe renderings we can see that both Reyes-style methods rapidly split distorted patches. To see the details, please zoom in using the electronic version.*

space. By splitting such patches, we expect our method to better adapt to such irregular regions. We regard *DiagSplit* as a ground truth tessellator.

Fig. 8 shows our quality analysis on the *Killeroo* model. The left column visualizes the tessellation density relative to the desired rate (by measuring the screen space triangle size). Note that due to our metric described in Sec. 4.2 we purposely have a higher density on silhouettes. *FracSplit* successfully removes most over- and undertessellation issues from the surface of the mesh, but it does not reach the quality of *DiagSplit*. The two rightmost columns show wireframe renderings in more detail for two areas of the surface. The eye region contains some highly distorted quad patches, which cannot be tessellated uniformly. On the other hand, the topology around the ear is more regular. For the latter case, *FracSplit* produces a tessellation that closely matches the appearance of *DiagSplit*, but near the eyes it leaves some fractional subpatches with suboptimal tessellation.

To provide a better quantitative measurement of the quality, we also plot two histograms for each algorithm. The first one is simply a distribution of projected triangle areas, thus a different representation of the heatmap. We can see that the average density (marked by a red line) is approximately halfway between *HWTess* and *DiagSplit*. The second histogram is a quality metric of the triangles often used in Delaunay triangulation: the greater the ratio between the radius of the incircle and circumcircle, the closer it is to an equilateral triangle. This metric is interesting as thin triangles typically generate more overhead during rendering. Our method does not improve the surface in this aspect.

Limitations The main quality limitations of our method are direct consequences of the choice of our continuous LoD scheme. The first one is directly inherited from fractional tessellation, which typically generates more triangles than ideally required (refer to Fig. 1). Unfortunately the number of fractional edges is likely to increase exponentially after each *split* iteration. The second problem is that at transitional regions (where our split factors become fractional), some subpatches become distorted. In the heatmap of Fig. 8 these subpatches are clearly visible in red, but we can also see that their number is relatively small. Furthermore, these factors together often produce very small and thin triangles, that might potentially lead to numerical issues during sampling and shading. Sampling is not an issue in our current application, since it is successfully addressed by the fixed-function arithmetics used by the rasterizer. However, it might result in artifacts in a ray tracer, the evaluation of which is left for future work. We also did not experience shading artifacts, even when the tessellation density was adjusted to the micropolygon range (where our method is no longer optimal). There the main cause of artifacts are shader derivatives: we have visualized parametric derivatives in the pixel shader, but did not experience any outliers.

5.2. Performance in a Rasterization Pipeline

Fig. 9 shows performance statistics using four subdivision surfaces from two different perspectives each, using a distant and a close-up view. All surfaces were tessellated to approximately 3-pixel-sized triangles, and we used an edge split threshold of 16. The *Lizard* model has been rendered using our method inside a commercial game engine, but using a different configuration (NVIDIA GTX660Ti). Therefore the timings are not directly comparable. We performed all other tests on an NVIDIA GFX Titan Black GPU.

We have also conducted a more detailed breakdown of all three algorithms inside the rasterization pipeline. Their efficiency depends on the interplay of multiple factors, such as the cost of evaluating a patch (domain shading), visibility sampling and shading, and in case of split-dice, the per-patch processing. To measure these, we prepared a simple experiment which gradually increases the tessellation density from a static viewpoint over time. Fig. 10 shows these measurements using the *Killeroo* asset.

In the results the overall rendering time did not improve significantly over *HWTess* when using split-dice. In fact, for the *Killeroo* the rasterization itself gets up to 10% faster below a certain triangle size (about 4 pixels), but the overhead of the computational stage seems to cancel it out. However, our primary goal is not to outperform the hardware in its standard domain, but rather to extend its capabilities into a more flexible adaptive scheme. Also, the trends indicate that going lower with the tessellation, split-dice methods would have a consistently better performance, but that range is no longer covered by the resolution of the hardware.

Looking at the other plots in Fig. 10, we can clearly see that all split-dice methods use fewer triangles and domain shader invocations than *HWTess*. The latter is not trivial, since the number of subpatches increases exponentially on each split level, and shared domain vertices are shaded redundantly. This means that the improvement in tessellation can compensate against this effect. Another major overhead of *FracSplit* and *DiagSplit* comes from the extra bandwidth used during computational stage and the hull shader invocations, which are per-patch costs.

DiagSplit consistently outperforms our method: due to integer tessellation and the further limitations outlined in Section 5.1, it should be preferred for micropolygon rendering. However, popping is usually an important issue with larger polygon sizes.

6. Conclusion and Future Work

We have presented a method which bridges the gap between the performance and quality of hardware tessellation and micropolygon renderers. To our knowledge this is the first real-time implementation of split-dice, which is *both* crack-free and temporally coherent. The requirement of continuous LoD also meant a compromise in terms of quality and speed when compared to integer-based micropolygon tessellators. While we believe that fractional splitting opens up interesting possibilities for hardware tessellation, in the future it would be useful to automatically transition towards integer tessellation beyond a certain triangle size.

In the future we would like to combine our method with other surface subdivision techniques. The feature-adaptive method of Niessner et al. [NLMD12] is particularly promising, as it can represent finer edge or vertex-based details of subdivision surfaces (hierarchical edits and creases [DKT98]). It would be also interesting to investigate asymmetric fractional tessellation techniques, given we have already used directed edges, as well as methods that better adapt to out transitional patches.

Acknowledgements

We would like to thank Vaclav Kyba and Xiaomao Wu for their helpful ideas and support, and the rest of Crytek's Cinebox team. We are very grateful for the assets used throughout this paper. The *Lizard* asset was modeled by Hanno Hagedorn. The *DragonBaby* is from the Sintel movie by the Blender Foundation (www.sintel.org) and the *Killeroo* is the courtesy of Headus Inc. (www.headus.com.au). The first author of this paper was funded by Crytek GmbH.

References

[AG99] Apodaca A. A., Gritz L.: *Advanced RenderMan: Creating CGI for Motion Picture*, 1st ed. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999. 2

[Can11] Cantlay I.: Directx 11 terrain tessellation. *Nvidia whitepaper* (2011). 5

[CC78] Catmull E., Clark J.: Recursively generated b-spline surfaces on arbitrary topological meshes. *Computer-Aided Design 10*, 6 (1978), 350–356. 5

[CCC87] Cook R. L., Carpenter L., Catmull E.: The Reyes image rendering architecture. *Computer Graphics (Proc. SIGGRAPH) 21*, 4 (1987), 95–102. 1, 2

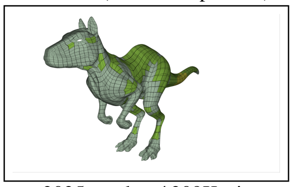
[DKT98] DeRose T., Kass M., Truong T.: Subdivision surfaces in character animation. SIGGRAPH'98, pp. 85–94. 7

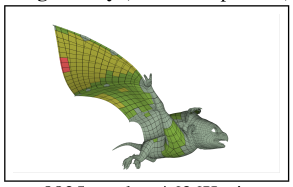
[FFB*09] Fisher M., Fatahalian K., Boulos S., Akeley K., Mark W. R., Hanrahan P.: DiagSplit: parallel, crack-free, adaptive tessellation for micropolygon rendering. *ACM Transactions on Graphics (Proceedings of SIGGRAPH Asia) 28*, 5 (2009), 150:1–150:10. 3

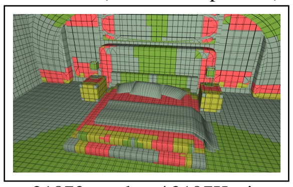
[LCWB80] Lane J., Carpenter L., Whitted T., Blinn J.: Scan line methods for displaying parametrically defined surfaces. *Communications of the ACM 23*, 1 (1980), 23–34. 2

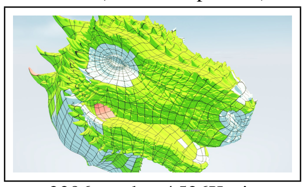
[LSNC09] Loop C., Schaefer S., Ni T., Castano I.: Approximating subdivision surfaces with Gregory patches for hardware tessellation. *ACM Transactions on Graphics (Proceedings of SIGGRAPH Asia) 28*, 5 (2009), 151:1–151:9. 5

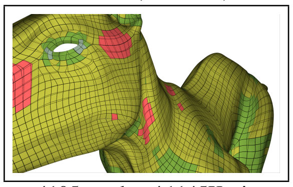
Killeroo (2894 base patches) **DragonBaby** (8154 base patches) **Bedroom** (15038 base patches) **Lizard**(2000 base patches)

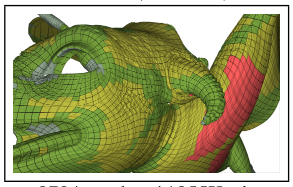

2035 patches / 309K tris
0.48 ms (0.11/0.37)

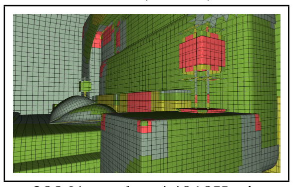

8825 patches / 626K tris
1.85 ms (0.17/1.68)


21873 patches / 2197K tris
1.9 ms (0.3/1.6)


2396 patches / 526K tris
2.43 ms (0.38/2.05)


4105 patches / 1145K tris
1.02 ms (0.15/0.87)


8704 patches / 1855K tris
3.40 ms (0.25/3.15)


29961 patches / 4019K tris
2.73 ms (0.43/2.3)

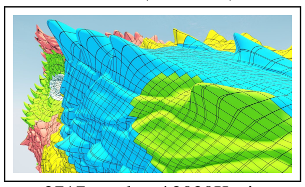

3717 patches / 2020K tris
12.47 ms (0.67/11.8)

Figure 9: *Performance statistics of our method. We report the number of subpatches sent for rasterization and the rasterized polygons. The first number of the timings show the total rendering time, which we break down as* FracSplit / Rasterization. *All images were captured at Full-HD resolution. For the DragonBaby and Lizard back-patch culling was disabled.*

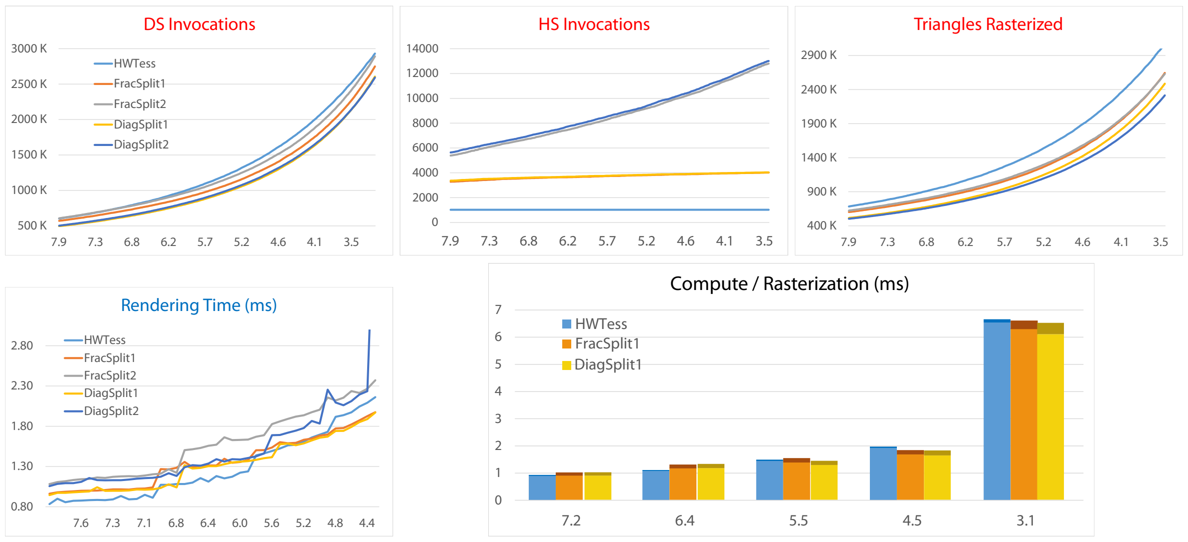


Figure 10: *Performance metrics of adaptive tessellation methods, as a function of target triangle size (the horizontal axis of each plot). The numbers in the names mean the split iterations (e.g.* ***FracSplit2*** *has two). We profiled shader invocations as well as timings from a static view (the setting was identical to Fig. 8, and the image resolution was* 3840 × 2160*).*

[MHAM08] MUNKBERG J., HASSELGREN J., AKENINE-MÖLLER T.: Non-uniform fractional tessellation. In *Proceedings of Graphics Hardware* (2008), pp. 41–45. 2

[Mor01] MORETON H.: Watertight tessellation using forward differencing. In *Proceedings of Graphics Hardware* (2001), pp. 25–32. 1, 2, 3, 4

[NL13] NIESSNER M., LOOP C.: Analytic displacement mapping using hardware tessellation. *ACM Transactions on Graphics 32*, 3 (2013), 26. 2

[NLMD12] NIESSNER M., LOOP C., MEYER M., DEROSE T.: Feature-adaptive GPU rendering of Catmull-Clark subdivision surfaces. *ACM Transactions on Graphics 31*, 1 (2012), 6:1–6:11. 7

[PO08] PATNEY A., OWENS J. D.: Real-time Reyes-style adaptive surface subdivision. *ACM Transactions on Graphics (Proceedings of SIGGRAPH Asia) 27*, 5 (2008), 143:1–143:8. 2, 5

[SM88] SEDERBERG T. W., MEYERS R. J.: Loop detection in surface patch intersections. *Computer Aided Geometric Design 5*, 2 (1988), 161–171. 5