DirectX 11 Reyes Rendering

Lihan Bin
University of Florida
lbin@cise.ufl.edu

Vineet Goel
AMD/ATI,
Vineet.Goel@amd.com

Jorg Peters
University of Florida
jorg@cise.ufl.edu

ABSTRACT

We map reyes-rendering to the GPU by leveraging new features of modern GPUs exposed by the Microsoft DirectX11 API. That is, we show how to accelerate reyes-quality rendering for realistic real-time graphics. In detail, we discuss the implementation of the split-and-dice phase via the Geometry Shader, including bounds on displacement mapping; micropolygon rendering without standard rasterization; and the incorporation of motion blur and camera defocus as *one* pass rather than a multi-pass. Our implementation achieves interactive rendering rates on the Radeon 5000 series.

Keywords: reyes, motion blur, camera defocus, DirectX11, real-time , rasterization, displacement

1 INTRODUCTION

The reyes (render everything you ever saw) architecture was designed by Pixar [CCC87] in 1987 to provide photo-realistic rendering of complex animated scenes. Pixar's PhotoRealistic RenderMan is an Academy Award-winning offline renderer that implements reyes and is widely used in the film industry. A main feature of the reyes architecture is its adaptive tessellation and micropolygon rendering of higher-order (not linear) surfaces such as Bézier patches, NURBS patches and Catmull-Clark subdivision surfaces. (A micropolygon is a polygon that is expected to project to no more than 1 pixel.)

The contribution of our paper is to judiciously choose features of the Microsoft DirectX11 API to achieve real-time performance. We make heavy, non-standard use of the Geometry Shader and of the tessellation engine, instancing and multiple-render-targets to minimize the number of GPU passes. In particular,

- the Geometry Shader streams out alternatively to a split- or a dice-queue so that all data remains on the GPU,
- we replace triangle rasterization by point rendering,
- we accommodate displacement mapping, and
- we combine blur and de-focus in a single pass followed by a rendering pass to obtain real-time performance. Fig. 1 gives a high-level view of the framework; Fig. 4 provides a more detailed view.

Overview. After a brief discussion of prior related work, Section 2 reviews the reyes architecture stages and known efficient algorithmic options. Section 3 shows how our implementation takes specific advantage of the Direct X11 pipeline to obtain real-time performance. Section 4 analyzes the effect of implementation-specific choices and discusses alternatives. We conclude with Section 5 discussing limitations of the present implementation and pointing out how current hardware evolution will impact our framework.

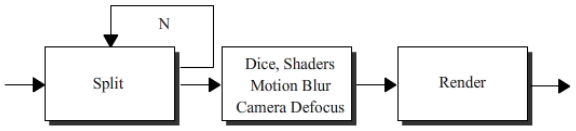

Figure 1: The three types of passes in our framework:the split stage (consisting of *N* adaptive refinement passes), one pre-rendering pass and one final composition pass.

1.1 Related Prior Work

The paper assumes basic familiarity with the shader stages of a modern GPU pipeline. In particular, we focus on the MS Direct X11 standard. In *Real-time reyes-style adaptive surface subdivision*, Patney and Owens [PO08] port the split-and-dice stage (see Section 2) on the GPU using CUDA. Micropolygons are rendered via OpenGL so that the approach is too slow for real-time rendering. *RenderAnts* by Kun Zhou et al. [ZHR+09] is an implementation of the complete reyes-pipeline based on a special, newly created, hence non-standard GPGPU programming language, called BSGP. The main contribution of the RenderAnts approach is a software layer which attempts to load-balance within various stages. Our approach uses the hardware to balance the load within different shader stages, avoiding the software overhead. Several recent publications time of this submission) underscore the importance of defocus and motion blur in real-time (non-reyes) settings: *Micropolygon Ray Tracing with Defocus and Motion*

WSCG 2010 conference proceedings, ISBN 80-903100-7-9
WSCG'2010, September 7-10 – September 7-10, 2010
Brno, Czech Republic.

Blur [HQL+10] proposes a fast ray traveral data structure for micropolygons, *Real-Time Lens Blur Effects and Focus Control* [LES10] simlutates precise lense effects via (non-micropolygon) ray tracing, accelerated by depth-peeling and [HCOB10] focuses on *Using blur to affect perceived distance and size*. Our approach to motion blur and defocus is not based on ray tracing and different from all three. *Data-Parallel Rasterization of Micropolygons* [FLB+09] proposes an efficient micropolygon rasterization (on the CPU). We adapt this algorithm to the GPU setting and add camera defocus and motion blur as well as displacement mapping in the same pass. We also leverage the simple split-and-dice strategy, *DiagSplit* of Fisher et al. [FFB+09] to reduce the number of micropolygons generated from a diceable patch. However, we base our estimate on a proper bounding box rather than just the corner coefficients.

2 THE REYES ARCHITECTURE STAGES AND KNOWN GPU IMPLEMENTATION STRATEGIES.

The reyes architecture stages defined by Cook, Carpenter and Catmull [CCC87] are as follows.

Input:	high-order patches
Splitting:	Adaptive recursive patch split until the screen space projection is sufficiently small.
Dicing:	Uniform split into micropolygons.
Shading:	Shaders (Displacement Shader, Surface Shader and Lighting Shader) are applied to each vertex.
Sampling:	Multiple samples are generated at different time and different lens positions (using stochastic sampling).
Composition:	Samples/Fragments on the same screen location are depth-sorted and blended.

Below we discuss known good reyes-implementation strategies that influenced our implementation; and we discuss the basics of motion blur and camera defocus.

2.1 Splitting

Splitting of the higher-order patch means adaptive subdivision of its domain to partition it into pieces. The goal is to obtain sub-patches that yield micropolygons (that map to one pixel) when uniformly tessellated (diced). The standard split algorithm computes a screen bound of an input patch. If this bound is less than the pixel threshold, the patch is sent to the dice stage where it is tessellated uniformly; otherwise it is split into sub-patches until the condition is satisfied. The algorithm works well when the patch control points are equally spaced. But when the control points are highly non-uniform, then uniform tessellation over-tessellates in some regions or under-tesselates in others.

DiagSplit [FFB+09] is a heuristic for avoiding over-tessellation (and taking care of T-joints, where one sub-patch is split but not its neighbor). The idea is to give shorter edges lower tessellation factors and thereby adjust the domain tessellation to the size of the projected image (screen variation). For, if the edge has non-uniform screen variation, uniform domain tessellation in the dicing stage cannot guarantee the micropolygon property: we may get some larger polygons and some extremely small polygons.

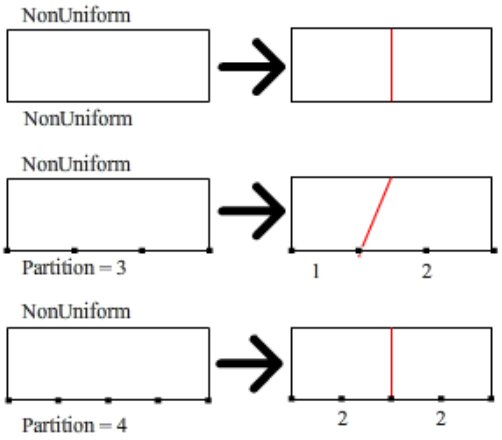

Figure 2: Three DiagSplit scenarios: simple but effective.

In [FFB+09] good correspondence is called *uniform* and poor correspondence *non-uniform*. To check uniformity of domain tessellation with the range tessellation, i.e., uniformity of screen variation under uniform domain tessellation, we step in equal interval through the domain and sample $n+1$ points p_i for each edge. Let P be the projection to screen space, $\ell_i := \|Pp_{i-1} - Pp_i)\|$ and $m := \max_i \ell_i$. If

$$\sigma := |mn - \sum \ell_i| \leq 1 \qquad (1)$$

then $\ell_i \sim m$ for all i, i.e. that the range is uniformly partitioned and we have good correspondence. If we then set the edge tessellation factor τ to $\tau := mn+1$, i.e. partition the domain edge into mn segments, then one domain segment should correspond to one pixel in the range.

Given the edge tessellation factors τ_0 and τ_2 of two opposing edges of a quad, we split the patch according to one of three choices (see Fig. 2).

— If both $\sigma_0 > 1$ and $\sigma_2 > 1$, we split at the mid-points.
— If $\sigma_k > 1$ for only one $k \in \{0,2\}$ then
  — if τ_k is even, split at the mid-point.
  — if τ_k is odd, split at $\lceil \tau_k/2 \rceil$.

Watertightness is maintained when shared edges have the same edge tessellation factor.

2.2 Motion blur and Camera defocus

Motion blur is an important tool to convey a sense of speed. Long exposure of a moving object creates a

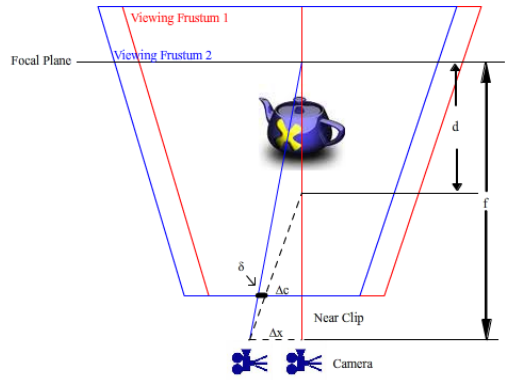

Figure 3: Defocus: simulating defocus by camera shift of less than Δ_c and projection shift of less than Δ_x so that $\delta < 1$.

continuous and blurred image on optical film. To approximate the effect of the Model matrix changing from M_0 to M_1, one averages the images obtained from rendering with n_{mb} intermediate matrices $(1-\frac{i}{n_{mb}+1})M_0 + \frac{i}{n_{mb}+1}M_1$, $i = 1,\ldots,n_{mb}$.

Camera defocus is used to guide viewer attention: as light travels through a lens, only the objects that are within a certain range appear sharp. In order for the light to converge to a single point, its source needs to be in an *in-focus* plane, a certain distance away from the lens. Anything not in or near the plane is mapped to a circle of confusion (CoC) that increases with the distance from the in-focus plane (and the size of the lens). When the CoC is smaller than the film resolution, we call the range in focus and get a sharp image. (In standard rendering, the lens is a 'pinhole', i.e. the lens has zero size yielding always sharpness.)

We can simulate camera defocus, analogous to motion blur, by rendering the scene multiple times. To obtain an un-blurred image within d of the focal point while blurring the remaining scene, we shift the viewing frustum by maximally Δ_x within a plane (cf. Fig. 3) and adjust the viewing direction towards the focus so that the image of the focal object project to the same 2D screen location. (If the camera shift Δ_x were too large, only the focal point would be sharp.) That is (c.f. Fig. 3), for a given distance d_{near} to the near clipping plane, focal-region-width d and distance to the in-focus plane f, we need to determine upper bounds Δ_x and Δ_c by setting $\delta \in \{(\pm 1, \pm 1)\}$, i.e. the allowable perturbations of at most one pixel, We obtain Δ_x and Δ_c by solving two linear equations

$$\frac{\Delta_x}{f-d_{near}} = \frac{\Delta_c}{f}, \qquad \frac{\Delta_x - \delta}{f-d_{near}-d} = \frac{\Delta_c}{f-d}$$

arising from similar triangles,

3 OUR REYES IMPLEMENTATION.

One update of the image in our implementation uses $N+2$ passes as outlined in Fig. 1 and in more detail in Fig. 4: There are N passes for adaptive refinement (splitting), one heavy pre-rendering pass that combines dicing, shading (computation of normals) and sampling (displacement, motion blur and defocus), and a final pass for composition. The implementation uses three buffers: PatchBuffer[0], PatchBuffer[1] and diceablePatchBuffer.

3.1 Implementing the reyes-Splitting Stage

In pass 1, in the Geometry Shader, we take PatchBuffer[read] where read = 0 as input and test if a patch is diceable or not. If it is diceable, we stream out to the diceablePatchBuffer, otherwise we stream out to PatchBuffer[write] where write = 1. In the next passes, up to some pass N when the size of PatchBuffer[write] is 0, we satisfy the reyes requirement by repeatedly switching the read and the write PatchBuffer and subdividing the patches. Once the PatchBuffer[write] ends up empty, we switch to the pre-rendering pass.

In more detail, in the split passes, we test each edge of the patch in screen space for uniformity according to (1). Our algorithm takes displacement mapping into account when computing the edge tessellation factor τ. (Displacement mapping moves vertices and thereby changes the size of tessellated polygon violating the micropolygon property. The effect, pixel dropout, is shown in Fig. 9, *top*.) To maintain the micropolygon property, we perform one additional test on each edge: we find the difference between the maximum displacement and minimum displacement, or edge-width, on the edge, as explained in the next paragraph. If the difference is large, we subdivide the edge.

Let the *edge-width* of displacement map be the maximal surface perturbation along a domain edge due to the displacement map. To avoid searching through the displacement map to determine the edge-width, i.e. to avoid a repeated for-loop, we convert the displacement map to a mip-map structure on the GPU. This approach follows the spirit of the CPU algorithm [MM02], in building a mipmap-like structure, but, instead of averaging 2×2 pixels, we compute the *max displacement and the min displacement* values of the 2×2 pixels. This allows us to read off the edge-width for any subdivided patch by looking up the max and min entries at the level of the displacement mip-map corresponding to log(edge-length).

The approach works well, except that it becomes inefficient if the patch has a markedly high-frequency displacement map. Then the upper bound on the edge tessellation factor is high and easily results in overestimation and hence poor performance. We address this by

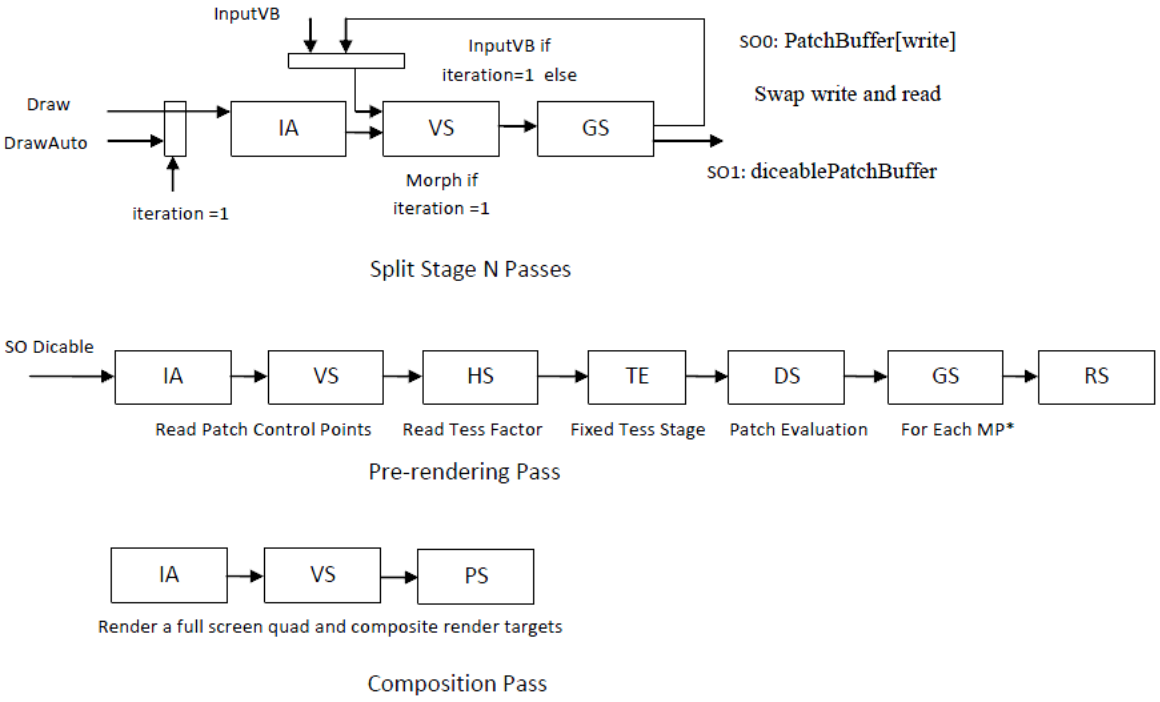

Figure 4: Implementation overview: N initial splitting passes, followed by a heavy pre-rendering pass and a final pass compositing render targets. The pre-rendering pass dices into micropolygons (MP*) using the tessellation engine (Section 3.2); the pass also applies instancing and multiple-render-target in the Geometry Shader to enable defocus and motion blur (Section 3.4); and it computes the micropolygon normal and displacement (with the distortion by the displacement already accounted for into the split stage – see Section 3.1). Abbreviations: IA = Input Assembly, VS = Vertex Shader, HS = Hull Shader, TE = Tessellation Engine, DS = Domain Shader, GS = Geometry Shader, RS = Rasterizer, PS = Pixel Shader, VB= Vertex Buffer, MP=Micropolygon.

clamping to a maximal entry and having the Geometry Shader output sufficiently many points to represent the such unusually big distortion triangles (see Algorithm 1). The result is displayed in Fig. 9, *bottom*.

3.2 Implementing the reyes-Dice Stage

We apply the DirectX 11 *tessellation engine* using the edge tessellation factors τ computed in the Splitting stage. The input to this stage is the diceablePatchBuffer, the output are micro-polygons. Note that this stage, as well as the Shading and the Sampling stages of reyes-rendering are folded into one pre-rendering pass.

3.3 Replacing the GPU-Rasterization Stage

With the output of the DirectX 11 Domain Shader stage in the form of micro-triangles of size at most 1 pixel (see Fig. 5), we rasterize in the Geometry Shader. Current hardware rasterization is designed to rasterize *large triangles*. In particular, it can test a 4x4 'stamp' of samples against each polygon to increase throughput and parallelism. However, for each micropolygon this approach streams out at most one pixel wasting all 4×4 parallelism. Therefore we generate pixel-sized output in the Geometry Shader and send the resulting *point* (not triangle) *per micropolygon* through the hardware rasterizer (which cannot be skipped in current hardware). While the one-primitive-per-clock-cycle rule of the hardware means that the micro-triangle and the point would in principle result in equal speed, using one point increases throughput in the setup, since we send one vertex (point) instead of the three triangle vertices (see Fig. 6).

To determine a correct point sample, we compute the micropolygon's bounding box and then the potentially covered pixels with *findOverlappedPixelGrid*. As explained in Algorithm 1, we then generate a point from any sample that has a successful point-in-triangle test. If all samples fail the point-in-triangle test, there is no need to draw a point since neighboring triangles will provide points for the corresponding pixels. This is akin to Interleaved Sampling [KH01]. We note that without displacement, the projection of any micropolygon is smaller than 1 pixel and then at most four pixels are covered, as illustrated in Fig. 5; but with our correction for high-frequency displacement, more pixels can be generated.

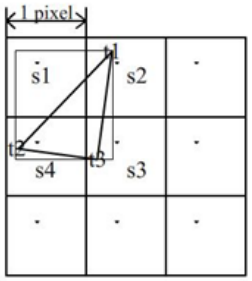

Figure 5: Pixel-footprint of a micropolygon returned by findOverlappedPixelGrid applied to its bounding box. The pointInTriangle test will output $s4$.

input : float4[3] pos4D; – triangle vertices
Uses: pointInTriangle(q,T,u,v) – true if q is in the triangle T; returns corresponding u,v
output: Point p

```
pos2D[0..2] = ProjectToScreen(pos4D[0..2]);
BoundingBox bbox =
computeBoundingBox(pos2D);
PixelGrid = findOverlappedPixelGrid(bbox);
for q ∈ PixelGrid do
    if pointInTriangle(q,pos2D,u,v) then
        p ← pos4D,u,v;
        sendToRasterizer(p);
    end
end
```

Algorithm 1: Compute correct point sample as alternative to rasterization

3.4 Implementing the reyes-Sampling Stage

The sample stage typically requires one pass per sample for a total of n passes. We leverage the Multiple Render Target call to generate n samples in one pass so that our sampling and composition requires only two passes: a pre-rendering pass and a final composition pass. We use the DirectX 11 OutputMerge stage for the z-test and for alpha-blending. (As usual, for transparency, patches need to be kept ordered in the split stage.)

For camera defocus, we shift the camera by at most Δ_c and the near-clip-plane by at most Δ_x, projecting with step size Δ_c/n_{cd}, respectively Δ_x/n_{cd} around the initial position. (This yields $(2n_{cd}+1)(2n_{cd}+1)$ projections). Here n_{cd} is a user-specified number with larger n_{cd} providing better results at higher cost. To combine an n_{mb}-transformation motion blur with a n_{cd}-sampling camera defocus, we, off-hand, have to render a frame $n_{mb}n_{cd}$ times as shown in Algorithm 2.

However, instancing and the DX11 multiple-render-target call allow us to send objects with different matrices to different render targets. We leverage this to approximate motion blur and defocus *in one pass* in place of $n_{mb}n_{cd}$ passes (see Fig. 8). This is followed by a final composition pass that blends the entries to generate the image. To meet current limitations of the number of render targets, we set $n_{cd} = 1$ generating 8 different Model matrices (leaving out the unperturbed projection) by calling 8 instances of the Geometry Shader and having the ith instance render to the ith render target. The composition pass then combines these targets.

```
for i = 0 to n_mb do
    for j = 0 to n_cd do
        Model View Projection Matrix = Model[i]
        * View[j] * Projection[j]
    end
end
```

Algorithm 2: Standard algorithm of complexity $n_{mb}n_{cd}$. Our implementation approximates it by a single pass using multiple-render-targets.

Although, for interactive motion blur, we may assume small changes in the Model matrix, there may be triangles whose projection is stretched to violate the micropolygon certification. Just as with extreme displacement (Section 3.1, last paragraph) we then have the Geometry Shader output multiple pointis.

3.5 Shadows

To obtain interactive shadows (see e.g. Fig. 8), we interleave passes computing a *shadow map* by reyes rendering from the light position. Here we do not need the all reyes stages but only split, dice and sample; there is no need to shade and composite. By varying the size of the micropolygons, the artist can adjust the shadow quality.

4 ANALYSIS OF THE IMPLEMENTATION

Due to hardware limitations, the choice of the edge tessellation factor τ in Equation (1) (which that determines the number of segments on each edge in the tessellation engine) cannot be arbitrarily large. In fact, in our implementation, we set a *diceable threshold* $\bar{\tau}$ as an upper bound on τ since we can improve throughput by setting it lower than the current hardware tessellation bound of 64. Table 1 shows how the number of sub-patches varies with $\bar{\tau}$ while the number of evaluation points remains within the same order of magnitude due to adaptive tessellation in the splitting stage. The threshold influences the best load balancing strategy since a lower $\bar{\tau}$ results in a larger number of small patches with low tessellation factor. We tested five different $\bar{\tau}$ values: 8,12,16,20 and 24 (higher values were not competitive). Our implementation achieves the best performance for $\bar{\tau} = 24$.

$\bar{\tau}$	tessellation vertices	number of patches
8	3446260	37134
12	2932396	15251
16	2763775	9378
20	2602425	6151
24	2528523	3835

Table 1: Influence of the diceable threshold $\bar{\tau}$ on the number of patches.

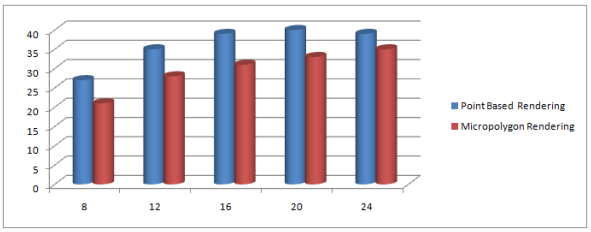

Figure 6: Point-based vs Micropolygon rendering. Performance for a diceable threshold $\bar{\tau} \in \{8, 12, 16, 20, 24\}$ (y-axis:fps, x-axis: $\bar{\tau}$).

As a canonical test scene, we chose the scene in Fig. 9, i.e. a teapot with displacement map and a resolution of 1280×1024 pixels. Fig. 6 shows that rendering a point is consistently superior to rendering the micropolygon via the rasterizer. Adding motion blur and defocus as another Geometry Shader operation in the pre-rendering pass affects the performance *sub-linearly* in the number of projections: projecting 8 times only reduces the frames per second to one fourth rather than one eighth. We conclude that the pre-rendering pass is not compute-bound. That is, rendering could be faster were it not for the limitation of current GPU hardware to process one primitive per clock cycle: if more than one point could be processed per cycle downstream from the Geometry Shader, the throughput would increase correspondingly. (The simple solution of rendering a polygon the size of several micropolygons would not match the stringent requirements of reyes-rendering.) The main bottleneck, however, is the splitting stage rather than the heavy pre-rendering pass. Fig. 7 shows the extra cost of multiple passes for splitting by juxtaposing the (correct) split+dice performance with a dice-only performance that generates roughly the same number of polgons (but may result in incorrect pixel-dropout).

5 DISCUSSION AND FUTURE WORK

Since the pre-rendering pass is not compute-bound, we can accommodate more complex surface and lighting shaders. However, we did not invest into our own complex shaders, since we anticipate that Renderman shaders will in the future be compiled directly on the Geometry Shader obsoleting any custom-made reyes shaders.

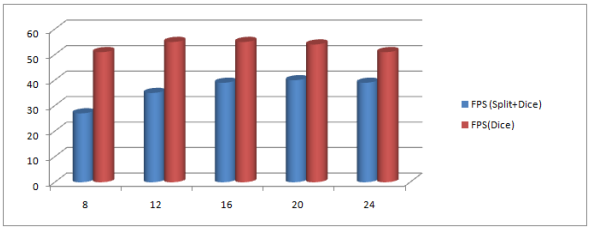

Figure 7: The extra cost of multiple passes for splitting is the difference between correct split+dice and incorrect dice-only (y-axis:fps, x-axis: $\bar{\tau}$).

The proposed framework can accommodate (and is being tested for) order-independent transparency but at the cost of slower performance and additional buffer space, depending on the depth of transparency sorting.

In summary, the contribution of the paper is an efficient use of the DX11 pipeline: split-and-dice via the Geometry Shader, micropolygon rendering without standard rasterization and motion blur and camera defocus as *one* pass rather than a multi-pass via MRT.

ACKNOWLEDGMENTS

Work supported in part by NSF Grant CCF-0728797 and by ATI/AMD.

REFERENCES

[CCC87] Robert L. Cook, Loren Carpenter, and Edwin Catmull. The Reyes image rendering architecture. In Maureen C. Stone, editor, *Computer Graphics (SIGGRAPH '87 Proceedings)*, pages 95–102, July 1987.

[FFB+09] Matthew Fisher, Kayvon Fatahalian, Solomon Boulos, Kurt Akeley, William R. Mark, and Pat Hanrahan. DiagSplit: parallel, crack-free, adaptive tessellation for micropolygon rendering. *ACM Transactions on Graphics*, 28(5):1–8, December 2009.

[FLB+09] Kayvon Fatahalian, Edward Luong, Solomon Boulos, Kurt Akeley, William R. Mark, and Pat Hanrahan. Data-parallel rasterization of micropolygons with defocus and motion blur. In *HPG '09: Proceedings of the Conference on High Performance Graphics 2009*, pages 59–68, New York, NY, USA, 2009. ACM.

[HCOB10] Robin Held, Emily Cooper, James O'Brien, and Martin Banks. Using blur to affect perceived distance and size. In *ACM Trans. Graphics*, 2010.

Figure 8: (*left*) Blurring with $n_{mb} = 8$ and (*right*) defocus with $n_{cd} = 1$. See also the accompanying video.

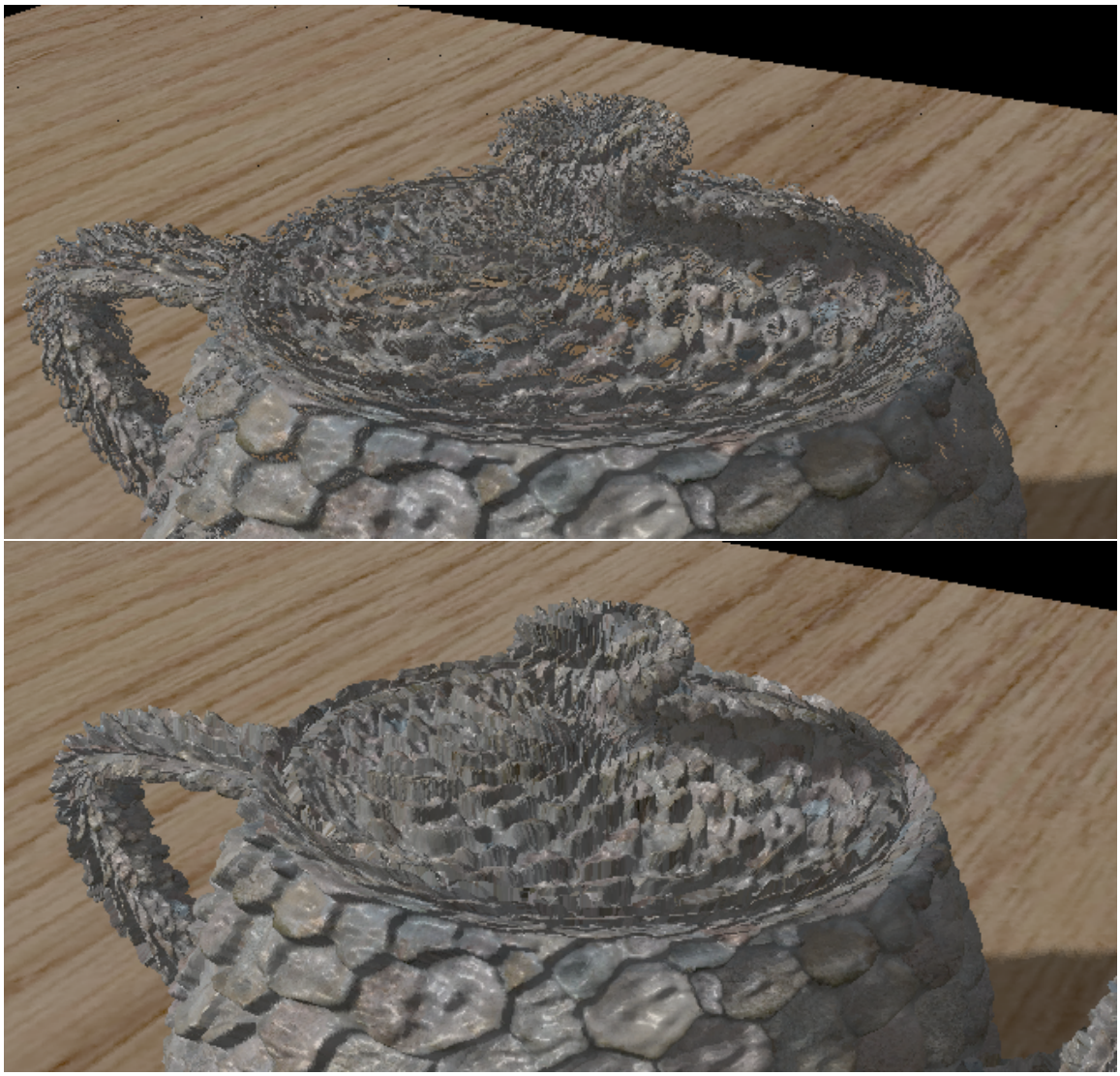

Figure 9: Displacement mapping. (*top*) Pixel dropout due to incorrect treatment with [MM02]. (*bottom*) The problem is fixed by applying the strategy of Section 3.1.

[HQL+10] Qiming Hou, Hao Qin, Wenyao Li, Baining Guo, and Kun Zhou. Micropolygon ray tracing with defocus and motion blur. In *ACM Trans. Graphics, 29(3), 2010 (Proc. ACM SIGGRAPH 2010)*, 2010.

[KH01] Alexander Keller and Wolfgang Heidrich. Interleaved sampling. In Steven J. Gortler and Karol Myszkowski, editors, *Rendering Techniques*, pages 269–276. Springer, 2001.

[LES10] Sungkil Lee, Elmar Eisemann, and Hans-Peter Seidel. Real-time lens blur effects and focus control. In *ACM Trans. Graphics, 29(3), 2010 (Proc. ACM SIGGRAPH 2010)*, 2010.

[MM02] Kevin Moule and Michael D. McCool. Efficient bounded adaptive tessellation of displacement maps. In *Proc. Graphics Interface*, pages 171–180, May 2002.

[PO08] Anjul Patney and John D. Owens. Real-time reyes-style adaptive surface subdivision. *ACM Trans. Graph*, 27(5):143, 2008.

[ZHR+09] Kun Zhou, Qiming Hou, Zhong Ren, Minmin Gong, Xin Sun, and Baining Guo. Renderants: interactive reyes rendering on GPUs. *ACM Trans. Graph*, 28(5), 2009.