Real-Time Rendering Methods With Adaptive Levels of Detail for Fast Rendering of Parametric Objects on Modern GPUs

Johannes Unterguggenberger, Lukas Lipp, Michael Wimmer, Markus Steinberger, Bernhard Kerbl, and Markus Schütz

***Abstract*—Parametric functions are an extremely efficient representation for 3D geometry, capable of compactly modelling highly complex objects. Once specified, parametric 3D objects allow for visualization at arbitrary levels of detail (LOD), at no additional memory cost, limited only by the amount of evaluated samples. However, mapping the sample evaluation to the hardware rendering pipelines of modern graphics processing units (GPUs) is not trivial. In this article, we propose a general method for efficient rendering of parametrically-defined 3D objects on modern hardware architectures. Our method adaptively analyzes, allocates and evaluates parametric function samples to produce high-quality renderings. Geometric precision can be modulated from few pixels down to sub-pixel level, enabling real-time frame rates of several 100 frames per second (FPS) for various parametric functions. We propose a dedicated LOD stage, which outputs patches of similar geometric detail to a subsequent rendering stage that uses either a hardware tessellation-based approach or performs point-based software rasterization. Our method requires neither preprocessing nor caching, and the proposed LOD mechanism is fast enough to run each frame. Hence, our approach also lends itself to animated parametric objects. We demonstrate the benefits of our method over a state-of-the-art spherical harmonics (SH) glyph rendering method and over classical LOD approaches, while showing its flexibility on a range of other demanding shapes.**

***Index Terms*—Parametric functions, adaptive levels of detail (LOD), real-time rendering, tessellation, point rendering.**

I. Introduction

MICROPOLYGON rendering techniques such as Epic Games' Nanite [6] allow rendering of geometry in such high detail that the triangles are approximately pixel-sized with respect to screen space resolution. Nanite achieves real-time frame rates by utilizing an elaborate streaming system that fetches and renders clusters of the base mesh—which have been prepared in a precomputation step—in suitable geometric detail. Its main advantage over conventional approaches is higher rendering quality and continuous LOD transitions, while the former often suffer from noticeable visual artifacts when switching a model from one discrete LOD to another [15]. Conventional and modern approaches both suffer from high storage requirements and memory pressure during rendering due to dynamically loading clusters or discrete LODs.

In this paper, we propose using parametric objects which can help to relieve memory pressure in geometry stages of rendering pipelines, and comes with virtually zero storage requirements: *Parametric functions* are entirely defined in code while still producing attractive or useful shapes with possibly high geometric detail, limited only by machine floating point precision. Parametric functions are a mathematical description of an object's surface. Alternative mathematical descriptions include implicit functions and signed distance functions (SDF) [19]. However, rendering such representations efficiently remains challenging, as no algorithm can render arbitrary implicit surfaces both quickly and accurately [16]. In recent years progress has been made for SDF rendering [10], yet peak performance is still achieved by specialized rendering solutions targeted to specific mathematical objects, such as Peters et al. [21] for spherical harmonics (SH) glyph rendering. It uses polynomial root finding with ray tracing to achieve good rendering performance. SH glyphs can, however, also be expressed parametrically which—and this is one of the key features of our approach—does not require any preprocessing, derivatives, or root finding but instead, directly evaluates the parametric expression. Our approach is more general, supporting a wide range of 3-dimensional objects representable through parametric expressions, and exceeds the rendering performance for specialized implicit surfaces such as higher order SH glyphs.

Our proposed technique takes as input only a parametric function. Compared with previous work, our solution makes few assumptions about the sampled functions, supporting a wide range of complex shapes like those shown in Fig. 1: SH glyphs, seashell models defined as described by Wilson [34], or yarn and fiber curves as described by Crane [4]. It is applicable to a wide range of parametric functions that are described according to the general conditions defined in Fig. 2. We propose a multi-step pipeline: a dedicated LOD analysis stage adaptively determines the sampling density to be used by a subsequent rendering stage.

Received 23 September 2024; revised 19 August 2025; accepted 24 November 2025. Date of publication 2 December 2025; date of current version 17 June 2026. This work was supported by WWTF - Instant Visualization and Interaction for Large Point Clouds and Netidee (Project Math2Model, Project call #18, Project ID: 6890, TU Wien Bibliothek, under Project ICT22-055. Recommended for acceptance by C. Garth. *(Corresponding author: Johannes Unterguggenberger.)*

Johannes Unterguggenberger is with the TU Wien, Institute of Visual Computing & Human-Centered Technology, 1040 Vienna, Austria, and also with Huawei Technologies, 1220 Vienna, Austria (e-mail: junt@cg.tuwien.ac.at).

Lukas Lipp, Michael Wimmer, Bernhard Kerbl, and Markus Schütz are with the TU Wien, Institute of Visual Computing & Human-Centered Technology, 1040 Vienna, Austria (e-mail: lukas.lipp@cg.tuwien.ac.at; wimmer@cg.tuwien.ac.at; kerbl@cg.tuwien.ac.at; mschuetz@cg.tuwien.ac.at).

Markus Steinberger is with Huawei Technologies, 1220 Vienna, Austria, and also with Graz University of Technology, 8010 Graz, Austria (e-mail: steinberger@icg.tugraz.at).

The source code is available at https://github.com/cg-tuwien/FastRenderingOfParametricObjects.

Digital Object Identifier 10.1109/TVCG.2025.3638697

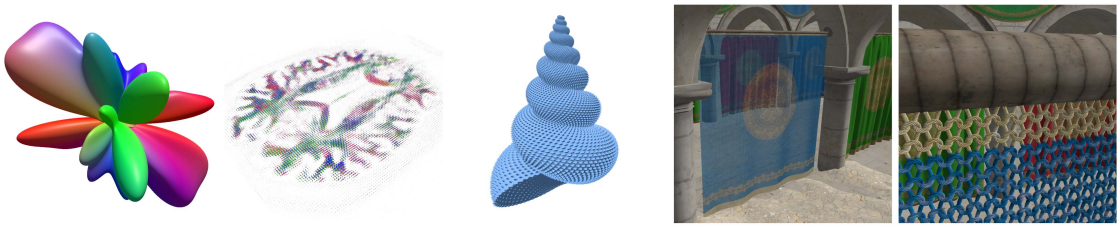


Fig. 1. Our technique is able to render a variety of parametrically defined objects with diverse properties in real time. The frames per second for (a) to (e) on a mid-range previous-generation NVIDIA RTX 3070 GPU are 248 FPS, 270 FPS, 349 FPS, 98 FPS, 138 FPS, respectively (Fig. 1(a) and (b) are rendered at 1440×1440 resolution, (c) to (e) at 1920×1080 with 4×SSAA and 8×MSAA).

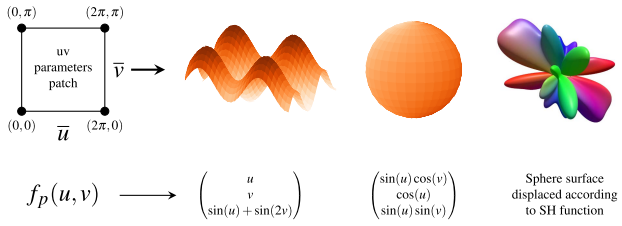


Fig. 2. We consider vector-valued parametric functions $f_p(u, v)$ with $f_p : \mathbb{R}^2 \to \mathbb{R}^3$, mapping independent 2D variables called parameters to 3D positions in euclidean space. The examples show the construction of a sine wave surface in $\mathbb{R}^3$, a sphere by interpreting $\mathbb{R}^2$ as spherical coordinates and $\mathbb{R}^3$ as the corresponding cartesian coordinates, and an SH glyph by further transforming the sphere using the respective SH functions.

Depending on the chosen sampling resolution, our technique can render highly tessellated patches (typically producing triangles which span over one or only a few screen pixels) or directly render the parametric function point-wise (typically sampling the function once or multiple times per screen pixel). Our point-based method draws inspiration from recent advances in point cloud rendering [25]. Common to both rendering approaches is their emphasis on utilizing the compute capabilities of GPUs and their ease of integration into existing rendering applications, possibly relieving memory pressure within rasterization-based graphics pipelines, and allowing to improve the overall visual fidelity of rendered scenes through ultra-detailed geometric objects. Examples for suitable applications of our technique could be curtains made of thousands of yarn curves in architectural pre-visualization in real-time, power-up items or terrains in video games, or detailed visualizations of medical measurements such as SH glyphs from magnetic resonance imaging (MRI). Objects rendered with our method can help to fully utilize GPU resources if compute throughput is not fully saturated, but memory throughput is already at its limit.

In summary, the contributions of our paper are:

1) We describe a general method to render a wide range of parametric functions with close to pixel-perfect geometric accuracy, which is fast enough to be used in conjunction with super sampling (SS).
2) We describe an efficient compute shader-based LOD selection algorithm that generates view-dependent parameter patches for generic parametric functions, leading to approximately uniform geometric detail in the rendered output across the object.
3) We describe different variants to render the patches from Contribution 2, including point-based rendering, rasterization using the hardware tessellator, and a hybrid technique to select the optimal rendering variant per patch.
4) We evaluate our method on a range of demanding parametric shapes and compare it to the state of the art in terms of SH glyph rendering by Peters et al. [21], and to a conventional LOD approach that switches between discrete LOD models.

This paper extends our previous work [32] with a more detailed description of rendering setups and configurations in Sections IV-B and IV-C. We describe how to achieve anti-aliasing with point-based rendering in Section IV-C, and a hybrid rendering approach in Section IV-D which can help to achieve more stable frame rates. The performance of these rendering variants is analyzed in Section V. Furthermore, we provide supplemental material for more insights into our hybrid rendering approach, and for guidelines on parametric object construction.

II. Related Work

Most previous work on rendering parametric curves or surfaces focuses on specific parametric shapes, such as the efficient rendering of rational Bézier patches [5], [24], Catmull-Clark subdivision surfaces [14], [17], [20], [35], or similar patches such as B-Spline or NURBS curves and surfaces [35]. Poirier et al. [22] focus on rendering SH glyphs for visualizing measurements from diffusion magnetic resonance imaging (dMRI) scans. Some of these techniques utilize specialized data structures [14], [20]. They all rely on a triangulated representation of a given type of parametric function for rendering. Some techniques create triangle primitives in software to be rendered by the hardware rasterizer or perform tessellation in software [5], [20], [24], [35], while others make use of hardware tessellation units [17], which are standard features on modern desktop GPUs. The technique by Kuth et al. [14] uses mesh shaders, which were first introduced to desktop GPUs with NVIDIA's Turing architecture [18] in 2018.

Especially older techniques do not try to produce ultra-detailed geometry but instead optimize for a visual error metric to be less than a pixel [5], [20], which is also the approach taken by the more recent technique by Worchel et al. [35]. Eisenacher et al. [5] is conceptually similar to our approach, insofar as it subdivides patches until a certain metric is satisfied. In contrast to our approach, they use uniform subdivision of patches, and their technique is tailored to Bézier patches. Their method could be incorporated into the structure of our method by using the same error metric from their "oracle" step in our Patch Subdivision stage instead of our generally applicable screen distance-based metric. Our tessellation-based rendering variant would be perfectly suitable for rendering Bézier patches, possibly requiring a crack avoidance procedure between neighboring patches.

The work by Poirier et al. [22] on SH glyph rendering takes a more pragmatic approach in trying to evaluate and set suitable tessellation levels, but fails to prevent visual inaccuracies. Another recent approach can produce and render ultra-detailed geometry at real-time frame rates on modern GPUs [14], but

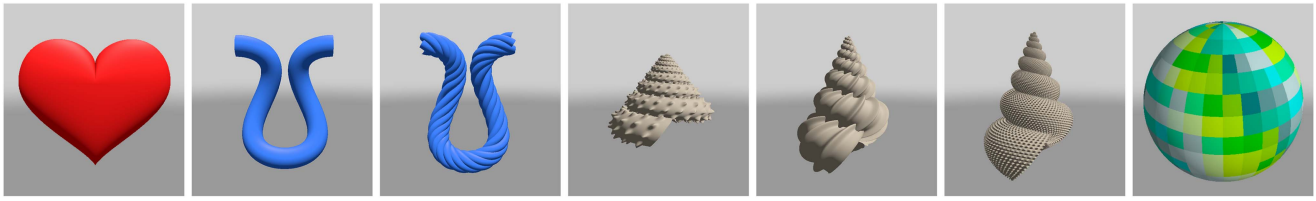


Fig. 3. These images show various parametrically defined objects. Slight variations in the parametric functions can lead to different shapes, as can be seen in (d) to (f), which all use the same underlying parametric function with slight variations in auxiliary parameter values. (g) shows a possible set of patches produced by our PATCH SUBDIVISION stage for a parametrically defined sphere.

focuses on Catmull-Clark subdivision surfaces only. In terms of SH glyph rendering, Peters et al. [21] achieve pixel-perfect geometric detail by intersecting a ray with the SH glyph through polynomial root finding for each screen pixel. Their visual results constitute a significant improvement over the results from Poirier et al. [22], but suffer from exceedingly decreasing performance with increasing SH order and visual artifacts with SHs of orders 10 and higher. In terms of rendering configuration, our proposed method is more similar to the approach by Poirier et al. [22] insofar as we also use the graphics pipeline. However, in terms of rendering quality and in terms of its performance characteristics, our method is much more similar to the ray tracing-based approach by Peters et al. [21]: both methods scale performance-wise relative to the number of rendered pixels: The method of Peters et al. traces one ray per pixel, while our solution selects suitable levels of geometric detail using a screen distance-based metric. Further usages of glyphs in the context of medical or scientific visualization include comparisons between healthy and infected persons [36], [37], [38] or comparisons between ensembles of stress tensor fields [1].

Point rendering can arguably also be described as an approach to render ultra-detailed geometry, given a sufficient number of samples. Recent work shows that modern GPUs are capable of processing and rendering 50 to 144 billion points per second [25], [26]. Schütz' earlier work [25] describes being limited by memory bandwidth. Their more recent work improves throughput through compression [26]. By sampling points on parametric functions, we avoid memory fetches as the bottleneck—potentially trading it for a compute-based bottleneck.

Epic Games' Nanite technology [6] enables rendering of ultra-detailed 3D meshes. It achieves this with careful preprocessing of meshes into clusters of 128 triangles each. At runtime, clever LOD switching enables seamless transitions between different LODs on a per-cluster basis and ensures that relevant clusters are streamed to GPU memory on demand. Clusters are selected so that rendering their triangles for a given camera position produces approximately pixel-sized triangles if the input mesh provides enough explicit detail. Nanite employs both hardware and software rasterization; they report that the latter is up to $3\times$ faster for rendering small triangles [9]. To combine software rasterization and hardware rasterization into the same render target, values are written into a single-channel 64-bit integer image by compute shaders during software rasterization, and from fragment shaders for hardware-rasterized triangles. Storing depth in the most significant bits of the written 64-bit integer values is crucial: this way, it serves as an equivalent to the depth test. While Nanite can render impressively detailed 3D meshes, it is still limited to static geometry. Unterguggenberger et al. [31] extend this approach partially to supporting animated skinned meshes of high geometric detail. Similar to Nanite, our approach combines the strengths of the graphics and compute pipelines of modern GPUs to produce geometry with close-to-pixel detail. However, we instead target parametric functions, which can be sampled with virtually arbitrary fidelity. Furthermore, our solution involves no preprocessing and recomputes the levels of detail from scratch each frame. Thus, it also lends itself to animated shapes with erratic changes in appearance.

III. PARAMETRIC FUNCTION DEFINITION

Our method considers parametric functions $f_p(u, v)$ that transform two input parameters (u, v) into Cartesian coordinates (x, y, z) of the three-dimensional euclidean space, i.e., $\mathbb{R}^2 \rightarrow \mathbb{R}^3$, or intuitively a transformation of a quad/rectangle into a three-dimensional surface as illustrated in Fig. 2. We illustrate the input to $f_p(u, v)$ with parameter patches that range from lower bound parameter values u_{min}, v_{min} to upper bound parameter values u_{max}, v_{max}, where we use the notations $\overline{u}$ and $\overline{v}$ refer to the whole parameter ranges, namely, $\overline{u} = [u_{min}, u_{max}]$ and $\overline{v} = [v_{min}, v_{max}]$. The arithmetic definition enables a compact representation of geometric shapes with varying complexity and desirable properties, such as C^∞ continuity.

```
#define PI        3.14159265359
// Creates the surface "Johi's Heart".
// Input: u ... first  parameter in the range [0, PI  )
//        v ... second parameter in the range [0, 2*PI)
vec3 sampleJohisHeart(float u, float v) {
    // Start with a sphere shape:
    vec3 p = vec3(sin(u) * cos(u), cos(u), sin(u) * cos(v));
    // Distort it into a heart shape:
    if (u < PI / 2.0) {
        p.y *= 1.0 - (cos(sqrt(sqrt(abs(p.x * PI * 0.7)))) * 0.8);
    } else {
        p.x *= sin(u) * sin(u);
    }
    p *= vec3(0.9, 1.0, 0.4);
    return p;
}
```

Listing 1. GLSL source code of a parametric function which produces a heart shape based on two input parameters u and v.

In the context of our method, a *parametric function* is expressed directly in source code; in addition to mathematical elements, it may also contain logical operators and flow control (conditional statements, loops, and recursions). This facilitates the intuitive generation of features that are harder to express mathematically, such as creases or discontinuities. An example of such a parametric function is given in Listing 1. It uses `if` statements to scale parts of a sphere base shape such that the resulting surface forms the heart shape shown in Fig. 3(a).

IV. METHOD

In this section, we describe our approach, which is able to render a wide variety of parametric functions in real time with controllable precision. Depending on the current frame's camera

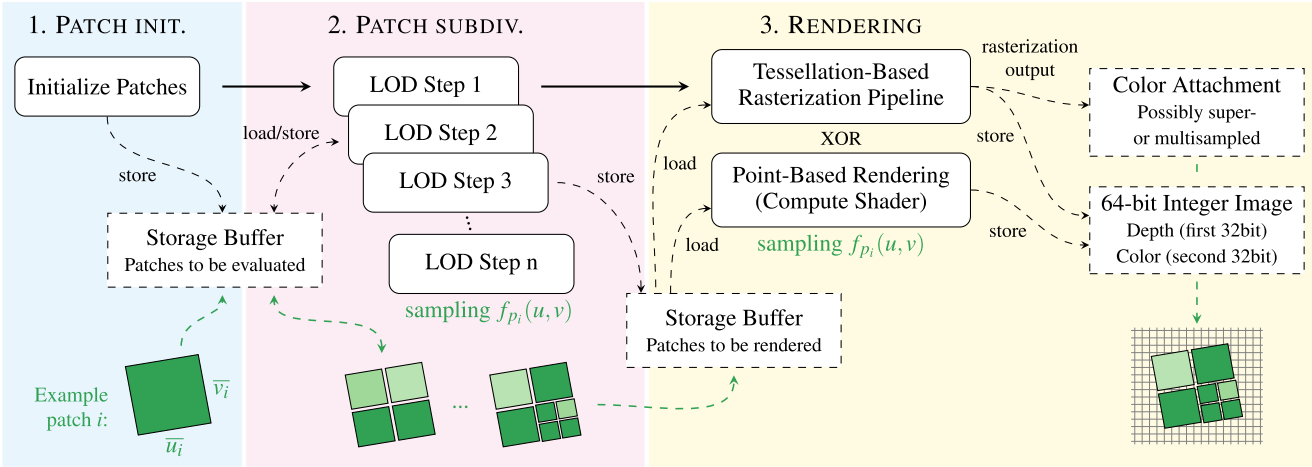


Fig. 4. Overview of the stages of our method, and the buffers and textures which are accessed in each stage. Patches are evaluated and subdivided, and then re-evaluated up to n times in the n LOD steps, before they are stored in the "Patches to be rendered" buffer. The rendering stage reads all the patches from that buffer and either generates a triangle mesh to render them via hardware tessellation or performs point-based rendering. The latter variant cannot be used with a renderpass that rasterizes into a framebuffer. Instead, it must perform `atomicMin` writes into a 64-bit integer buffer or image.

position, orientation, projection, and screen coordinates, a LOD stage produces patches of similar size in screen space, which are subsequently rendered with one of two different rendering variants. Fig. 4 depicts the overview of our method and its major stages:

1) Patch Initialization: A compute shader stores one patch per parametric object in the buffer of patches to be evaluated, in particular, the parameter ranges $\overline{u_i}$ and $\overline{v_i}$, along with some auxiliary data, such as the type of object—which refers to a particular $f_{p_i}(u,v)$—and which material shall be used for shading. For objects that are known to be very detailed, $\overline{u_i}, \overline{v_i}$ can already be uniformly subdivided in this stage, which ensures a minimal number of output patches to be forwarded to the rendering stage.
2) Patch Subdivision: The second stage executes up to a predefined number of n LOD steps, which subdivide the parameter ranges $\overline{u_i}, \overline{v_i}$ until their approximated screen-space extents e_{u_i}, e_{v_i} when evaluated with $f_{p_i}(u,v)$ no longer exceed screen-space thresholds t_u, t_v. This procedure aims to create patches of similar sizes to be forwarded to the stage to optimally utilize the GPU.
3) Rendering: All the patches which have been scheduled for rendering in one of the preceding LOD steps during Patch Subdivision are rendered in one of two manners:
   a) Tessellation-based: Patches are rendered with a tessellation-enabled graphics pipeline with either fixed or adaptive tessellation levels. Vertices generated by the tessellator are positioned according to $f_{p_i}(u,v)$.
   b) Point-based: Patches are sampled by $f_{p_i}(u,v)$ at fixed steps across parameter ranges $\overline{u_i}$ and $\overline{v_i}$. Results are projected into screen space and written to a 64-bit integer target image.

A. Patch Subdivision Stage

To determine whether and how a patch i should be subdivided, its parameter range $\overline{u_i}, \overline{v_i}$ is sampled and evaluated using its associated function $f_{p_i}(u,v)$. The evaluated samples are projected to screen space and analyzed separately to determine the subdivision pattern. In order to achieve adaptive subdivision, we sample along lines in u- and v-direction independently. Concretely, we take 8 samples in u-direction at 4 fixed v-values $v_1, \ldots, v_4$ (for a total of 32 samples), and the same for the v-direction at 4 fixed u-values $u_1, \ldots, u_4$. For each line, we compute the sum $e_{u_{ik}}$ or $e_{v_{ik}}$ respectively of screen-space extents between the projected sample points, and compare them to user-defined thresholds t_u, t_v. If for any line, $e > t$, then the patch half in which the line is located is split along the direction of the line. For example, if this happens to at least one of the lines in u-direction at v_1 or v_2, then the first half of the v-range needs to be split into two u-intervals and its parts are scheduled for re-evaluation by the subsequent LOD step. The resulting subdivision patterns are shown in Fig. 5.

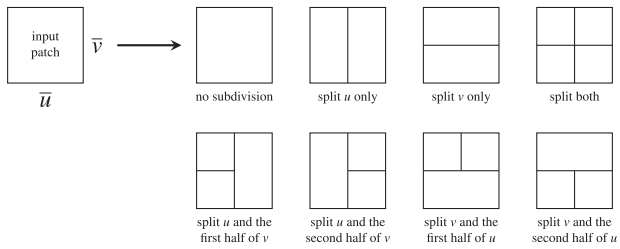
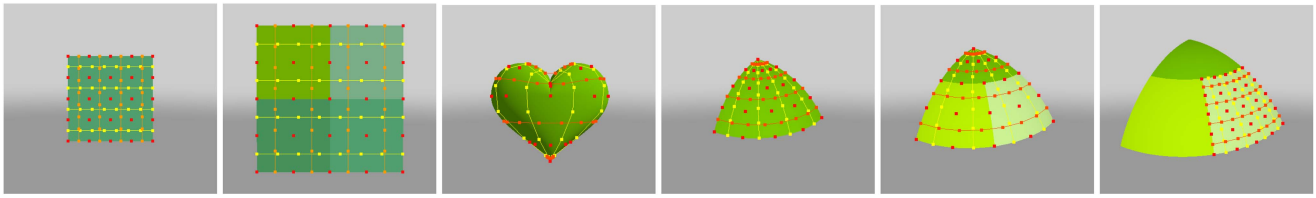


Fig. 5. During an LOD step, an input patch can either not be subdivided or split according to the seven patterns shown in this figure when being scheduled for re-evaluation.

The 32 samples correspond to a typical subgroup size on NVIDIA and Intel GPUs. After one subgroup has taken 2×32 samples for the patch splitting decisions, the same subgroup takes another 25 samples of the parametric function, which is crucial to prevent false-positive culling decisions for cases where, e.g., only a small part of a patch corner reaches into the viewing frustum. Our evaluation scheme is illustrated in Fig. 6, showing the lines used for patch evaluation in u and v directions, and the 25 extra samples.

The subdivision shader is executed 24 times, corresponding to constructing up to 24 levels of detail—sufficient for all setups analyzed in Section V. After sufficient patch subdivision has been achieved in a certain LOD step, no further LOD step will store any more patches into the "Patches to be evaluated" storage buffer and only store sufficiently subdivided patches into the "Patches to be rendered" storage buffer—how both buffers are accessed is shown in Fig. 4. All of the dispatch calls in the



(a) Patch evaluation found that this patch does not need to be split. (b) Patch evaluation found that this patch shall be split across $\bar{u}$ and $\bar{v}$. (c) Patch evaluation pattern on the example of Johi's Heart (d) Evaluation of one patch of a sphere segment (LOD step 1) (e) Non-uniform subdivision of a sphere segment (LOD step 1) (f) Evaluation of one of the now subdivided patches (LOD step 2)

Fig. 6. Each of our LOD steps analyzes a given $\overline{u_i}, \overline{v_i}$ patch by sampling $f_{p_i}(u, v)$ at 89 locations to determine if and which splits are necessary. Samples to approximate the screen-space extents across $\overline{u_i}$ are drawn in yellow. Samples to approximate the screen-space extents across $\overline{v_i}$ are drawn in orange. The 25 extra samples to help with the frustum culling decision are indicated by red dots. (c) shows why multiple evaluations across a patch are crucial in many cases: The first and fourth measurements in v-direction ($e_{v_{i\,1}}$ and $e_{v_{i\,4}}$) are close to the poles, where the parameters are very condensed. The middle two measurements ($e_{v_{i\,2}}$, $e_{v_{i\,3}}$) fall into usable positions. (d) to (f) show an example of evaluating a sphere segment. Assuming a resolution of 512×512 pixels and user-defined screen thresholds of $t_u = t_v = 256$, LOD step 1 finds that the patch in (d) does not exceed the thresholds and hence, no subdivisions are required. With the camera closer to the object in (e), the lower parts of the sphere segment exceed t_v. The patch is non-uniformly subdivided and the new parts are scheduled for evaluation in LOD step 2. (f) shows the evaluation pattern on one of these patches during LOD step 2.

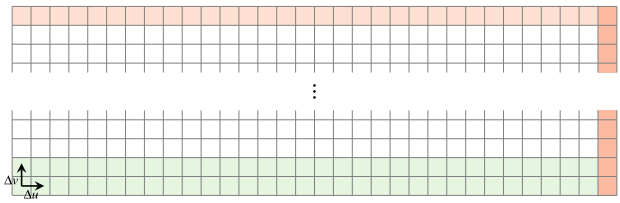


Fig. 7. With the POINT-BASED variant, parameter range $\overline{u_i}$ is processed in "columns" of 32 samples. Within such a column, subgroups sample the patch to be rendered "row-wise" in v parameter direction. A subgroup always keeps the data of two rows in registers, which are marked in green. Data of neighboring u samples is shared through subgroup operations. The samples that do not write pixels are marked in red—they take auxiliary samples which are used for tangent and bitangent calculations.

PATCH SUBDIVISION stage are indirect dispatch calls. For those LOD steps for which no patches are left to be evaluated this means that the GPU still has to process the respective dispatch commands, but will find that the number of workgroups to be processed is zero, and therefore, no compute invocations will be executed [29]. The empty dispatch calls did not incur any noticeable or measurable overhead in our tests. Alternatively, the number of dispatch calls could be adapted based on the highest required LOD step from the previous frame, if latencies of a few frames to reach the appropriate number of dispatch calls are acceptable by an application.

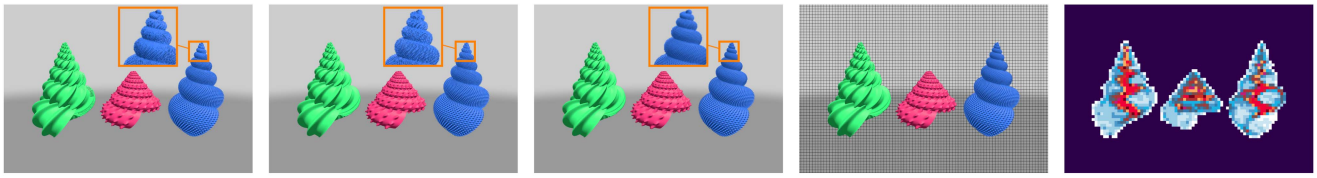
In many scenarios, the PATCH SUBDIVISION stage is very fast, often taking less than 10% of the total frame time, which is typically the case for the tests presented in Figs. 12 and 14(a) even when the camera is near the parametric objects, so that many screen pixels are covered. The exact percentage depends on the particular object and scene setup. In the test presented in Fig. 13, the LOD stage has to determine patch sizes for 358 k fiber curves (i.e., 358 k initial patches), which leads to the LOD stage taking up to 50% of the frame time for this particular rendering configuration. One key factor of the PATCH SUBDIVISION stage's low impact on frame times in our implementation is its heavy use of subgroup operations. They allow the sharing of data between compute invocations [27]. For example, to compute the screen distance between two adjacent samples, we let the two threads that computed the samples share the results via subgroup communication.

The second key factor for achieving fast rendering performance is non-uniform patch subdivision, which is illustrated in Fig. 5. There are eight possible cases when a patch is evaluated in the LOD step: If the approximated screen-space extents $e_{u_{i\,k}}, e_{v_{i\,k}}$ do not exceed thresholds t_u, t_v across $\overline{u_i}, \overline{v_i}$ of patch i, no subdivision is performed. In this case, patch i is not scheduled for re-evaluation but is instead stored in the buffer for patches to be rendered. If, however, an exceedance of t_u, t_v was detected for any $e_{u_{i\,k}}$ or $e_{v_{i\,k}}$, patch i is split according to the seven patterns shown in Fig. 5—except in the very last LOD step, where (possibly split) patches are scheduled for rendering in any case. Non-uniform patch subdivision has little impact on the performance of the PATCH SUBDIVISION stage but improves performance in the stage, as the resulting patches are more consistent in their screen-space extents. The reduction in total frame time when comparing non-uniform patch subdivision to uniform patch subdivision—that is, a constant patch subdivision scheme of one patch into four—amounts to about 15% for a simple parametrically defined sphere like shown in Fig. 3(g), and can be as high as 50% for the close-up view of a SH glyph like shown in Fig. 12(b).

B. Rendering via Hardware Tessellation

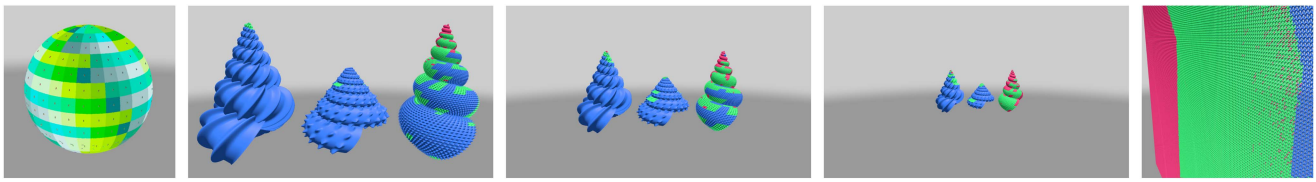
Rendering patches that have been scheduled for rendering with our TESSELLATION-BASED variant is straightforward: Each scheduled patch i is rendered as a quad with fixed inner and outer tessellation levels l [28]. Each vertex produced by the tessellator is set to the position produced by $f_{p_i}(u_x, v_y)$, where u_x and v_y refer to the interpolated parameter locations within $\overline{u_i}$ and $\overline{v_i}$ ranges, produced by the tessellator. In GLSL, these can be computed with the help of `gl_TessCoord.xy` in tessellation evaluation shaders [13]. The maximum tessellation level supported on modern GPUs is typically 64, which means that an edge (of a triangle or quad to be tessellated) is subdivided into 64 parts. A perfectly screen-aligned edge with a screen-space extent of 64 (which could be the result of using a threshold value $t = 64$) would therefore be subdivided such that there is one segment for each pixel. In many cases, choosing such fine subdivisions is counter-productive due to the resulting impact on rendering performance [11], [12]. Therefore, we propose to use fixed tessellation levels only for parametric functions which are expected to produce sub-pixel geometric detail or resort to adaptive tessellation with SS, which might capture sub-pixel detail even better and produce more uniform geometric density.

Adaptive tessellation levels are based on the actual approximated maximum screen-space extents e_{u_i}, e_{v_i} of a given patch



(a) Point rendering, aiming for 1spp; Holes appear with the green seashell on the left.
(b) Point rendering, aiming for 9spp (oversampling 3× in both, u and v directions)
(c) Point rendering into a local framebuffer with 3×3 pixels per output pixel, enabling up to 9xSS if all pixels get values assigned.
(d) There is one local framebuffer for each 16×16 region of the screen.
(e) Heatmap indicating the numbers of patches assigned to each 16×16 tile. Red regions represent higher patch counts.

Fig. 8. These images show different results of our point rendering variants. (a) and (b) are results from directly rendering into the 64-bit integer image, while (c) shows a result from first rendering into small, virtual, local framebuffers per 16×16-pixel tile (shown in (d)). Relevant patches are first assigned to each tile. The corresponding heatmap illustrating the assigned patch counts per tile is shown in (d).



(a) Spot-checks are done in the middle of patches for a region of 7×3 pixels.
(b) With the camera close to the seashell models, it can be seen that only few patches are scheduled for super-sampled rendering.
(c) With the camera moving away, more patches are deemed to require super sampling.
(d) With the camera further away, all the patches of the seashell model on the right have been found to show sub-pixel detail.
(e) Sub-pixel feature detection applied to a curtain model consisting of 60k yarn curves

Fig. 9. (a) Shows the positions of the spot-checks for sub-pixel feature detection during the PATCH SUBDIVISION stage. In (b) to (e), the effects of a two-level sub-pixel feature detection approach are visualized: The patches colored in blue have not been found to have sub-pixel features in native resolution and are consequently scheduled for rendering with noAA. The patches colored in green have been found to have sub-pixel features with respect to native resolution and have been scheduled for re-evaluation once, but haven't been found to have sub-pixel features when tested against a 8xMS framebuffer configuration. The patches colored in red have been detected to have sub-pixel features also with respect to the increased resolution of 8xMS and have consequently been scheduled for rendering with a 4xSS+8xMS rendering variant.

i. These extents are measured during PATCH SUBDIVISION by taking the maximum of the four measurements $e_{u_{ik}}$ and the maximum of the four measurements $e_{v_{ik}}$, as illustrated in Fig. 6. For example, splitting the parameter range $\overline{u_1}$ with approximated screen extents e_{u_1} based on threshold t_u can lead to split patch sizes with extents $e_{u_2} \approx \frac{t_u}{2}$ if e_{u_1} is just slightly less than t_u. Adaptive tessellation in that context means to scale a tessellation level l as follows:

$$l = \text{clamp}\left(\frac{e_{u_2} l}{\min(t_u, l)}, l_{min}, l_{max}\right).$$

The result is clamped to a minimum tessellation level l_{min} (e.g., $l_{min} = 8$) and a maximum tessellation level l_{max} (e.g., $l_{max} = 64$).

Render Targets: Since our TESSELLATION-BASED rendering variants are rendered with classical graphics pipelines, their outputs are typically rasterized into a framebuffer's color attachment. This is the only option to profit from hardware-accelerated multi-sampling (MS) and its resolve operation. MS evaluates the depth test for multiple sub-samples of one fragment, while limiting the color samples—or fragment shader invocations—to one per fragment. I.e., it enables higher depth resolution in a hardware-accelerated manner, which is especially beneficial for fine-structured geometry, such as the curtains shown in Fig. 1(d) and (e), which consist of many small yarn or fiber curves like those shown in Fig. 3(b) and (c). Rendering those with SS or MS produces much less Moiré patterns in the rendered output thanks to the higher geometry sampling density. Eight sub-samples per pixel are currently the maximum on modern NVIDIA and AMD GPUs [33]. If more evaluations are desired for a single fragment, a combination of SS and MS has to be established. For many of our evaluations presented in Section V, we utilized a configuration of 4xSS (i.e., resolution doubled in each dimension) combined with 8xMS framebuffers. In total, this leads to 32 depth evaluations, and four color evaluations for each fragment. The final rendering results are produced by first resolving MS through hardware resolve [29], and then averaging the resulting four corresponding fragments of the high-resolution color attachment into one.

We use color attachments as render targets for all our tests presented in Section V that use a TESSELLATION-BASED rendering variant, which is generally faster than the alternative: Storing the rendering output into a 64-bit integer image, written in software through atomic operations from fragment shaders. By combining depth and color values into a single 64-bit integer, as illustrated below, `atomicMin` operations ensure that parallel writes produce the correct rendering result.

To support more complex material systems, this data format could be changed from RGB color values to storing a material ID instead—the same approach as proposed Nanite, which performs shading for each material in a subsequent step by rendering a full-screen quad and using hardware depth tests to make it efficient [9].

Since our POINT-BASED rendering variants operate within compute shaders, they cannot use the hardware's depth testing functionality, which is only accessible when rendering via graphics pipelines into framebuffers with an attached depth attachment. Hence, using a 64-bit integer image and storing the rendering results via atomic operations remains the only option. In general, it is undesirable to leave holes in the rendered result of a connected parametrically defined surface. Therefore, useful rendering settings will rather try to oversample a parametric function to some degree, so that at least one sample is taken per pixel. This leads to some amount of overdraw—meaning

multiple `atomicMin` operations accessing the same fragment in the 64-bit output image—which is generally a tradeoff that must be taken to avoid holes with our implementation.

Configuration Options for Tessellation-Based Rendering: One of the most impactful configuration options for the TESSELLATION-BASED rendering variant are the screen distance thresholds t_u and t_v, which affect output patch sizes from the PATCH SUBDIVISION stage. Further major configuration options are different tessellation levels, and the sample count per pixel, which depends on the framebuffer format: noAA, MS, SS, or a combination of MS and SS. In combination with the tessellation levels, patches can be generated that lead to sub-pixel-sized triangles, or to triangles covering multiple pixels in the resulting framebuffer. Adaptive tessellation levels, as described in Section IV-B, lead to more homogeneous triangle sizes, while fixed tessellation levels can help with avoiding artifacts on patch borders and for generating patches with sub-pixel geometric detail for the stage. The latter is especially useful in combination with MS or SS framebuffers so that multiple samples can be captured per pixel, leading to super-sampled and anti-aliased results. In terms of MS, the typical maximum number of sub-samples on modern GPUs is eight [33]. SS framebuffers—possibly in combination with MS—can be used to render even more sub-samples per pixel. For example, combining a 4xSS framebuffer with an 8xMS framebuffer format, gives 32 samples for one pixel. In Section V we mainly analyze TESSELLATION-BASED with noAA, and TESSELLATION-BASED with 4xSS+8xMS, which still renders at high FPS despite SS and MS.

C. Point-Based Rendering

With the POINT-BASED variant, a patch i of range $\overline{u_i}, \overline{v_i}$ to be rendered is sampled point-wise by $f_{p_i}(u, v)$ in equidistant steps along each parameter direction. The resulting color values are stored in an image at the respective screen-space coordinates. Since color attachments are incompatible with this rendering variant, we use a 64-bit image as the target to receive the rendering output as described in Section IV-B. Samples of an input patch are produced in the following manner: The screen-space threshold parameters t_u, t_v determine the size of a patch. Given a defined workgroup size of N, we divide the parameter range $\overline{u_i}$ by the smallest multiple of N that is larger than t_u, i.e., by $\lceil \frac{t_u}{N} \rceil$, and use that as the total number of samples taken across $\overline{u_i}$ for each parameter v. Parameter range $\overline{v_i}$ is sampled in steps of size $\Delta v = \frac{v_{imax} - v_{imin}}{t_v} - \varepsilon$, where ε can be used to decrease the step size to prevent holes in the rendered output. We use $N = 31$, which is not arbitrary, but rather tied to our compute shader-based implementation: 32 subgroups—which is a typical subgroup size on NVIDIA and Intel GPUs, while it is typically 64 on AMD GPUs—sample the patch "row-wise" in $\lceil \frac{t_u}{31} \rceil$ "columns" (if we call parameter direction u a "row" and parameter direction v a "column" for the sake of tangible description) and share data among neighboring subgroups. Our approach is illustrated in Fig. 7. Each subgroup keeps the data of two rows in registers so that neighboring values in v parameter direction are available. Neighboring values in u parameter direction are read from neighboring subgroups through `subgroupShuffle` operations. The values of the neighboring samples in u and v directions are used to calculate the normal vector for the current sample u_x, v_y through the cross product of tangent vector $\mathbf{t} = f_p(u_{x+1}, v_y) - f_p(u_x, v_y)$ and bitangent vector $\mathbf{b} = f_p(u_x, v_{y+1}) - f_p(u_x, v_y)$. The normal $\mathbf{n} = \mathbf{t} \times \mathbf{b}$ is required for shading computations. We avoid sampling $f_p(u, v)$ multiple times with the same parameters. The last u column does not produce points since it does not have a neighbor with a higher subgroup index. Therefore, only 31 subgroups write pixel values, while the last subgroup only provides its data to the second to last subgroup.

Our POINT-BASED rendering variant also features an *adaptive sampling* configuration, where the number of samples in u and v directions is not calculated based on the threshold parameters t_u, t_v, but instead on the approximated screen distance extents e_{u_i}, e_{v_i}, which are measured during PATCH SUBDIVISION. The adaptive sampling configuration processes patch i in $\lceil \frac{e_{u_i}}{31} \rceil$ columns, taking row-wise steps of size $\Delta v = \frac{v_{imax} - v_{imin}}{e_{v_i}} - \varepsilon$. The adaptive sampling variant assumes $f_{p_i}(u, v)$ to distribute samples somewhat uniformly across $\overline{u_i}, \overline{v_i}$, since it takes approximately one sample across the screen space-based extents e_{u_i} and e_{v_i}. This variant often leads to much faster rendering speeds but runs the danger of producing small holes (often the size of 1×1 pixel) in the rendered output. In some cases, they can be prevented by increasing the ε values.

Anti-Aliasing with Point-Based Rendering: The POINT-BASED rendering approach described in Section IV-C writes its results into a 64-bit image, as described in Section IV-B. A potential problem with this approach is that even when a given pixel is oversampled, one of the samples—the one with the smallest depth value—overwrites all the other samples, potentially producing aliased results. In this section, we describe an alternative variant that first gathers samples in small, virtual, local framebuffers in shared memory and then resolves them in software, producing anti-aliased results. We refer to this variant by POINT-BASED$_{\text{local FB}}$, and denote the variant which writes directly into the 64-bit image as POINT-BASED$_{\text{direct}}$.

With the POINT-BASED$_{\text{local FB}}$ variant, the render target is subdivided into 16×16-pixel local framebuffers that reside in shared memory. Since it would be highly inefficient to render each patch into each of these tiles, we propose a patch-to-tile assignment step that selects all the potentially relevant patches per tile based on each patch's axis-aligned bounding box (AABB). Whenever a patch's AABB intersects a tile's bounds, it is assigned to this tile. Dividing the screen into 16×16-pixel tiles can look like shown in Fig. 8(d). Depending on the chosen screen distance threshold values t_u and t_v, a heatmap showing how many patches have been selected per tile can look like shown in Fig. 8(e) for the different seashell variations from Fig. 3(d) to (f).

The patch to tile assignment step must be scheduled after the PATCH SUBDIVISION stage and, naturally, before the stage (i.e., between stages 2 and 3 in Fig. 4). It is implemented in a compute shader with one thread per patch that has been scheduled for rendering by the PATCH SUBDIVISION stage. This thread assigns the patch to all tiles that overlap with the patch's AABB. In our tests, 16×16 pixel tiles showed favorable performance over larger tile sizes. Overall, this intermediate step has a very low impact on performance: Even in a relatively demanding scenario—namely 60 k yarn curves, 358 k fiber curves, and six seashell models, covering the entire screen so that all tiles get patches assigned—this step does not take longer than 0.14 ms for a resolution of 1920×1080 on an RTX 3070.

Local framebuffers have the same format as the global render target: 64-bit integer as described in Section IV-B. For 2×2 super-sampling, 32×32-pixel local framebuffers must be

allocated; for 3×3 super-sampling they are sized 48×48-pixel accordingly. For each tile, these steps are performed:

1) Initialize each pixel in the local framebuffer to the clear color.
2) Point-render each selected patch into the local framebuffer in the same manner as described in Section IV-C.
   - Discard points outside of the local framebuffer bounds
   - `atomicMin` operations now access shared memory
3) Average each 2×2/3×3-pixel region in the local framebuffer, producing one output pixel. Write the latter into the global render target.
   - Regard only entries that have been set, i.e., are different from the clear value

The main benefit of POINT-BASED$_{\text{local FB}}$ are super-sampled results, which are not possible with our POINT-BASED$_{\text{direct}}$ variant. The effect can be observed Fig. 8: While just increasing the sample count succeeds in filling some holes, POINT-BASED$_{\text{direct}}$ is unable to get rid of Moiré patterns, which can be seen when comparing Fig. 8(b) to (a). Only POINT-BASED$_{\text{local FB}}$ is able to reduce them, as can be seen in Fig. 8(c)—most notably on the blue seashell on the right, which has a lot of tiny geometric details on the surface.

Configuration Options for Point-Based Rendering: The screen distance thresholds t_u and t_v have a big effect for POINT-BASED rendering variants as well, since they scale the number of compute workgroups to be spawned for pixel filling. Additional major configuration options are varying the sample density per patch, and the choice of using POINT-BASED$_{\text{direct}}$ or POINT-BASED$_{\text{local FB}}$. The main differences between these two are that the former generally leads to better rendering performance, while only the latter is able to produce anti-aliased rendering results. Varying t_u and t_v typically has an impact on performance: While decreasing them leads to higher computational load in the PATCH SUBDIVISION stage (more subdivisions, more LOD steps), it often has a positive impact on rendering performance since the workload per patch decreases accordingly so that the overall workload can potentially be distributed better across the GPU. Increasing the sample density per patch also affects the computational workload during rendering and leads to more memory transfers due to overdraw, but is often necessary to fill holes in the rendered output. For our evaluations presented in Section V we mainly focus on POINT-BASED$_{\text{local FB}}$ with 4xSS and leave out POINT-BASED$_{\text{direct}}$, since it is unable to produce anti-aliased rendering results.

D. Hybrid Rendering

Depending on the underlying parametric function of a parametric object, super-sampled rendering might be required to accurately render a surface, while for smooth surfaces noAA configurations are sufficient. In terms of performance it would be desirable to render with only those patches with super sampling that require it. In this section, we describe a hybrid rendering variant that performs spot checks, which sample a small part of a parametric surface in order to determine whether a surface has sub-pixel detail and adapts the rendering variant accordingly. We chose this approach because such spot checks are a very general approach, supporting a wide variety of parametric functions. Whether or not an object shows sub-pixel features not only depends on the parametric function, but also on the distance to the camera and the framebuffer resolution. The curtains consisting of several thousands of yarn or fiber curves shown in Fig. 1(d) and (e) would be an example of parametric objects leading to sub-pixel detail. Some of the seashell variants (shown in Fig. 3(d) to (f)) produce sub-pixel detail on the surface—but maybe not everywhere: Being self-repeating shapes, their geometric features get scaled smaller towards their top regions, eventually leading to sub-pixel geometric detail at some point. Such a situation is shown in Fig. 8(a) to (c): This is most obvious with the blue seashell model on the right, which has small bumps across its surface. Pixel-level sampling in Fig. 8(a) and (b) leads to noticeable aliasing, while sub-pixel-level sampling and averaging leads to an anti-aliased result in Fig. 8(c).

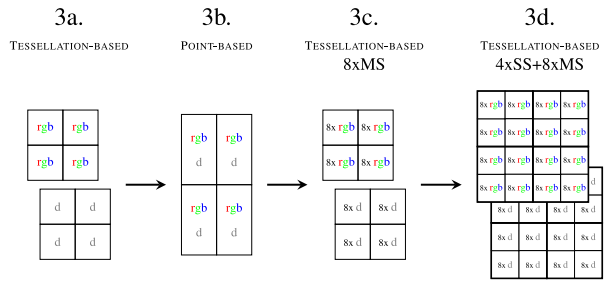


Fig. 10. Different rendering variants require different render targets. TESSELLATION-BASED performs best in conjunction with framebuffers with color and depth attachments, and they enable MS, providing multiple sub-samples per pixel (e.g., eight depth and color samples). POINT-BASED rendering targets 64-bit integer images which store both color and depth in one pixel as described in Section IV-B. The order of rendering variants indicated in this figure is important to not lose any detail: Render variants targeting lower resolution targets are rendered first, the highest resolution comes last. Depth and color data must be transferred between the different render target formats.

If our proposed spot-check detects sub-pixel variations, the following actions are performed:

1) Do not schedule the patch in question for rendering.
2) Mark the patch to indicate that it has sub-pixel details shall be rendered in an anti-aliased manner.
3) Re-schedule the patch for evaluation in the next LOD step.

At each of the 21 locations in the middle of a patch, our implementation of detecting subpixel features takes nine additional samples from all combinations of three different u and v offsets at $\frac{1}{2}$, $\frac{1}{3}$, $\frac{1}{5}$, uses them to compute normals, and compares them with the bi-linearly interpolated normals from a ≈1-pixel-sized region. The sub-sample offsets were chosen to be fractions of the first three prime numbers, so that they capture a wide range of different frequencies of the parametric object's surface. The effect of our hybrid rendering approach can be observed in Fig. 9: It shows a three-tier hybrid approach, first evaluating patches against the native resolution with noAA, then against an 8xMS increased resolution, and finally rendering the patch with 4xSS+8xMS if evaluation at the 8xMS-level still shows sub-pixel variations.

Care must be taken when creating an application that shall be able to dynamically use different rendering variants since they might require different render target formats and the application needs to switch between them during a frame. While TESSELLATION-BASED render variants preferably render into a framebuffer's color attachment (as described in Section IV-B), POINT-BASED render variants write into 64-bit integer images (as described in Section IV-C). Depth and color information must be transferred between these two formats. Compute shaders must be used to read and extract color and depth values from 64-bit integer images, or to combine and write them to 64-bit integer images. For MS framebuffers, transferring color and depth values from previous rendering variants into all the relevant samples must be ensured as well. Fig. 10 shows a setup

supporting different render variants and indicates the order of rendering them within the stage: It is important to start with the render variants that use the coarsest render targets and progress to render targets that provide finer detail. Failing to follow this order will have the effect of losing previously rendered detail. Keeping this order is essential to get correct rendering results at the geometry borders since framebuffers of higher resolution capture more depth samples. In our implementation, we transfer the the previously rendered noAA results into an 8xMS framebuffer by rendering a full-screen quad. Writing color and depth values in a fragment shader ensures that all the sub-samples get the already rendered color and depth values assigned. Also, color and depth transfer from the 8xMS framebuffer into the 4xSS+8xMS framebuffer attachments is performed through a full-screen quad for the same reason. We show in Section V that such a hybrid rendering setup can help to keep frame rates constant, but it must be noted that our implementation also incurs some significant overhead from transferring the data between the different render targets as shown in the following table:

	3a. → 3b.	**3b. → 3c.**	**3c. → 3d.**	**Total**
Values written	2.1 M	16.6 M	66.4 M	85.0 M
Bytes transferred	17 MB	133 MB	530 MB	680 MB
Time	0.05 ms	0.11 ms	0.78 ms	0.94 ms

The first line refers to the number of color or depth values that have to be written by the render output units for a resolution of 1920×1080. Color values of 8 bits per channel + 32-bit depth values amount to the stated bytes transferred. The milliseconds have been measured on an NVIDIA RTX 3070 using GPU timestamp queries, without rendering any patches. It shall be noted that timestamp queries synchronize between measurements and that in an actual usage scenario, the actual data transfer overhead might be less severe since a GPU might be able to schedule other work in parallel. However, the overhead is not insignificant—especially the data transfer of the 8xMS data into the 4xSS+8xMS framebuffer.

V. Results

We evaluate our method and its rendering variants based on the following parametric functions, which have different characteristics and therefore pose different challenges to a rendering method: SH glyphs, parametric plain-knit yarn curves, and parametric seashells.

One challenge when rendering SH glyphs—especially such of higher order—is that each sample is computationally expensive. Furthermore, parameters are distorted very non-uniformly across the entire parameter range. These partly extreme distortions can lead to parameter patches being extremely stretched. Our method's Patch Subdivision stage identifies such cases and ensures similar geometric density during rendering across all patches. The challenge with rendering plain-knit yarn curves is the vast amount that is required for a typical scene. Knit yarns and parametric seashell models produce very small-scale surface features, leading to sub-pixel geometric detail for most of our test setups.

All results were gathered after a GPU warm-up phase of two seconds from multiple camera positions. The camera is positioned at a certain distance to the object(s) for the first

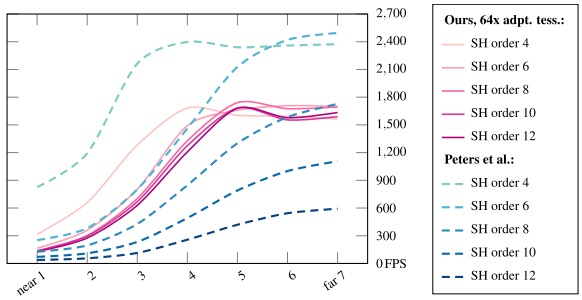


Fig. 11. These diagrams shows the performance impact of using different SH orders for rendering a single SH glyph, measured on an NVIDIA RTX 3070. $t_u = t_v = 84$, noAA, adaptive tessellation, and $l_{max} = 64$ have been used as the configuration for our method.

measurement and moves either away or closer with every further measurement. Each measurement result (e.g., FPS, number of pixels written, number of render patches) represents the averaged data over a five-second period, during which the camera moved around the center of the object or dataset in one full circle, requiring the Patch Subdivision stage to generate and forward a different set of patches to the stage every frame.

SH Glyph Rendering and Comparison with Previous Work: Spherical harmonics are mathematical functions defined on the surface of a sphere and are parameterized using spherical coordinates (θ, φ) with $\theta \in [0, \pi]$ and $\varphi \in [0, 2\pi]$. These functions result from a linear combination of a set of orthonormal basis functions, an important property that makes them useful in a wide range of fields. They are described via band index ℓ and parameter m. ℓ also states the *order* of an SH. Higher orders allow the representation of higher frequencies but also involve increased computational complexity, making them expensive to evaluate and challenging to visualize. Thanks to their spherical parametrization, one way of representing them is by using a sphere with the distance from its surface to its center set to the result of the evaluated SH at the corresponding location (θ, φ), which aligns them with the definition of parametric functions.

The SH glyphs we use represent measurements from a so-called high angular resolution diffusion imaging (HARDI) [30] dataset of a brain scan [8], captured via dMRI and shown in Fig. 1(b). We compare our method with the state-of-the-art ray tracing-based SH glyph rendering method by Peters et al. [21]: Fig. 11 shows the effect of using different SH orders for rendering. Our method shows remarkably consistent performance for varying SH orders, while the ray-traced approach suffers from a strongly decreasing performance with each SH order increase due to increased complexity from root finding. While it renders more than 2000 FPS on an NVIDIA RTX 3070 for SH order 2, our method outperforms it for SH orders 8 and higher in single SH glyph rendering and already for SH orders 4 and higher for the brain scan dataset, for which we provide a diagram in our previous work [32]. Since higher SH orders reveal more detail about the measured data they represent, we have focused further tests on SH order 12: For single glyph rendering of SH order 12, Fig. 12(f) shows that our method shows better performance across all GPUs. The Tessellation-based variant is optimal for rendering SH glyphs due to their smooth surface. Our recommended configurations for close-up and distant rendering are described in Fig. 12(f) and (g), respectively, and produce almost pixel-perfect geometric detail. With fixed

tessellation factors, our method still outperforms the method by Peters et al. across all GPUs, but does not improve rendering quality despite the higher geometric detail. Our method avoids the artifacts shown in Fig. 12(a), which emerge due to numerical issues [21]. However, it also introduces small, less noticeable artifacts near the poles of the scaled sphere. An example is shown in Fig. 17(a) and (b).

Our PATCH SUBDIVISION stage creates a suitable number of patches to be rendered from a single initial patch. Fig. 12(f) shows that up to 30 k patches are created if the camera is close to the SH glyph, while only a few hundred are required when the camera is further away. With super-sampled rendering, the number of patches increases according to the effectively higher resolution, as shown in Fig. 12(g). In some cases it might be beneficial to already subdivide the initial patch into a constant number of patches, e.g., 2×2, in order to evaluate more initial samples of the parametric object with the pattern shown in Fig. 6(a). Sticking to one single initial patch might lead to premature frustum culling for very distant camera positions.

Structural similarity index measure (SSIM), peak signal-to-noise ratio (PSNR), and root mean squared error (RMSE) values of our results and those by Peters et al. with respect to the reference rendered in 16xSS+8xMS are listed in the following table:

	SSIM	PSNR	RMSE
Single glyph, Peters et al.	0.9985	36.83 dB	3.67
Single glyph, ours noAA	0.9996	43.48 dB	1.71
Brain scan, Peters et al.	0.9776	28.47 dB	9.62
Brain scan, ours noAA	0.9665	26.13 dB	12.59
Brain scan, ours 4xSS+8xMS	0.9890	32.11 dB	6.32

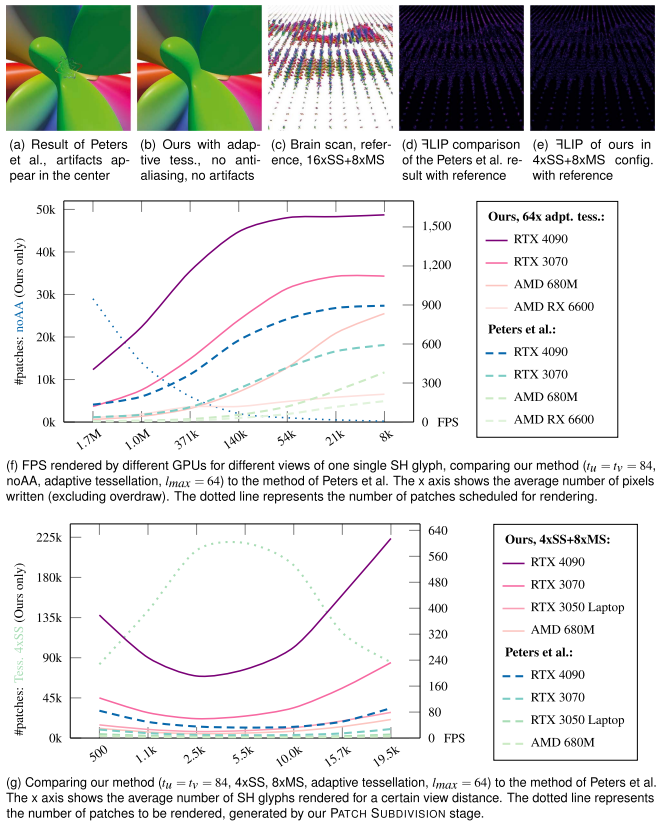


Fig. 12. Comparing (a) with (b) reveals that our method avoids the artifacts near the center of the SH glyph. (f) and (g) show the performance of our variants in comparison to the method by Peters et al. For the first measurement, the camera starts at the view producing (a) and (b) and moves away for subsequent measurements. (c) shows a part of our reference image of the brain scan dataset, rendered with 16xSS+8xMS. (d) and (e) show the results of the ꟻLIP image difference evaluator for images produced with our method and with the method by Peters et al., respectively.

While our method produces better SSIM, PSNR, and RMSE numbers when viewing a SH glyph from up close (single glyph) even in the non-anti aliased configuration, our noAA configuration produces slightly worse numbers for the brain scan dataset, when the glyphs are rendered from a distance. However, our method provides plenty of headroom for rendering in 4xSS and 8xMS. This configuration not only produces superior quality—as shown by SSIM, PSNR, and RMSE numbers and in Fig. 12(d) and (e), which were generated with NVIDIA's difference image evaluator ꟻLIP [3]—but also significantly outperforms the method by Peters et al. (in non-anti aliased configuration) across all tested GPUs. We provide further evaluations in our previous work [32].

Plain-Knit Yarn: Gröller et al. [7] provided an early parametric description of knitwear, but we use the more recent parametric plain-knit yarn curves described by Crane [4]. Its fundamental shapes of yarn curves and fiber curves are shown in Fig. 3(b) and (c), respectively. We use an extruded version of them for our evaluations by taking the yarn direction ("tangent") as parameter u, and constructing circles around it via parameter v. We get an orthogonal vector ("normal") to the tangent and rotate it using Rodrigues' rotation formula [23] around the tangent by an angle θ, which is parameter v in the range $[0, 2\pi)$. The source code repository containing the code for Crane's plain-knit yarn curves lists several implementations, none of which is able to render a large number of yarn curves in real time with good rendering quality—which our method enables for at least 358 k curves, as our results in Fig. 13(a) show. We show the performance comparisons of three different variants of our method on the example of a blue curtain composed of 60 k yarn curves or 358 k fiber curves in Fig. 13. The configurations compared are a TESSELLATION-BASED variant with adaptive tessellation, $t_u = t_v = 62$, $l_{max} = 19$, and no anti-aliasing (noAA), the same configuration with 4xSS and 8xMS, and a POINT-BASED variant with 4xSS. The noAA configuration suffers from severe aliasing artifacts such as Moiré patterns, which emerge due to the high numbers of fiber curves and the sub-pixel geometric detail produced by them. The artifacts get more severe with increasing camera distances. The configurations with SS produce visually satisfying results like those shown in Fig. 1(d) and (e). In most situations, the TESSELLATION-BASED rendering variant shows better performance numbers, but there's also a noticeable sweet spot for the POINT-BASED variant, where it outperforms the former—namely when rendering a limited number of yarn curves or fiber curves at a relatively small scale. However, our implementation of the POINT-BASED variant suffers from the problem that it produces a rendering result equivalent to that of conservative rasterization, which can impair visual quality, especially when rendering thin geometry.

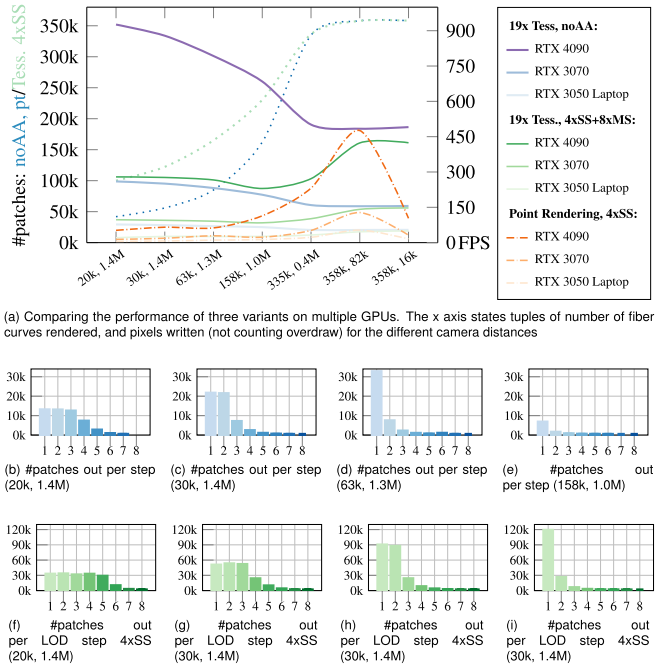


Fig. 13. (a) shows performance results and numbers of patches to be rendered for 358 k fiber curves, arranged as a curtain as shown in Fig. 1(d) (rendered without the "Sponza" 3D model for these evaluations). (b) to (e) show the numbers of patches generated by the PATCH SUBDIVISION stage for the noAA and point rendering variants during each LOD step, while (f) to (i) show the corresponding numbers for the 4xSS configuration.

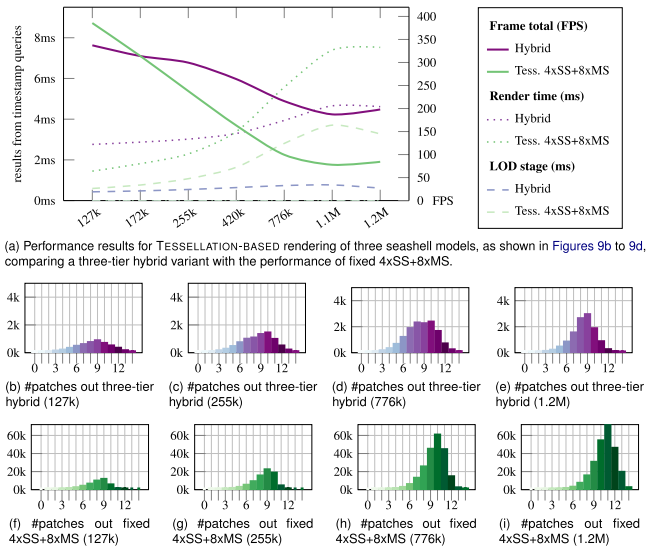


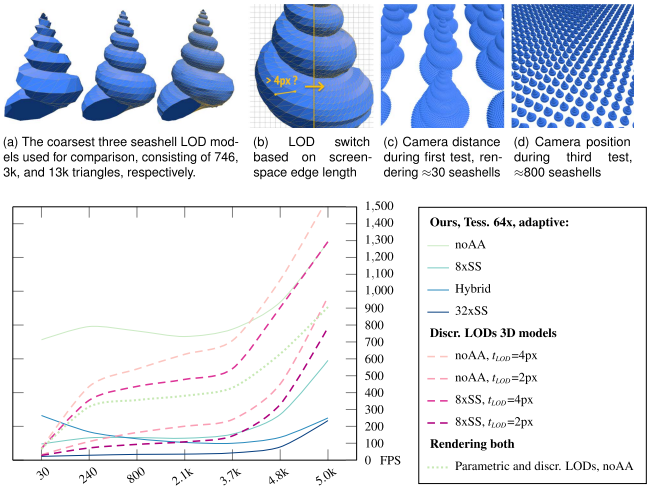
Fig. 14. These diagrams show performance results achieved with a three-tier hybrid rendering technique, which dynamically schedules patches for rendering with Tess. noAA, Tess. 8xMS, or Tess 4xSS+8xMS rendering variants The x axis in (a) shows the number of pixels covered on the screen as the camera moves from a distance towards the parametric objects. The dashed lines show the milliseconds for rendering all the patches, and the dotted lines represent the milliseconds how long all the LOD steps of the PATCH SUBDIVISION took. (b) to (e) put the numbers of output patches after each LOD step in perspective to (f) to (i), showing an effect of the hybrid technique.

In contrast to the results of single SH glyph rendering, where all the geometry is raised from one single initial patch, Fig. 13(b) to (i) show different subdivision characteristics since PATCH INITIALIZATION already creates 358 k patches. Many of them are culled for near-camera positions. Hence, only ≈12 k remain after the first LOD step in Fig. 13(b), all of which need to be subdivided due to the camera distance and screen resolution. As the camera moves away, more fiber curves become visible but relatively fewer patches need to be subdivided. The SS variants in Fig. 13(f) to (i) generally require more patch subdivisions due to the higher effective resolution.

Parametric Seashells, Hybrid Method, and Discrete LODs: The parametric seashell model follows roughly the construction guidelines by Wilson [34], which allows creating a virtually unlimited number of seashell variants by altering constant parameters. Some variants are shown in Fig. 3(d) to (f). Achieving satisfying rendering results for the variant shown in Fig. 3(f) is challenging since its parametric function creates many tiny bumps on the surface, leading to sub-pixel geometric detail when viewed from a distance or rendered in low resolution.

The subpixel features produced by this seashell variant make it a great test candidate for our hybrid rendering approach. Its performance results are presented in Fig. 14, showing that a hybrid approach can lead to much more stable frame rates as PATCH SUBDIVISION selects a suitable render method which ultimately keeps render load more consistent. In contrast, rendering the whole scene with 4xSS+8xMS shows varying frame rates and leads to severe FPS drops when the camera is near to the parametric objects. Notably, the LOD stage takes less time for the hybrid approach, which is a result of fewer patch subdivisions being required with the hybrid approach. The fixed 4xSS+8xMS configuration unconditionally aims for much smaller patch sizes. This behavior can be observed well in Fig. 14(h) and (i): As the camera is close to the parametric objects, they get subdivided into relatively large numbers of small patches. In contrast, the hybrid approach determines that super-sampled rendering is not required for many patches and hence, stops subdividing earlier, as can be seen in Fig. 14(d) and (e), scheduling much fewer patches for rendering. Overall, this leads to a much less pronounced FPS drop when the camera moves closer to the parametric seashells. However, the chart also shows the fixed overhead of a hybrid rendering setup, as described in Section IV-D. This leads to the hybrid rendering technique not being able to outperform the fixed 4xSS+8xMS configuration in all cases. Evaluations with yarn curves are provided in our supplementary material.

We compare the performance of our TESSELLATION-BASED configurations to another baseline, namely to the conventional approach of discrete LODs [15]. It selects one of several pre-computed 3D meshes of different geometric detail. We use a total of eight different resolutions of the seashell model shown in Figs. 1(c) and 3(f). The first three are shown in Fig. 15(a), each level quadrupling the previous level's number of triangles. Level eight suffices for pixel-perfect geometric precision for camera distances as in Fig. 15(c) or further away. We refrain from including more detailed LOD models since level nine would require 2.5 GB of video memory (for uncompressed vertex data), with the FBX file being ≈ 900 MB in size, possibly leading to long loading times. The combined rendering performance of rendering both, parametric seashell objects and the corresponding discrete LOD models, in the same frame is



(a) The coarsest three seashell LOD models used for comparison, consisting of 746, 3k, and 13k triangles, respectively.

(b) LOD switch based on screen-space edge length

(c) Camera distance during first test, rendering ≈30 seashells

(d) Camera position during third test, ≈800 seashells

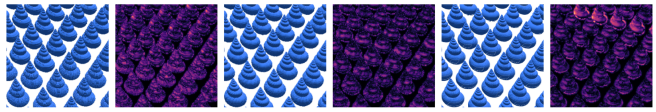
(e) FPS rendered with an RTX 4090 for different camera distances of 5k seashell objects. Our results are rendered with the TESSELLATION-BASED rendering variant with adaptive sampling and up to 64x tessellation, and $t_H = t_V = 84$ PATCH SUBDIVISION. Discrete LOD results render appropriate 3D models of different detail based on t_{LOD} values. The x axis shows the average number of seashells rendered. The dotted line represents results from rendering 10k seashell models—half of them as parametric objects, the other half as 3D models.

Fig. 15. The plot in (e) shows performance comparisons between rendering parametrically-defined seashells to a classical LOD system that switches between different discrete 3D model versions of the seashell. The coarsest three 3D models are shown in (a). (b) shows that the decision of switching to a more detailed LOD is made based on edge lengths in screen space: t_{LOD}=4px in this example. (c) and (d) show the initial and third (of seven) camera distances during our tests.

better than rendering both types sequentially in our tests. This is represented by the dotted line in Fig. 15(e) and shows rendering performance improvements between 2% and 29% across the different measurements from different camera distances. Rendering parametric functions mainly demands computational load from shading units, while rendering discrete LOD models is typically limited by memory transfer—the different GPU utilization characteristics give leeway for performance optimization in an application. The following table lists SSIM, PSNR, and RMSE values of different rendering configurations for the performance evaluations in Fig. 15. References were rendered in 32xSS using parametric seashells.

	Near distance			Far distance		
	SSIM	PSNR	RMSE	SSIM	PSNR	RMSE
Discr. LODs, noAA, t_{LOD}=4px	0.9909	29.41 dB	8.63	0.9592	23.41 dB	17.22
Discr. LODs, noAA, t_{LOD}=2px	0.9937	30.88 dB	7.29	0.9604	23.26 dB	17.52
Discr. LODs, 8xSS, t_{LOD}=4px	0.9936	30.97 dB	7.21	0.9675	24.43 dB	15.32
Discr. LODs, 8xSS, t_{LOD}=2px	0.9956	32.31 dB	6.18	0.9749	25.28 dB	13.88
Ours, noAA	0.9933	30.70 dB	7.44	0.9709	24.92 dB	14.48
Ours, 8xSS	0.9961	32.99 dB	5.72	0.9789	26.26 dB	12.40
Ours, Hybrid	0.9944	31.49 dB	6.79	0.9675	23.91 dB	16.25

The decision of when to switch to a more detailed LOD is made based on a value t_{LOD}, which refers to the longest edge's length projected into screen space as shown in Fig. 15(b). Values between 4px and 2px lead to approximately the same geometric density as rendering the corresponding parametric object through our technique. Comparing noAA configurations shows favorable SSIM, PSNR, and RMSE numbers for our method in almost all cases. Additionally, the discrete LOD switches are much more noticeable than the continuous changes of our technique, which is performed on patch level, not on model



(a) Discrete LODs, t_{LOD}=2px (b) ꟻLIP of Figure 16a (c) Ours, tess.-based, noAA (d) ꟻLIP of Figure 16c (e) Ours, tess., hybrid method (f) ꟻLIP of Figure 16c

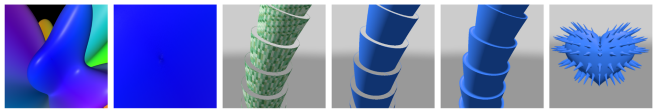
Fig. 16. Some selected results from rendering 5 k seashells, along with their ꟻLIP image comparisons to a parametric 32xSS reference

level. 8xSS configurations generally lead to greatly improved visual quality. Our technique shows favorable results over using discrete LOD models, like noticeably higher rendering speeds for close-up views. Fig. 16 shows some selected evaluations, revealing different patterns of visual artifacts in the ꟻLIP image comparisons. Our hybrid rendering technique again manages to keep frame times relatively consistent. Unfortunately, it produces noticeable seams between some neighboring patches rendered with different configurations, which we were unable to fully resolve.

Point-based Rendering Results and Performance: In our tests about seashell rendering, configurations with SS and MS produce much more satisfying visual results. Performance of POINT-BASED rendering generally trails behind TESSELLATION-BASED configurations. Performance results are included in our previous work [32] and in our supplemental material. The POINT-BASED$_{\text{local FB}}$ variant writes its results into small, virtual, super-sampled local framebuffers first. The overdraw with this variant is absorbed on local framebuffer-level (in contrast to the global framebuffer with other configurations). The single resulting color value after the software-resolve operation leads to only one single `atomicMin` operation into the global 64-bit integer target. While it would be reasonable to expect some performance improvement from this approach—due to the atomic operations targeting shared memory instead of global memory—this did not show in our measurements. We attribute the performance decline of POINT-BASED$_{\text{local FB}}$ over configurations without local framebuffers to increased computational load within the point rendering shaders, requiring the additional clearing and resolve steps (as described in Section IV-C). The additionally required patch-to-tile assignment step never took more than 0.4 ms in our measurements.

VI. CONCLUSION AND FUTURE WORK

We have presented a general method for sampling and rendering parametric functions in real time. Although a general method, it outperforms recent solutions for rendering individual, high-detail SH glyphs and large glyph datasets in terms of rendering speed and quality. For large datasets, our noAA configuration achieves higher frame rates on a *mobile* AMD 680 M GPU than the method by Peters et al. [21] on a dedicated *desktop* NVIDIA RTX 4090 GPU. Even at the higher-quality 4xSS and 8xMS configurations, our method running on the much weaker RTX 3050 Laptop is competitive with theirs on an RTX 4090. We found non-uniform patch subdivision to be a key factor for our method's good rendering performance, contrary to the recommendation of Eisenacher et al. [5] to always split patches 1:4. Additional experiments for organic shapes and fabric yield several hundred FPS while avoiding prominent artifacts. A parametric object description rendered with our technique not only produces generally favorable visual results



(a) Artifacts at SH poles (b) Closeup of Figure 17a (c) Jump discontinuities (d) Point rendering (e) Rendered with triangles (f) Johi´s Heart with small spikes

Fig. 17. Current limitations of our method.

when compared to a conventional discrete LOD approach, but also has the potential to save hundreds of megabytes of video memory and to reduce the initial loading time significantly. Our hybrid rendering approach is a tool that can help to keep frame rates stable by super sampling only those patches that have been found to show subpixel variations.

There are multiple avenues for future work: Currently, our PATCH SUBDIVISION ignores the concept of holes and subdivides patches strongly near discontinuities as illustrated in Fig. 17(c). An early exit criterion could improve performance in such cases. While the sampling pattern shown in Fig. 6 works well even for challenging objects like shown in Fig. 17(f), it might fail to capture extremely fine geometric details. Future work might investigate more dense and irregular sampling during the PATCH SUBDIVISION stage. With our proposed approach, a fixed patch subdivision during PATCH INITIALIZATION can help to better evaluate challenging surfaces. Another area of improvement would be gaps that are left open with POINT-BASED rendering, while TESSELLATION-BASED closes them with the triangles created by the tessellator, as shown in Fig. 17(d) and (e). Besides the proposed screen distance-based metric for deciding whether to subdivide a patch, we hope to explore further metrics based, e.g., on the derivatives of $f_p(u, v)$. We also plan to investigate relevant performance factors for our POINT-BASED variant. A more optimized implementation might be able to increase the performance of POINT-BASED$_{\text{direct}}$ and POINT-BASED$_{\text{local FB}}$ rendering variants. This would make the latter perfectly suitable for the super sampling path in a hybrid rendering approach, avoiding the expensive data transfers between different framebuffer formats. It is thinkable to extend our hybrid rendering approach by rendering not only parametrically-described objects but also discrete LOD models, which might be beneficial for controlling overall rendering performance and utilizing a GPU's different hardware units in a more balanced manner. *Work Graphs* [2] are a new feature of modern GPUs and would allow the number of dispatch calls for the PATCH SUBDIVISION stage to be determined and scheduled within a frame, eliminating the latency of an adaptive approach like described in Section IV-A.

Since PATCH SUBDIVISION is fast enough to run every frame, our method is well-suited for use with animated objects. In this work we have only scratched the surface. We believe that our general method can unlock a wide range of elaborate, animated parametric functions, enable glyph-based visualization of time-varying medical data with high frame rates, or allow for smooth morphing between data sets in real time.

ACKNOWLEDGMENTS

The authors thank Christoph Peters for his detailed guidelines for getting the implementation of their SH glyph rendering method [21] right and reasonably optimized, so that fair comparisons could be established. The authors thank him and Tark Patel for providing the data set they have used in their paper and pointing us to its origin [8]. The "Sponza" 3D model shown in Fig. 1(d) and (e) is based on a modified version by Morgan McGuire of the original version by Frank Meinl, Crytek. This research has been funded by WWTF (project *ICT22-055 - Instant Visualization and Interaction for Large Point Clouds*) and Netidee (project *Math2Model, project call #18, project id: 6890*). The authors acknowledge TU Wien Bibliothek for financial support through its Open Access Funding Programme.

REFERENCES

[1] A. Abbasloo, V. Wiens, M. Hermann, and T. Schultz, "Visualizing tensor normal distributions at multiple levels of detail," *IEEE Trans. Vis. Comput. Graphics*, vol. 22, no. 1, pp. 975–984, Jan. 2016.

[2] Advanced Micro Devices, Inc. Announcing, "GPU work graphs in vulkan," 2023. Accessed: Apr. 08 2024. [Online]. Available: https://gpuopen.com/gpu-work-graphs-in-vulkan

[3] P. Andersson, J. Nilsson, T. Akenine-Möller, M. Oskarsson, K. Åström, and M. D. Fairchild, "FLIP: A difference evaluator for alternating images," *Proc. ACM Comput. Graph. Interactive Techn.*, vol. 3, no. 2, pp. 15:1–15:23, 2020, doi: 10.1145/3406183.

[4] K. Crane, "A simple parametric model of plain-knit yarns," 2023. Accessed: Feb. 29 2024. [Online]. Available: https://github.com/keenancrane/plain-knit-yarn

[5] C. Eisenacher, Q. Meyer, and C. Loop, "Real-time view-dependent rendering of parametric surfaces," in *Proc. Symp. Interactive 3D Graph. Games*, 2009, pp. 137–143.

[6] Epic Games, Inc, "Nanite virtualized geometry in unreal engine| Unreal engine 5.0 documentation," 2024. Accessed: Jan. 01 2024. [Online]. Available: https://docs.unrealengine.com/5.0/en-US/nanite-virtualized-geometry-in-unreal-engine/

[7] E. Gröller, R. T. Rau, and W. Straßer, "Modeling and visualization of knitwear," *IEEE Trans. Vis. Comput. Graphics*, vol. 1, no. 4, pp. 302–310, Dec. 1995.

[8] S. HashemizadehKolowri, R.-R. Chen, G. Adluru, and E. V. R. DiBella, "Jointly estimating parametric maps of multiple diffusion models from undersampled q-space data: A comparison of three deep learning approaches," *Magn. Reson. Med.*, vol. 87, no. 6, pp. 2957–2971, 2022, doi: 10.1002/mrm.29162.

[9] B. Karis, R. Stubbe, and G. Wihlidal, "A deep dive into nanite virtualized geometry," in *Proc. ACM SIGGRAPH 2021 Courses Adv. Real-Time Rendering Games*, 2021, Part 1, pp. 1–155.

[10] M. J. Keeter, "Massively parallel rendering of complex closed-form implicit surfaces," *ACM Trans. Graph.*, vol. 39, no. 4, 2020, Art. no. 141.

[11] B. Kerbl, L. Horváth, D. Cornel, and M. Wimmer, "An improved triangle encoding scheme for cached tessellation," in *Proc. 43rd Annu. Conf. Eur. Assoc. Comput. Graph.*, 2022, pp. 53–56, doi: 10.2312/egs.20221031.

[12] J. Khoury, J. Dupuy, C. Riccio, and W. Engel, "Adaptive GPU tessellation with compute shaders," *GPU Zen: Adv. Rendering Techn.*, 2, pp. 3–17.

[13] Khronos Group, "gl_TessCoords - OpenGL 4 reference pages," 2011-2014. Accessed: Mar. 08 2024. [Online]. Available: https://registry.khronos.org/OpenGL-Refpages/gl4/html/gl_TessCoord.xhtml

[14] B. Kuth, M. Oberberger, M. Chajdas, and Q. Meyer, "Edge-friend: Fast and deterministic Catmull-Clark subdivision surfaces," *Comput. Graph. Forum*, vol. 42, no. 8, 2023, Art. no. e14863, doi: 10.1111/cgf.14863.

[15] D. Luebke, M. Reddy, J. D. Cohen, A. Varshney, B. Watson, and R. Huebner, *Level of Detail for 3D Graphics*. San Francisco, CA, USA: Morgan Kaufmann, 2002.

[16] T. Möller and R. Yagel, "Efficient rasterization of implicit functions," Tech. Rep., 1995. [Online]. Available: https://vda.cs.univie.ac.at/research/publications/publication/5055/

[17] M. Nießner and C. Loop, "Analytic displacement mapping using hardware tessellation," *ACM Trans. Graph.*, vol. 32, no. 3, 2013, Art. no. 26.

[18] NVIDIA Corporation, "NVIDIA turing GPU architecture," *Whitepaper*, 2018. [Online]. Available: https://www.nvidia.com/content/dam/en-zz/Solutions/designvisualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf

[19] S. Osher and R. Fedkiw, "Signed distance functions," in *Level Set Methods and Dynamic Implicit Surfaces*. New York, NY, USA: Springer, 2003, pp. 17–22, doi: 10.1007/0-387-22746-6_2.

[20] A. Patney, M. Ebeida, and J. Owens, "Parallel view-dependent tessellation of Catmull–Clark subdivision surfaces," in *Proc. Conf. High Perform. Graph.*, 2009, pp. 99–108, doi: 10.1145/1572769.1572785.

[21] C. Peters, T. Patel, W. Usher, and C. R. Johnson, "Ray tracing spherical harmonics glyphs," in *Proc. 28th Int. Symp. Vis. Model. Vis.*, 2023, pp. 21–31, doi: 10.2312/vmv.20231223.

[22] C. Poirier, M. Descoteaux, and G. Gilet, "Accelerating geometry-based spherical harmonics glyphs rendering for dMRI using modern OpenGL," in *Proc. 12th Int. Workshop Comput. Diffusion MRI*, 2021, pp. 144–155.
[23] O. Rodrigues, "Des lois géométriques qui régissent les déplacements d'un système solide dans l'espace, et de la variation des coordonnées provenant de ces déplacements considérés indépendants des causes qui peuvent les produire," *J. Mathématiques Pures et Appliquées*, vol. 5, pp. 380–440, 1840.
[24] M. Schwarz and M. Stamminger, "Fast GPU-based adaptive tessellation with CUDA," *Comput. Graph. Forum*, vol. 28, no. 2, pp. 365–374, 2009.
[25] M. Schütz, B. Kerbl, and M. Wimmer, "Rendering point clouds with compute shaders and vertex order optimization," *Comput. Graph. Forum*, vol. 40, no. 4, pp. 115–126, 2021.
[26] M. Schütz, B. Kerbl, and M. Wimmer, "Software rasterization of 2 billion points in real time," *Proc. ACM Comput. Graph. Interactive Techn.*, vol. 5, no. 3, pp. 1–17, Jul. 2022.
[27] The Khronos Group Inc., "Subgroups :: Vulkan documentation project," 2022-2023. Accessed: Mar. 01 2024. [Online]. Available: https://docs.vulkan.org/guide/latest/subgroups.html
[28] The Khronos Group Inc, "Tessellation :: Vulkan documentation project," 2022-2023. Accessed: Mar. 03 2024. [Online]. Available: https://docs.vulkan.org/spec/latest/chapters/tessellation.html
[29] The Khronos Group Inc, "Vulkan 1.3.279 - A specification (with all registered Vulkan extensions)," 2024. Accessed: Mar. 01 2024. [Online]. Available: https://registry.khronos.org/vulkan/specs/1.3-extensions/html
[30] D. S. Tuch, T. G. Reese, M. R. Wiegell, N. Makris, J. W. Belliveau, and V. J. Wedeen, "High angular resolution diffusion imaging reveals intravoxel white matter fiber heterogeneity," *Magn. Reson. Med.*, vol. 48, no. 4, pp. 577–582, 2002, doi: 10.1002/mrm.10268.
[31] J. Unterguggenberger, B. Kerbl, J. Pernsteiner, and M. Wimmer, "Conservative meshlet bounds for robust culling of skinned meshes," *Comput. Graph. Forum*, vol. 40, no. 7, Oct. 2021, Art. no. 13, doi: 10.1111/cgf.14401.
[32] J. Unterguggenberger, L. Lipp, M. Wimmer, B. Kerbl, and M. Schütz, "Fast rendering of parametric objects on modern GPUs," in *Proc. Eurographics Symp. Parallel Graph. Visual.*, 2024, Art. no. 12, doi: 10.2312/pgv.20241129.
[33] S. Willems, "GPU hardware info database," 2016–2024. Accessed: Sep. 12 2024. [Online]. Available: http://gpuinfo.org
[34] B. Wilson, "Building a parametric seashell," 2022. Accessed: Feb. 29 2024. [Online]. Available: https://observablehq.com/@bronna/parametric-seashell
[35] M. Worchel and M. Alexa, "Differentiable rendering of parametric geometry," *ACM Trans. Graph.*, vol. 42, no. 6, pp. 1–18, 2023.
[36] C. Zhang, M. Caan, T. Höllt, E. Eisemann, and A. Vilanova, "Overview detail visualization for ensembles of diffusion tensors," *Comput. Graph. Forum*, vol. 36, no. 3, pp. 121–132, 2017, doi: 10.1111/cgf.13173.
[37] C. Zhang, T. Höllt, T. Caan, E. Eisemann, and A. Vilanov, "Comparative visualization for diffusion tensor imaging group study at multiple levels of detail," in *Proc. Eurographics Workshop Vis. Comput. Biol. Med.*, 2017, pp. 53–62, doi: 10.2312/vcbm.20171237.
[38] C. Zhang, T. Schultz, K. Lawonn, E. Eisemann, and A. Vilanova, "Glyph-based comparative visualization for diffusion tensor fields," *IEEE Trans. Vis. Comput. Graphics*, vol. 22, no. 1, pp. 797–806, Jan. 2016.

Johannes Unterguggenberger received the BSc, MSc, and PhD degrees from TU Wien. After having been university assistant and project assistant with TU Wien, he joined Huawei Technologies as senior computer graphics architect. His research interests revolve around real-time rendering.

Lukas Lipp received the bachelor of science and master's degrees from the Vienna University of Technology. He is currently working toward the PhD degree specializing in the field of differentiable rendering. His research focuses on advancing computer graphics, specifically exploring techniques that merge rendering with machine learning and optimization.

Michael Wimmer received the MSc and PhD degrees from TU Wien, in 1997 and 2001. He is a full professor with the Institute of Visual Computing and Human-Centered Technology, TU Wien, where he heads the Rendering and Modeling Group. He is also the director of the Center for Geometry and Computational Design (GCD).

Markus Steinberger received the MSc and PhD degrees from the Graz University of Technology, in 2010 and 2013. He is a full professor with the Graz University of Technology, Austria. His biggest honors include the promotion sub auspiciis prasidentis rei publicae, being the first Austrian to win the GI Dissertation Prize, and winning the Heinz Zemanek Prize.

Bernhard Kerbl received the MSc and PhD degrees from the Graz University of Technology. After a postdoc with INRIA, he was a visiting researcher in the Robotics Institute, Carnegie Mellon University and is currently a co-principal investigator with TU Wien. His research focuses on parallel processing, real-time and differentiable rendering, especially radiance fields.

Markus Schütz received the PhD degree from TU Wien, in 2021. He is a postdoc researcher with TU Wien. His research focus is on point-based rendering, specifically on GPGPU-based rasterization algorithms and LOD construction, as well as adaptions of point-based rendering algorithms for virtual reality and web browsers.