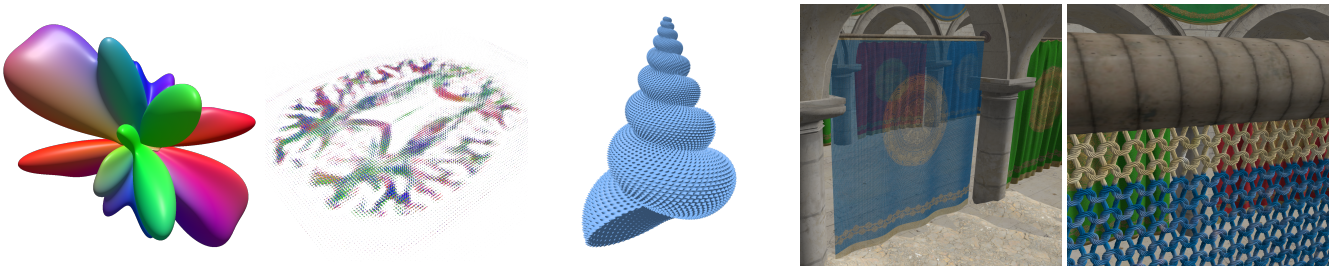
Eurographics Symposium on Parallel Graphics and Visualization (2024)
D. Pugmire, G. Reina, S. Rizzi (Editors)

Fast Rendering of Parametric Objects on Modern GPUs

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

TU Wien, Institute of Visual Computing & Human-Centered Technology, Austria



(a) *SH glyph order 12* **(b)** *19,600 SH glyphs* **(c)** *Parametric seashell* **(d)** *358k fiber curves* **(e)** *Fiber curves close-up*

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

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, 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. This has given rise to several specialized solutions, each targeting interactive rendering of a constrained set of parametric functions. In this paper, we propose a general method for efficient rendering of parametrically defined 3D objects. Our solution is carefully designed around modern hardware architecture. 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 level-of-detail (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 softare 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, while showing its flexibility on a range of other demanding shapes.

CCS Concepts

• ***Computing methodologies*** → ***Rasterization;*** • ***Human-centered computing*** → *Scientific visualization;*

1. Introduction

Across the multi-faceted landscape of interactive graphics applications, we see a growing demand for solutions that reduce overall memory load. The increasing demand for cloud rendering solutions encourages developers to reduce data traffic across high-latency networks. For devices with dedicated graphics hardware, minimizing slow CPU-GPU transfers across slow bus systems is key to avoiding stutter and lagging. Even for on-chip processing, modern consumer-grade GPUs encourage high arithmetic load over memory load: in the last decade, GPU compute throughput has progressed much quicker than memory transfer speed. As an example, consider NVIDIA's RTX 4090, which has 2.5× the compute performance of the previous generation's RTX 3090 (73 TFLOPS vs. 29 TFLOPS), while memory bandwidth increased by only 7.8%

© 2024 The Authors.
Proceedings published by Eurographics - The European Association for Computer Graphics.
This is an open access article under the terms of the Creative Commons Attribution License, which permits use, distribution and reproduction in any medium, provided the original work is properly cited.

DOI: 10.2312/pgv.20241129

delivered by EUROGRAPHICS DIGITAL LIBRARY www.eg.org diglib.eg.org

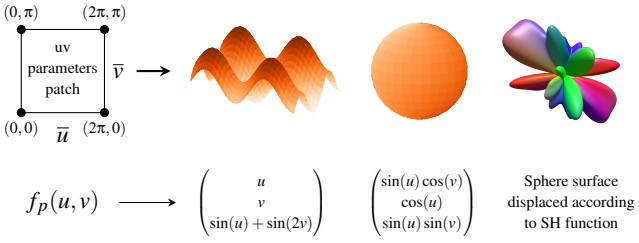


Figure 2: *We consider vector-valued parametric functions $f_p(u,v)$ with $f_p : \mathbb{R}^2 \to \mathbb{R}^3$ mapping independent 2D variables called parameters to 3D positions in Euclidean space. The arithmetic definition enables a compact representation of geometric shapes with varying complexity and desirable properties, such as C^∞ continuity. We illustrate the input to such $f_p(u,v)$ with parameter patches that range from lower bound parameter values u_{min}, v_{min} to upper bound parameter values u_{max}, v_{max}, where the notations $\overline{u}$ and $\overline{v}$ refer to the whole ranges, i.e., $\overline{u} = [u_{min}, u_{max}]$ and $\overline{v} = [v_{min}, v_{max}]$.*

(1008 GB/s vs. 935 GB/s). This development was observable in GPU generations released by NVIDIA and AMD in the last decade, and is especially evident in their last two respective GPU generations. At first glance, these trends run counter to the users' expectations for modern graphics applications: ultra-detailed geometry (≈one triangle per screen pixel [KSW21]) is expected for interactive entertainment. One answer to this apparent dilemma can be found in procedural content, such as *parametric functions*.

Procedural content can be created from coarse inputs by computing transient data for fine-grained details on the fly. At the application level, we can exploit suitable procedural geometry representations, like parametric functions [PDG21, Cra23, Wil22]. These can efficiently yield detailed 3D shapes by evaluating surface or volume samples according to the function's mathematical definition, like illustrated in Figure 2. However, previous work limits the types of functions that can be efficiently sampled, requiring specialized real-time rendering solutions for different function classes.

At the hardware level, procedural content generation is facilitated by modules like the tessellation shader [The23b]. Coarse base geometry is processed in compute units, where the tessellation engine produces new geometry primitives before passing them to the rasterizer. Hence, tessellation trades increased computational load for reduced memory transfers, resulting in overall increased rendering speed compared to not using the tessellator and transferring geometry in high detail [NKF*16]. However, their historic orientation toward triangle meshes makes it unclear how to exploit these modules for other representations, including parametric functions.

In this paper, we consolidate the use of procedural geometry representations—specifically, parametric functions—with recent hardware trends to achieve fast, high-quality rendering of complex mathematical shapes. 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 (see Figure 1). It requires neither derivatives nor preprocessing. We propose a multi-step pipeline: a dedicated LOD analysis stage adaptively determines the sampling density to be used by a subsequent rendering stage. Depending on the chosen sampling resolution, our technique can render highly tessellated patches (where each resulting triangle spawns approximately 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 [SKW21]. Common to both rendering approaches is their emphasis on utilizing the compute capabilities of GPUs and their ease of integration into existing rendering applications. We also account for the emerging trend in recent years toward ultra-high geometric detail in real time, as heralded by Epic Games' Nanite [Epi24]. Nanite can render static geometry at such high detail that, after an LOD selection step, rasterization is usually performed on triangles not much larger than a single pixel [KSW21]. Similarly, our technique can render almost pixel-perfect geometric detail for parametric functions that show limited variance at a sub-pixel level when rendering at screen resolution. With super-sampled (SS) configurations, our tessellation-based or point-based rendering variants are able to capture and render sub-pixel geometric detail in real time.

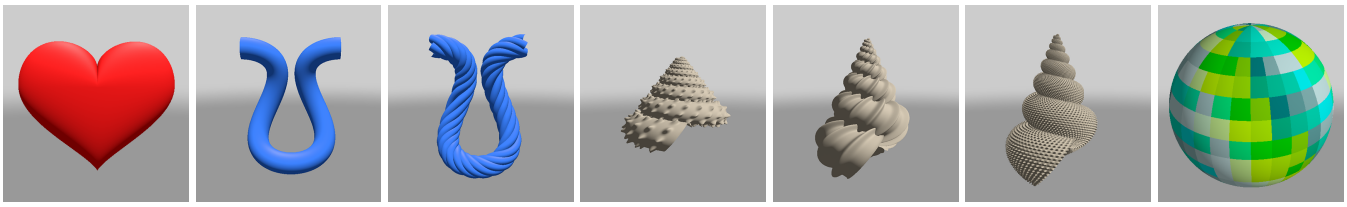
In summary, the contributions of our paper are:

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 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 entire parametric object.
3. We describe two different variants to render the patches from Contribution 2.: One performs point-based rendering into a 64-bit integer image, while the other uses hardware tessellation.
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. [PPUJ23], showing that our method surpasses it in terms of rendering speed and rendering quality for higher SH orders; in large datasets already for SH order 4.

2. Related Work

Most previous work on rendering parametric curves or surfaces focuses on specific parametric shapes, such as the efficient rendering of rational Bèzier patches [EML09, SS09], Catmull-Clark subdivision surfaces [PEO09, NL13, KOCM23, WA23], or similar patches such as B-Spline or NURBS curves and surfaces [WA23]. Poirier et al. [PDG21] focus on rendering Spherical Harmonics (SH) glyphs for visualizing measurements from diffusion magnetic resonance imaging (dMRI) scans. Some of these techniques utilize specialized data structures [PEO09, KOCM23]. All of them 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 [PEO09, SS09, EML09, WA23], while others make use of hardware tessellation units [NL13], which are standard features on modern desktop GPUs. The technique by Kuth et al. [KOCM23] uses mesh shaders, which were first introduced to desktop GPUs with NVIDIA's Turing architecture [NVI18] in 2018.

© 2024 The Authors.
Proceedings published by Eurographics - The European Association for Computer Graphics.



(a) *Johi's Heart* **(b)** *Yarn Curve* **(c)** *Fiber Curve* **(d)** *Seashell variant 1* **(e)** *Seashell variant 2* **(f)** *Seashell variant 3* **(g)** *Sphere*

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

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 [EML09, PEO09], which is also the approach taken by the more recent technique by Worchel et al. [WA23]. The method of Eisenacher et al. [EML09] 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 Bèzier 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 Bèzier patches, possibly requiring a crack avoidance procedure between neighboring patches.

The work by Poirier et al. [PDG21] 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 is able to produce and render ultra-detailed geometry at real-time frame rates on modern GPUs [KOCM23], but focuses on Catmull-Clark subdivision surfaces only. In terms of SH glyph rendering, Peters et al. [PPUJ23] 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. [PDG21], but suffer from exceedingly decreasing performance with increasing SH order and visual artefacts 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. [PDG21] 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. [PPUJ23]: 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 [ZHC*17, ZCH*17, ZSL*15] or comparisons between ensembles of stress tensor fields [AWHS15].

Point rendering can arguably also be described as an approach to render ultra-detailed geometry, under the condition that sufficiently many samples are used. Recent work shows that modern GPUs are capable of processing and rendering 50 to 144 billion points per second [SKW21, SKW22], with the former being limited by memory bandwidth and the latter improving the throughput through compression. By sampling points on parametric functions, we avoid memory fetches as the bottleneck, but potentially trade it for a compute-based bottleneck.

Epic Games' Nanite technology [Epi24] 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× faster for rendering small triangles [KSW21]. 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. [UKPW21] 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 arbitrary fidelity, limited only by machine floating point precision. 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.

3. Parametric Function Definition

Our method considers parametric functions that transform two input parameters (u, v) into Cartesian coordinates (x, y, z) of the three-dimensional Euclidean space, i.e., $\mathbb{R}^2 \to \mathbb{R}^3$, or intuitively a transformation of a quad/rectangle into a three-dimensional surface as illustrated in Figure 2. Interpreting $\mathbb{R}^2$ as, for example, spherical co-

© 2024 The Authors.
Proceedings published by Eurographics - The European Association for Computer Graphics.

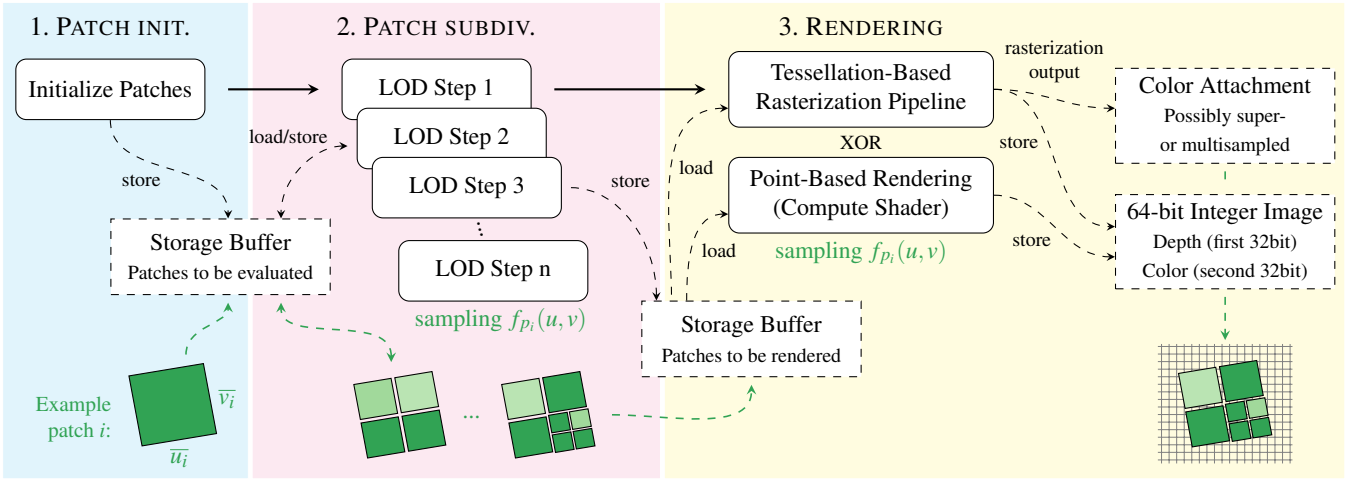


Figure 4: *Overview of the stages of our method, and the buffers and textures which are accessed in each stage. Patches are evaluated and subdivided and then re-evaluated up to n times in the n LOD steps, before they are stored in the "Patches to be rendered" buffer. The rendering stage reads all the 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.*

ordinates and $\mathbb{R}^3$ as the cartesian coordinates of the corresponding points on a sphere allows us transforming the quad into a sphere. We can then further transform the sphere into spherical harmonic glyphs using the respective SH functions.

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 Figure 3a.

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

```
#define PI        3.14159265359
// Creates the surface "Johi's Heart".
// Input: u ... first  parameter in range [0, PI  )
//        v ... second parameter in 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;
}
```

4. 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 position, orientation, projection, and screen coordinates, an LOD stage produces patches of similar size in screen space, which are subsequently rendered with one of two different rendering variants. Figure 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. The point of this procedure is to create patches of similar sizes to be forwarded to the RENDERING stage in order to optimally utilize GPUs.
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.

© 2024 The Authors.
Proceedings published by Eurographics - The European Association for Computer Graphics.

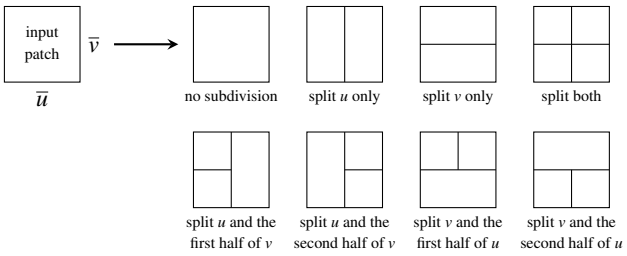


Figure 5: *During an LOD step, an input patch can either not be subdivided, or split according to the seven patterns shown in this figure when being scheduled for re-evaluation.*

4.1. 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 $v_1,\dots,v_4$ (for a total of 32 samples), and the same for the v-direction at 4 fixed u-values $u_1,\dots,u_4$. 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 Figure 5.

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 Figure 6, showing the lines used for patch evaluation in u and v directions, and the 25 extra samples.

A fixed number of compute shader invocations is dispatched every frame to perform these evaluations in multiple subsequent LOD steps. We use 12 dispatch calls, which is enough to subdivide a (perfectly screen-aligned) 32k pixels-wide patch down to 8 pixels and should suffice for the vast majority of cases. 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 Figure 4. All of the dispatch calls in the 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 [The24]. 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 previous frame's highest required LOD step, if latencies of a few frames to reach the appropriate number of dispatch calls are acceptable by an application.

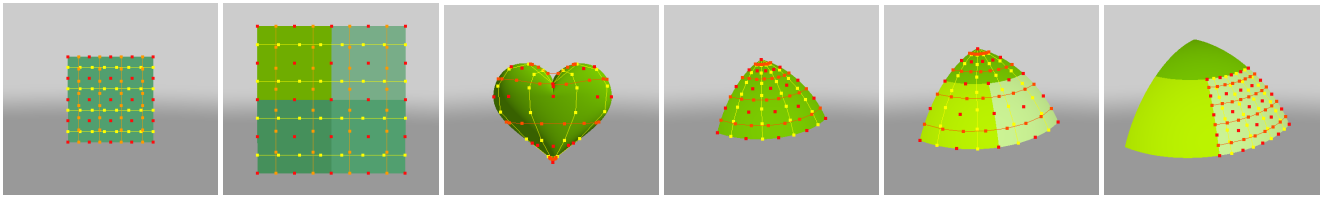
In many scenarios, the PATCH SUBDIVISION stage is very fast, often taking less than 10% of the total frame time, which is typically the case for the tests presented in Figures 9 and 12 even when the camera is near the parametric object so that many screen pixels are covered. The exact percentage depends on the particular object and scene setup. In the test presented in Figure 11, the LOD stage has to determine patch sizes for 358k fiber curves (i.e., 358k 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 sharing data between compute invocations [The23a]. For example, in order 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 Figure 5. There are eight possible cases when a patch is evaluated in the LOD step: If the approximated screen-space extents $e_{u_{ik}}, e_{v_{ik}}$ 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 $e_{u_{ik}}$ or $e_{v_{ik}}$, patch i is split according to the seven patterns shown in Figure 5—except in LOD step n, where (possibly split) patches are scheduled for rendering in any case. Non-uniform patch subdivision doesn't influence the performance of the PATCH SUBDIVISION stage too much, but to a greater degree leads to better performance in the RENDERING stage, since the resulting patches are more similar in terms of 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 already 15% for a simple parametrically defined sphere like shown in Figure 3g, and can be as high as 50% for the close-up view of a SH glyph like shown in Figure 9c.

4.2. Rendering Tessellated Patches

Rendering patches that have been scheduled for rendering with our TESSELLATION-BASED variant is very straightforward: Each scheduled patch i is rendered as a quad with fixed inner and outer tessellation levels l [The23b]. 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 [Khr14]. 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 so that there is one segment for each pixel. In many cases, choosing such fine subdivisions is counter-

© 2024 The Authors.
Proceedings published by Eurographics - The European Association for Computer Graphics.



(a) *Patch evaluation resulted in no splits.* **(b)** *Patch evaluation resulted in $\overline{u}$ and $\overline{v}$ splits.* **(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 the now subdivided patch (LOD step 2)*

Figure 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. Figure 6c 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 very near 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. Figures 6d to 6f show an example of evaluating a sphere segment. Assuming a resolution of 512×512 and user-defined screen thresholds of $t_u = t_v = 256$, LOD step 1 finds that the patch in Figure 6d does not exceed the thresholds and hence, no subdivisions are required. With the camera closer to the object in Figure 6e, 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. Figure 6f shows the evaluation pattern on one of these patches during LOD step 2.*

productive due to the resulting impact on rendering performance [KDR18, KHCW22]. 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 geometry density.

Adaptive tessellation levels are based on the actual approximated maximum screen-space extents e_{u_i}, e_{v_i} of a given patch i. These extents are measured during PATCH SUBDIVISION by taking the maximum of the four measurements $e_{u_{i_k}}$ and the maximum of the four measurements $e_{v_{i_k}}$, as illustrated in Figure 6. For example, splitting 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:

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

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., $l_{max} = 64$).

The outputs of a graphics pipeline are typically rasterized into a color attachment. This is the only option to profit from hardware-accelerated multi-sampling (MS) and its resolve operation. We use color attachments as render targets for all our tests presented in Section 5 that use the TESSELLATION-BASED variant since it is generally faster than the alternative—which is storing the rendering output into a 64-bit integer image. The latter is fundamentally the same approach as taken by Nanite [KSW21]. Depth writes and also the depth test of a graphics pipeline can be disabled in this case. The resulting depth and color values are instead written in software through atomic operations from fragment shaders. By combining depth and color values into a single 64-bit integer value, as illustrated below, `atomicMin` operations ensure that parallel writes will produce the correct rendering result.

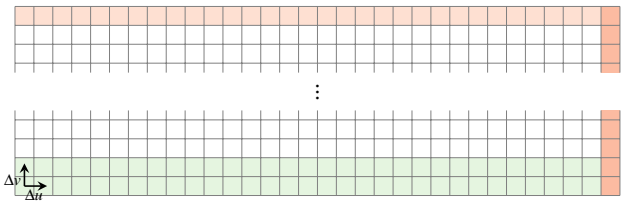
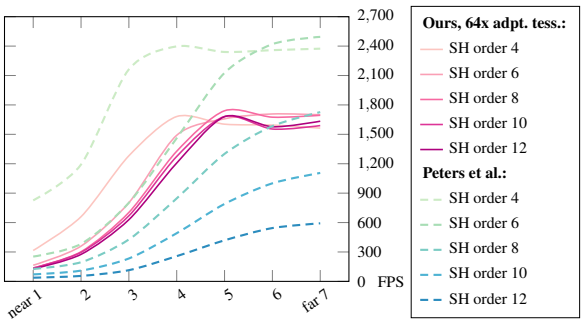


Figure 7: *With the POINT-BASED variant, parameter range $\overline{u_i}$ is processed in "columns" of 32 samples. Within such a column, subgroups sample the patch to be rendered "row-wise" in v parameter direction. A subgroup always keeps the data of two rows in registers, which are marked in green.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

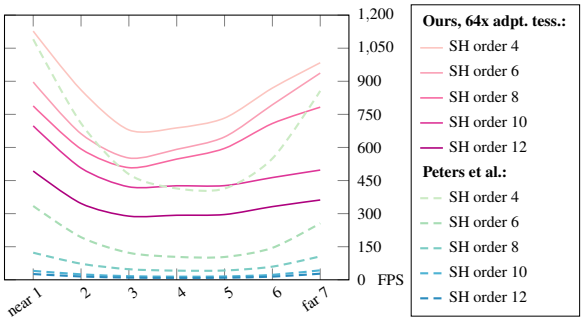
4.3. Point-Based Rendering

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. 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 4.2. 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 $\Delta v = \frac{v_{i_{max}} - v_{i_{min}}}{t_v} - \varepsilon$, where ε can be used to decrease the step size in order 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 $\lceil \frac{t_u}{31} \rceil$

© 2024 The Authors.
Proceedings published by Eurographics - The European Association for Computer Graphics.



(a) *FPS comparison 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.*

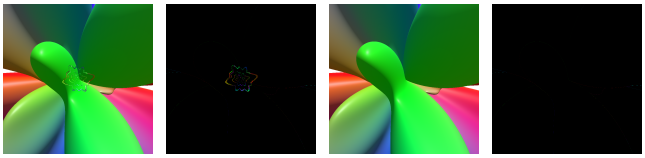


(b) *FPS comparison of using different SH orders for rendering the HARDI dataset containing 19,600 SH glyphs on an NVIDIA RTX 3070. $t_u = t_v = 84$, noAA, adaptive tessellation, and $l_{max} = 64$ have been used for our method.*

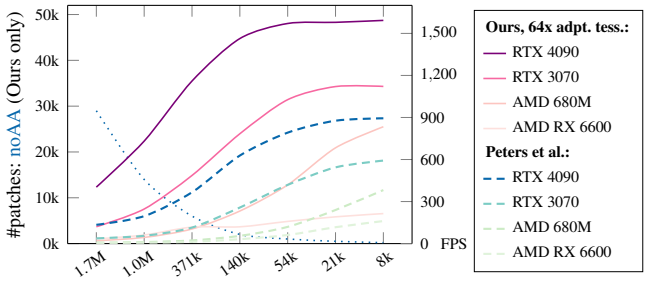
Figure 8: *These diagrams show the impact of using different SH orders for rendering the SH glyphs.*

"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 Figure 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 higher subgroup index. Therefore, only 31 subgroups write pixel values, while the last subgroup only provides its data to the second to last subgroup.

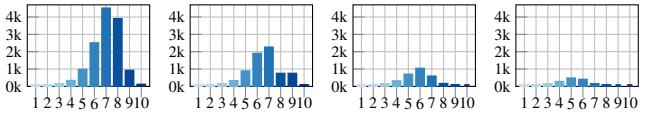
Our POINT-BASED rendering variant also features an **adaptive sampling** configuration, where the number of samples in u and v directions are not calculated based on the threshold parameters t_u, t_v, but instead on the approximated screen distance extents e_{u_i}, e_{v_i}, which are measured during PATCH SUBDIVISION. The adaptive sampling configuration processes patch i in $\left\lceil \frac{e_{u_i}}{31} \right\rceil$ 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.



(a) *Result of Peters et al.* **(b)** *Diff. between reference and 9a* **(c)** *Ours with adaptive tess.* **(d)** *Diff. between reference and 9c*



(e) *FPS rendered by different GPUs for different views of one single SH glyph, comparing our method ($t_u = t_v = 84$, noAA, adaptive tessellation, $l_{max} = 64$) to the method of Peters et al. The x axis shows the average number of pixels written (excluding overdraw). The dotted line represents the number of patches to be rendered, output by* PATCH SUBDIVISION.



(f) *#patches out per LOD step (1.7M)* **(g)** *#patches out per LOD step (1.0M)* **(h)** *#patches out per LOD step (371k)* **(i)** *#patches out per LOD step (140k)*

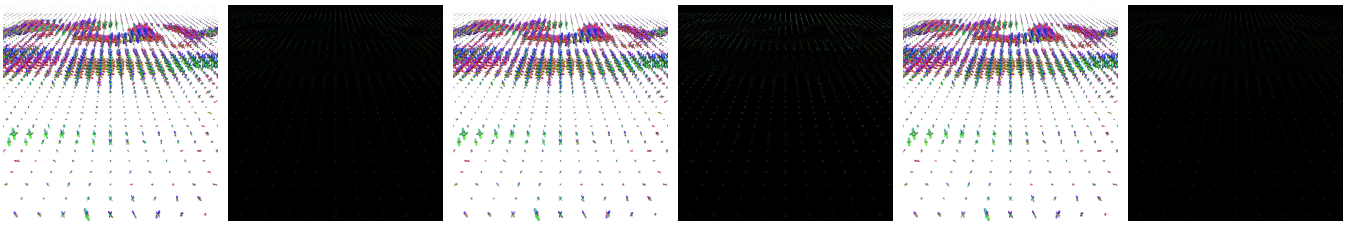
Figure 9: *For an SH glyph of order 12, Figures 9a to 9d show qualitative comparisons of the rendered results compared with a reference image produced with 16xSS and 8xMS. For the first measurement, the camera starts at the view producing Figures 9a and 9c and moves away for subsequent measurements. Figure 9e presents the performance of our variants in comparison to the method by Peters et al. Ours avoids their closeup artifacts. It also shows the number of patches to be rendered, produced in our* PATCH SUBDIVISION *stage—while Figures 9f to 9i show the average numbers of patches to be evaluated during each LOD step. No patches remain to be evaluated after LOD step 10 in any of these test setups.*

5. Results

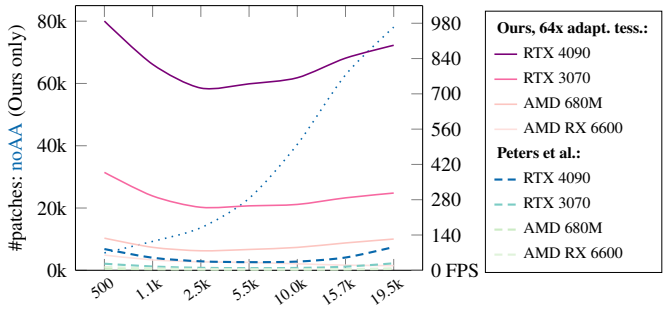
We evaluate our method and its rendering variants based on the following parametric functions, which have different characteristics and therefore pose different challenges to a rendering method:

- SH glyphs
- Parametric plain-knit yarn curves
- Parametric seashells

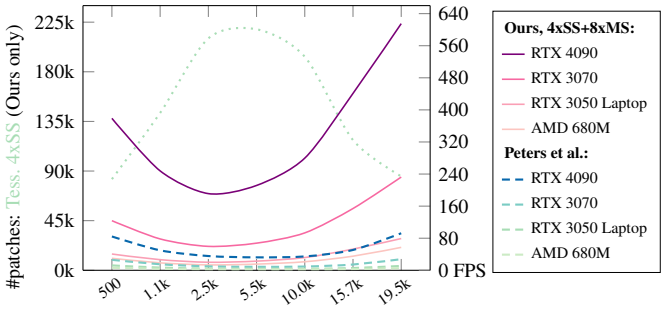
© 2024 The Authors.
Proceedings published by Eurographics - The European Association for Computer Graphics.



(a) *Rendered with the method by Peters et al.* **(b)** *Difference between 10a and reference* **(c)** *Rendered with our method, noAA* **(d)** *Difference between 10c and reference* **(e)** *Rendered with our method, 4xSS+8xMS* **(f)** *Difference between 10e and reference*



(g) *FPS rendered by different GPUs for different views of the HARDI dataset, comparing our method ($t_u = t_v = 84$, noAA, adaptive tessellation, $l_{max} = 64$) to the method of Peters et al. The x axis shows the average number of SH glyphs rendered for a certain view distance. The dotted line represents the patches output by our* PATCH SUBDIVISION *stage.*



(h) *Comparing our method ($t_u = t_v = 84$, 4xSS, 8xMS, adaptive tessellation, $l_{max} = 64$) to the method of Peters et al. The x axis shows the average number of SH glyphs rendered for a certain view distance. The dotted line represents the number of patches to be rendered, generated by our* PATCH SUBDIVISION *stage, which is much higher than in Figure 10g due to 4xSS.*

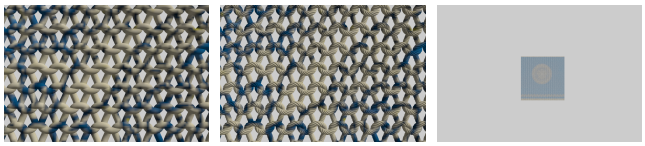
Figure 10: *For 19,600 SH glyphs of order 12, Figures 10a to 10f show qualitative comparisons between the results and a reference image which has been produced with 16xSS and 8xMS. Figures 10g and 10h compare the performance of our variants with the method by Peters et al. The camera starts at a view distance similar to the one producing Figures 10a, 10c and 10e and moves away for subsequent measurements.*

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. The challenge with rendering plain-knit yarn curves is the vast amount that is required for a typical scene. While these two parametric functions produce smooth surfaces, the parametric seashell model has very small-scale surface features, leading to sub-pixel geometric detail for most of our test setups. The results presented in Figures 9 to 12 were gathered after a GPU warm-up phase of 2 seconds from multiple camera positions. The camera is located close to the object or center of the dataset for the first measurement and moves away with every further measurement. Each measurement result (such as FPS, number of pixels written, or number of render patches) represents the averaged data over a 5 seconds long time span, 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 RENDERING stage every frame.

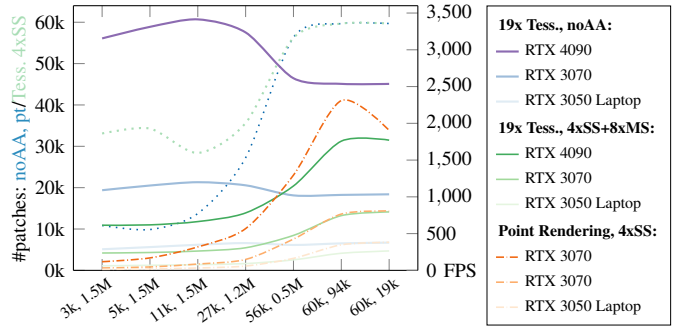
SH Glyphs: Spherical harmonics are mathematical functions defined on the surface of a sphere and are parameterized using spherical coordinates (θ, φ) with $\theta \in [0, \pi]$ and $\varphi \in [0, 2\pi]$. These functions result from a linear combination of a set of orthonormal basis functions, an important property which 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 to represent higher frequencies but also come with an increase in 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) [TRW*02] dataset of a brain scan [HCAD22], captured via dMRI and shown in Figure 1b. We compare our method with the state-of-the-art SH glyph rendering method by Peters et al. [PPUJ23], which uses ray tracing to render high-quality SH glyphs in real time. Figure 8 shows the effect of using different SH orders for rendering. Our method shows remarkably consistent performance for varying SH orders, while the method by Peters et al. suffers from strongly decreasing performance with each SH order increase. 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 (Figure 8a) and already for SH orders 4 and higher for the large dataset (Figure 8b). 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, Figure 9e 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 configuration is shown in Figures 9c and 9e, which

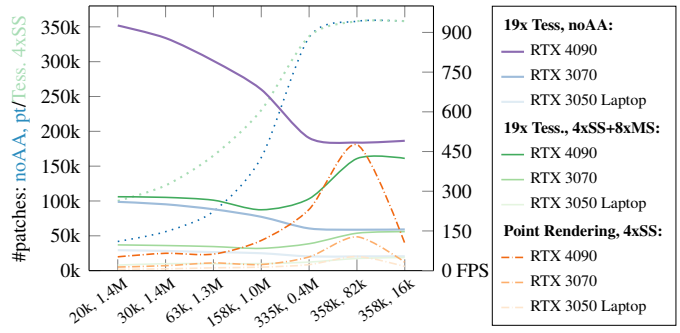
© 2024 The Authors.
Proceedings published by Eurographics - The European Association for Computer Graphics.



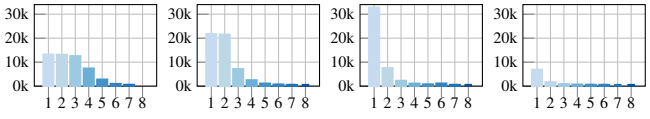
(a) *Initial camera position (yarn curves)* **(b)** *Initial camera position (fiber curves)* **(c)** *Final camera position*



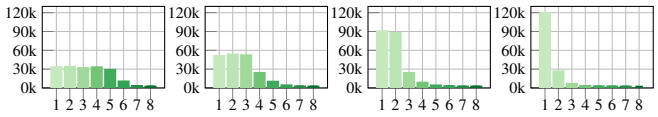
(d) *Comparing the performance of three variants on multiple GPUs. The x axis states number of fiber curves rendered, and pixels written (not counting overdraw) for the different camera views between Figures 11a and 11c.*



(e) *Comparing the performance of three variants on multiple GPUs. The x axis states tuples of number of fiber curves rendered, and pixels written (not counting overdraw) for the different camera distances between Figures 11b and 11c.*



(f) *#patches out per step (20k, 1.4M)* **(g)** *#patches out per step (30k, 1.4M)* **(h)** *#patches out per step (63k, 1.3M)* **(i)** *#patches out per step (158k, 1.0M)*



(j) *#patches out per LOD step 4xSS (20k, 1.4M)* **(k)** *#patches out per LOD step 4xSS (30k, 1.4M)* **(l)** *#patches out per LOD step 4xSS (30k, 1.4M)* **(m)** *#patches out per LOD step 4xSS (30k, 1.4M)*

Figure 11: *Performance results and number of patches to be rendered of different configurations for 60k yarn curves (Figure 11d), and 358k fiber curves (Figure 11e). For the latter, Figures 11f to 11i show the numbers of patches to be evaluated for the noAA and point rendering variants for each LOD step, while Figures 11j to 11m show the corresponding numbers for the 4xSS configuration.*

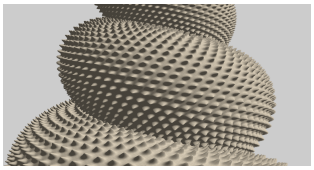
produces almost pixel-perfect geometric detail. Even a configuration of our method with fixed tessellation factors outperforms the method by Peters et al. across all GPUs, but does not improve rendering quality despite the higher geometric detail produced. Our method avoids the artefacts produced by the method by Peters et al., which are shown in Figure 9b, but it also introduces small, much less noticeable artefacts near the poles of the scaled sphere, an example of which is shown in Figures 13a and 13b. 9f to 9i show how PATCH SUBDIVISION increases the number of patches uniformly during the first four LOD steps to create sufficient geometric detail. Steps 5 and higher subdivide fewer patches or subdivide non-uniformly for larger distances to the camera, since the subdivisions suffice already in more cases. Rendering the HARDI dataset, our method outperforms the method by Peters et al. even more strongly, as shown in Figure 10g. It even provides headroom for a 4xSS and 8xMS variant, as shown in Figures 10e and 10h. The image quality of this configuration is superior, which is especially noticeable during camera movements and can also be observed in Figure 10f.

Plain-Knit Yarn: Gröller et al. [GRS95] provided an early parametric description of knitwear, but we use the more recent parametric plain-knit yarn curves described by Crane [Cra23]. Its fundamental shapes of yarn curves and fiber curves are shown in Figures 3b and 3c, respectively. We use an extruded version of them for our evaluations, namely one which takes the yarn direction ("tangent") as parameter u, and constructs a circle around each point along that direction via parameter v. We get an orthogonal vector ("normal") to the tangent and rotate it using Rodrigues' rotation formula [Rod40] 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 amount of yarn curves in real time with good rendering quality and offers real-time changes to the code—our method enables this for at least 358k curves, as our results in Figure 11e show.

We show the performance comparisons of three different variants of our method on the example of a blue curtain composed of 60k yarn curves or 358k fiber curves in Figure 11. 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. In both test cases, the noAA configuration suffers from severe aliasing artefacts such as Moiré patterns, which is due to the high number of yarn curves or fiber curves and the sub-pixel geometric detail produced by them, especially for larger camera distances. The configurations with SS produce visually satisfying results like those shown in Figures 1d and 1e. In most situations, TESSELLATION-BASED 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. In contrast to the results of single SH glyph rendering, where all the geometry is raised from one single initial patch, Figures 11f to 11m show different subdivision characteristics since PATCH INITIALIZATION al-

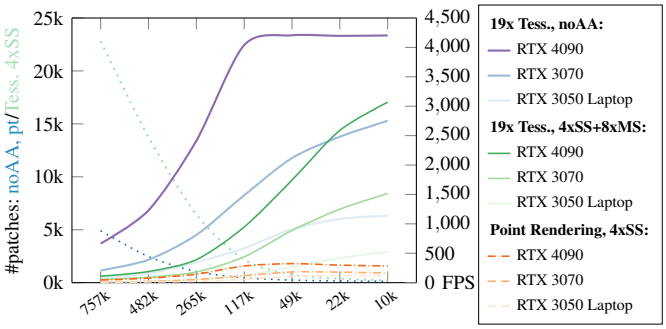
© 2024 The Authors.
Proceedings published by Eurographics - The European Association for Computer Graphics.

ready creates 358k patches. Many of them are culled for near camera positions. Hence, only ≈12k remain after the first LOD step in Figure 11f, 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 Figures 11j to 11m generally require more patch subdivisions due to the higher effective resolution.

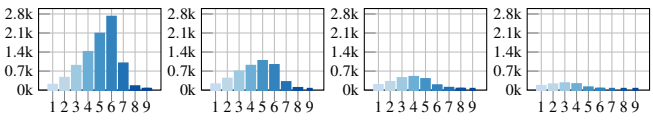


(a) *Initial camera position*

(b) *Final camera position*



(c) *Comparing the performance of three variants on multiple GPUs. The x axis represents different camera distances between Figures 12a and 12b, stating the number of pixels written on average (not counting overdraw).*

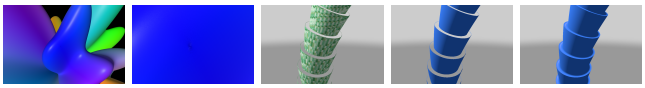


(d) *#patches out per LOD step (757k)* **(e)** *#patches out per LOD step (482k)* **(f)** *#patches out per LOD step (256k)* **(g)** *#patches out per LOD step (117k)*

Figure 12: *Results of a parametric seashell with sub-pixel detail. Figures 12d to 12g show the numbers of patches evaluated per LOD step for the noAA and point rendering configurations.*

Seashell: The parametric seashell model follows roughly the construction guidelines by Wilson [Wil22], which allows creating a myriad of seashell variants by parameter variations, such as the ones shown in Figures 3d to 3f. Achieving satisfying rendering results for the variant shown in Figure 3f is challenging since its parametric model creates many tiny bumps on the surface, which lead to sub-pixel geometric detail when viewed from a distance or rendered in low resolution. For our tests with the parametric seashell function we use the same configurations that we have used for the yarn curves tests, except for $l_{max} = 64$ to capture sub-pixel detail produced by the parametric seashell model for many view positions. Also in this case, configurations with SS produce much more satisfying visual results. The POINT-BASED performance trails behind TESSELLATION-BASED configurations. Results of different configurations are shown in Figure 12.

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



(a) *Artefacts at SH poles* **(b)** *Closeup of Figure 13a* **(c)** *Jump discontinuities* **(d)** *Point rendering* **(e)** *Rendered with triangles*

Figure 13: *Current limitations of our method*

6. 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 680M GPU than the method by Peters et al. [PPUJ23] 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 the good performance, contrary to the recommendation of Eisenacher et al. [EML09] to always split patches 1:4. Additional experiments for organic shapes and fabric yield several hundred FPS while avoiding prominent artifacts.

There are multiple avenues for future work: We intend to investigate better handling of holes in parametric objects. Our PATCH SUBDIVISION ignores the concept of holes and would currently subdivide patches strongly near the discontinuity as illustrated in Figure 13c. An early exit criterion could improve performance in such cases. Furthermore, the rendered output of a parametrically defined palm tree trunk shown in Figures 13d and 13e currently differs between POINT-BASED and TESSELLATION-BASED since point rendering does not fill gaps, which are closed by the triangles produced by the tessellator. 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, the speed of which we found to depend heavily on the test setup used. Most interestingly, POINT-BASED performance increased in some cases when the performance of TESSELLATION-BASED declined. Hence, a combined rendering variant is conceivable, where noAA tessellation-based rendering is used for coarse patches and point-based rendering for all others. Such a dynamic technique may guarantee improved rendering quality without compromising performance. A new feature of modern graphics APIs called *Work Graphs* [PR23, Adv23] 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 4.1.

Since PATCH SUBDIVISION is fast enough to run every frame, our method is well suited for use with animated objects or time-varying data. Although flowing curtains consisting of hundreds of thousands of fiber curves have their appeal, we have only scratched the surface: we believe that our general method can unlock a wide range of elaborate, animated parametric functions, and enable glyph-based visualization of time-varying medical data with high frame rates, or smooth morphing between data sets in real time.

© 2024 The Authors.
Proceedings published by Eurographics - The European Association for Computer Graphics.

Acknowledgements

We would like to sincerely thank Christoph Peters for his detailed guidelines for getting the implementation of their SH glyph rendering method [PPUJ23] right and reasonably optimized, so that we were able to compare it to our method in a fair manner. Furthermore, we thank him and Tark Patel for providing the data set they have used in their paper and pointing us to its origin [HCAD22]. We thank Hiroyuki Sakai for his help with LaTeX and TikZ. The "Sponza" 3D model shown in Figures 1d and 1e is a version modified by Johannes Unterguggenberger (bundled with the rendering framework *Auto-Vk-Toolkit* [Res24]), which 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).

References

[Adv23] ADVANCED MICRO DEVICES, INC.: Announcing GPU Work Graphs in Vulkan. https://gpuopen.com/gpu-work-graphs-in-vulkan, 2023. [Accessed 08-April-2024]. 10

[AWHS15] ABBASLOO A., WIENS V., HERMANN M., SCHULTZ T.: Visualizing tensor normal distributions at multiple levels of detail. *IEEE transactions on visualization and computer graphics 22*, 1 (2015), 975–984. 3

[Cra23] CRANE K.: A Simple Parametric Model of Plain-Knit Yarns. https://github.com/keenancrane/plain-knit-yarn, 3 2023. [Accessed 29-February-2024]. 2, 9

[EML09] EISENACHER C., MEYER Q., LOOP C.: Real-time view-dependent rendering of parametric surfaces. In *Proceedings of the 2009 symposium on Interactive 3D graphics and games* (2009), pp. 137–143. 2, 3, 10

[Epi24] EPIC GAMES, INC.: Nanite Virtualized Geometry in Unreal Engine | Unreal Engine 5.0 Documentation. https://docs.unrealengine.com/5.0/en-US/nanite-virtualized-geometry-in-unreal-engine/, 2024. [Accessed 01-Jan-2024]. 2, 3

[GRS95] GRÖLLER E., RAU R. T., STRASSER W.: Modeling and visualization of knitwear. *IEEE Transactions on Visualization and Computer Graphics 1*, 4 (1995), 302–310. 9

[HCAD22] HASHEMIZADEHKOLOWRI S., CHEN R.-R., ADLURU G., DIBELLA E. V. R.: Jointly estimating parametric maps of multiple diffusion models from undersampled q-space data: A comparison of three deep learning approaches. *Magnetic Resonance in Medicine 87*, 6 (2022), 2957–2971. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/mrm.29162, arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/mrm.29162, doi:https://doi.org/10.1002/mrm.29162. 8, 11

[KDR18] KHOURY J.-N., DUPUY J., RICCIO C.: Adaptive gpu tessellation with compute shaders. URL: https://api.semanticscholar.org/CorpusID:198116975. 6

[KHCW22] KERBL B., HORVÁTH L., CORNEL D., WIMMER M.: An Improved Triangle Encoding Scheme for Cached Tessellation. In *Eurographics 2022 - Short Papers* (2022), Pelechano N., Vanderhaeghe D., (Eds.), The Eurographics Association. doi:10.2312/egs.20221031. 6

[Khr14] KHRONOS GROUP: gl_TessCoords - OpenGL 4 Reference Pages. https://registry.khronos.org/OpenGL-Refpages/gl4/html/gl_TessCoord.xhtml, 2011-2014. [Accessed 08-March-2024]. 5

[KOCM23] KUTH B., OBERBERGER M., CHAJDAS M., MEYER Q.: Edge-friend: Fast and deterministic catmull-clark subdivision surfaces. *Computer Graphics Forum 42*, 8 (2023), e14863. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.14863, arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1111/cgf.14863, doi:https://doi.org/10.1111/cgf.14863. 2, 3

[KSW21] KARIS B., STUBBE R., WIHLIDAL G.: A deep dive into nanite virtualized geometry. In *ACM SIGGRAPH 2021 Courses, Advances in Real-Time Rendering in Games, Part 1*. 2021. https://advances.realtimerendering.com/s2021/index.html [Accessed 10-September-2021]. 2, 3, 6

[NKF*16] NIESSNER M., KEINERT B., FISHER M., STAMMINGER M., LOOP C., SCHÄFER H.: Real-time rendering techniques with hardware tessellation. *Computer Graphics Forum 35*, 1 (2016), 113–137. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.12714, arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1111/cgf.12714, doi:https://doi.org/10.1111/cgf.12714. 2

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

[NVI18] NVIDIA CORPORATION: NVIDIA Turing GPU Architecture. https://images.nvidia.com/aem-dam/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf, 2018. [Accessed 12-April-2021]. 2

[PDG21] POIRIER C., DESCOTEAUX M., GILET G.: Accelerating geometry-based spherical harmonics glyphs rendering for dmri using modern opengl. In *Computational Diffusion MRI* (Cham, 2021), Cetin-Karayumak S., Christiaens D., Figini M., Guevara P., Gyori N., Nath V., Pieciak T., (Eds.), Springer International Publishing, pp. 144–155. 2, 3

[PEO09] PATNEY A., EBEIDA M., OWENS J.: Parallel view-dependent tessellation of catmull–clark subdivision surfaces. pp. 99–108. doi:10.1145/1572769.1572785. 2, 3

[PPUJ23] PETERS C., PATEL T., USHER W., JOHNSON C. R.: Ray Tracing Spherical Harmonics Glyphs. In *Vision, Modeling, and Visualization* (2023), Guthe M., Grosch T., (Eds.), The Eurographics Association. doi:10.2312/vmv.20231223. 2, 3, 8, 10, 11

[PR23] PATEL A., RIDDELL T.: D3D12 Work Graphs Preview. https://devblogs.microsoft.com/directx/d3d12-work-graphs-preview, 2023. [Accessed 08-April-2024]. 10

[Res24] RESEARCH UNIT OF COMPUTER GRAPHICS | TU WIEN: Auto-Vk-Toolkit. https://github.com/cg-tuwien/Auto-Vk-Toolkit, 2024. 11

[Rod40] RODRIGUES O.: 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. *Journal de Mathématiques Pures et Appliquées 5* (1840), 380–440. 9

[SKW21] SCHÜTZ M., KERBL B., WIMMER M.: Rendering point clouds with compute shaders and vertex order optimization. *Computer Graphics Forum 40*, 4 (2021), 115–126. URL: https://www.cg.tuwien.ac.at/research/publications/2021/SCHUETZ-2021-PCC/. 2, 3

[SKW22] SCHÜTZ M., KERBL B., WIMMER M.: Software rasterization of 2 billion points in real time. *Proceedings of the ACM on Computer Graphics and Interactive Techniques 5*, 3 (July 2022), 1–17. 3

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

[The23a] THE KHRONOS GROUP INC.: subgroups :: Vulkan Documentation Project. https://docs.vulkan.org/guide/latest/subgroups.html, 2022-2023. [Accessed 01-March-2024]. 5

© 2024 The Authors.
Proceedings published by Eurographics - The European Association for Computer Graphics.

[The23b] THE KHRONOS GROUP INC.: Tessellation :: Vulkan Documentation Project. https://docs.vulkan.org/spec/latest/chapters/tessellation.html, 2022-2023. [Accessed 03-March-2024]. 2, 5

[The24] THE KHRONOS GROUP INC.: Vulkan® 1.3.279 - A Specification (with all registered Vulkan extensions). https://registry.khronos.org/vulkan/specs/1.3-extensions/html, 2024. [Accessed 01-March-2024]. 5

[TRW*02] TUCH D. S., REESE T. G., WIEGELL M. R., MAKRIS N., BELLIVEAU J. W., WEDEEN V. J.: High angular resolution diffusion imaging reveals intravoxel white matter fiber heterogeneity. *Magnetic Resonance in Medicine 48*, 4 (2002), 577–582. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/mrm.10268, arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/mrm.10268, doi:https://doi.org/10.1002/mrm.10268. 8

[UKPW21] UNTERGUGGENBERGER J., KERBL B., PERNSTEINER J., WIMMER M.: Conservative meshlet bounds for robust culling of skinned meshes. *Computer Graphics Forum 40*, 7 (Oct. 2021), 13. URL: https://www.cg.tuwien.ac.at/research/publications/2021/unterguggenberger-2021-msh/, doi:10.1111/cgf.14401. 3

[WA23] WORCHEL M., ALEXA M.: Differentiable rendering of parametric geometry. *ACM Transactions on Graphics (TOG) 42*, 6 (2023), 1–18. 2, 3

[Wil22] WILSON B.: Building a Parametric Seashell. https://observablehq.com/@bronna/parametric-seashell, 2022. [Accessed 29-February-2024]. 2, 10

[ZCH*17] ZHANG C., CAAN M., HÖLLT T., EISEMANN E., VILANOVA A.: Overview + detail visualization for ensembles of diffusion tensors. *Computer Graphics Forum 36*, 3 (2017), 121–132. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.13173, arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1111/cgf.13173, doi:https://doi.org/10.1111/cgf.13173. 3

[ZHC*17] ZHANG C., HÖLLT T., CAAN M. W. A., EISEMANN E., VILANOVA A.: Comparative Visualization for Diffusion Tensor Imaging Group Study at Multiple Levels of Detail. In *Eurographics Workshop on Visual Computing for Biology and Medicine* (2017), Bruckner S., Hennemuth A., Kainz B., Hotz I., Merhof D., Rieder C., (Eds.), The Eurographics Association. doi:10.2312/vcbm.20171237. 3

[ZSL*15] ZHANG C., SCHULTZ T., LAWONN K., EISEMANN E., VILANOVA A.: Glyph-based comparative visualization for diffusion tensor fields. *IEEE transactions on visualization and computer graphics 22*, 1 (2015), 797–806. 3

© 2024 The Authors.
Proceedings published by Eurographics - The European Association for Computer Graphics.