Eurographics Symposium on Rendering 2019
T. Boubekeur and P. Sen
(Guest Editors)

Volume 38 (2019), Number 4

Tessellated Shading Streaming

J. Hladky [1] H. P. Seidel [1] M. Steinberger [2]

[1]Max-Planck-Institute for Informatics, Saarland Informatics Campus, Saarbrücken, Germany
[2]Graz University of Technology, Austria

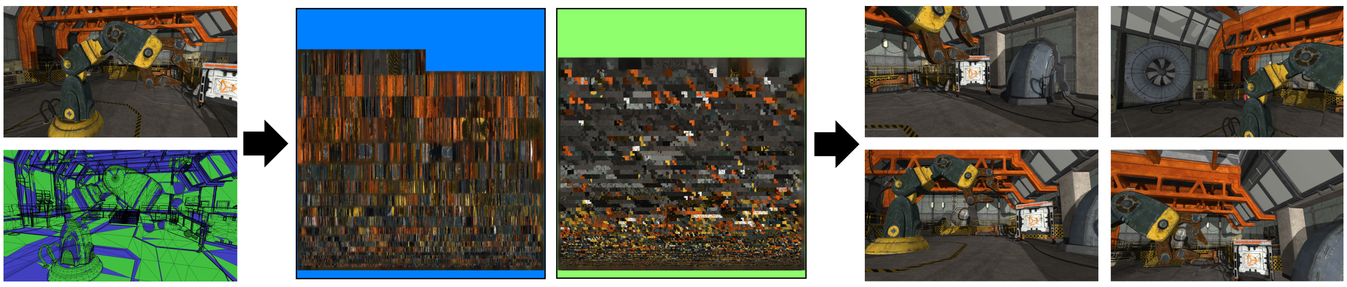


Figure 1: *We divide the scene geometry using triangle footprints to gather shading via Oversampling (blue) and L-packing (green) into a shading atlas. With this data, we construct near ground-truth novel views in under 1 ms on a desktop and with full 60 FPS on a smartphone.*

Abstract

Presenting high-fidelity 3D content on compact portable devices with low computational power is challenging. Smartphones, tablets and head-mounted displays (HMDs) suffer from thermal and battery-life constraints and thus cannot match the render quality of desktop PCs and laptops. Streaming rendering enables to show high-quality content but can suffer from potentially high latency. We propose an approach to efficiently capture shading samples in object space and packing them into a texture. Streaming this texture to the client, we support temporal frame up-sampling with high fidelity, low latency and high mobility. We introduce two novel sample distribution strategies and a novel triangle representation in the shading atlas space. Since such a system requires dynamic parallelism, we propose an implementation exploiting the power of hardware-accelerated tessellation stages. Our approach allows fast de-coding and rendering of extrapolated views on a client device by using hardware-accelerated interpolation between shading samples and a set of potentially visible geometry. A comparison to existing shading methods shows that our sample distributions allow better client shading quality than previous atlas streaming approaches and outperforms image-based methods in all relevant aspects.

CCS Concepts

• ***Computing methodologies*** $\rightarrow$ ***Rendering; Texturing; Virtual reality;*** *Image-based rendering;*

1. Introduction

Ideal virtual reality (VR) solutions should be able to deliver high-fidelity content at imperceivable latencies without any restrictions on the user's movement. Current solutions always fall short on at least one of these three aspects. Tethered head mounted display (HMD) systems such as Oculus Rift or HTC VIVE offer superb movement tracking and visual fidelity, but they restrain the user's movement by being bound by a thick cable to a high-end PC. Untethered solutions allow full freedom of movement. However, they only have a mobile GPU, e.g., Oculus Go, Microsoft HoloLens, Google Daydream, or Samsung Gear. Due to thermal and power restrictions mobile GPUs are unable to render high-fidelity content.

A solution to displaying high-fidelity content without movement restrictions is streaming rendering. Some vendors (Sony, Nvidia, Google) offer a commercial cloud gaming service over a wide area network. In this case, the bandwidth and latency of wireless networks are the main problem, as VR games have to be instantly responsive [CCT*11, MHUC12]. The latency and bandwidth requirements are barely met by edge servers [CWSR12] and wireless local area networks [LCC*15].

The introduction of wireless adaptors (HTC) further underlines the movement towards streaming the 3D content. While these adaptors offer unrestricted movement, they add additional weight to the already bulky HMDs and offer poor battery life.

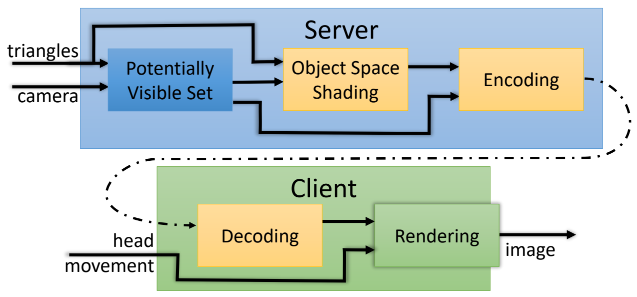


Figure 2: *Our split rendering pipeline for VR determines potentially visible geometry, performs object space shading and transmits the shading data to the client. The client uses the pre-shaded information to render novel views with no perceived latency to head movement. In this paper, we focus on shading gathering (yellow).*

Building on the fact that mobile GPUs are sufficient to display pre-shaded content, we propose a combination of object space shading [RKLC*11, HY16] with a texture atlas to split the traditional rendering pipeline between a powerful server and a lightweight client, as shown in figure 2: The server computes which geometry might become visible under head movements (determining a potentially visible set (PVS)). These triangles are sampled and shaded in object space and gathered in a texture atlas. The atlas is transmitted to the client, which renders novel views from this data until new shading information arrives. Concurrently with our work, Shading Atlas Streaming (SAS) [MVD*18], a similar approach has been published, underlining the importance of the topic.

Traditional object-space shading [RKLC*11] has the advantage that shading costs can be amortized for multiple objects [Bak16, Che15]. However, their global object-space map size is proportional to the scene size and thus it becomes too expensive to maintain and stream for large scenes [LD12]. Thus, streaming approaches must carefully manage texture space. Gathering shading in a predefined rectangular packing [MVD*18], achieves good texture space utilization, but creates subpar sample distributions which either lead to high bandwidth requirements or low quality.

In this paper we propose a novel method to dynamically gather shading samples akin to how geometry is actually sampled on screen. We do not cover the potentially visible set (PVS) computation here, as we have covered it in our previous work [HSS].

In our method the shading is computed and stored per triangle. The sample distribution is chosen on a per-triangle basis to best fit its screen-space footprint. After determining the color of the shading samples we store them into shading atlas in a representation that minimizes wasted atlas space. The client then uses the shading atlas to shade the PVS; thus providing novel views.

For shading gathering, we make the following contributions:

- **Tessellation dynamic parallelism**. We introduce a novel approach to leveraging the tessellation stages to dynamically spawn shading threads and infer additional data per tessellation-spawned sample. This allows to process the geometry in blocks of varying sizes within one dispatch, which is not possible with the current compute-mode execution on the GPU.
- **Two sampling strategies**. We propose two complementary strategies for distributing object-space shading samples on the surface of a triangle. For each method we present efficient encoding/decoding functions and mapping into a shading atlas.
- **Novel triangle representation**. We introduce a novel representation of triangle samples as L-shapes, which pack efficiently into one another and minimize the space in a shading atlas.

By having *full control* of the shading quality per primitive, we easily support per-object priorities, foveated rendering, and can reduce the shading sampling density when the PVS is large. Our sampling distribution strategies support effective *compression* and efficient *filtering* on the client.

We outperform traditional image-based and object-space shading strategies in terms of quality, speed, and memory. In comparison to SAS [MVD*18], we do not require preprocessing, achieve better quality, and avoid artifacts for slanted triangles. Our end-to-end implementation shows that our approach even works well on an untethered headset using an off-the-shelf smartphone.

2. Related work

Most related to our approach are the areas of object-space shading, image-based rendering (IBR) and remote rendering.

2.1. Object-space shading

Object-space shading is a popular alternative to image-space shading [Bak16], as it is able to exploit temporal and spatial coherence [RKLC*11]. Many object-space methods propose GPU extensions and can thus not be used in practice [BFM10, CTM13, CTH*14, AHTAM14]. On current hardware, a pre-charted texture per visible object can be used for simple scenes [Bak16]. For complex scenes, fine-grained visibility and pre-charted mip-mapped textures can be used [HY16]. Unfortunately, this is not suitable for streaming as large portions of the textures remain empty.

Considering alternative texture layouts, techniques like Ptex [BL08] and Mesh Color Textures [Yuk17] become interesting, as they map distinct samples to each triangle. As our approach dynamically chooses per-triangle sampling rates, there is a similarity to those approaches. Systematically, Reyes rendering [CCC87] is also related, as it dynamically determines the number of shading sampling on a per-primitive level. In any case, the major difference to all aforementioned approaches is that our approach does not require a predefined mapping to texture space or any specific shape. We dynamically determine the number and organization of per-triangle shading samples and map them into a texture.

2.2. Image-based rendering

Image-based rendering (IBR) is omnipresent to hide latency for VR headsets. Most well known is asynchronous time warping (ATW) [Ocu18], which hides the latency during rapid view offsets by warping the viewport inside a slightly overscanned framebuffer. ATW cannot reveal dis-occlusions and thus can only show perspectively wrong images to the user, which becomes apparent under high latencies or faster head movements. More advanced warping methods are able to handle full 3D motion [DER*10]. For example by represent the source view as a set of points [CW93], as layered depth images [SGHS98] or unstructured lumigraphs [BBM*01]. However, the most popular representation suitable for real-time warping remains a geometric proxy from the depth buffer. The shading samples are mapped to the proxy using projective texture mapping [MMB97]. To optimize the performance and reduce memory footprint, a geometric simplification can be applied [DRE*10,YTS*11,BMS*12,LCC*15]. These methods suffer from the resolution limits of the depth buffer.

2.3. Remote rendering

Remote rendering systems fundamentally differ by the data they transmit [SH15]. A minimal system transfers no geometry and thus forces the client to use pure ATW. The rendering data present on the server side can be used for color+depth compression [PHE*11] or efficient MPEG encoding [NCO03].

Using only the rendered depth buffer to derive the proxy geometry makes IBR independent of the scene complexity. The transmitted data can be represented as color+depth images, from which a novel view can be obtained by splatting or texture-mapping [CG02, SNC12]. This is compatible with speculative rendering [LCC*15] or residual image transmission [YN00, BG04, CWC*15]. Static scenes can be pre-processed into an IBR database, using view-dependent texture maps [COMF99], sparse [TL01] or dense [BCC16] impostors or geometry images [SMSW11].

In many video games the background geometry remains static. This fact can be exploited as the static geometry can be used directly for perspective texture mapping, optionally with further simplification [RKR*16]. Keeping the full model data on client and server allows to reduce transmission only to images after factoring in low-frequency illumination [Lev95], disocclusions [MCO97] or removing complex indirect illumination effects [CLM*15].

Similar to our approach Shading Atlas Streaming (SAS) [MVD*18] also splits the rendering pipeline between a sever and client. While previous split rendering approaches tried to temporally or spatially augment shading on a client [CWC*15], SAS reduces the geometry on the client and performs all complex shading on the server. While their pipeline is similar to ours, their shading atlas can be characterized as a dynamic version of virtual textures for rectangular shapes, similar to the ones in Far Cry [Che15]. Their main contribution is a dynamic memory management for those rectangular texture blocks in shader on the GPU. Using rectangular blocks comes with the downside that each mesh requires preprocessing into rectangular patches. Furthermore, when triangles are very small or elongated—which is natural during perspective projection—no good rectangular representation exists. Then, SAS wastes shading samples or falls back to lower quality.

Our approach does not require preprocessing and adapts to each triangle's shape on screen. Thus, it can handle completely dynamic geometry, e.g., supporting tessellation. Additionally, it does not introduce color bleeding between adjacent triangles, e.g., across edges on a cube. Most importantly, we never create excessive unused shading samples, as we always choose a fitting representation in the texture atlas. Thus, our results are sharper while using less memory and avoid visible artifacts across triangle boundaries.

3. Method

When gathering shading samples for streaming rendering into a shading atlas, we have the following objectives:

O1 The number of shading samples generated per primitive should be similar to the number of samples the primitive covers on screen. This is important to bound the shading and transmission costs to be similar to standard shading.

O2 The distribution of shading samples should be close to the samples generated for the expected view. Perspective transformations generate non-equal distances between samples which must be captured to generate sharp output images.

O3 Shading samples should be packed effectively into the texture atlas, using the available space efficiently. This is important to reduce bandwidth requirements during streaming and allows for more efficient encoding.

O4 The captured shading samples must allow for efficient reconstruction. As the client is usually light-weight, efficient reconstruction must be supported, e.g., using bilinear interpolation.

O5 Preprocessing should be avoided. Modern graphics content is highly dynamic, making use of tessellation and highly dynamic objects. Thus any preprocessing strongly limits the applicability.

To fulfill these objectives, we introduce two methods to sample shading on the surface of a triangle: *L-packing* and *Oversampling*. Both draw inspiration from the built-in hardware tessellation patterns of modern GPUs and are complementary: Oversampling is well suited for slanted triangles and L-packing for all other cases.

3.1. Dynamic parallelism and tessellation

To gather shading samples, similar to the way a triangle is projected for a reference frame (**O1**), one needs to dynamically adjust the shading samples generated per triangle. Traditionally, one would use rasterization to invoke fragment shader instances for all samples. However, this only works efficiently, if the samples are generated continuously and can be covered by a triangle. As we will show later, violating these prerequisites has advantages.

Using standard compute execution to spawn threads on the GPU would require a complex approach. One would have use multiple kernel launches, where the first one determines the total number of samples per triangle and writes out to global memory. Then, a prefix sum could determine and allocate thread blocks of varying sizes. Block communication and synchronization would be necessary to handle corner cases, overall slowing performance significantly.

Recently, Nvidia introduced mesh shaders [Mou18]. While they allow to dynamically launch workers per input meshlet, there are two major restrictions. First, current hardware internally uses only a single group size to be launched: 32 threads. Thus, mesh shaders cannot be used effectively for small triangles, i.e., if only a few samples are needed. Second, even if a different group size were supported, it must be constant for the entire launch. Thus, the sample count cannot dynamically adjust to a triangle's screen size.

Finally, tessellation provides a way to dynamically launch

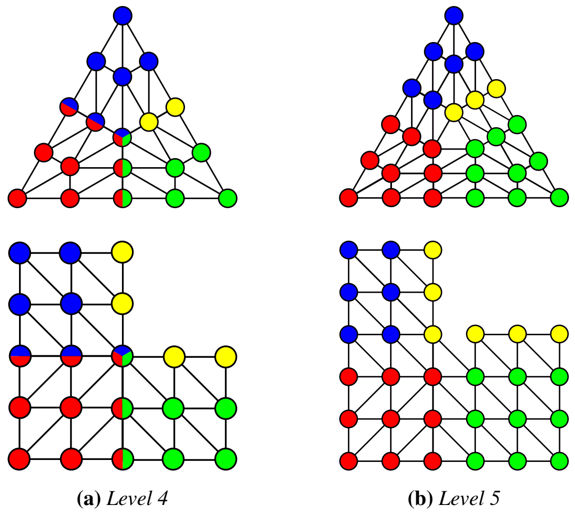


Figure 3: *The vertices of the tessellation pattern correspond to the locations for sample gathering. To pack the samples into an L-shape, we cut the triangle open along the yellow samples and duplicate them to allow for interpolation. We duplicate at most 32 samples per triangle using the highest available tessellation level.*

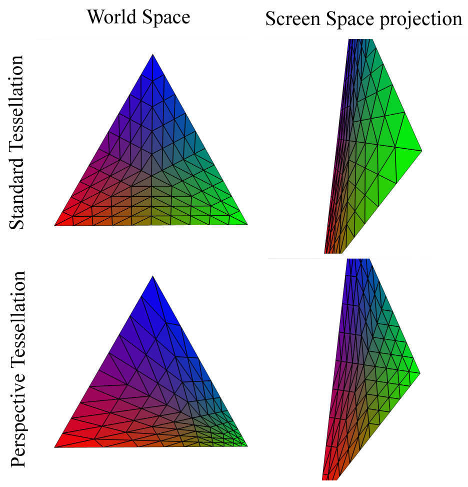


Figure 4: *The effects of perspective-corrected sample positions [MHAM08]. Note that we cannot achieve true screen-space uniformity on the red-blue edge of the triangle in the bottom right.*

threads using hardware acceleration. However, the control over the tessellation pattern of a triangle is limited. Users can specify four tessellation factors $\mathbf{L} = \begin{bmatrix} l_{o0} & l_{o1} & l_{o2} & l_i \end{bmatrix}$ ranging from 1 to 64 to control the subdivision along the three edges of the triangle ($l_{o0} - l_{o2}$) and the number of inner circles (l_i). A method for specifying an exact number of threads spawned or a sample id within a patch is not provided.By in-depth analysis of the tessellation patterns, we have derived formulas for computing this additional data, which allows us to use the dynamically launched threads of the tessellation stage for our needs. The only remaining limit is the number of threads spawned per patch—3169 on current hardware (all four levels set to 64). For our shading approach we add a pre-tessellation step to alleviate this limitation.

To reduce complexity, we always set all four tessellation factors to the same value l. The outer tessellation factors would only change the number of threads (vertices) spawned for the respective edges, but not influence the inner pattern, which is the major source for spawning threads. This yields the following equations for spawned threads (vertices) $v(l)$ and generated triangles $t(l)$:

$$v(l) = 3\left(\left\lfloor\frac{l}{2}\right\rfloor + 1\right)\left(\left\lfloor\frac{l}{2}\right\rfloor + l \bmod 2\right) + (l+1) \bmod 2, \quad (1)$$

$$t(l) = l \bmod 2 + 3(l + l \bmod 2)\left\lfloor\frac{l}{2}\right\rfloor. \quad (2)$$

3.2. L-packing

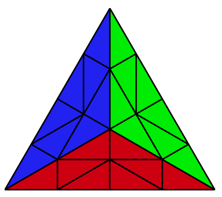
L-packing is best suited for near-equilateral triangles. It takes inspiration from the uniform triangle tessellation pattern and mesh color textures [Yuk17]. Each vertex spawned by the tessellator represents a sample on the surface of the triangle, which we directly use to sample the shading information (figure 3). This sampling pattern is well suited to represent shading on a triangle, as samples are placed on the triangle edges and are naturally distributed over the triangle [Yuk17]. Thus, shading information can be reconstructed for any point within a triangle.

Using this type of sample distribution is arguably well suited if a equilateral triangle is viewed straight on. However, the more a triangle's shape deviates from an equilateral, the less uniform the samples are distributed in object space. More importantly, perspective projection not only changes the triangle shape, but also the between sample distance (**O2**). To counteract this effect, we adjustment the samples considering the perspective projection [MHAM08], which increases the screen-space uniformity of the pattern (see figure 4). Although the sample locations are moved, the connectivity of the tessellation does not change and thus the pattern shown in figure 3 still holds—which we will use for encoding. Note that this perspective adjustment is only possible when working on a triangle basis and not on patches [MVD*18].

Encoding & Atlas layout Analyzing the pattern generated by the uniform tessellation (see figure 3), one can see an inherent quad structure underlying the tessellation. Thus, locally bilinear interpolation can be used to interpolate between four adjacent sample locations. This points towards an efficient way of encoding (**O3**) and interpolating the shading data (**O4**). The underlying quad structure can be brought forth by identifying six different sections of a triangle and cutting it along one subsection, which unfolds the triangle into an L-shape, as shown in figure 3.

We introduce a lightweight function that maps from a sample's barycentric coordinates to its position within the L-shape. There are two ways of identifying the sample's location on the triangle based on the maximum and minimum barycentric coordinates as illustrated in figure 5. For the given tessellation level l we can determine two key distances in the barycentric coordinates space: inner distance i and outer distance o:

$$o(l) = \frac{1}{l}, \qquad i(l) = \frac{2}{3l}. \quad (3)$$


(a) *Side identification*

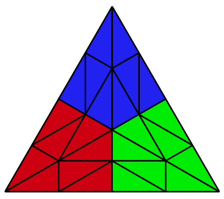

(b) *Tip identification*

Figure 5: *Given a sample on the surface of the triangle we identify different areas where the sample lands based on its barycentric coordinates. We distinguish between three sides (a) and three tips (b) used in Oversampling and L-packing, respectively.*

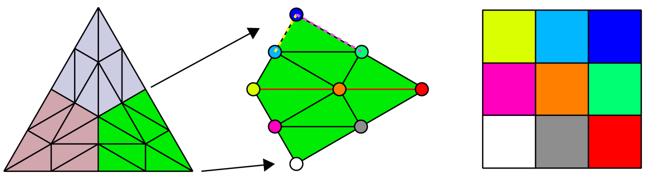


Figure 6: *In-detail view on samples spawned for the green tip of the triangle (left). (middle) The yellow dashed line represents the inner distance $i(l)$ and the magenta dashed line the outer distance $o(l)$ (equation 3). The red line depicts the block's diagonal under which we flip the row/column coordinates for atlas mapping (equation 4). (right) The samples are arrangement into a block; three such blocks (one for each tip) form a full L-shape of a triangle.*

These distances and the corresponding block-layout of the samples belonging to the green tip are depicted in figure 6.

Each sample generated by the tessellation pattern can be uniquely identified by its barycentric coordinates $\mathbf{B} = \begin{bmatrix}\lambda_1 & \lambda_2 & \lambda_3\end{bmatrix}$ where $\lambda_3 = 1 - \lambda_1 - \lambda_2$. For the barycentric coordinates $\mathbf{B}$, we identify the minimum, median and maximum values $\begin{bmatrix}\lambda_{min} & \lambda_{med} & \lambda_{max}\end{bmatrix}$ and obtain the corresponding row and column indices by:

$$row(\mathbf{B},l) = \frac{\lambda_{min}}{i(l)} \qquad column(\mathbf{B},l) = \frac{\lambda_{med} + \frac{\lambda_{min}}{2}}{o(l)} \tag{4}$$

For samples under the diagonal (red line in figure 6) the row and column indices must be flipped. Thus, we achieve a nice packing of samples from one section of the triangle into its corresponding square. The dimension of this square is

$$dim(l) = \lceil (l+1)/2 \rceil.$$

Then, we offset the squares based on the triangle section to which they belong (determined by the maximal barycentric coordinate). The blue section has its height offset by $dim(l)$, the green section has its width offset by $dim(l)$ and the red section stays. To achieve efficient packing of neighboring triangles in the atlas, we flip every other L-shape. The final result can be seen in figure 1.

As the triangle is cut open, we need to duplicate one line of samples. In the case of even tessellation levels, the samples that fall on the yellow dashed line in figure 3 are duplicated to both hands of the L-shape, resulting in a symmetric L. For uneven tessellation levels, the cut falls in-between two sample rows, thus we duplicate one of the two rows to the other hand of the L (also see figure 3b). Thus, one hand of the L becomes one row thicker than the other.

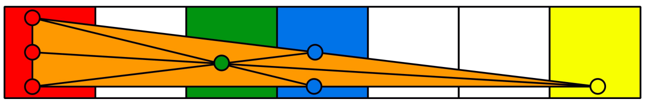


Figure 7: *An example of a very slanted triangle where the L-packing does not perform well: The triangle covers seven pixels, however the built-in tessellation pattern spawns samples at the depicted locations, which leaves vital parts of the triangle unshaded.*

Note that the samples are generated directly in the tessellation evaluation shader and are written to the texture from there. The rasterizer does not run. Generating very small triangles and forwarding them to the rasterizer to write to the texture would be inefficient, especially considering quad-shading.

Single-pixel primitives Using the smallest tessellation factor ($l = 1$) creates three samples. For very small triangles this is excessive (**O1**). To this end, we write single pixels directly to the texture in the tessellation control shader for tiny triangles. We then set $l = 0$ yielding a discard before tessellation evaluation.

Decoding On the client, decoding happens on a per fragment basis. We first obtain the barycentric coordinate of the fragment, but since we used the perspective correction [MHAM08], we apply an inverse perspective-encoding function. Starting from the barycentric coordinate $\begin{bmatrix}u & v & 1-u-v\end{bmatrix}$ we obtain inverse-encoded barycentric coordinates $\mathbf{B}^{-1} = \begin{bmatrix}u' & v' & 1-u'-v'\end{bmatrix}$ using:

$$\begin{aligned} u' &= \frac{uZ_1}{Z_0(1-u-v)+uZ_1+vZ_2} \\ v' &= \frac{vZ_2}{Z_0(1-u-v)+uZ_1+vZ_2} \end{aligned} \tag{5}$$

To assign the correct shading atlas location for the given fragment, we apply equation 4 to $\mathbf{B}^{-1}$ to identify the sample's correct location within the L. This supports fast lookups in novel view synthesis and allows for hardware-accelerated bilinear interpolation (**O4**).

3.3. Oversampling

The proposed L-packing approach is not suitable for very slanted triangles for which we get sub-optimal sample counts (**O2**) and/or distributions (**O3**) due to the nature of the tessellation pattern, as shown in figure 7. To decide whether a triangle can be handled with L-packing, we measure the ratio between the longest edge of the projected triangle and its height. This yields the "slantedness" ratio, which we threshold to decide between the two approaches.

Oversampling aims to find a simple arrangement of a projected triangle in the shading atlas, as shown in figure 9. Again, we need to dynamically spawn shading threads for this purpose, for which we again exploit tessellation. We derive equations for unique sample identification within the tessellation evaluation to obtain a linear thread id, which then allows us to create custom sample patterns for any triangle. Oversampling requires placing samples outside of the

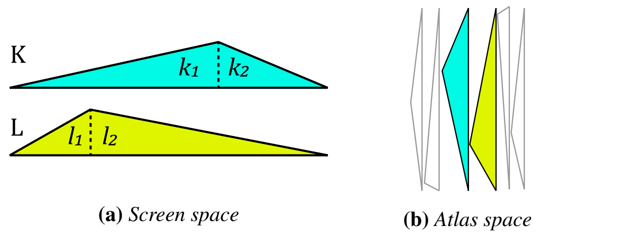


Figure 8: *Two slanted triangles K and L are split into right-angled triangles k_1,k_2 and l_1,l_2, respectively and encoded in the shading atlas next to another.*

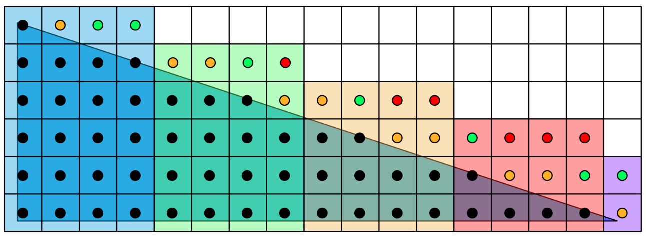


Figure 9: *An example of a slanted triangle with $w = 17$ and $h = 6$. Sample coloring indicates the challenges: black inside, orange spawned by conservative rasterization, green required for bilinear interpolation, red added by us to simplify computations. There are four blocks of height four (colored blue, green, yellow and red); the purple block is computed from the residual triangle length.*

triangle in order to allow efficient computation of the required sample count and hardware-accelerated interpolation over the whole surface of the triangle.

We do not use the rasterizer for Oversampling due to two reasons. First, samples outside of the triangle are required for bilinear interpolation, which not even conservative rasterization provides with any setting of current hardware-supported implementations. Second, using tessellation for both L-packing and Oversampling allows for a unified approach and reduce the number of passes.

To spawn an arbitrary number of samples, we mainly use the inner tessellation factor of the triangle which generates an exponential number of samples. We use the other tessellation factor to select threads between two exponential steps. To receive a consecutive thread id, we derive the inner layer of each sample using the previously derived inner and outer distances (equation 3):

$$l(\mathbf{B},\mathbf{L}) = \frac{\lambda_{min}}{i(l_i)}.$$

Summing the number of threads in all inner layers plus adding an offset within each layer, unrolls the layers into a consecutive thread id I. To determine the order in each layer, we again rely on tip and side identification, and compute the local id for each side using:

$$os(\mathbf{B},\mathbf{L},t) = \left\lfloor \left(t - l(\mathbf{B},\mathbf{L})i(l_i)\right)l_i \right\rceil, \tag{6}$$

where t is the outer tessellation level (see figure 5a). For a global id we identify the sample count in all previous layers as:

$$p(\mathbf{B},\mathbf{L}) = v\left(2\left(\left\lfloor \frac{l_i - 1}{2} \right\rfloor - l(\mathbf{B},\mathbf{L})\right) + \left\lfloor \frac{l_o + 1}{2} \right\rfloor\right) \tag{7}$$

and finally obtain the thread id I as:

$$I = O(\mathbf{B},\mathbf{L}) = o(\mathbf{B},\mathbf{L}) + p(\mathbf{B},\mathbf{L}) \tag{8}$$

For a more in depth derivation including formulas handling corner cases, see the supplemental material. The final formula is a non-recursive function and can be implemented as a series of simple arithmetic operations on GPU code without even using a loop.

Encoding & Atlas layout When a triangle is classified for Oversampling, we split it into two right-angled triangles based on its longest edge (base) and its corresponding height, as shown in figure 8. We compute the pixel coverage of base and height in screen space, which then gives the triangle's pixel-width w and pixel-height h. We use these values to derive the number of samples needed.

It is far from trivial to identify the exact number of pixels required for bilinear interpolation, i.e., determining the exact one neighborhood of texture samples that can be hit by any position within a triangle. To this end, we introduce additional samples conservatively to enable efficient computation. According to our tests, these additional samples stay within 25 to 35% and mostly stem from very small triangles, which notoriously show a high oversampling.

Our conservative overestimation relies on rectangular blocks, as shown in figure 9. Using w and h, we compute a block height R and block count P:

$$R = \left\lceil \frac{w}{h} \right\rceil, \qquad P = \frac{h}{R}. \tag{9}$$

These hold true for $w > h$ (slopes $s > 1$). If $s \leq 1$, we simply flip the width and height. The total number of samples is computed as:

$$r = R \cdot P \cdot (w - (P+1)/2) + 2h + w + 1 \tag{10}$$

After computing I, w and h we get the x,y coordinates within a triangle's grid as:

$$\begin{aligned} x &= b_{id}(I,w,h) \bmod b_w(I,w,h), \\ y &= \frac{b_{id}(I,w,h)}{b_w(I,w,h)} + \left\lceil \frac{h}{w} \right\rceil \cdot b(I,w,h), \end{aligned} \tag{11}$$

where $b_{id}(I,w,h)$ calculates the id of the sample within the current block (figure 9) and $b_w(I,w,h)$ calculates the current block width.

We need to transform the x,y grid coordinates onto the triangle's surface, i.e., into barycentric coordinates. We first obtain the ratio at which the longest side of the triangle is split. For triangle $T = (\mathbf{p}_0, \mathbf{p}_1, \mathbf{p}_2)$ with $\mathbf{p}_i = \begin{bmatrix} x_i & y_i & z_i \end{bmatrix}^T$ and $\mathbf{a} = p_1 - p_0, \mathbf{b} = p_2 - p_0$ we compute the split ratio as:

$$a(T) = \frac{\left| \frac{\mathbf{a} \cdot \mathbf{b}}{|\mathbf{a}|^2} \mathbf{a} \right|}{|\mathbf{a}|} \tag{12}$$

The resulting barycentric coordinates $\lambda'_1, \lambda'_2, \lambda'_3$ are computed as:

$$\lambda'_1 = \frac{y}{h} \qquad \lambda'_2 = a(T)(1-\lambda'_1) \pm \frac{x}{r} \qquad \lambda'_3 = 1 - \lambda'_2 - \lambda'_1.$$

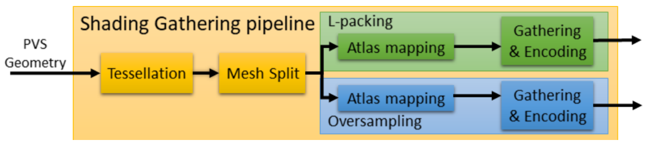


Figure 10: *Simplified diagram of the shading gathering stages.*

Since in the first step we split the triangle according to the longest side and its corresponding height, the $\pm$ sign in λ_2' stands for the left/right half of the slanted triangle. In the end, we again apply the perspective-correction offset to the new samples [MHAM08].

Decoding To locate each triangle in the atlas, we keep a buffer containing each triangle's dimensions and position in the shading atlas. Since we applied the perspective-correction at the end of sample gathering, we again modify the triangle's barycentric coordinates according to equation 5 for the texture lookup.

4. Implementation

We have implemented our pipeline, as shown in figure 10, using a combination of CUDA compute stages and OpenGL. The input to our method is a PVS computed for the current view cell [HSS].

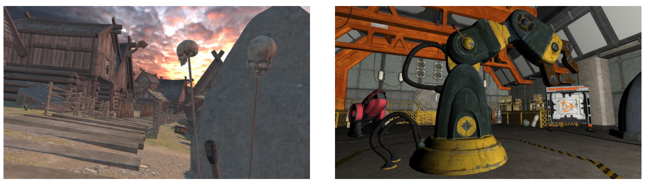
PVS Tessellation Since the number of samples be spawned by the tessellator is limited (typically by a tessellation factor of 64), we perform a first tessellation to split those triangles that go beyond this limit. We capture those triangles using transform feedback.

Mesh Split Next, we determine whether a triangle is subject to L-packing or Oversampling using the aspect ratio of the triangle's longest edge and its height in screen-space.

Atlas Mapping We determine the location of each triangle in the shading atlas using atomics, by first binning each triangle and then computing its offset in the bin. For L-packing, we use the triangle's tessellation level for binning (resulting in 64 bins) and an atomically increased linear counter for its location within the bin. For Oversampling, binning uses the triangle's base length and its height to compute the offset within the bin, as shown in figure 8.

Sample Gathering + Encoding In this stage both L-packing and Oversampling execute OpenGL's tessellation shaders to spawn the necessary number of shading samples as detailed before and write the shading data to the atlas. Finally, we use JPEG encoding on the completed atlas [Wal91]. We can ignore empty spaces in the atlas when computing the coefficients within individual JPEG blocks, efficiently using the available bandwidth without introducing sharp triangle boundaries, as shown in figure 1. Aligning triangles to 8x8 compression blocks could increase quality. However, as triangles with similar materials and color are naturally placed close to another (due to primitive order), we did not notice a need to enforce such boundaries, making more efficient use of the available space.

Decoding In order to determine the correct shading atlas locations, L-packing requires information about the tessellation level of a triangle and its position in the bin, which can be packed into four bytes of data per primitive. For decoding, Oversampling requires 16 bytes per primitive—its width/height and its position.



(a) *Viking Village (4.6M triangles) 1 shadow map* **(b)** *Robot Lab (472k triangles) 3 shadow maps*

Figure 11: *Examples of the tested scenes. Viking Village is a large outdoor scene, Robot Lab is a smaller indoor scene. The scenes' shading consists of image-based lighting computation, soft shadow maps and tone mapping. Robot Lab has more demanding shading computation than Viking Village.*

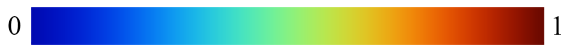


Figure 12: *The linear gradient used for color mapping of DSSIM.*

5. Results

To evaluate our approach, we test two scenes: Robot Lab (RL) and Viking Village (VV). For both scenes we set up complex shaders to highlight the costs of the different approaches as shown in figure 11. All tests were run on an Intel Xeon CPU E5-2643 @ 3.40 GHz with 32 GB of RAM and an NVIDIA Titan Xp. We consider a head movement of up to 30 cm and rotations of up to 120 degree. The camera fov is 60 degree. All client renderings are performed in 1920×1080. The configurations for our walkthroughs and the average size of the potentially visible geometry generated is shown in table 1. For all tests, the input is a PVS computed for a given view cell, which was created by sampling 256 views in 4K resolution. The shading atlas for the walkthrough is a 24 Mpix texture. We compare our method to Texel Shading [HY16] (TS) and Shading Atlas Streaming [MVD*18] (SAS). All quality comparisons use DSSIM with the color gradient in figure 12 for visual comparisons.

5.1. Computation time

In order to assess the overhead of the different methods, we measured the average per-sample computation time, as shown in ta-

Table 1: *Our test walkthroughs use three PVS configurations, with increasing translational and rotational movement support. The resulting PVS geometry ranges from 60k triangles up to 160k (percentages provide the average amount of slanted triangles).*

Label	Transl	$\sphericalangle$	Robot Lab	Viking Village
p1	10 cm	40 deg	64.3 K (30%)	109.6 K (41%)
p2	20 cm	80 deg	82.5 K (32%)	135.0 K (41%)
p3	30 cm	120 deg	97.7 K (33%)	158.7 K (41%)

Table 2: *The per shading sample cost in nanoseconds shows the overhead of the shading techniques. Robot Lab has more complex shading but less overdraw than Viking Village. L-Packing (LP) and SAS are pretty close to forward shading (FW) for Viking Village. Oversampling (OS) and Texel Shading (TS) show more overhead. For Robot Lab, all methods show a higher relative overhead.*

Scene	FW	LP	OS	TS	SAS
RL	1.2 ns	3.05 ns	8.48 ns	11.21 ns	3.10 ns
VV	0.74 ns	0.75 ns	1.86 ns	1.35 ns	0.71 ns

ble 2. Forward rendering (FW) shows that RL has a higher shading cost than VV, which is mostly due using three shadow maps. On the other hand, VV has more overdraw, which slightly reduces the shading efficiency of FW in VV. In comparison, SAS and LP, show nearly the same per sample cost in VV. Note that SAS essentially renders non-overlapping quads into the atlas, resulting in nearly perfect conditions for the hardware. Thus, the performance of LP is highly competitive. For RL (where FW shows less overdraw overhead), the additional costs of LP and SAS become apparent, increasing the cost of these object space shading approaches to nearly $3\times$ FW. OS shows a $2-3\times$ overhead in comparison to LP, which underlines the cost of more complex id computations. Furthermore, texture access might be less efficient, as there is no guarantee the neighboring samples are executed on the SIMD lanes on the GPU. Finally, TS shows significant overheads, especially for RL.

5.2. Texel Shading Comparison

Due to the limitations of texel shading, we split the comparison to SAS and TS. TS requires unique texture spaces for all objects. Usually, the same textures (and even the same objects) are used multiple times in a scene, e.g., wall textures in RL and buildings in VV. To perform a meaningful comparison, we have re-baked the textures for both scenes into a big unique texture. Of course, this step blows up texture memory and reduces texture quality. In comparison to our approach, TS additionally needs an inverse texture mapping from texels to primitive ids. This disallows any dynamic mesh refinement and increase memory requirements further. Additionally, the mipmapped textures are only partially filled.

Table 3 shows the comparison between TS and our approach. Clearly, TS requires nearly an order of magnitude more time for shading and generates $5\times$ more shading samples. Still, it does not reach our quality in RL and is significantly worse in VV. As TS always shades in blocks of 8×8 pixels, they generate many shading samples which are not necessary. Our better image quality is due two facts: First, our sample generation is independent of the texture resolution and thus can capture shading information at arbitrary resolution. Furthermore, the mipmapping in TS may mix color information from different triangles, resulting in color bleeding.

Image quality comparisons to TS are given in figure 13, which clearly show the aforementioned issues. The only advantage of TS is efficient decoding, where a simple texture lookup suffices, whereas we need to consider L-packing and adjust for perspective effects. Nevertheless, our decode performance is sufficient to render with any currently supported display frame rate. Our approach is also suitable for streaming, while TS only partially fills the textures, which would blow up encoding times and bandwidth. Overall, our approach is superior in all relevant characteristics.

5.3. Shading Atlas Streaming Comparison

To compare to SAS, we use the original textures, which results in sharper images than in the TS comparison. As SAS organizes its texture atlas in blocks, it requires preprocessing to merge triangles into rectangles, which also prohibits dynamic geometry. Furthermore, their sample distribution aims to be uniform on a triangle, but as multiple triangles are combined within a block, their respective sample choices influence one another. As the sizes of different blocks in SAS' shading atlas are limited to powers of two, fine grained size selections are not possible. Finally, SAS does not consider perspective effects when choosing sample distributions.

These considerations are reflected in the results, shown in table 4. On average, SAS generates more samples than ours, is faster during sample gathering, but achieves lower overall quality. Interestingly, the number of samples collect by SAS remains mostly constant, while our approach adapts to the PVS size. While we keep the image quality approximately constant, SAS loses quality with increasing PVS size. As can be seen, the performance difference mainly comes from Oversampling being less efficient than L-packing. While Oversampling generates fewer samples than L-packing it requires more than twice the shading time. Note that we did not observe any temporal instability artifacts when a triangle switches between L-packing and Oversampling as we achieve a sufficient sampling density in both approaches.

Although SAS generates more samples, their lower quality hints that they do not use these samples effectively. They add unnecessary samples by always shading complete blocks. At the same time, they miss samples where perspective effects and slanted triangles would require denser sampling. These cases are clearly seen in Figure 14. Note that SSIM does not heavily punish those, although they are very visible when comparing the methods side-by-side. Overall, our approach is the more flexible as it works independently of a scene's triangulation quality and projection. The prize for our superior quality is an increased cost for shading gathering, which however happens on the server and may thus be tolerable.

Our decode time on the client is significantly lower than SAS. For L-packing, we compute atlas positions in the vertex shader and require few arithmetic operations to convert barycentric coordinates to the texture coordinates in the fragment shader (Equation 4). For Oversampling, we directly use the interpolated texture coordinates. SAS decodes block locations from a bit field and needs to recover texture coordinates from the rectangular packing established during preprocessing, which results in the observed overhead.

While it might appear that Oversampling is costly compared to L-packing, it is necessary to achieve consistent high quality, as can be seen in figure 15. In this example, the pipe consists of few very slanted triangles spanning over its whole length. L-packing would generate blurry artifacts; Oversampling handles these cases well.

Table 3: *The comparison to Texel Shading (TS) clearly shows the advantage of our approach. We gather significantly fewer samples (sum of L-packing (LP), Oversampling (OS) inside and outside), in way less time (total time) and achieve better quality (DSSIM - lower is better). TS achieves better decode times (in ms), however our times are also sufficient to easily achieve 120Hz on current devices.*

Config.	**Gather and encode**		**Total**	**Samples gathered**			**Our**	**Decode**	**Texel Shading**			
	LP	OS	**time**	LP	OS in	OS out	**DSSIM**	**time**	Samples	Time	DSSIM	Decode
RL-p1	26.38 ms	54.41 ms	83.36 ms	8.2 M	3.33 M	1.70 M	0.0143	0.41 ms	117.79 M	1779 ms	0.0191	0.24 ms
RL-p2	42.25 ms	83.84 ms	128.64 ms	13.7 M	5.42 M	2.71 M	0.0144	0.43 ms	139.11 M	1894 ms	0.0191	0.25 ms
RL-p3	55.91 ms	114.68 ms	173.21 ms	18.2 M	7.30 M	3.69 M	0.0144	0.45 ms	153.33 M	2003 ms	0.0192	0.25 ms
VV-p1	4.05 ms	17.25 ms	24.06 ms	4.9 M	3.62 M	1.24 M	0.016	0.35 ms	136.12 M	1347 ms	0.0729	0.14 ms
VV-p2	5.87 ms	24.44 ms	33.09 ms	7.9 M	5.25 M	1.73 M	0.0158	0.37 ms	145.44 M	1217 ms	0.0727	0.14 ms
VV-p3	7.51 ms	30.4 ms	40.69 ms	10.6 M	6.58 M	2.17 M	0.0159	0.38 ms	154.61 M	1135 ms	0.0722	0.15 ms

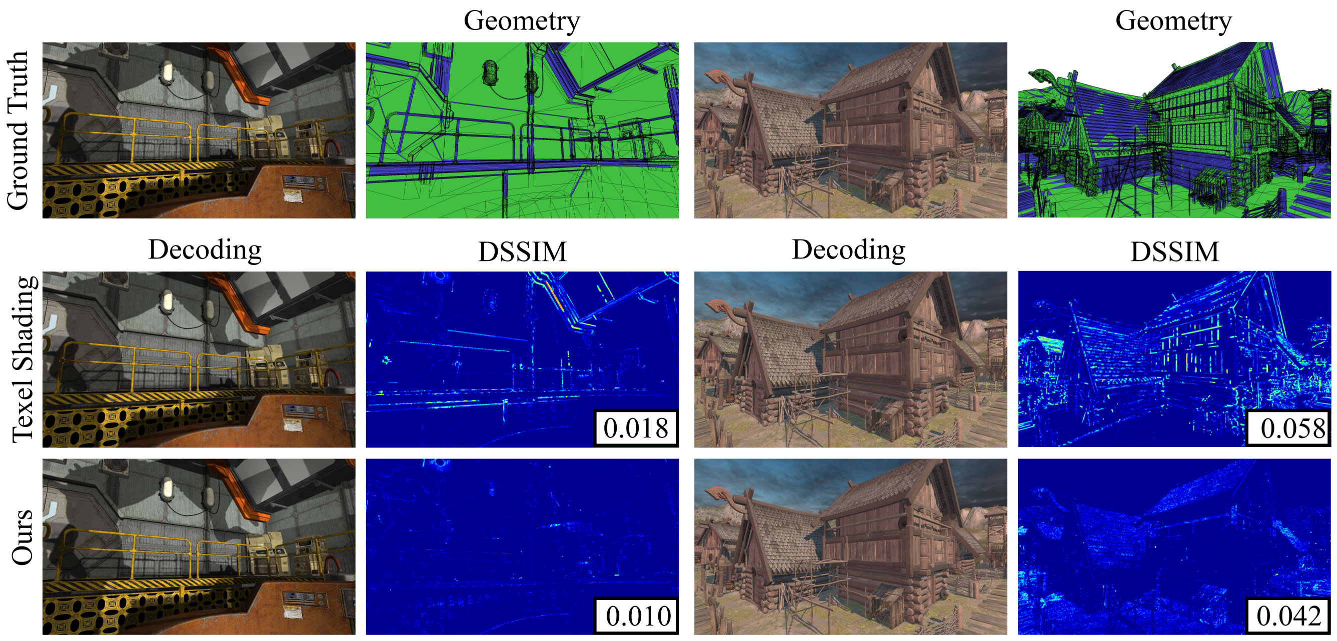


Figure 13: *The comparison between Texel Shading and our approach (L-Packing in green, Oversampling in blue) clearly shows the quality differences. For example, TS does not capture the high frequency detail on the railing in RL and shows clear quality issues along small and thin triangles, as on the sides of the buildings in VV. Our approach generates significantly sharper images and shows more uniform quality distributions. Please zoom in for details.*

Table 4: *The comparison to Shading Atlas Streaming (SAS) clearly shows that our approach clearly adapts more to the different usecases, increasing load and samples as the PVS increases and always achieves consistent quality. SAS generates significantly more shading samples but does not reach our quality. SAS is between* 1.4× *slower and* 2.5× *faster, depending on the amount of Oversampling we perform.*

Config.	**Gather and encode**		**Total**	**Samples gathered**			**Our**	**Decode**	**Shading Atlas Streaming**			
	LP	OS	**time**	LP	OS in	OS out	**DSSIM**	**time**	Samples	Time	DSSIM	Decode
RL-p1	26.71 ms	54.27 ms	83.52 ms	8.2 M	3.34 M	1.70 M	0.1208	0.42 ms	14.6 M	57.12 ms	0.1346	1.02 ms
RL-p2	42.99 ms	85.48 ms	131.01 ms	13.7 M	5.44 M	2.71 M	0.1209	0.43 ms	13.6 M	55.19 ms	0.1478	0.91 ms
RL-p3	55.77 ms	109.71 ms	168.01 ms	18.3 M	7.31 M	3.69 M	0.1208	0.45 ms	18.6 M	66.21 ms	0.1813	0.81 ms
VV-p1	4.29 ms	17.93 ms	24.95 ms	5.2 M	3.86 M	1.30 M	0.0711	0.39 ms	21.6 M	34.69 ms	0.0679	1.03 ms
VV-p2	6.31 ms	26.08 ms	35.16 ms	8.4 M	5.71 M	1.87 M	0.0712	0.4 ms	17.3 M	33.19 ms	0.0816	0.87 ms
VV-p3	7.99 ms	32.79 ms	43.58 ms	11.1 M	7.20 M	2.38 M	0.0712	0.42 ms	20.2 M	35.74 ms	0.1526	0.77 ms

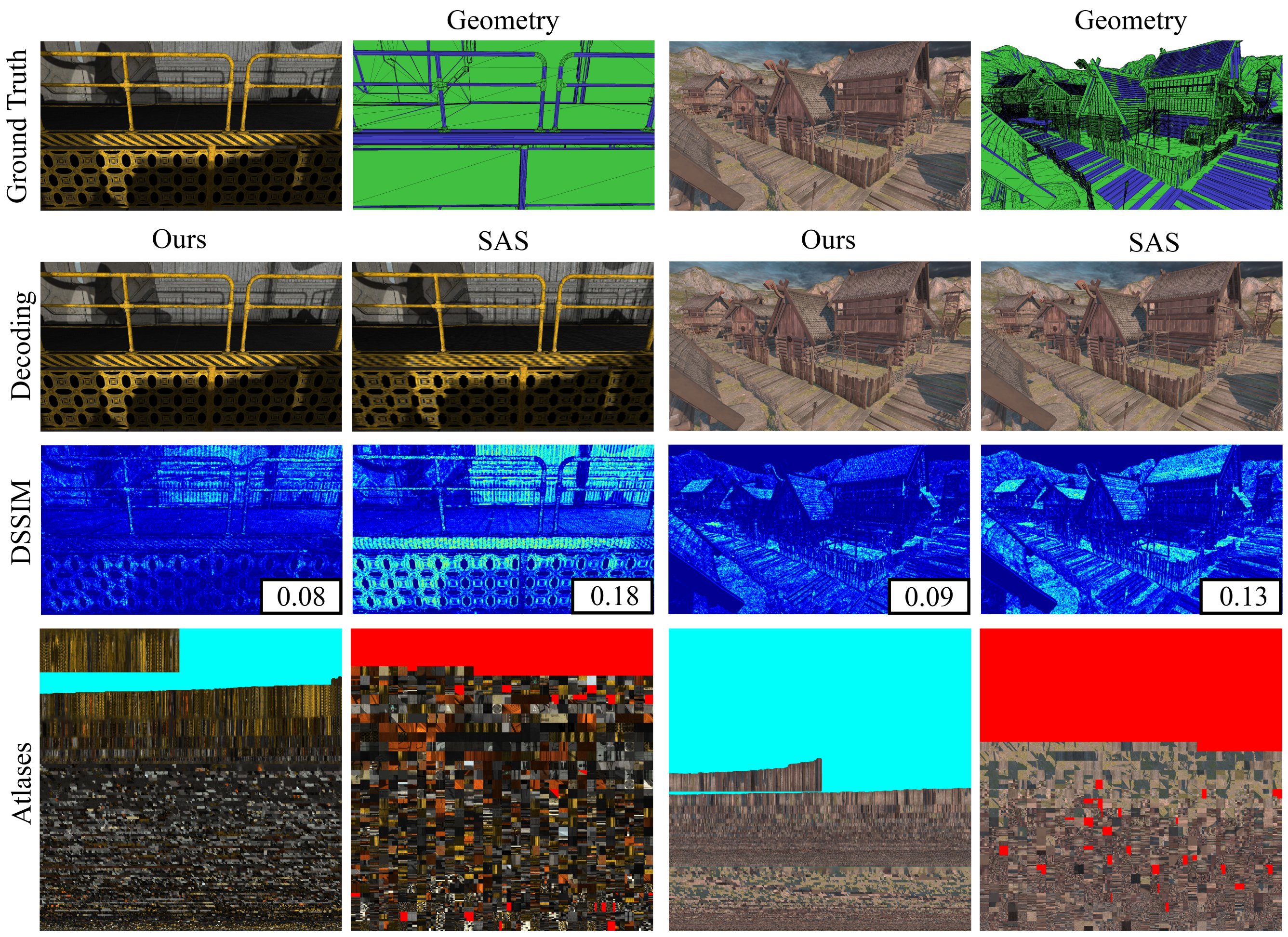


Figure 14: *The comparison between Shading Atlas Streaming and our approach (L-packing in green, Oversampling in blue) highlights the issues when capturing shading samples in rectangular blocks. The weaknesses of SAS are clearly visible when patches are large (lattice structure in RL) and triangles are slanted (railing in RL and roof in VV). Our flexibility in using the atlas space for different shapes is clearly visible when comparing the atlases. Please zoom in for details.*

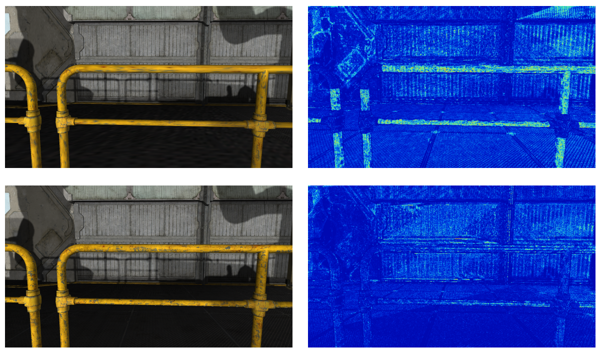


Figure 15: *An example where Oversampling (bottom) handles slanted triangles better than L-packing (top). The right column shows color-mapped DSSIM with the ground truth.*

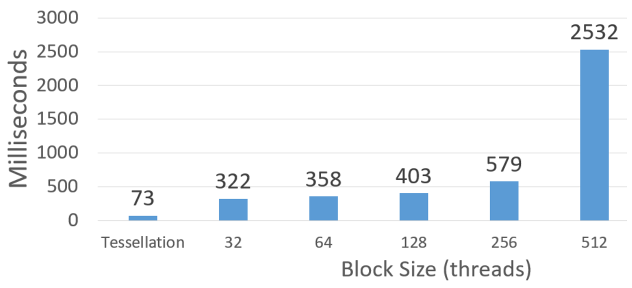


Figure 16: *The performance of gathering and encoding 5.64 M samples with Oversampling in Robot Lab: Using tessellation to dynamically spawn threads (left most ba), is clearly superior to using compute shaders with any fixed block size (32-512 threads).*

(a) *Intel compute stick (2015) HD Graphics 515* **(b)** *Google Daydream + Smartphone with Adreno 540*

Figure 17: *Thin client devices used for the streaming experiment. The Daydream is an untethered smartphone-based headset showing a frame from Robot Lab rendered at 2880×1440.*

5.4. Dynamic parallelism comparison

To evaluate the performance gained by using the tessellation stages, we decided to implement the Oversampling shading strategy using OpenGL compute shaders. The measurements in figure 16 show how fixed block size compute execution performs sub-optimally for tasks requiring dynamic workloads.

5.5. Real-life testing

To demonstrate the real-time capabilities of our method, we have implemented a client prototype running on a smartphone with Adreno 540 (early 2017) and on an Intel compute stick (thumb-sized PC) with Intel HD Graphics 515 (late 2015) shown in figure 17. The numbers we provide are for Robot Lab and Viking Village, respectively. For computing the visibility on the server we use the COS method [HSS]. The server runs with in average 10fps with the visibility pass taking about 70/80ms and shading pass 20ms. The compressed geometry updates come down to about 20KB/frame for either scene resulting in 4Mbps bandwidth at 10fps. The JPEG compression of our 8 Mpix atlas achieves a good quality with 40Mbps. With such timings we fit well within the bandwidth limitations of the current 802.11ac standard. The combined rendering and transfer latency over Wi-Fi comes down to about 150ms. The Adreno 540 can render this amount of geometry with 60 fps (the maximum supported refresh rate of the smartphone). Although considerably weaker, the compute stick still achieves 25fps.

6. Limitations and future work

One of the limitations of our method is a sub-optimal packing of triangles into the Oversampling shading atlas (see figure 1). A tighter packing could be achieved by changing the triangle representation or by employing multi-dimensional binning.

In its current state, our method supports only object-space shading effects. The atlas data structure does not store information about the screen-space neighborhood of a sample. For such effects we would have to store additional data per-sample or per-triangle.

If geometry is near-parallel to the viewing direction, artifacts can occur, as the screen-space footprint of the triangle changes rapidly under small camera movements. Similarly, a novel view far from the reference camera my show lower quality. Both issues can be fixed by considering the triangle's size under all potential views and applying more advanced filtering on the client.

While tessellation is an efficient way to generate shading samples (especially for L-packing), it still shows some overhead, especially when considering pre-tessellation and linear id computations for Oversampling. An alternative to tessellation for dynamic shading sample generation, could be dynamic scheduling [SKB*14] or a custom software rendering pipeline [KKSS18], especially considering that we do not employ the rasterizer.

By computing shading in tessellation there is no support for explicit derivative computations (`dFdx` in GLSL). However, in many cases implicit derivatives are sufficient. For example, we compute the mipmap level based on analytic sample distribution.

Our client always shows the shading information gathered during the last server frame, which might be inaccurate for dynamic/view-dependent effects. To alleviate this issue, the server could provide information to perform shading extrapolation between server frames. This would of course increase the load on the client.

7. Conclusion

By deriving a set of functions providing additional data in tessellation stages, we have introduced a novel approach to leveraging the dynamic parallelism of the hardware-accelerated tessellator. We have introduced two novel strategies to sample shading on the surface of a triangle: L-packing and Oversampling. L-packing provides a novel triangle representation within a shading atlas. Both strategies allow for efficient encoding and decoding and support hardware-accelerated interpolation between samples.

We have demonstrated a split rendering pipeline that uses encoded shading data for fast novel-view extrapolation. Our method works without any pre-processing and achieves near ground-truth novel view quality. Our tests show that our method is also suitable for use in a streaming-rendering scenario.

We consider our approach to launch and identify any number of threads using tessellation, a way to open up new possibilities for dynamic per-triangle work generation. Oversampling is just one example. Naturally, a novel tessellation pattern for true screen-space uniformity is another example to consider.

This research was supported by the Max Planck Center for Visual Computing and Communication, by the German Research Foundation (DFG) grant STE 2565/1-1, the Austrian Science Fund(FWF) grant I 3007, and NVIDIA Corporation who donated the GPU used for this research.

References

[AHTAM14] ANDERSSON M., HASSELGREN J., TOTH R., AKENINE-MÖILER T.: Adaptive texture space shading for stochastic rendering. *Computer Graphics Forum 33*, 2 (may 2014), 341–350. 2

[Bak16] BAKER D.: Object space lighting. GDC, 2016. 2

[BBM*01] BUEHLER C., BOSSE M., MCMILLAN L., GORTLER S., COHEN M.: Unstructured lumigraph rendering. In *Proc. SIGGRAPH* (2001), pp. 425–432. 3

[BCC16] BOOS K., CHU D., CUERVO E.: Flashback: Immersive virtual reality on mobile devices via rendering memoization. In *MobiSys* (2016), pp. 291–304. 3

[BFM10] BURNS C. A., FATAHALIAN K., MARK W. R.: A lazy object-space shading architecture with decoupled sampling. In *Proc. HPG* (2010), pp. 19–28. 2

[BG04] BAO P., GOURLAY D.: Remote walkthrough over mobile networks using 3-d image warping and streaming. *IEE Proceedings - Vision, Image and Signal Processing 151*, 4 (Aug 2004), 329–336. 3

[BL08] BURLEY B., LACEWELL D.: Ptex: Per-face texture mapping for production rendering. In *EGSR* (2008), pp. 1155–1164. 2

[BMS*12] BOWLES H., MITCHELL K., SUMNER R. W., MOORE J., GROSS M.: Iterative image warping. *Computer Graphics Forum 31*, 2pt1 (2012), 237–246. 3

[CCC87] COOK R. L., CARPENTER L., CATMULL E.: The reyes image rendering architecture. In *Proc. SIGGRAPH* (1987), pp. 95–102. 2

[CCT*11] CHEN K.-T., CHANG Y.-C., TSENG P.-H., HUANG C.-Y., LEI C.-L.: Measuring the latency of cloud gaming systems. In *Proc. Multimedia* (2011), MM '11, pp. 1269–1272. 1

[CG02] CHANG C.-F., GER S.-H.: *Enhancing 3D Graphics on Mobile Devices by Image-Based Rendering*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2002, pp. 1105–1111. 3

[Che15] CHEN K.: Adaptive virtual texture rendering in far cry 4. GDC, March 2015. 2, 3

[CLM*15] CRASSIN C., LUEBKE D., MARA M., MCGUIRE M., OSTER B., SHIRLEY P., SLOAN P.-P., WYMAN C.: CloudLight: A system for amortizing indirect lighting in real-time rendering. *Journal of Computer Graphics Techniques (JCGT) 4*, 4 (October 2015), 1–27. 3

[COMF99] COHEN-OR D., MANN Y., FLEISHMAN S.: Deep compression for streaming texture intensive animations. In *SIGGRAPH* (1999), pp. 261–267. 3

[CTH*14] CLARBERG P., TOTH R., HASSELGREN J., NILSSON J., AKENINE-MÖLLER T.: AMFS: Adaptive Multi-Frequency Shading for Future Graphics Processors. *ACM Trans. on Graph. 33*, 4 (jul 2014), 1–12. 2

[CTM13] CLARBERG P., TOTH R., MUNKBERG J.: A sort-based deferred shading architecture for decoupled sampling. *ACM Trans. on Graph. 32*, 4 (jul 2013). 2

[CW93] CHEN S. E., WILLIAMS L.: View interpolation for image synthesis. In *SIGGRAPH* (1993), pp. 279–288. 3

[CWC*15] CUERVO E., WOLMANY A., COXZ L. P., LEBECK K., RAZEENZ A., SAROIUY S., MUSUVATHI M.: Kahawai: High-quality mobile gaming using GPU offload. In *MobiSys* (May 2015), pp. 121–135. 3

[CWSR12] CHOY S., WONG B., SIMON G., ROSENBERG C.: The brewing storm in cloud gaming: A measurement study on cloud to end-user latency. In *Workshop on Network and Systems Support for Games (NetGames)* (Nov 2012), pp. 1–6. 1

[DER*10] DIDYK P., EISEMANN E., RITSCHEL T., MYSZKOWSKI K., SEIDEL H.-P.: Perceptually-motivated real-time temporal upsampling of 3D content for high-refresh-rate displays. *Computer Graphics Forum 29*, 2 (2010), 713–722. 3

[DRE*10] DIDYK P., RITSCHEL T., EISEMANN E., MYSZKOWSKI K., SEIDEL H.-P.: Adaptive image-space stereo view synthesis. In *15th International Workshop on Vision, Modeling and Visualization Workshop* (Siegen, Germany, 2010), pp. 299–306. 3

[HSS] HLADKY J., SEIDEL H.-P., STEINBERGER M.: Real-time potentially visible set for streaming rendering. 2, 7, 11

[HY16] HILLESLAND K. E., YANG J. C.: Texel shading. In *Proc. Eurographics: Short Papers* (2016), pp. 73–76. 2, 7

[KKSS18] KENZEL M., KERBL B., SCHMALSTIEG D., STEINBERGER M.: A high-performance software graphics pipeline architecture for the GPU. *ACM Trans. Graph. 37*, 4 (Nov. 2018). 11

[LCC*15] LEE K., CHU D., CUERVO E., KOPF J., DEGTYAREV Y., GRIZAN S., WOLMAN A., FLINN J.: Outatime - using speculation to enable low-latency continuous interaction for mobile cloud gaming. In *Proc. Mobile Systems, Applications, and Services* (2015). 1, 3

[LD12] LIKTOR G., DACHSBACHER C.: Decoupled deferred shading for hardware rasterization. In *Proc. I3D* (2012), pp. 143–150. 2

[Lev95] LEVOY M.: Polygon-assisted jpeg and mpeg compression of synthetic images. In *SIGGRAPH* (1995), pp. 21–28. 3

[MCO97] MANN Y., COHEN-OR D.: Selective pixel transmission for navigating in remote virtual environments. *Computer Graphics Forum 16* (1997), C201–C206. 3

[MHAM08] MUNKBERG J., HASSELGREN J., AKENINE-MÖLLER T.: Non-uniform fractional tessellation. In *Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware* (Aire-la-Ville, Switzerland, Switzerland, 2008), GH '08, Eurographics Association, pp. 41–45. 4, 5, 7

[MHUC12] MANZANO M., HERNANDEZ J. A., URUENA M., CALLE E.: An empirical study of cloud gaming. In *2012 11th Annual Workshop on Network and Systems Support for Games (NetGames)* (Nov 2012), pp. 1–2. 1

[MMB97] MARK W. R., MCMILLAN L., BISHOP G.: Post-rendering 3d warping. In *Proc I3D* (1997). 3

[Mou18] MOURS P.: Mesh shaders in turing. Talk at GPU Technology Conference Europe, 2018. 3

[MVD*18] MUELLER J. H., VOGLREITER P., DOKTER M., NEFF T., MAKAR M., STEINBERGER M., SCHMALSTIEG D.: Shading atlas streaming. *ACM Trans. Graph. 37*, 6 (2018), 199:1–199:16. 2, 3, 4, 7

[NCO03] NOIMARK Y., COHEN-OR D.: Streaming scenes to mpeg-4 video-enabled devices. *IEEE Computer Graphics and Applications 23* (01 2003), 58–64. 3

[Ocu18] OCULUSVR: Rendering to the oculus rift, 2018. Visited on March 30, 2018. 2

[PHE*11] PAJAK D., HERZOG R., EISEMANN E., MYSZKOWSKI K., SEIDEL H.-P.: Scalable remote rendering with depth and motion-flow augmented streaming. *Computer Graphics Forum 30*, 2 (2011), 415–424. 3

[RKLC*11] RAGAN-KELLEY J., LEHTINEN J., CHEN J., DOGGETT M., DURAND F.: Decoupled sampling for graphics pipelines. *ACM Trans. on Graph. 30*, 3 (may 2011), 1–17. 2

[RKR*16] REINERT B., KOPF J., RITSCHEL T., CUERVO E., CHU D., SEIDEL H.-P.: Proxy-guided image-based rendering for mobile devices. *Computer Graphics Forum 35*, 7 (oct 2016), 353–362. 3

[SGHS98] SHADE J., GORTLER S., HE L.-W., SZELISKI R.: Layered depth images. In *Proc. SIGGRAPH* (1998), pp. 231–242. 3

[SH15] SHI S., HSU C.-H.: A survey of interactive remote rendering systems. *ACM Comput. Surv. 47*, 4 (May 2015). 3

[SKB*14] STEINBERGER M., KENZEL M., BOECHAT P., KERBL B., DOKTER M., SCHMALSTIEG D.: Whippletree: Task-based scheduling of dynamic workloads on the GPU. *ACM Trans. Graph. 33*, 6 (Nov. 2014), 228:1–228:11. 11

[SMSW11] SHENG B., MENG W.-L., SUN H.-Q., WU E.-H.: MCGIM-based model streaming for realtime progressive rendering. *Journal of Computer Science and Technology 26*, 1 (jan 2011), 166–175. 3

[SNC12] SHI S., NAHRSTEDT K., CAMPBELL R.: A real-time remote rendering system for interactive mobile graphics. *ACM Trans. Multimedia Comput. Commun. Appl. 8*, 3s (Oct. 2012), 46:1–46:20. 3

[TL01] TELER E., LISCHINSKI D.: Streaming of complex 3d scenes for remote walkthroughs. *Computer Graphics Forum 20*, 3 (2001), 17–25. 3

[Wal91] WALLACE G. K.: The jpeg still picture compression standard. *Commun. ACM 34*, 4 (Apr. 1991), 30–44. 7

[YN00] YOON I., NEUMANN U.: Web-based remote rendering with ibrac (image-based rendering acceleration and compression). *Computer Graphics Forum 19*, 3 (2000), 321–330. 3

[YTS*11] YANG L., TSE Y.-C., SANDER P. V., LAWRENCE J., NEHAB D., HOPPE H., WILKINS C. L.: Image-based bidirectional scene reprojection. *ACM Trans. Graph. 30*, 6 (Dec. 2011), 150:1–150:10. 3

[Yuk17] YUKSEL C.: Mesh color textures. In *High Performance Graphics* (2017), HPG '17, pp. 17:1–17:11. 2, 4