Pulsar: Efficient Sphere-based Neural Rendering – Supplemental Document –

Christoph Lassner[1] Michael Zollhöfer[1]
[1]Facebook Reality Labs

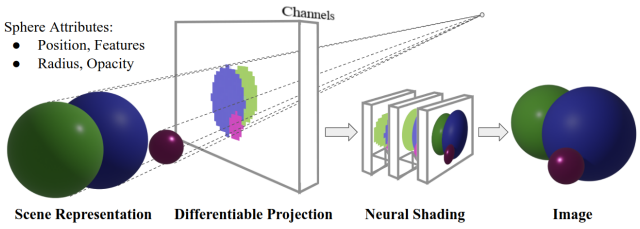

Figure 1: Reproduced visualization of the neural rendering pipeline from the main paper. In this supplemental document, we describe the technical details of the 'Differentiable Projection' step implemented by Pulsar.

In this supplemental document, we delve deeper into the implementation details of the Pulsar renderer (a full pipeline overview is reproduced in Fig. 1; this is Fig. 2 of the main paper). For this purpose, let's briefly revisit the scene description as described in the main paper. We represent the scene as a set $\mathcal{S} = \{(\mathbf{p}_i, \mathbf{f}_i, r_i, o_i)\}_{i=1}^{M}$ of M spheres with learned position $\mathbf{p}_i \in \mathbb{R}^3$, neural feature vector $\mathbf{f}_i \in \mathbb{R}^d$, radius $r_i \in \mathbb{R}$, and opacity $o_i \in \mathbb{R}$. Pulsar implements a mapping $\mathbf{F} = \mathcal{R}(\mathcal{S}, \mathbf{R}, \mathbf{t}, \mathbf{K})$ that maps from the 3D sphere-based scene representation $\mathcal{S}$ to a rendered feature image $\mathbf{F}$ based on the image formation model defined by the camera rotation $\mathbf{R}$, translation $\mathbf{t}$, and intrinsic parameters $\mathbf{K}$. $\mathcal{R}$ is differentiable with respect to $\mathbf{R}$, $\mathbf{t}$ and most parts of $\mathbf{K}$, *i.e.*, focal length and sensor size.

The Aggregation Function The rendering operation $\mathcal{R}$ has to compute the channel values for each pixel of the feature image $\mathbf{F}$ in a differentiable manner. To this end, we propose the following blending function (Eq. 1 of the main paper) for a given ray, associating a blending weight w_i with each sphere i:

$$w_i = \frac{o_i \cdot d_i \cdot \exp(o_i \cdot \frac{z_i}{\gamma})}{\exp(\frac{\epsilon}{\gamma}) + \sum_k o_k \cdot d_k \cdot \exp(o_k \cdot \frac{z_k}{\gamma})}. \qquad (1)$$

We employ normalized device coordinates $z_i \in [0, 1]$ where 0 denotes maximum depth and d_i is the normalized orthogonal distance of the ray to the sphere center. This distance, since always orthogonal to the ray direction, automatically provides gradients for the two directions that are orthogonal to the ray direction. Strictly speaking, for one ray this direction gradient could be non-existent if the ray hits the sphere in its center; or it could just provide gradients in one of the two remaining directions if it hits the sphere perfectly above or to the side of its center. We provide position gradients only for spheres that have more than three pixels projected radius because we observed that the gradients are numerically not stable otherwise. This means, that *if* position gradients are provided they can move spheres in all directions in space. We define $d_i = \min(1, \frac{||\vec{d_i}||_2}{R_i})$, where $\vec{d_i}$ is the vector pointing orthogonal from the ray to the sphere center. Like this, d_i becomes a linear scaling factor in $[0, 1]$.

A notable property of the proposed aggregation function is, that it is commutative w.r.t. the sphere order. This will become important in the following sections.

1. Data-parallel Implementation

Modern GPU architectures offer a tremendous amount of processing power through a large number of streaming multiprocessors and threads, as well as enough memory to store scene representations and rendered images in GPU memory. For example, even an NVIDIA RTX 2080 Ti consumer GPU has 4352 CUDA cores with 64 streaming multiprocessors with access to up to 11 GB of memory. The CUDA cores/threads are grouped in *warps* of 32 threads. Multiple warps again can work together in *groups*. Warps have particularly fast local shared memory and operations, however all threads in a warp execute exactly the same command on potentially different data, or a part of them must sleep. This computing paradigm is called 'Single instruction, multiple data' (SIMD). For example, if half of the threads follow a different execution path due to an '`if`' statement than the rest; in this case the half not following the branch will sleep while the first half executes the relevant commands.

All these architectural peculiarities must be taken into account to use GPU hardware efficiently. This requires making smart use of parallel code and finding good memory access patterns to not block execution through excessive IO loads. Because both of these aspects tightly connect, non-

intuitive solutions often turn out to be the most efficient and experimentation is required to identify them.

We found a way to keep the computation throughput high by elegantly switching between parallelization schemes and by using finely tuned memory structures as 'glue' between the computations. In the following sections, we aim to discuss these steps, the underlying memory layout and the parallelization choices made.

1.1. The forward pass

For the forward pass, the renderer receives a set of n spheres with position $\mathbf{p}_i$, features $\mathbf{f}_i$, radius r_i and opacity o_i for each sphere $i \in 1, \ldots, n$. Additionally, the camera configuration $\mathbf{R}$, $\mathbf{t}$ and $\mathbf{K}$ must be provided. Assuming we have a commutative per-pixel blending function, Eq. 1, the first fundamental choice to make is whether to parallelize the rendering process over the pixels or the spheres.

Parallelizing over the spheres can be beneficial through the re-use of information to evaluate the rendering equation for pixels close to each other. However, this approach leads to memory access collisions for the results (writing access to all pixel values must be protected by a mutex), which obliterates runtime performance. The second alternative is to parallelize rendering over the pixels. To make this strategy efficient, it is critical to find a good way to exploit spatial closeness between pixels during the search for relevant spheres. It is important to reduce the amount of candidate spheres (spheres that could influence the color of a pixel) for each pixel as quickly as possible. This can be achieved by mapping spatial closeness in the image to 'closeness' on GPU hardware: thread groups can analyze spheres together and share information. Overall, a two step process becomes apparent: 1) find out which spheres are relevant for a pixel (group), 2) draw the relevant spheres for each pixel. Both steps must be tightly interconnected so that memory accesses are reduced to a minimum.

By design of our scene parameterization the enclosing rectangle of the projection of each sphere is simple to find. But even in this simple case we would do the intersection part of the calculation repeatedly: every pixel (group) would calculate the enclosing rectangle for each sphere. This is why we separate the enclosing rectangle computation as step (0) into its own GPU kernel. Importantly, through this separation, we can parallelize step (0) over the spheres and use the full device resources.

1.1.1 Step 0: Enclosing Rectangle Calculation

This step is parallelized over the spheres. It uses $\mathbf{K}$, $\mathbf{p}_i$ and r_i to determine the relevant region in the image space for each sphere and to encode the intersection and draw information in an efficient way for the following steps. The standard choice for such an encoding is a k-d-tree, bounding volume hierarchy (BVH) or a similar acceleration structure. We experimented with (extended) Morton codes [3, 4] and the fast parallel BVH implementations [1, 2] and found their performance inferior[1] compared to the following strategy using bounding box projections.

Instead of using acceleration structures, the sphere geometry allows us to find the projection bounds of the sphere on the sensor plane. This is done with only a few computations for the orthogonal but also the pinhole projection model. In the pinhole model, the distortion effects make slightly more complex computations necessary; trigonometric functions can be avoided for higher numerical accuracy through the use of several trigonometric identities.

Additional steps must be taken to robustify the calculated boundaries for numerical inaccuracies. We make the design choice to have every sphere rendered with at least a size of one pixel: in this way, every sphere always receives gradients and no spheres are 'lost' between pixel rays. We store the results of these calculations in two data structures:

Intersection information This is a `struct` with four `unsigned short` values and contains the calculated x and y limits for each sphere. This data structure needs 8 bytes of memory. One cache line on the NVIDIA Turing GPUs holds $256 = 8 \cdot 32$ bytes, meaning that all 32 threads in a warp can load one of these data structures with one command. This makes coalesced iteration fast, which helps to process large amounts of intersection data structures in parallel.

Draw information This is a `struct` with all the information needed to draw a sphere once an intersection has been detected. We store the position vector, up to three feature value floats or a float pointer (in case of more than 3 features), as well as the distance to the sphere center and the sphere radius. This requires $8 \cdot 4 = 32$ bytes of storage per sphere. The importance of this step is to localize all required information and convert a 'struct of arrays' (separate arrays with position, radius, colors) to an 'array of structs' (one array of *draw information* structures) with the required information.

After this step, all input variables ($\mathbf{p}_i$, $\mathbf{f}_i$, r_i and o_i) are encoded in these two datastructures and only used from these sources. The computation and storage run in $0.22\,\mathrm{ms}$ for $1\,000\,000$ spheres. We additionally store the earliest possible intersection depth for each sphere in a separate array. For spheres that are outside of the sensor area, this

[1] We used our own implementation that closely follows Karras et al.'s papers but is likely slower than theirs. We evaluated the patented tr-BVH implementation in the NVIDIA OPTIX package (https://developer.nvidia.com/optix). However, OPTIX does not provide access to the acceleration structure and just allows to query it. This is insufficient for our use case because we need to find an arbitrary number of closest spheres to the camera; we decided not to use OPTIX to avoid the runtime hit of manual sorting.

value is set to infinity. Then, we use the CUB library[2] to sort the *intersection information* and *draw information* arrays by earliest possible intersection depth. This step takes another $3.2\,\mathrm{ms}$ for $1\,000\,000$ spheres. The sorting is important for the following steps: the sphere intersection search may be stopped early once it has been determined that no sphere with a greater distance can still have a notable impact.

1.1.2 Step 1: Intersection Search

The aim for this step is to narrow down the number of spheres relevant for pixels at hand as much and as quickly as possible, leveraging as much shared infrastructure as possible. That's why in a first processing step, we divide the entire image into nine parts[3] (an empirically derived value). The size of each of the nine parts is a multiple of thread block launch sizes (we determined this to be $16 \cdot 16$ pixels on current GPU architectures). All nine parts are processed sequentially. For each part, we first use the full GPU to iterate over the *intersection information* array to find spheres that are relevant for pixels in the region (we can again parallelize over the spheres). Using the CUB `select_flags` routine, we then quickly create arrays with the sorted, selected subset of *intersection information* and *draw information* data structures for all spheres (important: the spheres in this selected subset are still *sorted by earliest possible intersection depth*). From this point on, we parallelize over the pixels and use blocks and warps to use coalesced processing of spheres.

The next level is the block-wise intersection search. We use a block size of $16 \cdot 16 = 256$ threads, so eight warps per block. We observed that larger block sizes for this operation always improved performance, but reached a limit of current hardware at a size of 256 due to the memory requirements. This indicates that the speed of the proposed algorithm will scale favorably with future GPU generations.

We implement the intersection search through coalesced loading of the *intersection information* structures and testing of the limits of the current pixel block. The sphere *draw information* for spheres with intersections are stored in a block-wide shared memory buffer with a fixed size. This size is a multiple of the block size to always be able to accommodate all sphere hits. Write access to this buffer needs to be properly synchronized. If the buffer becomes too full or the spheres are exhausted, Step 2 execution is invoked to clear it. In Step 2, each pixel thread works autonomously and care must be taken to introduce appropriate synchronization boundaries to coordinate block and single thread execution. Additionally, each pixel thread can vote whether it is 'done' with processing spheres and future spheres would have not enough impact; if all pixels in a block vote 'done', execution is terminated. The vote is implemented through a thread-warp-block stage-wise reduction operation.

1.1.3 Step 2: the *Draw* Operation

The draw operation is executed for each pixel separately and for each sphere *draw information* that has been loaded into the shared memory buffer. Because every pixel is processed by its own thread, write conflicts for the channel information are avoided and each pixel thread can work through the list of loaded *draw information* at full speed. The intersection depths for each sphere are tracked: we use a small (in terms of number of spheres to track; this number is fixed at compile time) optimized priority queue to track the IDs and intersection depths of the closest five spheres for the backward pass. Additionally, updating the denominator of the rendering equation allows us to continuously have a tracker for the minimum required depth that a sphere must have for an an n percent contribution to the color channels. If set (default value: 1%), this allows for early termination of the raycasting process for each pixel.

1.1.4 Preparing for the Backward Pass

If a backward pass is intended (this can be optionally deactivated), some information of the forward pass is written into a buffer. This buffer contains for each pixel the normalization factor as well as the intersection depths and IDs of the closest five spheres hit.

We experimented with various ways to speed up the backward calculation, and storing this information from the forward operation is vastly superior to all others. It allows to skip the intersection search altogether at the price of having to write and load the backward information buffer. Since writing and loading can be performed for each thread without additional synchronization, it still turned out to be the most efficient way.

1.2. The Backward Pass

Even with the intersection information available, there remain multiple options on how to implement the backward pass. It is possible to parallelize over the spheres (this requires for each thread to iterate over all pixels a sphere impacts, but it avoids synchronization to accumulate gradient information) or over the pixels (this way each thread only processes the spheres that have been detected at the pixel position, but requires synchronization for gradient accumulation for each sphere). We found that parallelizing over the pixels is superior, especially since this implementation is robust to large spheres in pixel space.

[2] http://nvlabs.github.io/cub/

[3] During sorting, we also find the enclosing rectangle for all visible spheres and use this information for tighter bounds of the region to draw.

Again, minimizing memory access is critical to reach high execution speeds. To achieve this, we reserve the memory to store all sphere specific gradients. Additionally, we also allocate a buffer for the camera gradient information *per sphere*. We found that accumulating the sphere gradients through synchronized access from each pixel thread is viable, but synchronizing the accumulation of the camera gradients, for which every sphere for every pixel has a contribution, causes too much memory pressure. Instead, we accumulate the camera gradients sphere-wise and run a device-wide reduction as a post-processing step. This reduces the runtime cost to only $0.6\,\mathrm{ms}$ for $1\,000\,000$ spheres.

Overall, this implementation proved robust and fast in a variety of settings. Apart from being nearly independent of sphere sizes, it scales well with image resolution and the number of spheres. We found additional normalization helpful to make the gradients better suited for gradient descent optimization:

- sphere gradients are averaged over the number of pixels from which they are computed. This avoids parameters of small spheres converging slower than those of large spheres. In principle, large spheres have a larger impact on the image, hence receive larger gradients. However, from an optimization point of view, we found the gradients normalized by the number of pixels are much better suited for stable loss reduction with gradient descent techniques.
- camera gradients need to take the sphere size into account to lead to a stable optimization. We use the area that each sphere covers in the image as a normalization factor (together with the constant 1×10^{-3}, which we found approximately suitable to avoid having to scale sphere and camera gradients differently in gradient descent optimization). The area normalization makes this calculation very similar to Monte Carlo integration.

The gradient computation for each of the gradients is only performed if the gradients are required by the PyTorch autodiff framework. Overall, using these strategies we achieve very good scaling behavior as demonstrated in Fig. 3 of the main paper.

References

[1] Tero Karras. Maximizing parallelism in the construction of bvhs, octrees, and k-d trees. In *Proceedings of the Fourth ACM SIGGRAPH/Eurographics conference on High-Performance Graphics*, pages 33–37, 2012.

[2] Tero Karras and Timo Aila. Fast parallel construction of high-quality bounding volume hierarchies. In *Proceedings of the 5th High-Performance Graphics Conference*, pages 89–99, 2013.

[3] Guy M Morton. A computer oriented geodetic data base and a new technique in file sequencing. 1966.

[4] Marek Vinkler, Jiri Bittner, and Vlastimil Havran. Extended morton codes for high performance bounding volume hierarchy construction. In *Proceedings of High Performance Graphics*, pages 1–8. 2017.