Eurographics Symposium on Rendering (2025)
B. Wang and A. Wilkie (Editors)

Selective Caching in Procedural Texture Graphs for Path Tracing

Vincent Schüßler[1], Johannes Hanika[1], Basile Sauvage[2], Jean-Michel Dischler[2], and Carsten Dachsbacher[1]

[1]Karlsruhe Institute of Technology, Germany [2]ICube, Université de Strasbourg, France

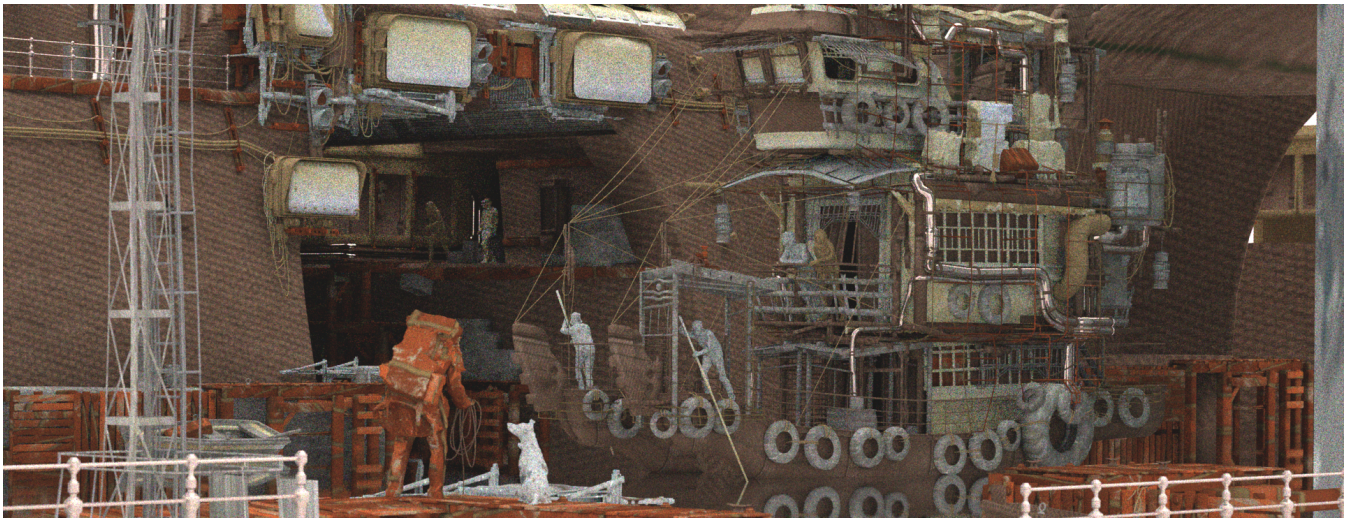


Figure 1: *The* PARTYTUG *scene with procedural textures, which we evaluate using our framework integrated into a path tracer. For each material in this scene, we instantiate one randomly selected procedural texture graph. Through caching, our approach reduces total render time for 256 samples per pixel to 38% of the time taken by rendering using shade-on-hit, i.e. evaluating each texture footprint individually.*

Abstract

Procedural texturing is crucial for adding details in large-scale rendering. Typically, procedural textures are represented as computational graphs that artists can edit. However, as scene and graph complexity grow, evaluating these graphs becomes increasingly expensive for the rendering system. Performance is greatly affected by the evaluation strategy: Precomputing textures into high resolution maps is straightforward but can be inefficient, while shade-on-hit architectures and tile-based caches improve efficiency by evaluating only necessary data. However, the ideal choice of strategy depends on the application context. We present a new method to dynamically select which texture graph nodes to cache within a rendering system that supports filtered texture graph evaluation and tile-based caching. Our method allows us to construct an optimized evaluation strategy for each scene. Cache-friendly nodes are identified using data-driven predictions based on statistics of requested texture footprints, gathered during a profiling phase. We develop a statistical model that fits profiling data and predicts how caching specific nodes affects evaluation efficiency and storage demands. Our approach can be directly integrated into a rendering system or used to analyze renderer data, helping practitioners to optimize performance in their workflows.

CCS Concepts

• ***Computing methodologies*** → ***Rendering; Ray tracing;***

DOI: 10.2312/sr.20251176

1. Introduction

Procedural materials are essential in modern 3D content creation, providing a controllable, resolution-independent approach to material generation. Tools like Blender [Ble25] and Adobe Substance 3D [Ado25] use procedural texture graphs (PTGs), where nodes define texture generators or filters, and edges determine information flow. By combining procedural primitives (e.g. noise functions) with operations like gradient mapping and distortion, users can efficiently create diverse, intricate textures. Adjusting node parameters reduces manual effort while achieving rich visual detail.

When rendering procedurally textured scenes, the task of evaluating many PTGs can become a substantial part of the total rendering work. Optimization is challenging especially in Monte Carlo-based rendering, e.g. path tracing, since the renderer requires random access to all textures. Procedural texture graphs provide a highly compact, generative representation and offer flexibility in how they are implemented: a rendering system can decide which parts of the texture to generate and when; this offers potential for more efficient resource management, balancing memory usage, performance, and visual fidelity.

A straightforward strategy is to precompute all output textures in high-resolution. This allows to fully decouple procedural texturing from rendering. Since textures are evaluated only once, multiple accesses to the same texture regions share results of a PTG evaluation. On the other hand, many texture regions will not be required in high resolution, due to partial occlusion, distance to the camera, or blur induced by camera lenses or a light scattering process. Since information about this is not available in advance, upfront evaluation potentially computes many unused results. In addition, precomputation increases latency and can require a significant amount of intermediate storage space.

Graph-based shading systems like Open Shading Language (OSL) [GSKC10] take the opposite approach by delaying graph evaluation until render time. This allows to evaluate graphs only on actual texture accesses by the renderer. Potential benefits include the reduction of computation and storage overhead, as well as balancing the use of memory and compute resources. However, as different texture accesses in OSL evaluate the PTG individually, they do not share evaluation results, thus impeding the overall efficiency. Tile-based caching can sometimes offset this drawback. However, it involves upfront work that has to amortize for it to increase efficiency, similar to full precomputation.

There is no universal answer to the question of which of these approaches is preferable in a given configuration. Therefore, our goal is to avoid a fixed choice in advance and leverage the full flexibility of the procedural representation instead. In this work, we introduce a novel framework for efficiently evaluating procedural texture graphs in a path tracer: We delay the choice of the evaluation *strategy* to the rendering process itself. This allows to decide whether to cache based on information about the rendered scene in a fine-grained way, at the level of individual parts of a PTG. Our framework also provides filtered PTG evaluation and predicting computational costs to guide optimization. We review the state of the art in the next section, before giving a more detailed overview of our framework.

2. Background and related work

Keeping up with increasing shading and texturing complexity is a recurring challenge in production rendering [CCC87; FHL*18; CFS*18; BAC*18; KCSG18; GIF*18]. Rendering systems typically expose powerful programmable interfaces to artists for controlling shading, through shader languages or shading graphs [HL90; GSKC10]. These allow to combine many layers of textures and procedurally defined patterns and noises. Executing many user-provided shader programs can become a bottleneck for rendering. Accessing textures involves lots of I/O, while procedural functions can be computationally intensive.

With the shift towards path tracing [KFF*15], this has become more difficult. In the typical shade-on-hit architecture, a shader has to be executed at every path vertex interaction. The incoherent nature of random paths leads to nonlocal memory access and divergent execution between different paths, hindering the use of SIMD instructions. Manuka [FHL*18] is designed around improving shading efficiency by fully decoupling shader execution and path tracing (shade before hit). Another approach is to try finding coherence in a larger batch of paths through sorting [ENSB13; ÁBWM16; LGXT17; CFS*18], sometimes with hardware support [NVI22; Int22]. In large and complex scenes usually there remains at least some level of incoherence.

Texture caching. Software caches got introduced early on [Pea90] to improve random texture access, with their overall design mostly standing the test of time [Gri08; CFS*18]. As the cache is discrete while texture space is continuous, accessing a texture cache involves reconstruction. The effectiveness of texture caching relies on the assumption that the required level of detail for reconstruction is often much lower than the original texture resolution is. Peachey [Pea90] describes this as the "principle of texture thrift", which states that the amount of texture information used for rendering should be limited by the output resolution, as any additional detail would be imperceptible. This ensures that the working set of textures remains bounded and, most importantly, independent of the original texture resolution, allowing it to fit efficiently into cache. However, this assumption holds only if lower-resolution versions of the texture can be efficiently stored and reconstructed, which inherently links caching to prefiltering. In practice, this is commonly achieved using resolution pyramids [Wil83] (mipmaps).

Footprints. Reconstruction bandlimit is usually represented as the linear transformation of a circular kernel, derived from a first-order approximation of the light transport process. This results in an elliptical *footprint*, which we can use for efficient filtering, e.g. using EWA [GH86; Hec89]. There exist various theories for computing appropriate footprints in ray and path tracing, including ray [Ige99] and path differentials [SW01], conical approximation [Ama84; ANA*19] or based on frequency analysis [BYRN17] or half-vector distributions [KHD14; HKD15]. Accurate footprints are generally difficult to obtain, since unidirectional methods neglect the (unknown) frequency content of either illumination or camera importance [BYRN17]. Bidirectional footprints are only available once a complete path is formed, presenting a challenge to incremental path construction. Due to discontinuities, in particular visibility, even a complete path does not carry the required infor-

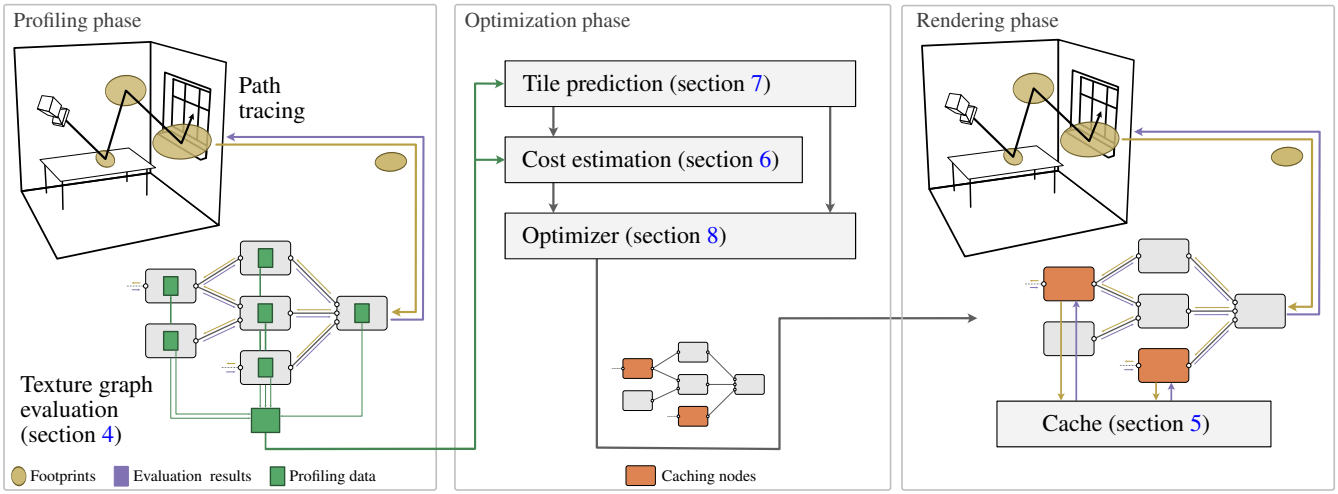


Figure 2: *Overview of our system. The algorithm is structured in three phases. First, during the profiling phase, path tracing generates texture footprints which are collected and used to query the texture graph with direct (uncached) evaluation, to collect profiling data. Then, an optimization is performed to determine the most cost effective position of cache nodes in the graph. Finally, the rendering phase continues tracing paths and querying the texture graph for footprints, using the optimized cache locations. See the indicated sections for more details.*

mation [ZD20] and aggregation would be necessary. Therefore, in practice footprints are rather approximate and often overestimated in order to favor caching.

Filtering with nonlinearities. Linear prefiltering is valid only for values that occur linearly in the rendering integral. Nonlinearities require more advanced methods [BN12]. This presents a challenge for caching in a procedural texture graph, as caching and thus filtering might occur before a nonlinear node. Note that in shading graphs, this problem is often ignored in practice to facilitate efficient caching using mip maps [GSKC10]. Heitz et al. [HNPN13; HNPN14] and Grenier et al. [GSDT22] filter color maps applied to Gaussian noises. Fournier and Sauvage [FS24] show that normal distributions are relevant approximations in the context of nonlinear texture transitions. Yang et al. [YB18] use normal distributions to automatically transform shader programs to bandlimited variants using genetic search and additional approximate filtering rules [DBLW15]. We also represent cached results as normal distributions in order to apply nonlinear operations to them.

Procedural texture graphs. Procedural texture graphs (PTG) are directed acyclic graphs that describe a process for synthesizing textures. Source nodes produce parametric patterns or noise textures [LLC*10], while inner nodes process and combine their inputs, e.g. through color transformations, blending or warping. Nodes in a PTG can access arbitrary pixels of their inputs, making them more powerful than common shading graphs [GSKC10]. PTGs thus present a very generic framework, in which lots of procedural techniques can be implemented as custom nodes. Recent works explore the inverse problem of obtaining graph parameters [SLH*20] or topologies [GHS*22] or both [LWS*25] from an example image, while we focus on improving the integration of PTGs into rendering and their efficient evaluation.

Caching procedurals. Reiner et al. [RLD*12] cache the evaluation of costly implicit functions to accelerate rendering on GPU hardware. Similarly, Fujieda et al. [FH22] cache the output of shading graphs. They always cache the largest subgraph that only depends on texture coordinates, in order to maximize work saved by reusing cached values. We instead try to maximize efficiency by explicit search for nodes to cache and in addition consider filtering of nonlinear results.

3. Overview

Our work revolves around maximizing shading performance by balancing different computational strategies. In particular, these are on-demand and upfront evaluation of footprints via tile-based caching. Evaluating each footprint individually when requested by the renderer limits shading work to the necessary, but misses the potential of sharing results between individual evaluations. Caching can leverage these redundancies, but necessitates resampling. This leads to upfront evaluation of grids of footprints, which has to amortize before any efficiency is gained. Available memory capacity and bandwidth further limit efficient reuse of shading results.

Which of the two strategies is more efficient depends on many parameters of the system and its use case, e.g. the cost of operations, available memory and Monte Carlo sample count. Further, the choice between the two options also depends on properties of the rendered scene. Shape and distribution of footprints is different for each material, since it depends on camera position and (indirect) visibility. The structure of the texture graph can make caching outputs of certain nodes preferable to others. A global choice for one of the strategies might therefore limit overall efficiency. Our goal is to leverage this unused potential by making more fine-grained choices to optimize efficiency for any specific use case individually.

In the following, we describe our framework to make these choices as the result of an optimization that is driven by data we collect from inside the rendering system in a profiling phase. We show an overview of the different phases and components in fig. 2.

In a path tracer, we compute elliptical footprints. Our method does not rely on a particular definition of these footprints, though we document our derivation in a supplemental document to facilitate reproducibility. We evaluate texture graphs on footprints using approximate filtering (section 4). This enables prefiltering inside the texture graph, which is a prerequisite for efficient caching at inner nodes. As a second option for graph evaluation, we introduce our texture cache in section 5. Its design is similar to a traditional tile cache commonly used in rendering systems. We discuss trade-offs of using caching, in particular possibly insufficient amortization and storage space requirements. These motivate a data-driven decision of whether to enable caching for each node.

This decision is inherently a global optimization problem. On one hand due to a global space constraint, which might require to prioritize caching of nodes that benefit the most. On the other hand, caching certain nodes reduces the value of caching others, e.g. there would be redundancy in caching a linear chain of nodes. In the end, we will formulate an ILP problem whose solution is the optimal selection of nodes to cache (section 8). This ILP problem requires several input parameters from the rendering system, namely evaluation cost and space requirements of different evaluation strategies. For this, we introduce a parametric measure of evaluation cost in section 6. We describe how we estimate cost for different strategies for a target sample count, based on data collected in the profiling phase. While the cost without caching is roughly linear in sample count and simple to extrapolate, we also require the total number of tiles that will be requested by the rendering system. This is needed to assess the cost of caching as well as its storage impact. To this end, we introduce a statistical model to predict the number of unique texture tiles requested at the target sample count (section 7).

4. Texture graph evaluation with filtering

In our work, we assume each material in the scene to be defined by a procedural texture graph (PTG) with a single output node. To evaluate a PTG for a given footprint requested by the path tracer, we evaluate the footprint at the output node of the PTG. Each node can then recursively evaluate its input nodes as necessary, possibly with transformed or duplicated footprints.

Prefiltering motivation. In our PTG, nodes are queried with footprints and return solutions to a filtering integral (see eq. (1)). We choose this approach over alternatives, such as stochastic evaluation on sampled points, for multiple reasons. While stochastic evaluation can sometimes amortize additionally introduced noise by reducing computation and memory traffic [PWSF24], nested integrals as they occur in PTGs are much more challenging to handle [WM24]. Further, for effective texture caching it is essential that sparsity of requested evaluation points is compensated by lower cache resolution. This is usually achieved by resolution pyramids and the size of footprints being roughly proportional to their spacing. Consequently, we need to carry footprint information through the graph to support caching inside of it, or for using mipmapping for image source nodes, or for leveraging built-in filtering of procedural models.

Filtering nonlinearities. Even though we use prefiltering inside the graph, our goal is to preserve the appearance of filtering at the output only. Results of footprint evaluation need to allow for the application of subsequent nonlinear operations as if applying them *before* prefiltering. As we show below, the distribution of values inside a footprint is a suitable representation of intermediate results inside the graph, while its mean would be insufficient. In practice, we approximate the full distribution using normal distributions, similar to previous work [HNPN13; HNPN14; GSDT22; YB18]. We assume that our graph output represents a value occurring linearly in the shading integral (e.g. albedo) although similar techniques are commonly used to filter normal maps [DHI*13].

4.1. Filtering theory

In the following, we show how the distribution of values allows to apply subsequent nonlinear operations to an evaluated footprint. Consider the node F in the graph in fig. 3 that represents a function f on texture coordinates $x \in \Omega$. Its filtered output is the inner product of f with a footprint k, which is usually some kernel function:

$$I_F = \int_\Omega f(x)\,k(x)\,\mathrm{d}x. \tag{1}$$

We take a probabilistic view and interpret k as the probability density function (PDF) of a random variable X. Following eq. (1), filtering F is equivalent to the expected value $\mathbb{E}[f(X)] = I_F$.

Returning to the more general problem, we want to compute an intermediate result from F that we can use to filter the downstream node G, which applies a function g pointwise to the outputs of F. Its filtered output is

$$I_G = \mathbb{E}[g(f(X))] \neq g(\mathbb{E}[f(x)]) = g(I_F), \tag{2}$$

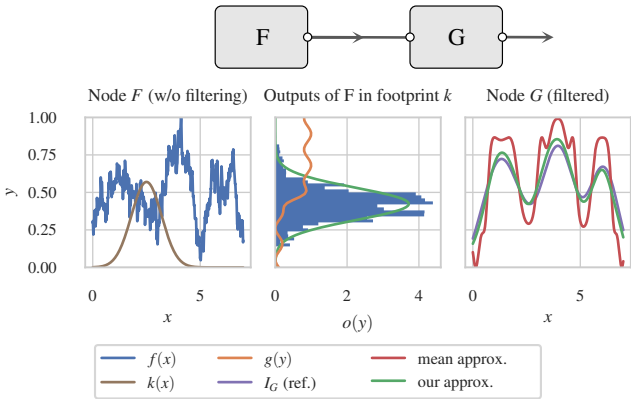
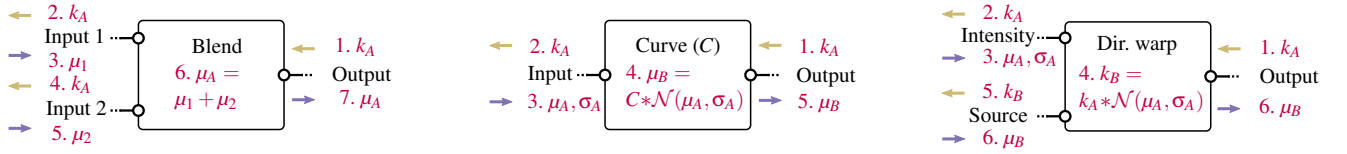


Figure 3: *We interpret the filter kernel $k(x)$ as a probability density of a random variable X and define $Y := f(X)$ with density $o(y)$. Evaluating a footprint to o, we can apply the nonlinear function $g(y)$ afterwards and still get correct results. In practice, we approximate o using a normal distribution.*



(a) *Additive blending is linear, thus we can evaluate the means of k_A for both inputs separately.*

(b) *For evaluating a non-linear transfer function C, we convolve a Gaussian approximation of the input in k_A with C.*

(c) *For directional warping, the "Intensity" input controls the strength of the warping applied to "Source". We use the Gaussian approximation to compute a new footprint k_B.*

Figure 4: *Three different nodes evaluating the mean value μ_A for a footprint k_A. The numbers signify the order of execution. For instance the Blend node receives a request for k_A (1) and passes this on to Input 1 (2), obtains μ_1 in return (3), repeats the same with Input 2 (4, 5), computes the output mean (6), before finally passing it on to the Output (7).*

which shows that we cannot trivially use I_F as an intermediate result. To compute I_G, we need to capture more information about the distribution of $f(X) =: Y$ (see fig. 3), which its PDF o_F achieves:

$$I_G = \mathbb{E}[g(f(X))] = \mathbb{E}[g(Y)] = \int_{\mathbb{R}} g(y)\, o_F(y)\, \mathrm{d}y. \tag{3}$$

Therefore we use the PDF o_F as the result of footprint evaluation.

4.2. Filtering approximation

Conceptually, nodes in our graph are queried with a footprint PDF k and return an output PDF o. We coarsely approximate both as normal distributions to make this approach efficient. In practice, only pairs of mean and covariance are exchanged between nodes. We show how some typical PTG nodes can be implemented in this framework in fig. 4.

Footprint PDF k approximation. We represent footprints as $2D$ normal distributions like in EWA filtering [GH86]. With this, linear transformation and convolution are simple to implement, which is useful when nodes query their inputs on a modified k (see fig. 4).

Output PDF o approximation. We approximate the output distribution of nodes as a normal distribution as well. We evaluate eq. (3) on the fly for its first and second moment, unless there is a more specialized implementation available for an operation. Other approximations like the unscented transform [Uhl95] might be a viable alternative for some operations. For multi-channel output, we treat each channel independently.

Filtering accuracy. In actual texture graphs, footprints and node outputs are not exactly normally distributed at all scales and node inputs and color channels can correlate, reducing the accuracy of our approach. We show results of our approximation in fig. 5 and timings and errors in table 1. We evaluate the output image for 1024×1024 pixels with one footprint per pixel. We compare to a point-sampled reference with 256 samples per pixel, as well as a naive filtering method (mean), which uses just the mean of o as the output of nodes. Mean filtering shows some obvious patterned artifacts, while our approximation produces plausible results with consistently lower error. Error tends to increase with wider footprints, as nodes with multiple inputs get no information about their correlation inside the footprint area. We also compare to reconstruction from the cache (+ cache), which section 5 describes in detail. To reconstruct a footprint, the cache will require smaller footprints to be evaluated than the original one. This generally reduces the error slightly compared to direct evaluation without the cache, but increases the time to evaluate the base resolution ($l = 10$). As spacing of evaluated footprints in the cache depends on their radius, the cache effectively decouples the density of evaluated footprints from the output resolution. Therefore, caching is significantly faster for coarse filtering ($l = 7,5$) evaluated at high resolution, i.e. if the density of footprints to reconstruct is high in relation to their size.

Discussion. In principle, we could make the filtering error arbitrary small by closer approximation of the involved PDF, or supersampling. However, coarse results of filtering are often accepted in practice, i.e. through mipmaps itself or nonlinear computations on linearly filtered values in shading graphs. While improving accuracy is an interesting avenue for future work, we believe our approximation serves as a basic model for investigating efficient graph evaluation and caching.

5. Texture cache

To reuse results of previously evaluated footprints, we introduce a texture cache into our framework. It follows a conventional design by managing texture tiles in a mipmap-like pyramidal structure. Each tile stores an 8×8 grid of evaluated isotropic footprints. In level i of the resolution pyramid, the footprint radius is 0.5^{i+1} and their spacing is 0.5^i in texture space, i.e. a greater level index corresponds to a finer level. Organization into tiles helps to amortize cache management and improves locality of operations. We assume a global cache, where evaluated results of any node from any texture graph can be stored. In summary, tiles are indexed by texture graph and node id, tile position and resolution level.

5.1. Footprint evaluation strategies

The introduction of the cache enables different options for evaluating a footprint at any node.

Direct evaluation. As before, we can evaluate a footprint by executing the operation defined by a node, while evaluating all required footprints from input nodes recursively.

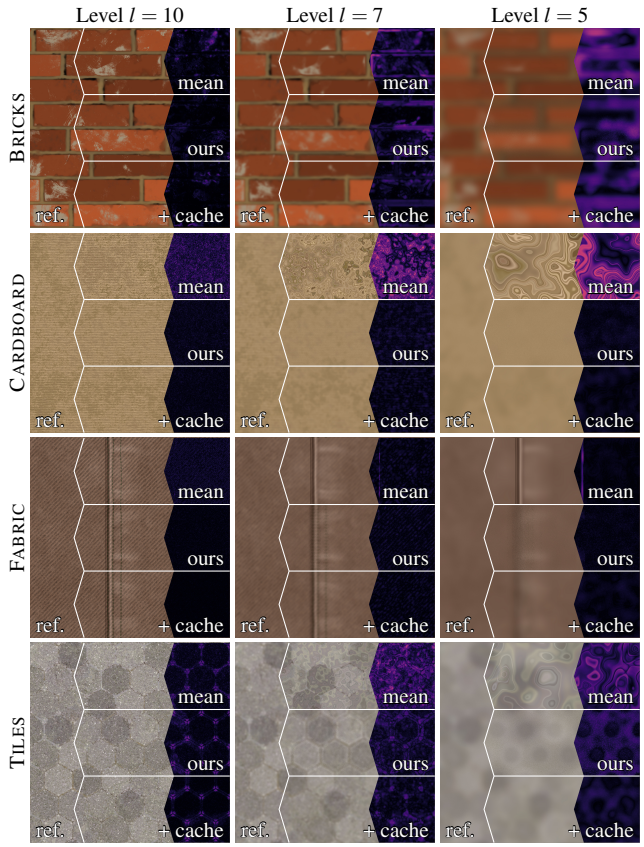


Figure 5: *Our Gaussian filtering approximation with footprint radius 0.5px ($l = 10$), 4px ($l = 7$) and 16px ($l = 5$) in comparison to naive filtering (mean). We overlay a ꟻLIP error map to a reference on the right. Our filtering preserves appearance better than using just the mean, which shows obvious patterned artifacts. Reconstructing our approximation from cache (+ cache) often improves results, since it evaluates the graph using smaller footprints.*

Table 1: *Timing and mean ꟻLIP errors for the comparison in fig. 5. We list per sample timings for the point sampled references (*).*

	time (s)			error (ꟻLIP $\cdot 10^{-3}$)		
Level l	$l = 10$	$l = 7$	$l = 5$	$l = 10$	$l = 7$	$l = 5$
BRICKS	*13.0	*12.4	*13.1			
mean	1.8	1.5	1.2	41.6	87.2	170.9
ours	4.9	4.6	4.1	27.5	81.2	154.0
+ cache	26.0	0.7	0.2	26.9	49.9	114.7
CARDBOARD	*6.5	*6.6	*6.2			
mean	3.7	3.4	3.2	129.9	237.9	301.9
ours	11.5	10.0	9.7	47.6	59.1	51.6
+ cache	61.0	1.7	0.3	39.6	52.4	50.4
FABRIC	*1.4	*1.4	*1.5			
mean	2.6	2.6	2.6	71.0	54.7	51.9
ours	7.4	7.6	7.7	30.5	50.0	43.5
+ cache	38.9	1.4	0.3	19.7	43.7	40.4
TILES	*46.4	*47.0	*47.1			
mean	2.0	1.8	1.6	70.2	173.1	179.6
ours	7.3	6.9	6.7	61.0	94.8	123.0
+ cache	38.1	1.2	0.3	46.5	78.5	77.6

Caching evaluation. If using caching at a node, we use EWA filtering to reconstruct arbitrary elliptical footprints from the cache of isotropic footprints. This usually involves interpolation between a coarse and a fine resolution level. We compute texture tiles on demand that are missing in the cache, but required to evaluate a footprint. For this, we use direct evaluation of the node on the footprint grid defined by the missing tile.

5.2. Choosing a strategy

For any footprint that we evaluate at a node, we now have two options for computing its result.Since there is not one universal strategy that is the most efficient one in all situations, we have to consider carefully which of the strategies fits best.

Amortization. Even though caching enables reuse of computed results, without sufficient redundancy in requested tiles it might not amortize its overhead. For caching, we generally have to reconstruct a footprint from multiple footprints of finer resolution in order to keep high frequencies. Therefore, the cost of reconstruction is inherently higher at first, and caching only pays off when sufficient reuse occurs. Evaluating isotropic footprints to reconstruct from is thus an investment of additional upfront computation for a future reduction in computation.

Available space. Another consideration for the efficiency of caching evaluation is the available space for storing results. If we would have to exceed available space because of newly evaluated tiles, we have to either evict old tiles or drop new ones. Both options make amortization less likely, as they can significantly reduce the period of potential reuse of a tile. Therefore, in cases were tiles would need to be evicted from the cache, it is usually more beneficial to never place them in the cache in the first place and instead use direct evaluation. In our work we thus try to avoid eviction by using caching only when we will not exceed the available storage space. Once a tile was computed, we thus assume it to stay in cache and not be evicted.

Data-driven caching decision. Since all of these considerations depend on the specific distribution of footprints at hand, our goal is to make a data driven decision about caching as part of the rendering system. We decide which evaluation strategy to use for each node of each PTG. While in principle a more fine-grained decision would be possible, e.g. treating regions of the texture differently, this adds additional complexity for reconstruction and for finding suitable texture partitions.

6. Measure for computational cost

As input to our optimization, we need to assess how efficient different strategies are at evaluating a texture graph on a set of footprints. To this end, we introduce a parametric model as a measure for the computational cost. This acts as a very simplified simulator: it allows us to compare different options without actually running all of them. It also allows us to reason about sensitivity of the outcome on different properties of hardware or rendering systems. Measured wall clock time can be incorporated into our model by setting parameter values according to timings.

6.1. Cost components and coefficients

The basic unit of our model is the local cost of evaluating a node with coefficients $\Theta = (c, \lambda_*)$ on a footprint x:

$$\begin{aligned}\text{cost}(x \mid \Theta) = c \cdot (&\lambda_{\text{const}} + \lambda_{\text{iso}} \cdot \text{anisotropy}(x) \\ &+ \lambda_{\text{none}} \cdot \text{size}(x)).\end{aligned} \tag{4}$$

We use the same equation for modeling the cost of reconstruction from cache with a global set of coefficients Θ_{cache}. It consists of the following components:

- c is a node-specific constant, e.g. measured time per operation. This could depend on parameters of the node and on the type of its output (gray or color).
- anisotropy(x) is proportional to footprint anisotropy. This models the cost of evaluation for a node for isotropic prefiltering, and simulates the cost of footprint assembly for anisotropic queries.
- size(x) is proportional to footprint area. This component models the cost of evaluation, if no efficient prefiltering is available for a node and simulates point sampling in base resolution.
- $\lambda_{\text{const}}, \lambda_{\text{iso}}, \lambda_{\text{none}}$ are node-specific weights to blend between the different filtering models, i.e. constant prefiltering, isotropic prefiltering, or no prefiltering. We constrain these to $\sum \lambda_* = 1$.

6.2. Aggregate cost

The local cost from the previous section does not include the cost of evaluating input nodes. In addition, for modeling the cost of cached evaluation, we need to consider a set of footprints at once. We now extend our definitions accordingly.

We begin by defining the set of input footprints as

$$\begin{aligned}\text{In}_i(x) = \{(j,y) \mid &\text{evaluating } x \text{ at node } i \text{ depends} \\ &\text{on evaluating } y \text{ at node } j\}.\end{aligned} \tag{5}$$

This is useful to express the total cost of the direct evaluation of a footprint x at node i, including the evaluation of inputs, recursively:

$$\text{cost}_\Sigma(x \mid \Theta_i) = \text{cost}(x \mid \Theta_i) + \sum_{(j,y)\in \text{In}_i(x)} \text{cost}_\Sigma(y \mid \Theta_j). \tag{6}$$

Next, let $\mathcal{F} = (F, w)$ be a weighted set of footprints, i.e. F is a set of footprints and $w\colon \mathcal{F} \mapsto \mathbb{R}^+$ is a weight function. The (total) cost for direct evaluation of $\mathcal{F}$ is then defined by

$$\text{cost}_*^{\text{direct}}(\mathcal{F} \mid \Theta) = \sum_{x\in F} w(x)\, \text{cost}_*(x \mid \Theta). \tag{7}$$

To model caching, we additionally introduce the function T

$$T(\mathcal{F}) = \{t \mid t \text{ is a tile required for reconstructing any } x \in F\}, \tag{8}$$

and the function z to map a set of tiles to their associated footprints

$$Z(\mathcal{T}) = (\{x \mid x \text{ is a footprint in tile } t, t \in \mathcal{T}\}, \mathbf{1}). \tag{9}$$

We can now define total cost of caching evaluation of $\mathcal{F}$ by

$$\begin{aligned}\text{cost}_\Sigma^{\text{cache}}(\mathcal{F} \mid \Theta) = &\underbrace{\text{cost}^{\text{direct}}(\mathcal{F} \mid \Theta_{\text{cache}})}_{\text{reconstruction cost}} \\ &+ \underbrace{\text{cost}_\Sigma^{\text{direct}}(Z(T(\mathcal{F})) \mid \Theta)}_{\text{cache fill cost}}.\end{aligned} \tag{10}$$

6.3. Computing the value per node in practice

In order to compare different evaluation strategies for a node, we require a representative set of footprints, on which we can evaluate our cost model. We use a profiling phase to record such a set, i.e. we render few samples per pixel using direct evaluation and store the set of requested footprints at each node.

Histogram representation. To represent these recorded sets compactly, we assume that the footprint queries are somewhat uniformly distributed for a node. In particular, we assume there is no correlation between footprint shape and a particular tile in the cache. We store the footprints in a 2D histogram at each node. Each bin is indexed by the $\log_2$ length of a footprint's major and minor axis. This information about a footprint is sufficient to evaluate all terms in our local cost model, i.e. its size and anisotropy. This histogram representation is approximate due to discretization.

Cache tiles. In addition, we record the set of required cache tiles $T(\mathcal{F})$ at each node. The number of unique tiles per level is sufficient information for evaluating the cost model for caching evaluation, as well as for estimating the required cache space.

Propagating footprint sets. For the purpose of computing the total cost of a node i, i.e. including every evaluated upstream (i.e. transitive input) node j, in general the footprint set at j will differ from the recorded set. This is because the recorded set includes every evaluation of j, even the ones not caused by an evaluation of i. Secondly, caching node i would change the footprint set at i from $\mathcal{F}$ to $Z(T(\mathcal{F}))$. While we can easily simulate this change at i using the recorded set of tiles, it is less obvious how it affects the footprint set at upstream nodes.

Therefore, we introduce an approximation to the function In (see eq. (5)), which we also record using the footprints we evaluated during profiling. It allows us to simulate the distribution of footprints at immediate input nodes, given the requested footprint distribution at a node. In this way, we propagate the recorded distribution to all upstream nodes and use it to evaluate the cost model. For approximation, we again rely on discretization and store a 2D histogram of changes between incoming and outgoing footprints of a node. We store separate histograms for each input node j to a node i, and for each cache level at i, i.e. the $\log_2$ minor axis length of incoming footprints at i. Each histogram is normalized to the number of requested footprints at i. The histogram bins represent the $\log_2$ change in footprint axis length between the footprints requested at i and j. We use these to transform a distribution of footprint requests from i to the one that is requested from j.

Cost measure evaluation accuracy. We compare our practical evaluation to the cost evaluated on the actual footprints in fig. 6. We use all combinations of our graphs and scenes (see section 9) and show the distribution of relative error, separated by the different cost terms. For cache fill cost to be accurate, our approximation needs to be able to simulate a change in the footprint distribution. For both direct and cache fill cost, constant cost, i.e. footprint count at each node, is predicted with almost no error, while footprint anisotropy ($\lambda_{\text{iso}} = 1$) has a low error. The error of approximate footprint size has high variance for the cache fill cost, while direct

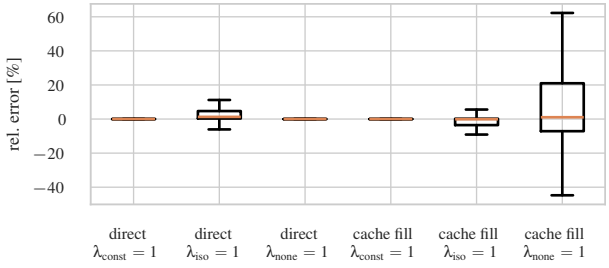


Figure 6: *Distribution of relative error of our practical cost evaluation. While we achieve low error for most cost terms, simulated cache fill cost are not fully reliable when $\lambda_{none} \neq 0$.*

cost is always accurate. We presume this is because deviations from the true footprint size have a quadratic influence on the cost. Consequently, we cannot fully rely on predicted cost for $\lambda_{\text{none}} \neq 0$, so we leave $\lambda_{\text{none}} = 0$ for almost all of our results.

7. Prediction of active texture area

For optimizing the choice of evaluation strategy, our goal is to estimate the cost at a target sample count, while observing only few profiling samples. While for direct evaluation the cost is roughly linear and can be scaled by sample count accordingly, caching behaves differently. In the beginning many newly requested tiles have to be evaluated, leading to a high upfront cost. Once the cache is filled sufficiently, this cost declines and only the usually much lower reconstruction cost remains.

Ultimately, we would like to keep the profiling phase as short as possible to make full use of potential optimization gains. Therefore the following section is concerned with predicting the progression of unique requested texture tiles at any sample count, based on data from the profiling phase. This is not only important for predicting the cost of caching for a higher sample count, but also for predicting the required cache space. The overall idea we follow is to use a Poisson process to model texture tile requests.

7.1. Idea

Let $A_n := T(\mathcal{F}_n)$ be the set of active texture tiles at sample count n, i.e. the ones that the renderer requested at sample count n. Our goal is to predict the number of unique tiles $\mathbb{E}[|A_n|]$, given statistics obtained from $k < n$ initial sample about A_k. For now, we assume that each tile $t \in A_\infty$ will be requested with uniform probability. This assumption is reasonable, since mipmap pyramids are designed to achieve this approximately. We assume further that tile requests are accurately described by a Poisson process with rate parameter λ, i.e. the frequency by which each tile is requested. Since each request is independent in the Poisson process, frequency is equal to probability for selecting any specific tile. The number of tiles in the limit will be the reciprocal of this probability, since we assumed it to be uniform:

$$\lambda = \frac{1}{|A_\infty|} \quad \Rightarrow \quad |A_\infty| = \frac{1}{\lambda}. \tag{11}$$

The Poisson assumption also allows us to predict the expected number of requested tiles at sample count n

$$\begin{aligned}\mathbb{E}[|A_n|] &= \mathbb{P}[\text{request count} > 0 \text{ at sample count } n]\,|A_\infty| \\ &= (1-\exp(-\lambda n))\,|A_\infty|.\end{aligned} \tag{12}$$

This reduces our problem to inferring the underlying Poisson rate of the tile request process.

In principle, we found the idea as described above to work well, even though tile requests are not truly independent, since a footprint evaluation can lead to multiple requested tiles. There are however some important considerations to take into account for making this approach practical, which we address in the following.

7.2. Zero-truncated Poisson mixture model

Actual data reveals significant inhomogeneity of the Poisson process, i.e. request probability is not actually uniform. This can be explained by e.g. occlusion, where tiles are only partially visible. In practice we solve this by using a mixture model, i.e. we assume a finite number of different Poisson rates. Further, since we only observe tiles that were requested at least once, our average tile count will be biased towards higher values. We correct this by modeling the average tile count using a zero-truncated Poisson distribution.

Definition. The Poisson distribution has the probability

$$\text{Pois}(k \mid \lambda) = \frac{\lambda^k e^{-\lambda}}{k!}, \tag{13}$$

which we use to define its zero-truncated version by

$$\text{Pois}_+(k \mid \lambda) = \frac{\text{Pois}(k \mid \lambda)}{1-\text{Pois}(0 \mid \lambda)} = \frac{\text{Pois}(k \mid \lambda)}{1-\exp(-\lambda)}. \tag{14}$$

Together with the weights π, we define the M component zero-truncated Poisson mixture, with which we model request counts:

$$p(k) = \sum_{j=1}^{M} \pi_j \,\text{Pois}_+(k \mid \lambda_j). \tag{15}$$

7.3. Model parameter estimation

Data. During the profiling phase, for each requested footprint at a node we compute which tiles would be requested and increment a per-tile counter for each request. Afterward, we compute a histogram of all observed request counts r. Given N different observed requests counts, the histogram is a sequence of tuples $(r_i, o_i)_{1 \le i \le N}$, where o_i is the occurrence count, i.e. how many tiles were requested r_i times.

Maximum likelihood estimate. To obtain the maximum likelihood estimate of a single zero-truncated Poisson component given a mean count $\bar{x}$, we solve the following equation in terms of the Lambert W function:

$$\frac{\lambda}{1-e^{-\lambda}} = \bar{x} \quad \Rightarrow \quad \lambda = \bar{x} + W(-\bar{x}e^{-\bar{x}}) \tag{16}$$

Expectation maximization (EM). We use the EM algorithm [DLR77] to compute a maximum likelihood estimate of our mixture model. In each iteration of the algorithm, it computes an assignment $a_{i,j}$ for each data point i to component j as the expectation under the current mixture:

$$a_{i,j} = \frac{\pi_j \,\mathrm{Pois}_+(r_i \mid \lambda_j)}{p(r_i)} \tag{17}$$

Then it maximizes the likelihood of component parameters, conditional on the assignments:

$$\pi_j = \frac{\sum_{i=1}^N a_{i,j}\, o_i}{\sum_{i=1}^N o_i}, \quad \bar{x}_j = \frac{\sum_{i=1}^N a_{i,j}\, o_i\, r_i}{\sum_{i=1}^N a_{i,j}\, o_i}. \tag{18}$$

We obtain λ_i as in eq. (16).

Initial mixture. To initialize the EM iteration, we begin with a dense initial mixture, i.e. we choose M in the order N, e.g. $M = 0.25N$. To choose each initial λ_j, we sample from the empirical data distribution D, i.e. the discrete distribution with $\Pr(R = r_i) \propto o_i$, and apply a uniform jitter to obtain an initial mean $\bar{x}_j$:

$$\begin{aligned} &R \sim D, \quad J \sim \mathrm{Uniform}(-0.5, 0.5), \\ &\bar{x}_j = \begin{cases} \max\{R + |J|, 1.01\} & \text{if R = 1,} \\ R + J & \text{otherwise.} \end{cases} \end{aligned} \tag{19}$$

Since the initial means follow the data distribution, we set $\pi_j = \frac{1}{M}$.

Merging. To make EM converge more quickly, we remove redundant components by merging after few initial iterations. We merge greedily and proceed in ascending order of λ_j. We merge two components with index j and $j+1$ if they are mutually contained within a tolerance τ of their standard deviations $\sqrt{\lambda}$, i.e. if

$$\lambda_j \geq \lambda_{j+1} - \tau\sqrt{\lambda_{j+1}} \text{ and } \lambda_{j+1} \leq \lambda_j + \tau\sqrt{\lambda_j} \tag{20}$$

Merging can usually reduce the number of components greatly. Afterwards, we proceed with EM iterations until convergence.

7.4. Prediction

We now extend our idea for predicting $\mathbb{E}[|A_n|]$ to our mixture model. Firstly, the inferred Poisson rates λ_j are with respect to the number of observations made and need to be normalized for eqs. (11) and (12) to be applicable.

For estimating $|A_\infty|$, we normalize λ_j to the total number of tile requests $t = \sum_{i=1}^N o_i\, r_i$. We divide these between components using the assignments from EM (eq. (17)):

$$t_j = \sum_{i=1}^N a_{i,j}\, o_i\, r_i. \tag{21}$$

Our estimate for the total number of unique tiles of component j is

$$|A_\infty^j| = \frac{t_j}{\lambda_j}. \tag{22}$$

We use the number of footprints at target sample count $\#\mathcal{F}_n$ and at profiling sample count $\#\mathcal{F}_k$ for estimating the expected number of requested tiles, since we expect $\#\mathcal{F}$ to grow linearly:

$$\mathbb{E}[|A_n|] = \sum_{j=1}^M \mathbb{E}\left[|A_n^j|\right] = \sum_{j=1}^M \frac{t_j}{\lambda_j}\left(1 - \exp\left(-\underbrace{\frac{\#\mathcal{F}_n\,\lambda_j}{\#\mathcal{F}_k}}_{=\frac{n\lambda_j}{k}}\right)\right). \tag{23}$$

Conservative estimation In absence of sufficient data, our approach will underestimate $|A_n|$. This is because mixture components with very low λ are unlikely to be observed. To prevent this systematic bias, we add a single component to our mixture to represent unobserved tiles, such that it has negligible influence on the prediction if enough data was observed. We extend our mixture to $M' = M + 1$ components, use a fixed $\lambda_{M'} = 0.01$, and then choose $t_{M'}$, such that when we plug it into eq. (22), $|A_\infty^{M'}|$ equals 10% of the total estimated unique tiles:

$$|A_\infty^{M'}| = 0.1 \sum_{j=1}^M |A_\infty^j| \quad \Rightarrow \quad t_{M'} = \left(0.1 \sum_{j=1}^M |A_\infty^j|\right) \lambda_{M'}. \tag{24}$$

Then we use eq. (23) as before, with M switched out for M'. Since $\lambda_{M'}$ is set very low, it will only influence the prediction if $k \ll n$.

Accuracy. We show results of our prediction in fig. 7, with varying samples per pixel (spp) for the profiling phase. We compare our predicted tile count to actual tile count in all our scenes (see section 9) and show the distribution of relative error. On average, our prediction is accurate even with few profiling samples, although

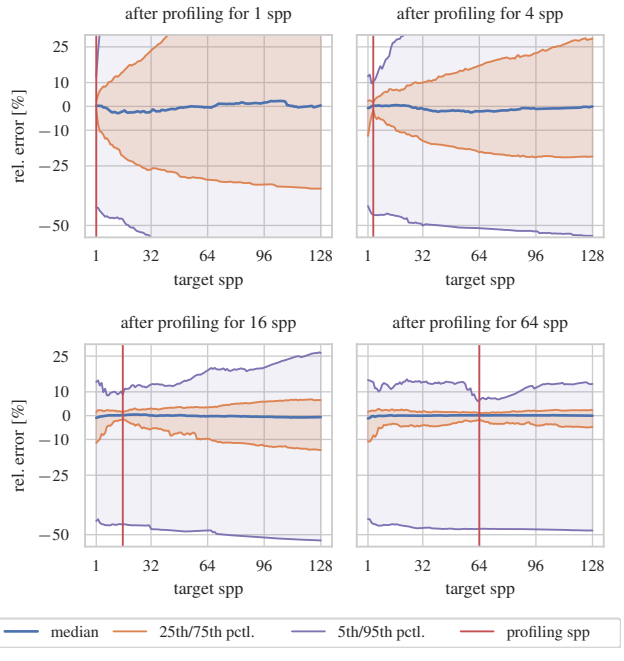


Figure 7: *Distribution of relative error of our tile number prediction. While the prediction is accurate on average and produces low error in most cases when taking sufficient profiling samples, some outliers remain.*

with high variance. Taking more profiling samples generally reduces variance and leads to a low error for most predictions. However, some outliers still remain, even for longer profiling phases. One possible explanation for this behavior could be that we predict the expected value of the number of tiles. Therefore, the error is not purely deviation of the ground truth from our estimate, but also includes the variance of the underlying stochastic process itself. While improving the robustness of this component of our framework is interesting for future work, we believe most error to be low enough for it to be useful in practice.

8. Optimization of caching decision

With the results from previous section, we can estimate how caching a node would affect overall efficiency. We are, however, still missing an algorithm for turning this information into a decision about which nodes to cache. Our task is to solve an optimization problem, where we want to maximize overall efficiency by picking nodes to cache where this increases efficiency, while staying below a threshold for cache capacity. This is essentially the classic knapsack problem, but we will have to add one further restriction by considering only one stage of caching. In the following, we explain this restriction and the inputs to our optimization, as well as an integer linear programming (ILP) formulation. To obtain a solution, we use a general ILP solver without much overhead.

Restriction to single-stage caching. We support only one stage of caching, i.e. we do not consider caching inputs to cached nodes. More formally, there can be at most one caching node on every path from any source node to the graph output (see fig. 8), since multiple stages of caches are often unnecessarily redundant.

8.1. Integer linear programming formulation

In order to share the cache capacity between all graphs in a scene, we perform a single global optimization after the profiling phase. We use the data gathered during the profiling phase to compute costs and predict tile counts.

Variables. For each node i of every graph, we define its weight w_i, i.e. its required cache space, using the expected number of tiles $\mathbb{E}[A_n]$ (see section 7), its number of output channels C_i and the number of footprints in each tile S:

$$w_i = \mathbb{E}[A_n]\, C_i\, S, \tag{25}$$

and its value v_i, i.e. the efficiency gained by caching the node (see section 6), using $\mathcal{F}_n$ to refer to the footprint set at node i:

$$v_i = \max\left\{\mathrm{cost}_\Sigma^{\mathrm{direct}}(\mathcal{F}_n \mid \Theta_i) - \mathrm{cost}_\Sigma^{\mathrm{cache}}(\mathcal{F}_n \mid \Theta_i),\, 0\right\}. \tag{26}$$

As output of our optimization, we introduce the binary variable x_i for every node i:

$$x_i = \begin{cases} 1 & \text{if node } i \text{ is cached,} \\ 0 & \text{otherwise.} \end{cases} \tag{27}$$

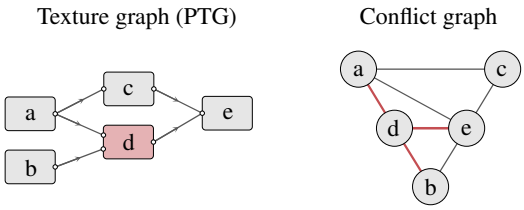


Figure 8: *Left: an example procedural texture graph with five nodes. Node d is marked as a cache node. Right: conflict graph, showing all nodes and edges between all nodes that are conflicted with respect to their cache state. Node b would be in conflict with d, since they share a sub-path on the way to the output node e, the edge between d and e in this example. Node c on the other hand is not in conflict with d and could be cached at the same time.*

Constraints. A solution to our optimization problem must not exceed the global available cache space W:

$$\sum_i w_i x_i \leq W. \tag{28}$$

To implement our single-stage restriction, we compute pairwise conflicts of nodes. Nodes i and j are in conflict if there is any path from a source node to the output node that includes i and j. We compute all conflicts using the reachability relation of all graphs, and add the following constraint for every conflict between a node i and j:

$$x_i + x_j \leq 1. \tag{29}$$

Objective. Under the above constraints, we seek to maximize evaluation efficiency by maximizing the total value

$$\sum_i v_i x_i. \tag{30}$$

9. Evaluation

Implementation. We provide source code of an implementation of our method [SHS*25], which includes the procedural texture graph evaluation with caching and the prediction steps. To obtain footprints from a path tracer and for final renders, we use a CPU-based research rendering framework. We use HiGHS [HSF*25; HH18] to solve our ILP problem instances (see section 8). It usually finds an optimal solution in the order of tens of milliseconds.

Methodology. We first record footprint requests when rendering each of the scenes, separated by material ID. We then run a single texture graph on footprints from one material ID at a time. By performing this for each available combination of scene, texture graph and material ID, we get a distribution of results. In some cases, we will report the distribution with one of these variables kept fixed. Since we are mainly interested in properties of footprint distribution and texture graphs and how they interact, we use our prediction results as a reference. We evaluate the accuracy of our introduced approximations separately in figs. 5 to 7. We presume our cost measurement computation to be accurate enough to not skew results heavily if not using λ_{none}. When our tile number prediction fails, it

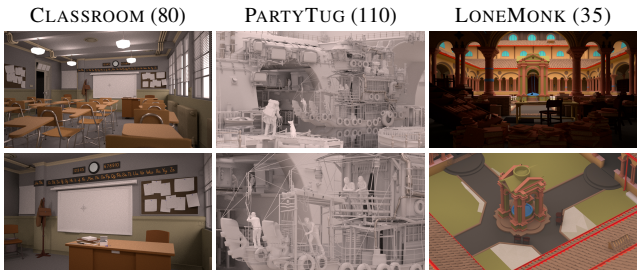


Figure 9: *Primary (top row) and secondary (bottom row) camera angle for the scenes used in our evaluation, with the number of materials next to scene names.*

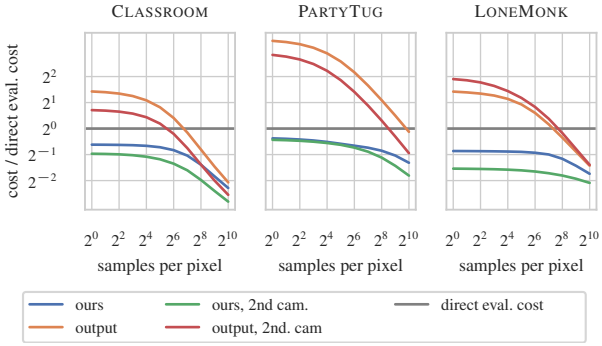


Figure 10: *Our method achieves lower total graph evaluation costs in the three scenes shown. Caching efficiency overall depends on camera angle, as the results with the secondary camera angle show.*

mostly underestimates the total number of tiles, which would be an advantage for other caching methods as well, e.g. always caching the output node.

Data. We use the three scenes shown in fig. 9 to evaluate our method. These are based on Blender demo files [Ble25]. We additionally use a secondary camera angle of each scene for some tests. We record footprints for an output resolution of 1280×720 and limit path length to 8 vertices. For texture graphs, we use the four procedural textures shown in fig. 5, as well as two additional ones (TOOTH, CONCRETE). These were converted from examples files from Adobe Substance Designer [Ado25] to our custom implementation and consist of 39–80 nodes.

Default parameters. We set the parameters of our cost measure (see section 6.1) to conform with our implementation. Since our cache reconstructs from isotropic footprints, we choose $\lambda_{\text{iso}} = 1$ for Θ_{cache}. We also assume only isotropic filtering to be available for source nodes and set $\lambda_{\text{iso}} = 1$ for those, too. For inner nodes we choose $\lambda_{\text{const}} = 1$, since they process footprints mostly independently of their shape or size in our graphs. We set the cost constant c for each node based on benchmarking our implementation on footprint evaluation. We use a profiling phase of $k = 16$ samples per pixel (spp) for predicting tile numbers, and leave the cache capacity unconstrained ($W = \infty$) for optimization.

Potential of caching. In fig. 11, we show the relative cost reduction of our approach compared to using direct evaluation of the texture graph. We compare this to the baseline method of caching requested texture tiles at the output node (*output*). While some texture graphs are more susceptible to caching than others, they display similar trends overall. Although caching will always pay off at some sample count, this can be quite high, as the *output* strategy shows. For a large share of material IDs, caching outputs will not amortize for the considered sample count of 1024, and even lead to an increase in evaluation cost by $2-4\times$. As the solution space of our optimization includes direct evaluation, it will never chose an option that leads to increased cost, given accurate predictions. It cannot improve results for very cacheable material IDs, since for these *output* is generally a good strategy. However, our median cost reduction is strictly better than caching outputs, although with varying significance depending on the graph.

Since relative cost reduction does not consider the significance of the material to the whole scene, we also show absolute cost reductions for whole scenes in fig. 10. We average results for the available texture graphs. Here, our method can show a consistent cost improvement of $2\times$ for lower sample count and up to $4\times$ for higher sample count, while caching outputs only pays off at high sample counts, with still higher total cost. We presume the secondary cameras to perform better for caching output nodes in CLASSROOM and PARTYTUG, since these are more close-up views of the scene.

Limited cache capacity. We investigate the effects of limiting cache capacity in fig. 12. For each material ID and graph combination, we limit cache capacity in relation to the space required for caching requested texture tiles at the output node (*output*). Generally, our method is often able to find good strategies with equal cache space usage as *output*. It also benefits from a slightly increased cache capacity, although this trend does not seem to continue for further increases. For limited capacity, our method transitions smoothly to equal cost with direct evaluation, but is still able to improve upon it with little cache capacity.

Cost measure parameters. We compare the overall effect of different choices of cost coefficients (see section 6.1) in fig. 13. We describe the following options in terms of their difference to the default choice (BASE), which is to set $\lambda_{\text{iso}} = 1$ for source nodes and cache reconstruction, $\lambda_{\text{const}} = 1$ for inner nodes:

CONSTALL: $\lambda_{\text{const}} = 1$ for all nodes and cache reconstruction,
ANISO: $\lambda_{\text{iso}} = 1$ for inner nodes,
SIZE: $\lambda_{\text{none}} = 1$ for all nodes,
CONSTSOURCE: $\lambda_{\text{const}} = 1$ for source nodes,
SIZESOURCE: $\lambda_{\text{none}} = 1$ for source nodes.

When comparing CONSTALL to our default coefficients, there is only a slight shift with our method, while caching outputs is much worse overall. This could be explained by a higher cost to fill the cache compared to direct evaluation. Respectively, we observe that when inner nodes have a cost depending on footprint shape (ANISO) or size (SIZE), caching output nodes is generally a good and reliable strategy. In these cases, cache fill cost is low compared to direct evaluation. Varying cost at source nodes, CONSTSOURCE makes it more costly to cache footprints compared to direct evalu-

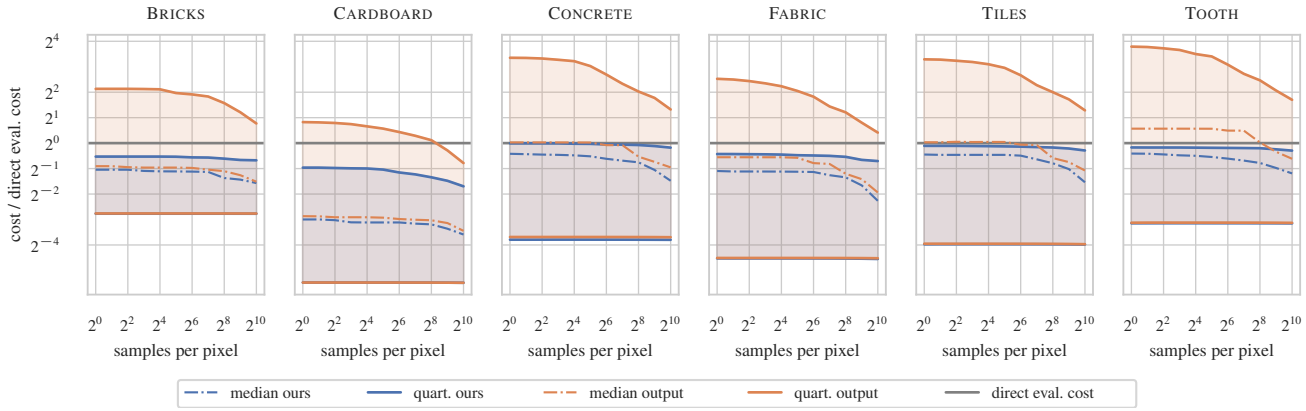


Figure 11: *Texture graphs vary in their suitability for caching. Generally, caching output nodes can achieve similar evaluation cost reduction as our method if the target sample count is sufficiently high, but can also results in increased cost for many material IDs. Our method avoids these cases and achieves lower cost on average.*

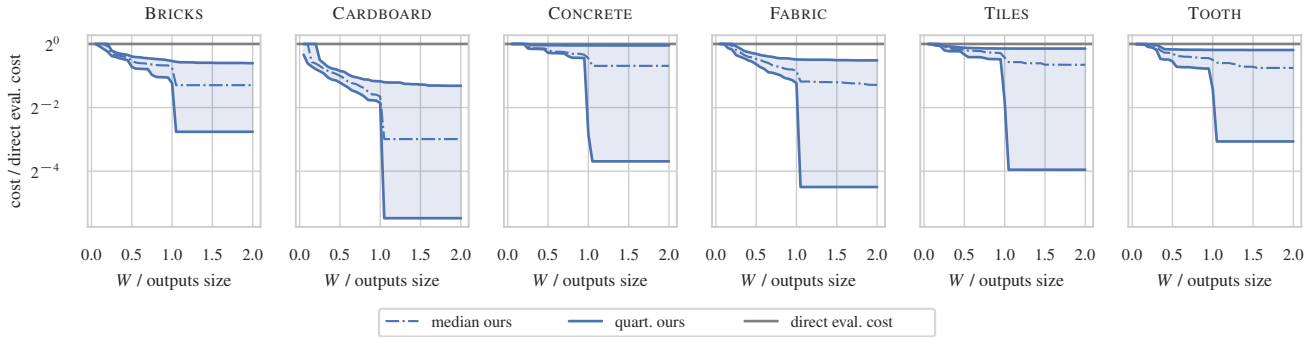


Figure 12: *Constraining the available cache space W reduces the benefit of caching, ultimately resulting in the same cost as direct evaluation of texture graphs. Our method is able to find compromises in between, trading an increase in cost for cache space. Providing more space than for caching output nodes does not seem to further increase efficiency.*

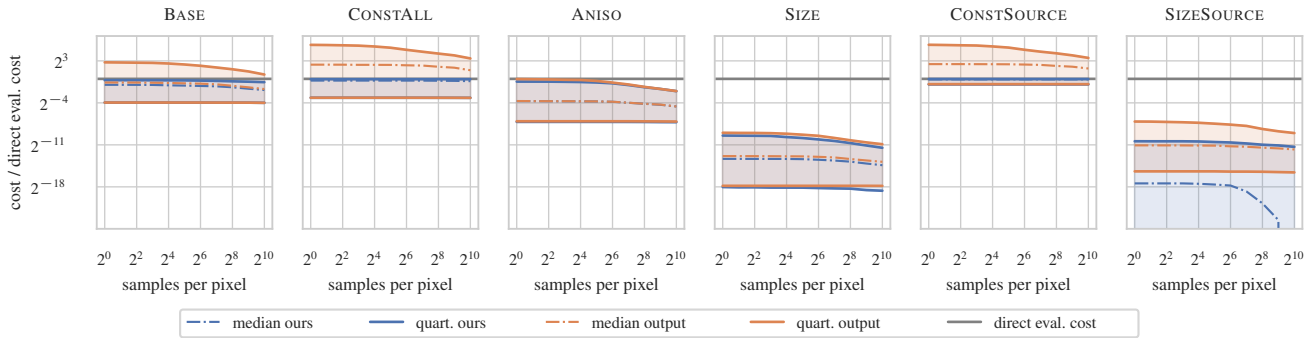


Figure 13: *Comparison of the effect of different parameters for our cost measure. Higher (*CONSTALL, CONSTSOURCE*) or lower (*ANISO, SIZE, SIZESOURCE*) relative cache fill cost make caching less or more attractive, respectively.*

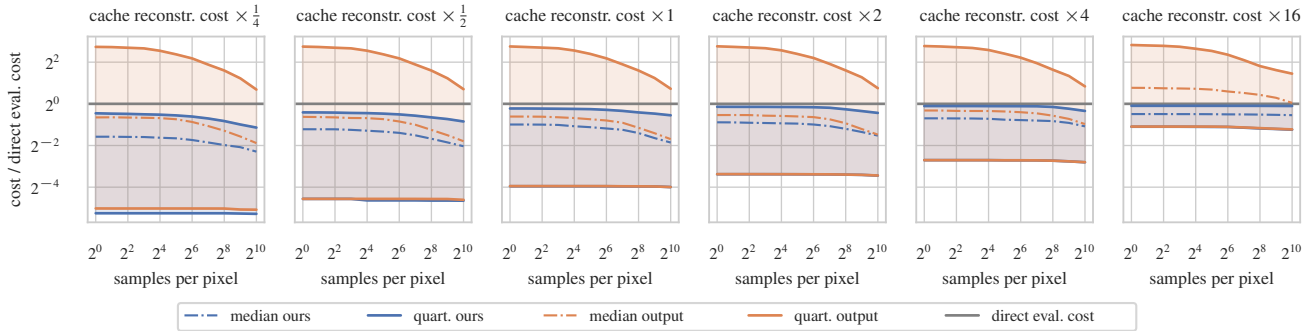


Figure 14: *Higher cost of reconstruction from cache reduces the best case benefits of caching overall. At the same time, worst case behavior of output caching is not affected much, since it is dominated by cache fill cost.*

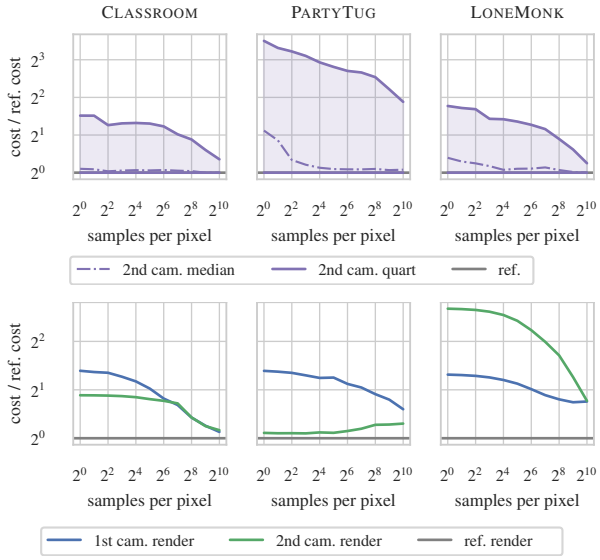


Figure 15: *Optimizing the evaluation strategy on a different footprint distribution leads to suboptimal results. Top: We optimize using our primary camera and render from the secondary camera. Relative cost of each material ID increases in many cases compared to optimizing on the secondary camera (ref.). Bottom: Total cost increases in the three scenes when using this setup. We also include the reverse case, i.e. optimizing on the secondary camera and rendering on the primary camera.*

ation, setting back amortization, while SIZESOURCE has the opposite effect.

We also consider varying cache reconstruction cost in fig. 14 by using a multiple of our default cache cost constant c. While reconstruction cost has a visible influence on the cost reduction that caching can achieve, it seems to not affect worst-case behavior. This could indicate, that in these cases cost is dominated by the cache fill cost.

Generalizability of solutions. We investigate how specific our evaluation strategies are to a footprint distribution in fig. 15. For this, we use our method to obtain an evaluation strategy on the primary camera and evaluate it on the footprints generated from the secondary camera. We compare the relative cost to an evaluation strategy that was optimized for the secondary camera. While many material IDs are not affected by the changed camera, in many cases cost is much higher without the specific optimization. We also show how the total cost of the scene is affected and include the reversed test of using the secondary camera to optimize. This demonstrates that the overall efficiency of evaluation strategies can depend significantly on the distribution of footprints.

10. Conclusion

We introduced a framework to examine the performance and resource usage of rendering systems with a procedural texture graph at the core of their shading system. We showed that the efficiency of a shading strategy and evaluations crucially depends on the frequency, distribution, and shape of the queries performed during rendering. These requests to the shading system originate from a path tracer and are decisively characterized by their pixel footprint. We provide a statistical prediction method to optimize the location of cache nodes in a procedural texture graph, taking performance, resource usage, and anticipated frequency of queries at a target sample rate into account. Our framework can indicate to rendering engineers whether shade-on-hit or precomputation are viable options for their application. Since our data collection and optimization scheme is relatively light weight, it can also be implemented into an offline rendering system directly, to select optimal cache node locations. There are several directions for future research, including adaptation to GPU rendering and more advanced path sampling algorithms, as well as extension to the temporal domain. GPU architectures might demand changes to the cost model as well as to the implementation of our profiling phase to accomodate for increased parallelism. More advanced sampling algorithms like path guiding could affect the accuracy of our prediction, since we assume path sampling to be invariable throughout rendering. The temporal domain might be interesting to support rendering movie sequences while exploiting shared, time-independent computations in the shading.

Acknowledgements

This work received funding from the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – project number 431478017.

References

[ÁBWM16] ÁFRA, ATTILA, BENTHIN, CARSTEN, WALD, INGO, and MUNKBERG, JACOB. "Local Shading Coherence Extraction for SIMD-efficient Path Tracing on on CPUs". *Proceedings of High Performance Graphics*. 2016. DOI: 10.2312/hpg.20161198.

[Ado25] ADOBE. *Substance 3D*. 2025. URL: https://www.adobe.com/products/substance3d.html.

[Ama84] AMANATIDES, JOHN. "Ray Tracing with Cones". *Computer Graphics (Proceedings of SIGGRAPH)* 18.3 (July 1, 1984), 129–135. DOI: 10/b64smh.

[ANA*19] AKENINE-MÖLLER, TOMAS, NILSSON, JIM, ANDERSSON, MAGNUS, et al. "Texture Level of Detail Strategies for Real-Time Ray Tracing". *Ray Tracing Gems: High-Quality and Real-Time Rendering with DXR and Other APIs*. 2019, 321–345. DOI: 10.1007/978-1-4842-4427-2_20.

[BAC*18] BURLEY, BRENT, ADLER, DAVID, CHIANG, MATT JEN-YUAN, et al. "The Design and Evolution of Disney's Hyperion Renderer". *ACM Transactions on Graphics* 37.3 (July 2018), 33:1–33:22. DOI: 10/gfjm8w.

[Ble25] BLENDER FOUNDATION. *Blender*. 2025. URL: https://www.blender.org.

[BN12] BRUNETON, ERIC and NEYRET, FABRICE. "A Survey of Non-Linear Pre-Filtering Methods for Efficient and Accurate Surface Shading". *IEEE Transactions on Visualization and Computer Graphics* 18.2 (Feb. 2012), 242–260. DOI: 10/chq5qh.

[BYRN17] BELCOUR, LAURENT, YAN, LING-QI, RAMAMOORTHI, RAVI, and NOWROUZEZAHRAI, DEREK. "Antialiasing Complex Global Illumination Effects in Path-Space". *ACM Transactions on Graphics (Proceedings of SIGGRAPH)* 36.4 (Jan. 2017). DOI: 10/ggfg5m.

[CCC87] COOK, ROBERT L., CARPENTER, LOREN, and CATMULL, EDWIN. "The Reyes Image Rendering Architecture". *Computer Graphics (Proceedings of SIGGRAPH)* 21.4 (1987), 95–102. DOI: 10/bk6dpr.

[CFS*18] CHRISTENSEN, PER, FONG, JULIAN, SHADE, JONATHAN, et al. "RenderMan: An Advanced Path-Tracing Architecture for Movie Rendering". *ACM Transactions on Graphics* 37.3 (Aug. 2018), 30:1–30:21. DOI: 10/gfznbs.

[DBLW15] DORN, JONATHAN, BARNES, CONNELLY, LAWRENCE, JASON, and WEIMER, WESTLEY. "Towards Automatic Band-Limited Procedural Shaders". *Computer Graphics Forum* 34.7 (2015), 77–87. DOI: 10.1111/cgf.12747.

[DHI*13] DUPUY, JONATHAN, HEITZ, ERIC, IEHL, JEAN-CLAUDE, et al. "Linear Efficient Antialiased Displacement and Reflectance Mapping". *ACM Transactions on Graphics (Proceedings of SIGGRAPH Asia)* 32.6 (Nov. 2013), 211:1–211:11. DOI: 10/gbd53f.

[DLR77] DEMPSTER, A. P., LAIRD, N. M., and RUBIN, D. B. "Maximum Likelihood from Incomplete Data Via the EM Algorithm". *Journal of the Royal Statistical Society: Series B (Methodological)* 39.1 (Sept. 1977), 1–22. DOI: 10.1111/j.2517-6161.1977.tb01600.x.

[ENSB13] EISENACHER, CHRISTIAN, NICHOLS, GREGORY, SELLE, ANDREW, and BURLEY, BRENT. "Sorted Deferred Shading for Production Path Tracing". *Computer Graphics Forum (Proceedings of the Eurographics Symposium on Rendering)* 32.4 (2013), 125–132. DOI: 10/gfzv8r.

[FH22] FUJIEDA, SHIN and HARADA, TAKAHIRO. "Progressive Material Caching". *SIGGRAPH Asia 2021 Technical Communications*. Nov. 22, 2022. DOI: 10.1145/3550340.3564223.

[FHL*18] FASCIONE, LUCA, HANIKA, JOHANNES, LEONE, MARK, et al. "Manuka: A Batch-Shading Architecture for Spectral Path Tracing in Movie Production". *ACM Transactions on Graphics* 37.3 (Aug. 2018), 31:1–31:18. DOI: 10/gfznbt.

[FS24] FOURNIER, ROMAIN and SAUVAGE, BASILE. "Mix-Max: A Content-Aware Operator for Real-Time Texture Transitions". *Computer Graphics Forum* 43.6 (Sept. 2024). DOI: 10.1111/cgf.15193.

[GH86] GREENE, NED and HECKBERT, PAUL S. "Creating Raster Omnimax Images from Multiple Perspective Views Using the Elliptical Weighted Average Filter". *IEEE Computer Graphics and Applications* 6.6 (1986), 21–27. DOI: 10.1109/MCG.1986.276738.

[GHS*22] GUERRERO, PAUL, HAŠAN, MILOŠ, SUNKAVALLI, KALYAN, et al. "MatFormer: A Generative Model for Procedural Materials". *ACM Transactions on Graphics (Proceedings of SIGGRAPH)* 41.4 (July 22, 2022), 46:1–46:12. DOI: 10/gqjn68.

[GIF*18] GEORGIEV, ILIYAN, IZE, THIAGO, FARNSWORTH, MIKE, et al. "Arnold: A Brute-Force Production Path Tracer". *ACM Transactions on Graphics* 37.3 (Aug. 2018), 32:1–32:12. DOI: 10/gfznb2.

[Gri08] GRITZ, LARRY. *OpenImageIO*. 2008. URL: https://github.com/AcademySoftwareFoundation/OpenImageIO.

[GSDT22] GRENIER, C., SAUVAGE, B., DISCHLER, J-M., and THERY, S. "Color-mapped noise vector fields for generating procedural micro-patterns". *Computer Graphics Forum (Proceedings of Pacific Graphics)* 41.7 (2022), 477–487.

[GSKC10] GRITZ, LARRY, STEIN, CLIFFORD, KULLA, CHRIS, and CONTY, ALEJANDRO. "Open Shading Language". *ACM SIGGRAPH Talks*. 2010, 1–1. DOI: 10.1145/1837026.1837070.

[Hec89] HECKBERT, PAUL. *Fundamentals of Texture Mapping and Image Warping*. Tech. rep. UCB/CSD-89-516. June 1989. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/1989/5504.html.

[HH18] HUANGFU, Q. and HALL, J. A. J. "Parallelizing the dual revised simplex method". *Mathematical Programming Computation* 10 (2018), 119–142. DOI: 10.1007/s12532-017-0130-5.

[HKD15] HANIKA, JOHANNES, KAPLANYAN, ANTON, and DACHSBACHER, CARSTEN. "Improved Half Vector Space Light Transport". *Computer Graphics Forum (Proceedings of the Eurographics Symposium on Rendering)* 34.4 (July 1, 2015), 65–74. DOI: 10/gfzv83.

[HL90] HANRAHAN, PAT and LAWSON, JIM. "A Language for Shading and Lighting Calculations". *Computer Graphics (Proceedings of SIGGRAPH)* 24.4 (1990), 289–298. DOI: 10/dttqmv.

[HNPN13] HEITZ, ERIC, NOWROUZEZAHRAI, DEREK, POULIN, PIERRE, and NEYRET, FABRICE. "Filtering Color Mapped Textures and Surfaces". *Proceedings of the Symposium on Interactive 3D Graphics and Games*. Mar. 2013, 129–136. DOI: 10/gfzp5z.

[HNPN14] HEITZ, ERIC, NOWROUZEZAHRAI, DEREK, POULIN, PIERRE, and NEYRET, FABRICE. "Filtering Non-Linear Transfer Functions on Surfaces". *IEEE Transactions on Visualization and Computer Graphics* 20.7 (2014), 996–1008. DOI: 10.1109/TVCG.2013.102.

[HSF*25] HUANGFU, QI, SCHORK, LUKAS, FELDMEIER, MICHAEL, et al. *HiGHS*. 2025. URL: https://highs.dev.

[Ige99] IGEHY, HOMAN. "Tracing Ray Differentials". *Annual Conference Series (Proceedings of SIGGRAPH)*. Vol. 33. Aug. 1999, 179–186. DOI: 10/c2t9t9.

[Int22] INTEL. *Intel® Arc™ Graphics Developer Guide for Real-Time Ray Tracing in Games*. 2022. URL: https://www.intel.com/content/www/us/en/developer/articles/guide/real-time-ray-tracing-in-games.html.

[KCSG18] KULLA, CHRISTOPHER, CONTY, ALEJANDRO, STEIN, CLIFFORD, and GRITZ, LARRY. "Sony Pictures Imageworks Arnold". *ACM Transactions on Graphics* 37.3 (Aug. 2018), 29:1–29:18. DOI: 10/gfjkn7.

[KFF*15] KELLER, ALEXANDER, FASCIONE, LUCA, FAJARDO, MARCOS, et al. "The Path-Tracing Revolution in the Movie Industry". *ACM SIGGRAPH Courses*. 2015, 24:1–24:7. DOI: 10/gfzp5t.

[KHD14] KAPLANYAN, ANTON S., HANIKA, JOHANNES, and DACHSBACHER, CARSTEN. "The Natural-Constraint Representation of the Path Space for Efficient Light Transport Simulation". *ACM Transactions on Graphics (Proceedings of SIGGRAPH)* 33.4 (July 2014), 102:1–102:13. DOI: 10/f6cz85.

[LGXT17] LEE, MARK, GREEN, BRIAN, XIE, FENG, and TABELLION, ERIC. "Vectorized Production Path Tracing". *Proceedings of High Performance Graphics*. 2017. DOI: 10.1145/3105762.3105768.

[LLC*10] LAGAE, A., LEFEBVRE, S., COOK, R., et al. "A Survey of Procedural Noise Functions". *Computer Graphics Forum* 29.8 (2010), 2579–2600. DOI: 10.1111/j.1467-8659.2010.01827.x.

[LWS*25] LI, BEICHEN, WU, RUNDI, SOLAR-LEZAMA, ARMANDO, et al. "VLMaterial: Procedural Material Generation with Large Vision-Language Models". *The Thirteenth International Conference on Learning Representations*. 2025.

[NVI22] NVIDIA. *Shader Execution Reordering*. 2022. URL: https://developer.nvidia.com/blog/improve-shader-performance-and-in-game-frame-rates-with-shader-execution-reordering/.

[Pea90] PEACHEY, DARWIN. *Texture On Demand*. Technical Memo 217. Pixar, Jan. 1990.

[PWSF24] PHARR, MATT, WRONSKI, BARTLOMIEJ, SALVI, MARCO, and FAJARDO, MARCOS. "Filtering after Shading with Stochastic Texture Filtering". *Proceedings of the ACM on Computer Graphics and Interactive Techniques* 7.1 (May 2024), 14:1–14:20. DOI: 10/gtvf3c.

[RLD*12] REINER, TIM, LEFEBVRE, SYLVAIN, DIENER, LORENZ, et al. "A runtime cache for interactive procedural modeling". *Computers & Graphics* 36.5 (2012), 366–375.

[SHS*25] SCHÜSSLER, VINCENT, HANIKA, JOHANNES, SAUVAGE, BASILE, et al. *Implementation of "Selective Caching in Procedural Texture Graphs for Path Tracing"*. June 2025. DOI: 10.5281/zenodo.15585724.

[SLH*20] SHI, LIANG, LI, BEICHEN, HAŠAN, MILOŠ, et al. "Match: Differentiable Material Graphs for Procedural Material Capture". *ACM Transactions on Graphics (Proceedings of SIGGRAPH Asia)* 39.6 (Nov. 27, 2020), 196:1–196:15. DOI: 10/ghnnkm.

[SW01] SUYKENS, FRANK and WILLEMS, YVES D. "Path Differentials and Applications". *Rendering Techniques (Proceedings of the Eurographics Workshop on Rendering)*. 2001, 257–268. DOI: 10/bq6pdj.

[Uhl95] UHLMANN, JEFFREY. "Dynamic map building and localization: New theoretical foundation". PhD thesis. University of Oxford, 1995.

[Wil83] WILLIAMS, LANCE. "Pyramidal Parametrics". *Computer Graphics (Proceedings of SIGGRAPH)* 17.3 (July 1, 1983), 1–11. DOI: 10/cq4xrd.

[WM24] WEST, REX and MUKHERJEE, SAYAN. "Stylized Rendering as a Function of Expectation". *ACM Transactions on Graphics (Proceedings of SIGGRAPH)* 43.4 (July 2024), 96:1–96:19. DOI: 10/gt48vh.

[YB18] YANG, Y. and BARNES, C. "Approximate Program Smoothing Using Mean-Variance Statistics, with Application to Procedural Shader Bandlimiting". *Computer Graphics Forum (Proceedings of Eurographics)* 37.2 (2018), 443–454. DOI: 10/gd2jb7.

[ZD20] ZIRR, TOBIAS and DACHSBACHER, CARSTEN. "Path Differential-Informed Stratified MCMC and Adaptive Forward Path Sampling". *ACM Transactions on Graphics (Proceedings of SIGGRAPH Asia)* 39.6 (Nov. 27, 2020), 246:1–246:19. DOI: 10/gsmm9d.