Karlsruhe Institute of Technology

Material Graph Memoization in Real-Time Path Tracing

Master Thesis of

Bastian Urbach

At the Department of Informatics
Institut für Visualisierung und Datenanalyse,
Lehrstuhl für Computergrafik

May 22, 2023

Reviewer:	Prof. Dr.-Ing. Carsten Dachsbacher
Second reviewer:	Prof. Dr. Hartmut Prautzsch
Advisor:	Vincent Schüßler

Abstract

Material graphs are commonly used in offline rendering and increasingly in real-time rendering to model the look of surfaces in a flexible, programmable way that is still accessible to artists. Using material graphs in a real-time path tracer is expensive because the path tracer must evaluate many graphs per pixel and frame and because the evaluation is often highly divergent for secondary rays.

In this thesis, I apply the memoization pattern to material graph evaluation in a real-time path tracer. Memoization refers to storing the result of a function in a cache for it to be reused in future calls with the same parameters. I build on an existing paper [FH22] that focuses on offline path tracing of individual images.

The main challenge of implementing material graph memoization is developing a cache data structure that supports fast and massively parallel reading and writing and uses hardware caches well. I describe two different approaches. Both store the data as a set of virtual textures in a hash table but the first one stores one texel per cache entry while the second one stores small tiles. The tiled variant is more complex but enables bilinear interpolation of cache entries and improves hardware cache locality. In both variants, cache lookup and update are safeguarded against concurrent access. I parallelize the computation of missing tiles in the tiled variant using wave operations. When the cache is full, an eviction strategy decides which entries will be overwritten first. I compare three eviction strategies: random eviction, least-recently-written and least-recently-used.

In my tests, I achieved cache hit rates of up to 99.98% and speedups of up to 680%. The speedup is highly scene-dependent and higher for small scenes with many complex material graphs. The tiled variant performs slightly better than the non-tiled one. Material graph memoization results in a slight loss in image quality. Caching only secondary hits still improves performance significantly but produces near-perfect images.

Kurzfassung

Material Graphs werden oft im Offline-Rendering und zunehmend auch im Echtzeit-Rendering verwendet, um das Aussehen von Oberflächen auf eine flexible, programmierbare Weise zu modellieren, die dennoch für Künstler zugänglich ist. Die Verwendung von Material Graphs in einem Echtzeit-Pathtracer ist teuer, weil der Pathtracer viele Graphen pro Pixel und Frame auswerten muss, und weil die Auswertung für Sekundärstrahlen stark divergent ist.

In dieser Arbeit wende ich das Memoization Entwurfsmuster auf Material Graphs in einem Echtzeit-Pathtracer an. Memoization bezeichnet das Speichern des Ergebnisses einer Funktion in einem Cache mit dem Ziel, es in zukünftigen Aufrufen mit den selben Parametern wiederzuverwenden. Ich baue auf einer exitierenden Arbeit auf [FH22], die sich auf Offline-Pathtracing einzelner Bilder konzentriert.

Das zentrale Problem bei der Implementierung von Material Graph Memoization ist die Entwicklung einer Cache-Datenstruktur, die schnellen und massiv parallelen, schreibenden und lesenden Zugriff ermöglicht und die die Hardware Caches effizient nutzt. Ich beschreibe zwei verschiedene Ansätze. Beide speichern die Daten als virtuelle Texturen in einer Hashtabelle, jedoch speichert der erste nur einen Texel pro Cache-Eintrag, während der zweite kleine Kacheln speichert. Die Kachel-Variante ist aufwendiger, ermöglicht aber bilineare Interpolation von Cache-Einträgen und nutzt die Caches der Hardware besser. In beiden Varianten sind Cache-Lookup und Cache-Update gegen konkurrenten Zugriff geschützt. Ich parallelisiere die Berechnung fehlender Kacheln in der Kachel-Variante mithilfe von Wave-Operationen. Wenn der Cache voll ist, entscheidet eine Eviction-Strategie, welche Einträge zuerst überschrieben werden. Ich vergleiche drei Eviction-Strategien: zufällige Eviction, least-recently-written und least-recently-used.

In meinen Tests habe ich Hit-Raten von bis zu 99.98% und Speedups von bis zu 680% erreicht. Der Speedup hängt stark von der Szene ab und ist höher in kleinen Szenen mit vielen teuren Material Graphs. Die Kachel-Variante ist etwas schneller, als die ohne Kacheln. Material Graph Memoization führt zu einem geringfügigen Verlust von Bildqualität. Den Cache nur für Sekundärtreffer zu nutzen verbessert die Performance immer noch signifikant, erzeugt aber nahezu perfekte Bilder.

Contents

1. Introduction

Realistic rendering of three-dimensional scenes requires not only a model of the geometry in the scene but also a model of how light interacts with the surface of each object. For a single surface point, this interaction is modeled by a bidirectional reflectance distribution function (BRDF). A separate problem is describing how the BRDF varies over the surface of an object. Many surfaces are not perfectly homogenous but have spatially varying reflective properties. For example, wooden surfaces typically have characteristic patterns of darker and brighter areas from the annual rings of the tree. Aside from such intrinsic variation, many surfaces have impurities, smudges or scratches. Including this variation in the geometric model is possible in theory but requires a very detailed geometric model, which is difficult to create and expensive to render. Instead, the traditional way of describing these phenomena is by mapping 2D raster textures to the surface of an object. These can for example be captured from the real world or drawn by an artist. However, as with other parts of a 3D rendering pipeline, making the appearance of a surface programmable provides more flexibility and is desirable for both artistic and technical reasons.

Material graphs are a flexible and artist-friendly way of modeling the look of a surface in 3D rendering programmatically. They are widely used in offline rendering and increasingly in real-time rendering as well, for example in Unity with Shader Graph [Uni23] or in the Unreal Engine with the Material Editor [Epi23]. Material graphs provide an intuitive, high-level mechanism for describing materials, that is decoupled from the more technical parts of 3D rendering. Nodes in a material graph represent operations and constants. A node can take several inputs and produce several outputs. The flow of values from output to input sockets is represented by the edges of the graph. While it is generally desirable to give artists more control and flexibility in how they design materials, it also comes at the risk of a severe performance impact. Fully or partially procedural materials can be much more expensive to evaluate than simply sampling a single texture per BRDF parameter. By hiding the complexity of procedural texturing, material graphs also hide its cost. This problem becomes even more relevant if a real-time path tracer is used instead of rasterization-based techniques. Due to the recursive nature of a path tracer, material graphs may be evaluated many times per frame and pixel. Furthermore, random sampling in a path tracer is likely to cause divergent branching as secondary rays hit different materials.

One very general approach for improving the performance of certain functions is the memoization pattern. This pattern stores the results of frequently evaluated functions in a cache indexed by the function parameters so that subsequent calls with the same parameters do not need to evaluate the function again. Memoization has successfully been used in combination with offline path tracing of individual images [FH22]. While the same approach can be used for real-time path tracing as well, there is still room for improvement. A common strategy in real-time path tracing is to make use of temporal continuity by reusing results from previous frames. Similarly, a cache that persists across frames combined with an effective eviction strategy has the potential to significantly improve performance as there is often a great deal of overlap between surface points visible in consecutive frames.

Compared to memoization on a CPU, implementing memoization in a GPU path tracer introduces some additional challenges and additional design decisions must be made. The cache data structure must support massively parallel reading and writing while keeping the content consistent. GPU performance also relies on the effective use of hardware caches as global memory bandwidth is a common bottleneck. Furthermore, like all GPU code, an efficient implementation should avoid divergent branching to make better use of SIMD parallelism.

In this thesis, I explore and compare different variants of material graph memoization in the context of real-time path tracing. I group the variants into one group that stores individual texels of virtual textures in a hashtable-based cache, and another one that works similarly but stores small tiles of texels. The more complex, tile-based approach enables bilinear filtering of cached values and has the potential to improve the locality of memory accesses. A central question is whether these advantages justify the higher complexity and overhead. I also compare different eviction strategies as well as different memory layouts for the tile-based approach. Eviction strategies differ in their computational overhead and additional memory usage and in how accurately they predict the reuse of cache entries. The choice of memory layout for the tile-based approach affects the locality of memory reads and thereby how well hardware caches are used.

I begin by providing context and groundwork for the techniques presented later, followed by an overview of related work. Then I describe the core ideas of my memoization techniques, followed by more detailed chapters on the implementation of the different variants, first for the non-tiled and then for the tile-based variants. Finally, I evaluate and compare the performance of the presented techniques and conclude with recommendations for their use as well as potential paths for further research.

2. Context and Groundwork

The memoization techniques that I present in this thesis are designed as one component of a real-time path tracer. This chapter provides an overview of some techniques and theoretical background that I use in my implementation and that I will refer to throughout the thesis. I begin with a brief explanation of path tracing and my choice of BRDFs. I also cover ray cones as an important building block for real-time path tracing in general and material graph memoization in particular. I explain the concept of a material graph as well as the generic optimization pattern of memoization. Finally, I give an overview of some relevant aspects of graphics hardware and APIs that I use in my techniques or that influenced their design.

2.1 BRDFs and Path Tracing

Rendering realistic images requires simulating how light interacts with a scene. The first step for this is to describe how light interacts with a single surface point. Surface points can emit light as well as reflect or transmit incoming light. The rendering equation [Kaj86, PJH16] describes these interactions:

$$L_o(x, \omega_o) = L_e(x, \omega_o) + \int_{S^2} f(x, \omega_o, \omega_i) L_i(x, \omega_i) |\cos \theta_i| d\omega_i \tag{2.1}$$

- x is the surface point,
- $L_o(x, \omega_o)$ is the outgoing radiance from x in direction ω_o,
- $L_e(x, \omega_o)$ is the radiance emitted by x in direction ω_o,
- S^2 is the unit sphere (or hemisphere for opaque surfaces), i.e. the set of all directions of incoming light,
- $f(x, \omega_o, \omega_i)$ is the bidirectional reflectance distribution function (BRDF),
- $L_i(x, \omega_i)$ is the incoming radiance at x from direction ω_i,
- θ_i is the angle between the surface normal and ω_i.

For simplicity, I assume all surfaces to be opaque in this thesis, i.e. I assume transmission of light through a surface to be zero. L_e is an inherent property of the surface point and non-zero only if x is a light source. As explained in more detail later on, L_i is approximated by a path tracer via ray tracing and recursive evaluation of the rendering equation at the first ray intersection. Various models are used for f. In the simplest possible case, f is constant:

$$f_{Lambert}(\omega_o, \omega_i) = f_{Lambert} = \frac{k_d}{\pi} \tag{2.2}$$

- $k_d \in [0, 1]$ is the diffuse color of the surface.

This models diffuse reflection on a perfectly smooth surface and is known as Lambertian reflection [PJH16]. I use this as an approximation for all diffuse reflection, though more accurate models for diffuse reflection on rough surfaces exist as well. I use the GGX BRDF to model specular reflection on rough surfaces. GGX is a microfacet BRDF, i.e. it models the surface as a collection of microscopic facets with a certain statistical distribution. The general form of a specular microfacet model is as follows [WMLT07]:

$$f(x, \omega_o, \omega_i) = \frac{F(\omega_i, \omega_h)G(\omega_i, \omega_o, \omega_h)D(\omega_h)}{4|\omega_i \cdot \omega_n||\omega_o \cdot \omega_n|} \tag{2.3}$$

- ω_n is the normal vector,
- $\omega_h = \frac{\omega_i + \omega_o}{||\omega_i + \omega_o||}$ is the half-vector,
- $F(\omega_i, \omega_h)$ is the Fresnel term,
- $G(\omega_i, \omega_o, \omega_h)$ is the shadowing-masking term,
- $D(\omega_h)$ is the microfacet distribution.

The Fresnel term F describes how much radiance is reflected (as opposed to transmitted) at the boundary between two materials. More light is reflected at grazing angles. I use Schlick's approximation [Sch94]:

$$F_{Schlick}(\omega_i, \omega_h) = F_0 + (1 - F_0)(1 - \omega_i \cdot \omega_h)^5 \tag{2.4}$$

F_0 is the reflection coefficient for $\omega_i = \omega_o = \omega_h$. It can either be computed from the indices of refraction or chosen artistically as the "specular color" of the surface.

The masking-shadowing term G describes what portion of the microsurface is shadowed (occluded by other microfacets as seen from ω_i) or masked (occluded by other microfacets as seen from ω_o). I use the separable Smith joint masking-shadowing function [Hei14].

$$G_{Smith}(\omega_i, \omega_o, h) = G_1(\omega_i, \omega_h)G_1(\omega_o, \omega_h) \tag{2.5}$$

$$G_1(\omega, \omega_h) = \frac{2|\omega_n \cdot \omega|\chi^+(\omega_h \cdot \omega)}{|\omega_n \cdot \omega| + \sqrt{\alpha^2 + (1 - \alpha^2)(\omega_n \cdot \omega)^2}} \tag{2.6}$$

- $\alpha \in [0, 1]$ is the roughness of the surface,
- $\chi^+(x) = \begin{cases} 1 & x > 0 \\ 0 & x \leq 0 \end{cases}$ is the Heaviside function.

The microfacet distribution D is the probability density of microfacet normals. Evaluating it at ω_h gives the differential probability that light from direction ω_i is reflected in direction ω_o. The GGX microfacet distribution is as follows [WMLT07]:

$$D_{GGX}(\omega_h) = \frac{\alpha^2\chi^+(\omega_n \cdot \omega_h)}{\pi((\omega_n \cdot \omega_h)^2(\alpha^2 - 1) + 1)^2} \tag{2.7}$$

Evaluating the rendering equation requires calculating an integral over the unit sphere or hemisphere. One term in the integrand is the incoming radiance L_i. In a path tracer,

$L_i(x, \omega_i)$ is calculated by tracing a ray from x in direction ω_i and evaluating the rendering equation recursively at the first intersection with the scene. However, this can only be done for a finite number of directions. The integral is therefore approximated with Monte Carlo integration.

Monte Carlo integration is a randomized numeric integration algorithm. It involves sampling random points in the integration domain and evaluating the integrand function at these points. The expected value of the function is then given by another integral, which is approximated by a sum over a finite number of samples [SM09]:

$$E\left[f(X)\right] = \int_A f(x)p(x)d\mu(x) \approx \frac{1}{N}\sum_{i=1}^{N} f(x_i) \tag{2.8}$$

- A is a domain,
- f is a function defined on A,
- $\mu(x)$ is a measure on A,
- p is a probability density on A,
- X is a random variable in A with probability density p,
- $x_1, x_2, \ldots, x_N$ are instantiations of X

By the law of large numbers, this approximation converges almost certainly towards the correct value for large N. By rearranging this formula, the original integral over f is approximated similarly:

$$\int_A f(x)d\mu(x) \approx \frac{1}{N}\sum_{i=1}^{N} \frac{f(x_i)}{p(x_i)} \tag{2.9}$$

The expected value of this approximation is the value of the integral but the approximation contains some variance, which is visible as noise in images generated by a path tracer. Importance sampling is an important technique for reducing the variance. Importance sampling refers to choosing a probability density p that is approximately proportional to f. As an edge case, if p is proportional to f then $\frac{f(x)}{p(x)}$ is constant and the approximation is exact.

To apply this algorithm to the global illumination problem, directions of incoming radiance at a surface point are sampled from the unit sphere or hemisphere and the rendering equation is evaluated recursively for the closest surface points in the sampled directions. Importance sampling is possible in various ways, for example by using a distribution that is proportional or approximately proportional to the BRDF or by sampling directions towards light sources with a higher probability.

The path tracer I wrote for this thesis is relatively simple in that it only supports point lights and decouples direct lighting from indirect lighting. For direct lighting, I sample from the finite set of point lights in the scene. For indirect lighting, I use BRDF importance sampling.

2.2 Ray Cones

High-frequency features in textures lead to aliasing artifacts when a texture is viewed from far away. A widely used solution to this problem is mipmapping [Wil83]. Mipmapping refers to providing a set of additional versions (mipmap levels) of a texture with exponentially decreasing resolution. The lower-resolution versions only contain low-frequency features. When a texture is sampled, the renderer decides dynamically which version of the texture is appropriate for the current sampling density. In rasterization-based rendering, this is approximated by the numerical derivatives of the texture coordinates in screen space. The length of these derivatives approximately describes the footprint of a pixel in texture space. The renderer then chooses a mipmap level where the pixel has a footprint of approximately one texel. The numerical derivatives are calculated efficiently using the texture coordinates at adjacent pixels.

In path tracing, no numerical screen-space derivatives are readily available but mipmapping is still beneficial. While aliasing artifacts are reduced by the Monte Carlo integration, aliasing still increases the variance and thereby the visible noise in the final image. Mipmapping is also important for the caching techniques I present in this thesis because using a lower resolution and thereby fewer texels where appropriate translates to more cache hits. Ray cones [AMCB+21] are a technique for computing approximate derivatives of texture coordinates that is compatible with path tracing. When using ray cones, rays used in path tracing are conceptually extended to cones that increase in thickness along the ray. This representation allows computing the approximate footprint of a pixel in a texture based on the width of the cone at the intersection. Intersections with the scene are still determined with ray tracing, ignoring the cone width. Primary rays have an opening angle corresponding to the size of one pixel on the screen. When a ray is reflected by a surface, the width and opening angle of the cone is carried over to the new ray and the opening angle is increased based on the curvature and roughness of the surface that was hit.

The width of a ray cone at an intersection is calculated using the small angle approximation:

$$w_{new} = w_{old} + \alpha t \tag{2.10}$$

- w_{new} is the width of the ray cone at the intersection,
- w_{old} is the width of the ray cone at the last surface interaction or zero for primary rays,
- α is the opening angle of the ray cone,
- t is the distance of the intersection along the ray.

The new opening angle after a surface interaction is calculated as $\alpha_{new} = \alpha_{old} + \beta_r + \beta_c$, where β_r describes the spread caused by the roughness of the surface and β_c describes the spread caused by the curvature of the surface. For GGX, β_r is calculated as follows:

$$\beta_r = 2\sqrt{\frac{\alpha_{GGX}^2}{2 - 2\alpha_{GGX}^2}} \tag{2.11}$$

- α_{GGX} is the roughness parameter of the GGX BRDF.

The approximate spread angle from curvature is calculated as follows:

$$\beta_c = -2k\frac{w}{n \cdot d} \tag{2.12}$$

- k is the curvature of the surface,
- n is the normal of the surface,
- d is the direction of the ray.

To compute derivatives of vertex attributes from a ray cone, an approximate elliptical footprint of the cone on the surface is determined by locally approximating the cone with a cylinder and the surface with a plane. The points of the ellipse that are furthest from

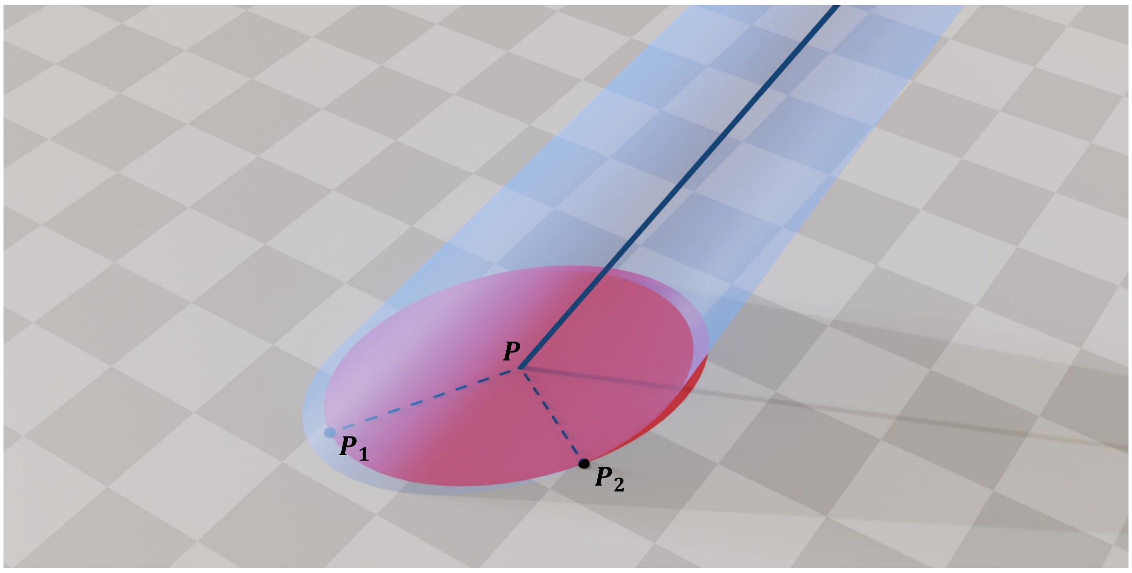


Figure 2.1: A ray (black), the corresponding ray cone (blue) and its approximate elliptical footprint (red). P_1 and P_2 are a vertex and a co-vertex of the ellipse and are used for computing derivatives.

its center along its major and minor axis are called its vertices and co-vertices respectively and are computed as follows:

$$h_1 = \text{ProjectOnPlane}(d, n) \tag{2.13}$$

$$h_2 = n \times h_1 \tag{2.14}$$

$$P_1 = P + \frac{h_1 r}{\|\text{ProjectOnPlane}(h_1, d)\|} \tag{2.15}$$

$$P_2 = P + \frac{h_2 r}{\|\text{ProjectOnPlane}(h_2, d)\|} \tag{2.16}$$

- $\text{ProjectOnPlane}(v, n) = v - (v \cdot n)n$,
- P is the ray intersection and center of the ellipse,
- P_1 is a vertex of the ellipse,
- P_2 is a co-vertex of the ellipse.

Values of vertex attributes at these points are obtained via barycentric interpolation. These values at the vertices and co-vertices are then used to compute two numerical derivatives of the vertex attribute. This is similar to how the values of adjacent pixels are used to compute numerical derivatives in rasterization-based renderers and the derivatives can be used in the same way for mipmapping and anisotropic filtering.

2.3 Material Graphs

Describing real-world materials such as stone or wood requires not just a local illumination model but also a description of how the reflective properties, such as roughness or albedo, vary across the surface. One way of doing this is by mapping 2D textures onto an object to supply parameters to a BRDF. A more flexible way is to describe it programmatically

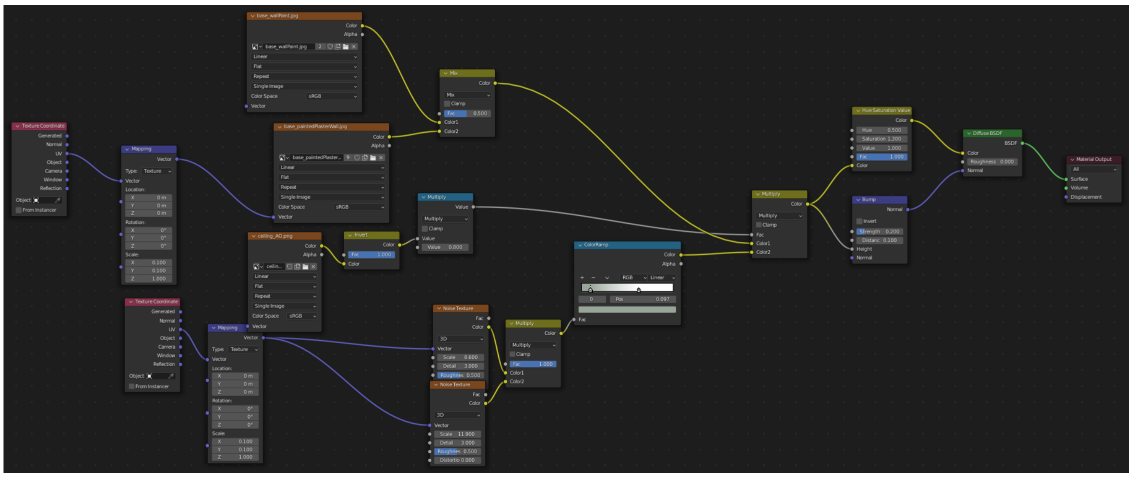


Figure 2.2: An example material graph from the *Classroom* scene in the Blender Shader Editor [Ble23].

in a shader. This can for example involve sampling one or multiple textures, distorting texture coordinates, evaluating noise functions and other procedural patterns as well as blending colors and other values in various ways. An advantage of this approach is that it allows creating materials with a great deal of variation with lower memory usage than a single large texture. Disadvantages include that complex materials are more expensive to evaluate and that writing shaders in text-based programming languages is not very artist-friendly. The latter is addressed by material graphs. Material graphs are a type of visual scripting that decouples material description from more technical aspects of shader programming. Material graphs follow the dataflow programming paradigm. They are usually translated to an imperative, text-based shading language such as HLSL or GLSL before being used in a renderer. The nodes of a material graph represent constant values and operations, such as sampling a texture, while the edges describe the flow of values between nodes. Material graph systems differ in how they treat BRDFs. For example, in the Blender Shader Editor [Ble23] (figure 2.2), BRDFs are simply nodes that can be combined in various ways. Unity Shader Graphs [Uni23] on the other hand only supply BRDF parameters to a single active target that handles all lighting calculations. I implemented my own material graph system backend following the latter approach for this thesis, which I describe in section 5.1.

2.4 Memoization

Memoization is a general pattern for improving the performance of commonly used functions. The core idea is that once a function has been evaluated, its result should be kept in memory in case it is needed again later. Subsequent calls of the same function with the same parameters return the value stored in memory instead of executing the function. If the function has parameters, multiple values must be stored in a cache data structure indexed by the parameters. The return values of the function must also depend exclusively on a known set of parameters and the function must not have side effects (i.e. the function must be *pure*). If the return value cannot be derived exclusively from the parameters, then storing them is not useful because the stored value would not necessarily be the correct one for a subsequent call with the same parameters. If the function has side effects, then it must be evaluated anyway to ensure that these side effects occur. Trivial examples of functions that cannot be memoized are a function that returns a random number or one that writes a value into a log.

If querying the cache is faster than evaluating the function, then calls where the value is already cached evaluate faster than without memoization. However, querying the cache before evaluating the function and storing the result after evaluation slows down function calls where the value was not already cached. For memoization to be effective, looking up and storing values should therefore be significantly faster than evaluating the function and function values should be reused frequently. The speedup is described by the following formula:

$$S = \frac{(T_f + T_r + T_w) \cdot (1 - h) + T_r \cdot h}{T_f} \tag{2.17}$$

- S is the speedup achieved by using memoization,
- T_f is the average time required to evaluate the function,
- T_r is the average time required for looking up a value in the cache,
- T_w is the average time required for writing a value to the cache,
- h is the probability of a cache hit.

A natural choice for the cache data structure is a hash table as it allows lookups in expected constant time. A hash table stores key-value pairs. Internally it uses an array to store the pairs. The index of a pair is given by a hash of its key. Implementations differ in how they treat hash collisions, i.e. in how they store multiple key-value pairs with the same hash. One approach is to store a linked list of key-value pairs at each index in the hash table. Another is linear probing, which searches the array linearly for a free index, starting at the index given by the hash. It is also possible to store a fixed-size array for each index in the hash table since the cache used for memoization is often limited in size anyway to prevent it from growing indefinitely. Regardless of which one of these approaches is used, if the cache is to be limited in size, decisions must be made on which values should be stored in the cache and when they should be overwritten. These decisions are made by an eviction strategy. Some general eviction strategies are used, such as the *least recently used* strategy, which dictates that whenever a value is to be stored in a full cache or a full section of a cache, the value that was used the least recently is to be overwritten with the new value. In addition to such generic strategies, it is also possible to use high-level domain knowledge about the function or the code that uses it in an eviction strategy.

2.5 Relevant Aspects of Graphics Hardware and APIs

Modern GPUs are complex, highly parallel processors with various hardware and firmware components designed to accelerate workloads commonly needed in computer graphics. Understanding their architecture and the APIs that expose it to the user is vital for implementing efficient path tracing with memoization.

2.5.1 SIMD Architecture

GPUs process data in many threads that are executed in parallel. The threads are organized in small groups of e.g. 32 threads [NVI09] called *waves*, *wavefronts*, *warps* or *subgroups*. I will use the term *wave* in this thesis. Waves are executed on SIMD processing units. The threads within a wave are executed in parallel using the SIMD approach. This means that all threads in a wave execute the same code paths at the same time on different sets of data. While the execution of a branch can be masked per thread to emulate per-thread branching, divergent branching is still a performance concern as masked threads have to wait for the branch to be completed instead of doing useful work. Multiple SIMD units can execute different waves in parallel. Furthermore, multiple waves can be *in flight* on a single SIMD unit. While only one of them is active, the unit can switch between them to hide long latencies, caused, for example, by global memory reads.

2.5.2 Wave Operations

The organization of threads in waves is mostly opaque to the shader programmer. However, newer versions of graphics APIs like Vulkan and DirectX provide instructions for efficient communication between the threads of a wave [Hen18, WBC$^+$21]. These include instructions for querying the wave size, reading values from other threads, and aggregation instructions such as the sum or the minimum of a value across the wave. In HLSL, they are represented by intrinsic functions with the prefix *Wave*, such as *WaveGetLaneCount*, *WaveReadLaneAt*, or *WaveActiveMin*. These instructions are useful for redistributing work within a wave in a way that makes better use of SIMD parallelism. For example, if a shader requires evaluating a function a varying number of times, the evaluations in a wave can be distributed evenly among the threads of the wave.

2.5.3 Atomic Operations and Synchronisation

GPU threads can read and write global memory concurrently. This can result in race conditions. For example, attempting to append a value to a list from two threads at the same time may result in one thread overwriting the value written by the other one. To alleviate this problem, GPUs provide instructions to perform various operations atomically [WCJS20]. In HLSL, these are represented by intrinsic functions with the prefix *Interlocked*, such as *InterlockedAdd*, *InterlockedMin* or *InterlockedExchange*. While only relatively simple operations can be performed atomically, they can be used to implement synchronization primitives and thereby eliminate race conditions for more complex tasks. For example, to implement a mutex, a thread uses InterlockedExchange to swap a value in a buffer with a value indicating that the mutex is locked. The value obtained from the buffer indicates to the thread whether it was the first thread to attempt locking the mutex.

2.5.4 Hardware Cache Hierarchy

Like most modern processors, GPUs employ a hardware cache hierarchy [NVI09] to speed up memory accesses that follow certain patterns. While this is always the case, even without the software caching techniques that I describe in this thesis, hardware cache locality should be considered for implementing an efficient software cache. Shader variables are typically stored in registers but when a shader attempts to read from a memory object, such as a texture or a buffer, the GPU will first query its caches for the requested value. The caches are organized in cache lines representing a small section of adjacent memory addresses. Reading the same amount of data from adjacent or nearby addresses is therefore faster than reading it from addresses that are spread over a larger section of memory. The caches are shared between multiple threads. In particular, the L1 cache is shared between the threads on one SIMD processing unit while the L2 cache is a single cache shared between all threads. This means that cache locality should not only be considered for consecutive memory reads within a thread but also among the threads in a wave or even globally. Graphics APIs provide different ways of making graphics memory available to shaders for different use cases. These include structured buffers as well as 1D, 2D and 3D textures. Choosing one of these indicates to the GPU driver what kinds of memory access patterns are to be expected. For example, accesses to a 2D texture should be expected to be local in the 2D texture space. This enables the GPU to use a memory layout that makes better use of the hardware cache. The exact memory layout used internally is opaque to the shader programmer.

2.5.5 Ray Tracing Hardware and APIs

Modern GPUs have dedicated hardware for parallelized ray tracing, which can for example be used via the DirectX Raytracing API (DXR) [Cor23]. DXR defines two ways of using hardware-accelerated raytracing. One option is to write a set of ray tracing shaders to be executed as part of a ray tracing pipeline. The entry point of a ray tracing pipeline is a ray generation shader. From within the ray generation shader, rays can be traced through an acceleration structure via an opaque instruction (*TraceRay* in HLSL). For each intersection with the scene or only the first one depending on the configuration, the GPU then invokes a hit shader that has access to information about the intersection, such as the primitive index or the barycentric coordinates of the intersection with a triangle. A payload struct can be used to pass input and output parameters between the caller and the callee. Hit shaders may also trace additional rays recursively. Different objects in the scene can be associated with different hit shaders to allow for material-specific code. Even though no explicit branching instruction is used, invoking the correct hit shader still involves branching. While evaluating one hit shader, threads that hit an object associated with a different hit shader are inactive. The other way of using ray tracing hardware is via ray queries. When using ray queries, only a single shader participates in ray tracing. This can be a compute shader or a fragment shader. Similarly to the ray tracing pipeline, the shader can trace rays with an opaque instruction but the information about the intersection is returned to the same shader instead of passed to a separate hit shader. Explicit branching is necessary to execute material-specific code. For both approaches, the user must first create an acceleration structure that is stored in graphics memory and contains all information about the scene that is necessary for ray tracing.

3. Related Work

My material graph memoization techniques are related to multiple groups of other techniques. First, memoization has been used for other purposes in path tracing. Second, memoization has been used to memoize final shading results in rasterization-based renderers. Third, my memoization cache acts as a set of virtual textures and therefore has some things in common with various virtual texturing techniques. Finally, the paper "Progressive Material Caching" [FH22] was the main inspiration for this thesis and describes a similar technique in the context of an offline path tracer. In this section, I discuss some of these techniques and their similarities with and differences to my techniques.

3.1 Other Applications of Memoization in Path Tracers

Memoization and caches are used in various ways to accelerate path tracing. An early example is irradiance caching [WRC88], which interpolates between and extrapolates from cached samples of diffuse irradiance to avoid the expensive Monte Carlo integration over the entire hemisphere. The cache is an octree with adaptive resolution to represent both high and low-frequency areas efficiently. Radiance caching [KGPB05] extends the technique to glossy reflection by caching the spherical or hemispherical radiance function at a surface point. A more recent variant of radiance caching [MRNK21] uses a neural network as the cache data structure. Visibility caching [PGSD13] uses a hashtable-based cache to memoize point-to-point visibility.

These techniques have in common that their main goal is to reduce the number of rays that need to be traced. By contrast, my techniques reduce the cost of material graph evaluation, which is a separate but in some scenes significant part of the total cost of path tracing.

3.2 Memoization of Final Shading Results

Memoization has also been applied to the final shading results in rasterization-based real-time rendering. This also includes material graph evaluation if material graphs are used.

Decoupled sampling [RKLC^{+}11] reuses shading results in multiple visibility samples to reduce shading times when many visibility samples per pixel are generated for antialiasing or effects like motion blur and defocus blur. The technique memoizes shading results in a hashtable-based cache. Before shading is performed for a visibility sample, the cache is queried and the cached shading result is used on a cache hit. Visibility samples have a position (x, y) on the screen, a position (u, v) on the lens and a time parameter t. To compute the cache key, they are projected to the screen position where the same surface point is visible in the center of the lens at the beginning of the exposure interval and then snapped to the closest pixel. That way, multiple visibility samples are mapped to the same cache key, which enables reuse. Decoupled sampling is related to my material graph memoization in that it memoizes the results of per-fragment operations in a hash table. An important difference to the techniques I describe is that the final, view-direction-dependent shading result is cached and cache entries are therefore not reused over multiple frames.

Caching of fragment shader results has also been proposed in the form of a hardware cache in mobile GPUs [APX14]. A single output color per fragment shader is stored in a set-associative cache indexed by a hash of the fragment shader inputs. To increase reuse, the eight least significant bits of the mantissa of a float input are ignored. Redundant fragments were found to occur mostly across consecutive frames and rarely within the same frame. This makes the cache ineffective in conventional GPUs because frames are too large and therefore the reuse distances too high for the relatively small hardware cache. However, in GPUs with parallel frame rendering, the reuse distances were found to be much lower and the cache enables the reuse of fragments across two frames that are processed in parallel. In principle, caching the final fragment color should hinder temporal reuse but mobile games often use 2D graphics or view-direction-independent shading, which makes this less of a concern.

3.3 Virtual Texturing and Memoization of Procedural Textures

Another group of techniques related to material graph memoization are techniques that sparsely cache textures in graphics memory either to reduce the cost of procedural textures or to reduce the graphics memory usage of large textures.

Shadermaps [JPC00] are a technique for accelerating procedural texturing. They require manually splitting a shader into a static and a dynamic phase. The static phase consists of computations that are likely to be reused across frames while the dynamic phase contains the remaining, typically view-direction-dependent computations, such as evaluating BRDFs. The results of the static phase are generated on-demand and stored in small tiles in a sparse mipmap, which acts as a cache indexed by texture coordinates. This is conceptually similar to the tile-based cache that I describe in this thesis. Shadermaps were found to enable around 95% reuse of the data produced by the static phase. The authors do not explain in detail how the sparse mipmap is implemented. In particular, they do not mention hash tables as a possible implementation. Shadermaps were described as a technique for offline rendering on CPUs. My thesis focuses on the implementation of a hashtable-based cache in the context of real-time GPU path tracing. My techniques do not require explicitly separating the static and dynamic phases but the representation as a material graph with a single, known BRDF implies such a separation.

Another system using a tile-based cache is unified texture management for arbitrary meshes as proposed by Lefebvre et. al. [LDN04]. The main goal of the system is to reduce the required graphics memory for large textures by storing only the currently needed parts of each texture. The system consists of three components: a texture load map, a texture cache and a texture producer. The texture load map determines which tiles of each texture are needed for the next frame. This is done by rasterizing scene objects in texture space with a resolution of one pixel per tile. The texture cache stores the currently active tiles. It consists of a tile pool containing the actual texture data and an indirection grid for each texture that maps tile coordinates to tile pool entries. Finally, the texture producer is responsible for loading the tiles requested by the texture load map. The texture producer can also decompress data or evaluate procedural textures. One key difference to my techniques is that the required tiles are determined before rendering of the current frame begins, which is more challenging in a path tracer. Another is the use of multiple indirection maps instead of a single hash table.

A recent paper [ZCJE22] that focuses on texture streaming in geographic information systems presents a technique that automatically groups nearby and similarly oriented primitives into texture atlases in a preprocessing step and then streams them from secondary memory (e.g. hard drives) into a sparse texture on the GPU. The streaming workload per frame is adapted to measured frame latencies to prevent stuttering. The system uses a perceptual importance metric calculated in the fragment shader to prioritize the streaming of textures that have a great influence on the overall look of the image. The paper is less closely related to my techniques because it focuses on streaming textures from secondary memory while my techniques fill virtual textures directly on the GPU by evaluating material graphs.

3.4 Progressive Material Caching

The techniques I present in this thesis are inspired by and can be viewed as an extension of progressive material caching [FH22]. Progressive material caching improves the performance of a path tracer by caching the outputs of nodes of a material graph. Values are stored in a sparse virtual mipmapped texture. The data structure used for this is a hash table indexed by node id, texture coordinates and mipmap level. View-direction-dependent nodes are not cached. Contrary to my version of material graph memoization, values are only reused within one frame and no eviction of cache entries is performed. This makes the technique more suitable for offline path tracing with many samples per pixel and frame as opposed to real-time path tracing.

4. Material Graph Memoization

The goal of the techniques in this thesis is to improve the performance of a real-time path tracer by reducing the time required to evaluate material graphs. A material graph must be evaluated for each ray hit and ultimately results in a BRDF that is then used to compute direct and indirect lighting. If BRDF importance sampling is used then the BRDF is required before tracing the next ray, which means that ray tracing and material graph evaluation are interlaced. Especially for secondary hits, material graph evaluation requires divergent branching as the randomized secondary rays hit different objects with different materials. Evaluating potentially complex material graphs divergently for each ray hit is a significant factor for the overall performance of the path tracer. Material graph memoization alleviates this problem by caching material graph outputs for future reuse.

4.1 Redundancy of Material Graph Evaluations

Two rays almost never hit exactly the same point and each point can produce a different set of material graph outputs. However, evaluating material graphs at a limited resolution is sufficient to render realistic images and ray cones provide a way of estimating the required resolution at each hit, similar to how they estimate the required texture resolution when used to select a mipmap level. Assuming this limited resolution given by ray cones, some material graph evaluations are redundant and can be prevented by storing and reusing the graph outputs. Furthermore, temporal continuity can be used to reduce material graph evaluations significantly as most points that are visible in the current frame (via primary or secondary rays) were likely already visible in previous frames. Especially when only relatively few paths per pixel are traced due to hardware limitations, temporal reuse becomes the main motivation for using a material graph cache. An important nuance is that while the set of relevant points for the entire frame exhibits a great deal of temporal continuity, the mapping of these relevant surface points to pixels is often a very chaotic one (depending on the roughness of the surfaces) due to the random sampling in the path tracer. This motivates a global approach for temporal reuse as opposed to a technique based on temporal reprojection.

4.2 Basic Structure of Material Graph Memoization

Using a material graph cache means that threads will traverse one of two different code paths for each ray hit (figure 4.1). Threads that had a cache hit read the material graph

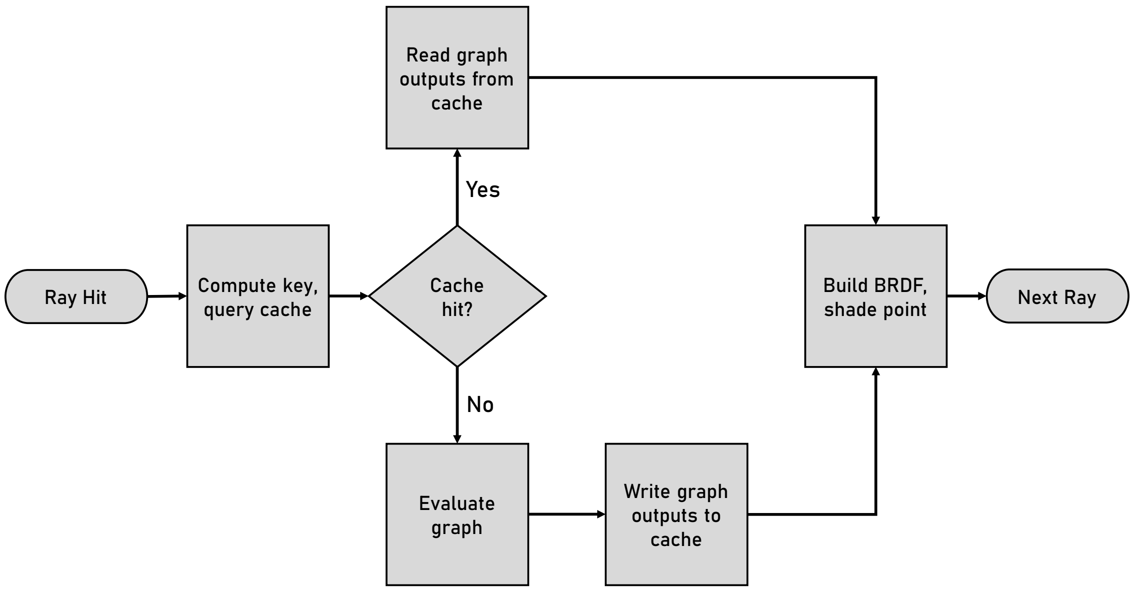


Figure 4.1: Abstract overview of material graph memoization in a path tracer. Threads processing a ray hit take one of two code paths depending on whether the point is already in the cache or not. On a cache hit, the thread reads the graph outputs from the cache. On a cache miss, the thread evaluates the graph and writes its outputs to the cache.

outputs from the cache and continue directly with shading. Threads that had a cache miss must first evaluate the material graph and then write the results into the cache before continuing with shading. At first glance, it may seem unwise to introduce additional divergent branching. However, because material graph evaluation is already highly divergent by itself, using a cache actually reduces divergence in scenes with many different materials. All threads that had a cache hit take the same code path while they would otherwise potentially take different ones to evaluate different material graphs.

Two important questions for designing a material graph cache are which values should be stored and by what keys they should be referred to. The relevant information gained from evaluating a material graph are mainly the emission and the BRDF. Storing an entire BRDF in a generic form would be expensive but material graphs typically use a fixed set of one or multiple parameterized BRDFs and only the parameters vary within the material. It is therefore sufficient to store the parameters of the BRDFs in the cache to obtain the same BRDF without reevaluating the entire graph. Typical parameters are roughness or base color. Additionally, emission should be cached, as well as normals if normal mapping is used.

Choosing appropriate keys for referring to values in the cache is not a trivial task because material graphs can receive a wide variety of input values. These include, for example, texture coordinates, surface normal, vertex colors, view direction, current time (for animated materials) and object id. The key should be chosen in such a way that two points with the same key also have identical output values, i.e. the relation of keys and values should be right-unique. On the other hand, the key should be small and fast to compute for performance reasons. It is also better for performance to calculate the key in the same way for all materials as it avoids divergent branching but that is not strictly necessary.

Vertex attributes do not vary completely independently. In particular, if there is no overlap in texture space, the relation of texture coordinates and any other vertex attribute is right-unique. In that case, including other vertex attributes in the key is unnecessary. In most

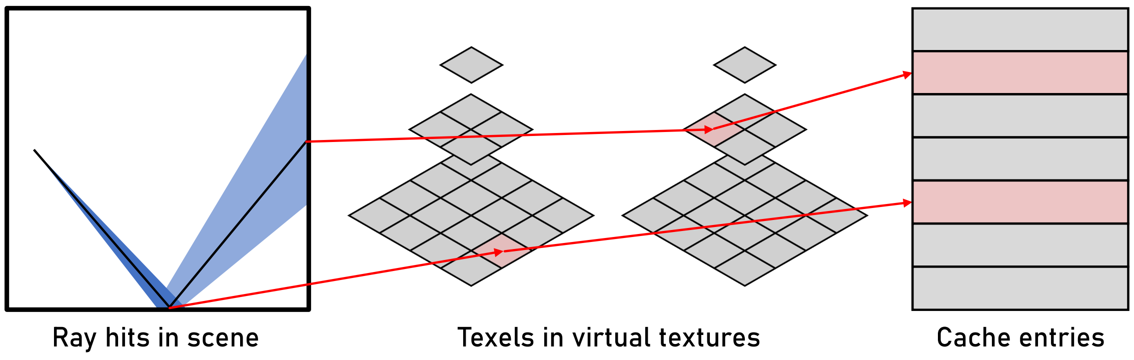


Figure 4.2: Organisation of the material graph cache as sparse, virtual, mipmapped textures. A ray hit in the scene is first mapped to a texel in a mipmap level of a virtual texture. Each virtual texture corresponds to one material graph. The mipmap level is determined by the ray cone footprint. The texels are then mapped to cache entries using a hash function. The virtual textures only exist as a conceptual intermediate step, not as separate objects in memory.

situations where UV coordinates do overlap, it is easy to prevent by instead repeating the texture space and mapping overlapping primitives to different copies of a texture. For example, while one may choose to use only a single leaf texture for a tree, the leaf texture can simply be set to repeat infinitely in texture space and each leaf can be mapped to a different instance of the same texture. Time dependency is a problem for material graph caching. While including the current time in the key is possible, it would significantly reduce the potential for reuse as values would never be reused across multiple frames. However, time dependency is rare and excluding animated materials from memoization is simple. View direction dependency mostly occurs when BRDFs are evaluated as part of the graph. However, even in systems that do represent BRDFs as nodes, BRDF evaluation and sampling can usually be cleanly separated into a second phase after the rest of the graph has been evaluated. For example, in Blender shading graphs, BRDFs can be combined with other nodes but no node produces a scalar, vector or color from a BRDF. It is therefore always possible to first compute all BRDF parameters and then feed them to a second graph containing all BRDFs. With these considerations, I decided to include texture coordinates and material id but no other graph inputs in the key.

4.3 Organisation of the Cache as Virtual Textures

As mentioned in section 4.1, it is necessary to limit the resolution of the cache based on ray cone footprints to enable reuse. This is achieved by treating the cache as a set of sparse, virtual, mipmapped textures (figure 4.2). Each cache entry represents one texel. The mipmap level is determined with ray cones and must be part of the cache key. To sample the virtual texture with nearest-neighbor filtering, the cache key is rounded to the nearest point on the texel grid. It is worth noting that this is still possible if other vertex attributes are also included in the key. Derivatives for all vertex attributes can be derived from ray cones and mipmapping works for any number of dimensions. The sparse texel storage makes high-dimensional virtual textures relatively unproblematic since the surface of an object is still a 2D space and only texels on the surface are stored. Better texture filtering than nearest-neighbor is not efficient in this variant. In principle, one could perform a separate cache lookup for each required texel. For example, bilinear filtering would result in four cache lookups. However, this makes using the cache significantly more expensive while also reducing the probability of a cache hit because all required texels must already be cached. I also describe a different variant using a tile-based cache (section 4.6) that solves these problems.

Even storing only texture coordinates, material id and mipmap level in the key would already require a substantial amount of memory relative to the size of the payload. Furthermore, comparing large keys is a problem for the performance of cache lookups. Smaller keys also have the advantage that they can be read atomically, which prevents race conditions without explicit synchronization. A simple way of reducing the size of the key to any desired size is by hashing it. This introduces the risk of false cache hits as a result of collisions in this hash function. While one may be tempted to use the same hash as was used to determine the location in the hash table, computing a second hash with a different hash function reduces the probability of false cache hits because both the location in the hash table and the second hash of the key must match. This is rare enough even with relatively short keys of 32 bits to not noticeably affect image quality.

4.4 Hash Table as Cache Data Structure

I use a hash table as the cache data structure. Hash tables are a good fit for this use case for several reasons. First, they allow lookups in expected constant time. This is vital because the cache must store several million entries and cache lookups must be significantly faster than evaluating the material graph for the cache to be effective. Furthermore, the flat internal structure of hash tables makes them well-suited for concurrent reading and writing. Writing an entry in a hash table affects only one small section of memory. This makes synchronization relatively simple and efficient.

With realistic cache sizes, it is common for multiple entries to be mapped to the same location in the hash table. Contrary to hash table implementations used for application logic, a hash table used as a cache is not strictly required to tolerate any number of such collisions. Not storing a value in the hash table affects performance but still yields the correct result. Nonetheless, tolerating a certain frequency of collisions by allowing the cache to store multiple entries for the same hash table location is desirable for performance reasons. I decided to use the linear probing approach for this but with a low maximum search distance (e.g. eight entries). During a cache lookup, I determine a base index in the hash table as usual but then compare the keys not only for that entry but for a few following entries as well. If any of those keys match, I report a cache hit and use the corresponding cached value. Otherwise, I report a cache miss. Similarly, when inserting a cache entry, I look through the same few entries following the base index for a suitable location to insert the entry. "Suitable" can mean several things as explained in section 4.5. In the simplest possible case, it means that the index is so far unused.

4.5 Eviction

Graphics memory is limited and the amount of data produced by caching the material graph outputs for each ray hit is high. Because temporal reuse is crucial to make the cache efficient in real-time path tracing, I cannot simply start each frame with an empty cache. Clearing the cache less often, for example, every N-th frame would make the cache more efficient on average but still does not work well for real-time applications because a real-time application is expected to deliver frames at a reasonably consistent rate. Periodically interrupting smooth user interaction with slow frames would negatively impact the user experience almost as much as a consistently low frame rate. Instead of such relatively simple solutions, an eviction method that continuously evicts certain cache entries to make room for new ones is needed. Choosing an eviction method is a tradeoff between predicting the reuse of cache entries accurately and the higher computational overhead and memory usage of more accurate techniques. Perfectly predicting reuse is not possible because neither future user interaction, nor the results of random sampling in the path tracer are known.

Aside from the option to not evict cache entries at all, I implemented and compared three eviction methods. The first method is to evict a random entry within the linear probing range. This method has the advantages that it is very easy to implement and that no memory accesses are required to choose a location, meaning that it introduces practically no overhead compared to not evicting entries at all. The disadvantage is that it does not attempt to predict which entries will be useful in the future.

The second method is to evict the entry from the linear probing range that was least recently written. This is based on the assumption that each cache entry has an interval of relevance and that the duration of this interval is very roughly similar for all cache entries. From this assumption follows that the oldest cache entry is the most likely one to have become irrelevant. The justification for the assumption is the same as for temporal reuse in general: camera and object motion are usually relatively smooth and it's rare for a point to be visible (directly or via reflection) for only a single frame. One problem in implementing this method is that standard HLSL provides no way of measuring time. However, the exact times are not relevant. It is sufficient to know the order in which cache writes occurred and even just knowing the frame when a cache entry was written is a good heuristic since object and camera motion occur between frame rendering. This method increases memory usage by a time (or "pseudo-time") field in each cache entry and introduces the computational overhead of finding the cache entry where it has the lowest value.

The third method is to evict the entry from the linear probing range that was least recently used. This method is similar to the previous one except that the time field is also updated when a cache entry is read. This increases the overhead because the field needs to be written more often. The reasoning behind this variant is that it relies less on the assumption that camera and object motion are smooth than the previous method. When a cache entry is being used, it is by definition still relevant and when a cache entry has not been used for a long time, a likely reason is that it is no longer relevant. It should be noted that this concept of "relevance intervals" is somewhat flawed in path tracing because some rays are only traced with very low probability, which means that the hit points may also be very unlikely to reappear in the next few frames. It is however still a good heuristic for the majority of ray hits.

4.6 Tile-Based Texel Storage

The variant described so far has two problems. First, only nearest-neighbor sampling of the virtual textures in the cache is efficient. Second, using a hash table has the disadvantage that any locality of cache reads in the texture space of the virtual textures is eliminated by the hash function. Particularly for primary rays, but also for certain secondary rays, the ray hits in a wave are likely to be close to each other both in world space and in texture space. However, even texels that are directly next to each other in texture space are usually mapped to very distant locations in memory by the hash function. This means that hardware caches are not used well.

A solution to both of these problems is tile-based texel storage. Instead of storing individual texels, small tiles of a fixed size are stored in each cache entry. Cache reads that are very close in texture space are likely to be on the same tile and thereby close in memory as well. Figure 4.4 visualizes the virtual texture tiles in an example scene. Bilinear filtering is efficiently possible within the tile by reading all four required texels from the same

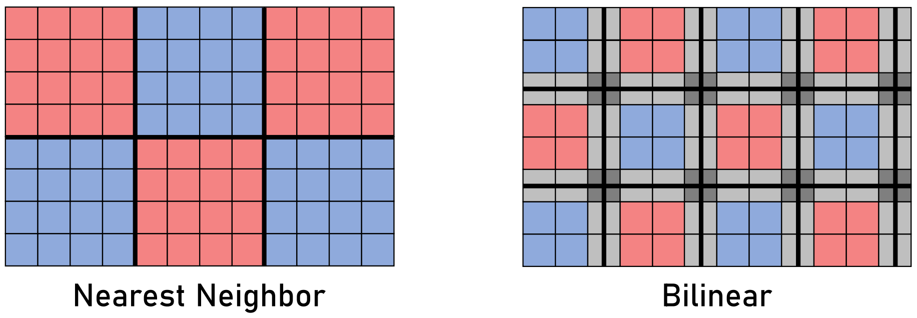


Figure 4.3: Organisation of texels in 4×4 tiles for nearest-neighbor filtering (left) and bilinear filtering (right). Tile borders are represented by thick black lines. To enable bilinear filtering without reading multiple tiles, tiles overlap by one texel. Each tile still stores 4×4 texels but only represents a 3×3 area in texture space. Boundary texels (gray) are stored by all adjacent tiles.

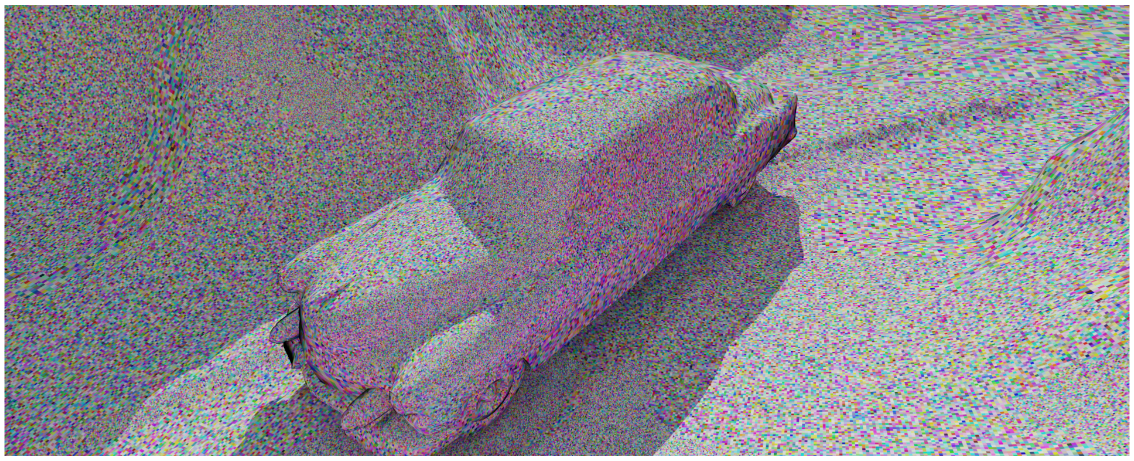


Figure 4.4: Visualisation of virtual texture tiles as randomly colored squares in one of the test scenes I used.

tile. If the requested point lies on the border of the tile, multiple tiles would need to be accessed. However, this can be avoided by making tiles overlap by one texel (figure 4.3). This increases the required memory for the cache and the number of texels that need to be computed but it ensures that the closest four texels to the requested point always lie on the same tile.

An important question for such a tiled material cache is how a new tile is calculated on a cache miss. The trivial solution would be to make the thread that requested the tile compute the entire tile by evaluating the material graph for each texel. This approach has two major problems. First, without coordination, it is likely that multiple threads in the same wave compute the same tile, which is redundant. Second, while the thread performs the very expensive operation of computing the tile, other threads in the wave are inactive. A better solution is to distribute the calculation of the tile among the threads of the wave. Using wave operations 2.5.2 for communication between lanes, I assign each texel of a tile to a different lane. The lanes then calculate their assigned texels in parallel. The calculation of a missing tile is well suited for this form of parallelization because the texels can be computed independently and the same operations must be performed for each texel. I explain the details of my algorithm for computing missing tiles along with other implementation details in section 5.4.

5. Implementation

In chapter 4, I explained the core ideas of my material graph memoization techniques at a relatively abstract level. In this chapter, I explain some important details of my implementation of the memoization techniques and of the renderer in which I tested them. I start with my material graph system and real-time path tracer. Then I first describe the non-tiled memoization variant and finally the additional challenges and design decisions of the tiled variant. Note that most of section 5.3 is also relevant for the tiled variant and not repeated in section 5.4. In that sense, the tiled variant should be understood as an extension or modification of the non-tiled one, rather than a completely separate technique.

5.1 Material Graph System

I decided to implement my own material graph backend for this thesis so that I have precise control over the generated code and more insight into the graphs. By "backend" I mean that my system translates a simple textual description of a material graph to an HLSL include file. I did not create my own graphical material graph editor but instead wrote an export script that translates Blender Shading Graphs into my own representation. In this section, I describe how I implemented my material graph system. While this is not the main focus of the thesis, I decided to include it as context for the memoization techniques presented later because integrating material graphs in a real-time path tracer is not a trivial task and there are different ways to do it.

The input to my system is a list of nodes. Each node description consists of an identifier for the node type (e.g. "Add" or "Normalize") and the inputs and outputs of the node as pairs of socket names and integer IDs. Sockets with matching IDs are connected in the graph. Notably, this graph representation contains no type annotations. Node inputs and outputs do have types but these are inferred based on the available node implementations and type conversions. In that way, it resembles graphical systems, which typically do not require the user to explicitly declare types either. For simplicity, I only allow a single parameterized BRDF per graph. BRDF parameters are treated as graph outputs. Listing 5.1 shows how a simple graph would be described in my input format. Figure 5.1 shows a visual representation of the same graph.

Mipmapped texture sampling requires derivatives of the texture coordinates. In path tracing, these can be provided via ray cones but the texture coordinates may be scaled or otherwise modified by the material graph. My material graph system therefore also generates

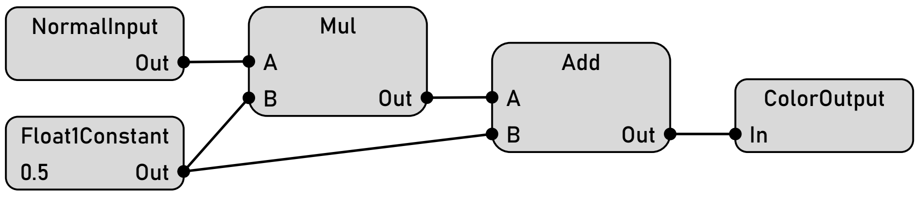


Figure 5.1: A very simple material graph that visualizes normal vectors by mapping them to colors. The graph is the same as in listing 5.1

Listing 5.1: A very simple material graph in my input format that visualizes normal vectors by mapping them to colors. The order of nodes is intentionally random here since ordering is handled by the code generator. The graph is the same as in figure 5.1.

```
Mul A<0 B<1 Out>2;
ColorOutput In<3;
Add A<1 B<2 Out>3;
Float1Constant 0.5 Out>1;
NormalInput Out>0;
```

code for computing derivatives where needed. Scalar and vector values with derivatives are represented by separate types. For example, a three-dimensional vector without derivatives is internally represented by the HLSL type float3 while a three-dimensional vector with derivatives is represented by the struct type float3_D, which contains the value itself along with two derivatives describing the ray cone footprint. I use the same type inference algorithm to infer base types as well as whether derivatives are needed. A limitation of my system for computing derivatives is that it cannot derive a value more than once. This is rarely a problem in practice but in theory, it can be necessary. For example, one node available in my system computes a normal vector from a position vector, which is useful for procedural displacement. Internally, it uses the derivatives of the position vector. Using this normal vector as texture coordinates for mipmapped texture sampling results in an error. The system cannot generate derivatives for the normal vector as that would require second derivatives of the position vector.

Node implementations are provided as function declarations in HLSL include files. I parse function signatures to build a list of available node implementations. Nodes can have multiple overloads. In particular, most nodes are available in a version that computes derivatives and one that does not. Node outputs are represented by output parameters to allow for more than one output socket per node. As an example, listing 5.2 shows the implementation of the cross-product node in my system.

The output of my system is an HLSL file that includes all node implementations used in the graph and declares a single function that evaluates the graph. This function consists of a list of variable declarations for the edges of the graph, followed by a list of function calls for the nodes of the graph. Each variable is used exactly once as output but may be used multiple times as input. In other words, the code is in static single assignment form (SSA). Since the output is in an imperative language, the nodes must be ordered in such a way that a variable is never used before it has been assigned. I do this with a depth-first search starting at the graph outputs.

Variables in HLSL are statically typed but the input graph does not contain type annotations. The system must therefore determine the type of each edge in the graph in a type inference phase. Type inference consists of selecting an overload for each node. The type

Listing 5.2: An example of a node implementation. The first overload does not compute derivatives, the second one does.

```
void Cross(in float3 A, in float3 B, out float3 Out) {
    Out = cross(A, B);
}

void Cross(in float3_D A, in float3_D B, out float3_D Out) {
    Out.v = cross(A.v, B.v);
    Out.x = cross(A.v, B.x) + cross(A.x, B.v);
    Out.y = cross(A.v, B.y) + cross(A.y, B.v);
}
```

Listing 5.3: The function generated for the example graph (see listing 5.1). *Convert* is a macro that invokes a type conversion function.

```
void EvaluateMaterial_Example() {
    float1 link0;
    float3 link1;
    float3 link2;
    float3 link3;

    Set(0.5, link0);
    NormalInput(link1);
    Mul(link1, Convert(float3, link0), link2);
    Add(Convert(float3, link0), link2, link3);
    ColorOutput(link3);
}
```

of an edge (and its variable) is then chosen to match the type of the corresponding output parameter. Overload selection happens in three phases. The first and second phase are a depth-first traversal of the graph. For each node, the available overloads are filtered based on the current list of overloads at adjacent nodes. The first phase starts at the sinks (i.e. graph outputs) while the second one starts at the sources (i.e. graph inputs and constants). The final phase performs an exhaustive search on the remaining combinations of overloads until it finds a solution for the graph. In principle, the final phase is sufficient but doing it on the entire search space would be too expensive.

5.2 Pathtracer

An important aspect of my thesis, particularly compared to existing work as mentioned in chapter 3 is the context of a real-time path tracer. I wrote a real-time path tracer for this thesis to serve as the framework for the material graph memoization system. This chapter provides insight into the design of this path tracer as the starting point for section 5.3 and 5.4.

I based the path tracer on DirectX ray tracing (DXR) (section 2.5.5). The GPU-side code is written in HLSL. More specifically, I implemented the path tracer in a single compute shader using ray queries for hardware-accelerated ray tracing. Contrary to the alternative approach of using the ray tracing pipeline, ray queries do not automatically branch into different hit shaders. Instead, such branching is implemented explicitly with conventional branching statements where necessary. This is important for my tile-based cache (section 5.4) because it gives me more control over which lanes in a wave are active, which I use to distribute the computation of new tiles over multiple lanes. The same could however also be achieved by using the ray tracing pipeline with a minimal hit shader that directly returns information about the intersection to the ray generation shader instead of performing material-specific code in the hit shader.

The path tracer dispatches one compute shader thread per pixel of the render target. Primary rays traced by a thread lie within the cone corresponding to the associated pixel. This is a form of stratified sampling that ensures that the same number of primary rays is traced for each pixel. It also simplifies writing the results to the render target as in my implementation, each thread only writes to this one pixel. Consequently, each pixel is only written by a single thread.

Each thread traces multiple paths for the same pixel. Each path consists of a fixed number of rays with each ray starting at the previous hit. Directions of secondary rays are determined with BRDF importance sampling. Additionally, I trace a shadow ray to a light source from each ray hit. Depending on the configuration, I either trace one primary ray per path or a single shared primary ray for all paths. The second option is an optimization based on the observation that the cone of a pixel is very narrow and not much variance is to be expected in the primary hits, except at edges, which can be handled well with various anti-aliasing techniques. In a production path tracer, one may even choose to generate primary hits via rasterization instead of by tracing individual primary rays. Which one of these approaches is chosen affects the performance of the material graph cache because primary hits are usually much more coherent than secondary rays. Primary hits of nearby pixels are likely to hit the same object and to be nearby in texture space as well. For secondary hits, this is generally not the case due to the random sampling. Tracing paths of a fixed length (as opposed to terminating paths with Russian roulette) makes the path tracer biased. However, it results in less divergent branching and still delivers good results for a real-time renderer.

For each ray hit, I first fetch information about the instance, the triangle and the vertices of the triangle from buffers. Next, I perform various material-independent computations, mainly related to ray cones (section 2.2). If material graph memoization is disabled, I then branch by material and evaluate the material graph. If material graph memoization is enabled, I perform the cache lookup (section 5.3, 5.4). On a cache miss, the material graph must still be evaluated but this happens as part of the cache lookup and is initiated by the memoization system. In a previous iteration of my implementation, the memoization system returned whether a cache hit or cache miss occurred and the path tracer evaluated the material graph on a cache miss and initiated a cache update. However, that organization does not work well for the tiled variant, which has to compute an entire tile on a cache miss. In the newer version, this logic is encapsulated in the memoization system and the rest of the path tracer never evaluates a material graph directly. Once the material graph outputs are known, I perform all BRDF-specific computations. This also requires branching but there are usually fewer types of BRDFs in a scene than there are materials. I sample one direction for the next ray of the path using BRDF importance sampling. Independently from that, I also sample one light source and trace a shadow ray to estimate direct lighting. For simplicity, I only support point and directional lights. Area lights can be emulated via emissive materials but I perform no importance sampling for them. Finally, I evaluate the BRDF for the two directions of incoming light, compute the radiance contribution of emission and direct light and continue with the next ray.

5.3 Material Graph Memoization

In section 4, I explained the core aspects of material graph memoization on a more abstract level. In this section, I explain my specific implementation of material graph memoization, which I evaluate in chapter 6. In particular, I explain the data structure of the cache, how

I compute cache keys and hashes, the eviction methods that I implemented, and how I handle concurrent cache access. This section focuses on the non-tiled variant while section 5.4 focuses on the additional design decisions and challenges of the tiled variant.

5.3.1 Data Structure

The cache is represented in memory by an HLSL `RWStructuredBuffer` of cache entries. Each cache entry stores a 32-bit hash of the key, a metadata struct defined by the eviction strategy and a cache value struct containing the outputs of the material graph. Because material graphs can have different outputs, the cache value struct is simply an array of scalar values, which may be interpreted differently for different graphs. The size of the array is fixed and must be set to the maximum number of fields required by any graph in the scene. Depending on the configuration, the material graph outputs are stored either as 32-bit or 16-bit floating point values, the latter using HLSL 16-bit float encoding and decoding intrinsics. 16-bit floats usually provide sufficient precision for material graph outputs such as colors or normal vectors. The non-tiled variant does not require any other global data structures than the cache buffer.

5.3.2 Key and Hashes

The first step of the cache lookup is to compute the key and its hashes. As explained in 4.2, I include texture coordinates, mipmap level and material ID in the key. The first step in computing the key is to select the mipmap level. My definition of a mipmap level in this context differs slightly from the usual definition in that I do not have a minimum or maximum level, nor does a mipmap level have a fixed size since each mipmap level of the virtual texture covers the entire 2D plane. Instead, mipmap levels should be thought of as levels of texel density in texture space. The texel density is the number of texels corresponding to one unit of length in texture space. I define the mipmap level m as the base-2 logarithm of the texel density d:

$$m = \log_2(d) \tag{5.1}$$

I compute the mipmap level from the partial derivatives of the texture coordinates as given by the ray cones (section 2.2):

$$\begin{aligned} l_x &= \left\| \frac{\partial t}{\partial x} \right\| && (5.2) \\ l_y &= \left\| \frac{\partial t}{\partial y} \right\| && (5.3) \\ l_{\min} &= \min\{l_x, l_y\} && (5.4) \\ m &= \operatorname{round}(\log_2(l_{\min}) + \text{bias}) && (5.5) \end{aligned}$$

- $t = (u, v)$ is the position of the ray hit in texture space,
- x and y are the directions of the ellipse axes of the ray cone footprint,
- round(x) rounds x to the nearest integer,
- bias is an optional bias applied to the mipmap level.

A noteworthy difference to, for example, OpenGL mipmap selection is that I use the shorter of the two derivatives instead of the longer one, which means that I prefer the higher texel density in anisotropic cases. This is because mipmapping usually mainly serves the purpose of preventing aliasing while in my case, it serves to limit the cache resolution to enable the reuse of texels. The material graph would typically already implement some form of anti-aliasing, e.g. by using mipmapping when reading from textures. Choosing the higher texel density prevents overblurring. Once the mipmap level is known, I convert the texture coordinates to texels and round to the nearest texel:

$$\hat{u} = \operatorname{round}(2^m u) \tag{5.6}$$

$$\hat{v} = \operatorname{round}(2^m v) \tag{5.7}$$

The next step is to compute two 32-bit hashes. The first hash is used to select the location in the hash table, the second one is stored in the cache entry and compared during a cache lookup to verify that the entry is the requested one. Computing a 32-bit hash from the more precise input values is a form of lossy data compression. It is therefore important to compute both hashes in different ways from the full key. It is not sufficient to compute one hash and then apply some modifications to it to get a second hash because then a collision in the first hash would always result in a collision in the second one as well, which renders the second hash useless. With that in mind, I compute the hashes and the hash table index as follows:

$$H_0 = \operatorname{hash}(m \oplus \operatorname{hash}(\hat{u} \oplus \operatorname{hash}(\hat{v} \oplus \mathrm{ID}_0))) \tag{5.8}$$

$$H_1 = \operatorname{hash}(m \oplus \operatorname{hash}(\hat{u} \oplus \operatorname{hash}(\hat{v} \oplus \mathrm{ID}_1))) \tag{5.9}$$

$$I = H_0 \mod (S - D) \tag{5.10}$$

- H_0 and H_1 are the hashes,
- hash(x) is a hash function for 32-bit integers, such as the PCG-hash [O'N14],
- $\oplus$ is the bitwise XOR operation,
- ID_0 and ID_1 are two different hashes of the material graph computed ahead of time,
- I is the index in the hash table where the linear probing starts,
- S is the size of the hash table,
- D is the maximum linear probing distance.

5.3.3 Linear Probing and Eviction

After a thread has determined the start index in the hash table, it performs a linear search over the following D entries to find one with a matching key. I implemented the linear search as a simple for loop that can be unrolled because the maximum linear probing distance is constant. Only the 32-bit hash of the key of each entry must be read in the loop. If no matching key is found, the thread evaluates the material graph and the eviction strategy chooses a location for the new cache entry in the same section of the hash table. This is trivial for the randomized eviction strategy. For the least-recently-updated and least-recently-used strategies, it requires another linear search to determine the entry where the time metadata field has the lowest value.

In both cases, cache entries may be overwritten by other threads during the linear search. I only protected the actual read and write operations against concurrent access (section 5.3.4) but not the linear search. This means that the index found by the linear search is not guaranteed to still have the correct key or to still be the optimal location to insert the new cache entry. I tolerate this and simply report a cache miss if the key has changed when reading a cache entry. I always insert a cache entry at the potentially incorrect location found by the linear search because choosing the ideal cache entry to evict is not necessary for the cache to work and deliver correct results. It is also less likely for two threads to access the same section of the hash table in the second linear search than in the first one because the second one is only performed on a cache miss.

The least-recently-updated and least-recently-used eviction strategies require a time metadata field. Determining the current time is not possible in standard HLSL but that is not necessary for eviction because only the order in which cache entries were read or written matters, not the time itself. I implemented two different methods of determining a pseudo-time value to be used for the time metadata field instead of a timestamp. The first one is to use a 32-bit CPU-side counter that is incremented between frames and passed to the GPU before rendering a frame. This method is very lightweight but not very precise since many cache entries are written in the same frame. Nonetheless, it is a good heuristic for an eviction strategy because camera and object motion are the main reasons to prefer evicting older entries and those take place between frame rendering. The second method is to use a GPU-side atomic counter that is incremented whenever a cache entry is written or read (the latter only if the least-recently-used strategy is used). This method gives the correct results but atomic operations on global memory are relatively expensive. Another problem is that a 32-bit counter may not be sufficient for counting all cache writes and is certainly not sufficient for counting all cache reads. For example, if eight cache reads per pixel are performed on a 1920×1080 pixel screen at 60 frames per second, a 32-bit counter would roll over approximately every four seconds:

$$\frac{2^{32}}{8 \cdot 1920 \cdot 1080 \cdot 60} \approx 4.3 \tag{5.11}$$

I solve this by simply using a 64-bit counter but that increases the size of the cache entries. Alternatively, one could reduce the precision by not storing all 64 bits in the cache entries. For example, using only bits 16 to 48 of a 64-bit counter would still be more precise than only storing a frame counter and result in a rollover interval of approximately 80 hours using the same parameters as above.

5.3.4 Concurrent Reading and Writing

The cache buffer is read and written concurrently by many different threads. This can lead to various problems. First, if two or more threads write the same cache entry simultaneously, the resulting entry may be an invalid combination of the correct entries, as shown in the following example sequence:

1. Thread_A writes Key_A,
2. Thread_B writes Key_B,
3. Thread_B writes Value_B,
4. Thread_A writes Value_A.

The resulting cache entry would consist of Key_B and Value_A. All threads reading the cache with Key_B afterward would use the wrong cached value (until the entry is overwritten again). Since the cache value cannot be written atomically, the result of concurrent writes could even be a new cache value that is a combination of the correct values.

To prevent inconsistent cache entries resulting from concurrent writes, I lock the cache entry before writing it. I do this by using `InterlockedExchange` to atomically swap the key in the cache with a special value `CACHE_LOCK`, which indicates that the cache entry is locked. If the value returned by `InterlockedExchange` is `CACHE_LOCK` as well, then the cache entry was already locked and the thread must abort writing. If it is not, the thread continues by writing first the cache value (and metadata if required by the eviction method) and then the cache key, which also unlocks the cache entry because the key is no longer `CACHE_LOCK`. I do not retry writing if the cache entry was locked. While this would be possible, it would mean that the cache entry that was just written is immediately evicted, and with the eviction methods that I implemented, there is no reason to prefer one cache entry over another that was written almost at the same time. Instead, I simply discard the cache entry. It is still used for the current frame by the thread that calculated it but it is not cached.

A slightly different problem may arise when one thread writes a cache entry and one or more other threads read it concurrently. A thread will never attempt reading an already locked cache entry because the key does not match. However, a thread can overwrite a cache entry after another thread has started but not completed reading it, as shown in the following example sequence:

1. The cache entry originally stores Key$_A$ and Value$_A$,
2. Thread$_A$ reads Key$_A$,
3. Thread$_B$ locks the entry,
4. Thread$_B$ writes Value$_B$,
5. Thread$_A$ reads Value$_B$,
6. Thread$_B$ writes Key$_B$ (which unlocks the entry).

In this case, Thread$_A$ would read and use the wrong value because the key matched. Note that the order of steps five and six does not matter, the problem would remain the same if they were swapped. This problem is less severe because it only has temporary consequences while the previous one resulted in a permanently incorrect cache entry (until overwritten). Nonetheless, it should be avoided.

In principle, this problem could be prevented in the same way as the previous one. Locking the cache entry not just for writing but also for reading would ensure that no thread overwrites a cache entry while another is reading it. However, cache entries are ideally much more often read than written and it is not a problem for multiple threads to read the same entry concurrently. Problems only arise when at least one thread writes the cache entry. Moreover, atomic operations on global memory are relatively expensive operations and it is an advantage to avoid them for the more common operation of reading a cache entry. The solution I settled on is to first read the key, then the value and finally the key again to test if it has changed since the read operation began. If the key no longer matches, a cache miss is reported. While this approach fixes the problem in the example case given above, there are in theory still situations where it fails as the following example sequence demonstrates (Thread$_A$ and Thread$_{A'}$ represent different threads with the same key):

1. The cache entry originally stores Key$_A$ and Value$_A$,
2. Thread$_A$ reads Key$_A$,
3. Thread$_B$ locks the entry,
4. Thread$_B$ writes Value$_B$,
5. Thread$_B$ writes Key$_B$ (which unlocks the entry),
6. Thread$_A$ reads Value$_B$,
7. Thread$_{A'}$ locks the entry,
8. Thread$_{A'}$ writes Value$_A$,
9. Thread$_{A'}$ writes Key$_A$ (which unlocks the entry),
10. Thread$_A$ reads Key$_A$ again.

To summarize, while Thread$_A$ is reading the cache entry, two full write operations are performed by other threads, the second of which reverts the key to its original value. Situations like this are unlikely but possible and result in Thread$_A$ reading the correct key before and after reading the wrong value. The root of the problem is that reading the same key twice is not a reliable indicator that it did not change between the two reads. I chose to ignore these situations in my implementation as they are unlikely enough to not have a noticeable effect on the final image. As mentioned before, the consequences are temporary, the cache is still consistent at the end of each frame. It is however possible to prevent these situations without locking the cache entry for each read at the cost of an additional integer field in each cache entry. In this alternative solution, a thread writing a cache entry would first lock the entry, then increment an update counter in the cache entry, then write the value and finally unlock the entry by writing the key. A thread reading a cache entry would first read the update counter, then the key, then the value and finally the update counter again. This would allow the thread to reliably detect changes to the cache entry because contrary to the key, the update counter can never be reverted to the previous value.

5.4 Tiled Material Graph Memoization

The tiled variant of material graph memoization is more complex to implement than the nontiled one. While some of the implementation details discussed in section 5.3 remain the same in the tiled variant, various additional problems must be solved and additional design decisions must be made. In this section, I explain the data structures I use for the tile-based cache, how I compute cache keys and hashes in the tiled variant and how I compute missing tiles efficiently by distributing the computation across multiple lanes in a wave.

5.4.1 Data Structures

Like the non-tiled version, the tiled cache also uses an `RWStructuredBuffer` as the hash table. This buffer always stores the keys as well as the metadata structs that may be required by the eviction strategy. Contrary to the non-tiled variant, each entry represents not just a single texel but a small (e.g. 4×4) tile of texels. I implemented and compared four different methods of storing the texels. One option is to include them in the main cache buffer, similar to the non-tiled variant. For example, with 4×4 tiles, each key and metadata struct would be followed by 16 texels, each of which contains all the outputs generated by the material graph. A disadvantage of this approach is that the fields read in the linear probing loop are spread out over a relatively large section of memory, which reduces hardware cache efficiency. The second option is therefore to store the texels in a separate buffer, which places them further from their keys and metadata but results in more tightly packed keys. The third option is to store the texels in an `RWTexture2D`. This has two advantages. First, hardware-accelerated bilinear filtering can be used. Second, the internal memory layout of a 2D texture is optimized for accesses that are close together in the 2D texture space. This is useful because bilinear sampling requires reading a 2×2 square of texels and because at least primary rays of nearby pixels are likely to hit nearby points in texture space. However, storing small tiles already has the same effect to a certain extent, even when stored line by line in a buffer. Furthermore, if the material graph produces more than four channels of outputs, as is usually the case, multiple tile-sized grids of texels must be allocated for each cache entry in the 2D texture. For example, eleven channels are necessary to store emission, base color, normal, roughness and metalness. This spreads out the values belonging to a single texel, which also negatively affects hardware cache efficiency as these values are always read and written together. This motivates the final method that I implemented, which is to store the tiles in a 3D texture. In this variant, I arrange the multiple tile-sized texel grids of each tile next to each other in the third dimension of the texture. 3D textures are optimized for accesses that are local in the 3D texture space, which means that both nearby texels and the other values belonging to the same texel should be nearby in memory. The downside is that 3D textures do not allow hardware-accelerated bilinear filtering but only trilinear filtering, which is more expensive but not required in this case.

5.4.2 Key and Hashes

The cache key in the tiled variant identifies only a tile, not a texel. It consists of the tile position in texture space, the mipmap level and the material ID. Additionally, the coordinates of the ray hit within the tile are required to sample the tile. Computing the mipmap level is the same as in the non-tiled variant. Computing the tile position is similar to computing the texel in the non-tiled variant except for the different grid resolution:

$$\hat{u} = \operatorname{floor}\left(\frac{2^m u}{s_u}\right) \tag{5.12}$$

$$\hat{v} = \operatorname{floor}\left(\frac{2^m v}{s_v}\right) \tag{5.13}$$

- $\hat{u}$ and $\hat{v}$ are the position of the tile in multiples of the tile size,
- $s = (s_u, s_v)$ is the tile size in texels.

If bilinear filtering is used, tiles overlap by one texel. In that case, s is the size of the area covered by the tile in texture space, i.e. one less than the number of texels stored in the tile. The coordinates within the tile are simply the fractional part of the previous formula:

$$\tilde{u} = \frac{2^m u}{s_u} - \hat{u} \tag{5.14}$$

$$\tilde{v} = \frac{2^m v}{s_v} - \hat{v} \tag{5.15}$$

- $\tilde{u}$ and $\tilde{v}$ are the coordinates within the tile in range $[0, 1]$.

I combine the tile position, mipmap level and material ID in the same way as in the non-tiled variant (equation 5.8 and 5.9).

5.4.3 Calculation of Missing Tiles

On a cache miss, the missing tile must be calculated and inserted into the cache. Instead of computing the tile entirely within the thread that had a cache miss, I distribute the calculation over the wave using wave operations (section 2.5.2). Each texel of a tile is calculated by a different lane. In my implementation, I do not allow more texels per tile than lanes per wave. I do however allow smaller tiles and compute multiple tiles in parallel within one wave if possible. For example, on GPUs that have 32 lanes per wave, two 4×4 tiles of the same material can be calculated simultaneously.

A sufficient number of lanes must not only exist but also be active within a wave when the calculation of missing tiles starts. Every branching instruction can cause lanes to become inactive. For example, one may want to stop tracing paths that hit the sky or ones that fail a Russian Roulette test. However, a naive implementation of this (e.g. by conditionally breaking out of a loop or returning from a function) would result in those lanes becoming inactive. Inactive lanes are not reachable with wave operations and do not execute any code. Inactive lanes are also a problem when the ray tracing pipeline is used instead of ray queries. The ray tracing pipeline allows assigning different hit shaders to different instances. A natural use case for this capability is to execute all material-specific code in the hit shader and perform no explicit branching by material. The problem with this approach is that when a hit shader is executed, only lanes that are executing the same hit shader are active. The ray tracing pipeline can still be used for the tiled variant if all objects use the same hit shader or if the calculation of missing tiles is performed in the ray generation shader.

Each lane queries the cache independently. Lanes that had a cache miss request the computation of their tile from the wave. Lanes that had a cache hit still participate in the computation of the missing tiles. Multiple lanes may request the same tile so the set of missing tiles in the wave may be smaller than the number of lanes that had a cache miss. I compute the missing tiles in several iterations. Each lane computes at most one texel per iteration but as mentioned before, the wave still computes multiple tiles in a single iteration if the tile size is small enough. Each iteration consists of the following steps:

1. The wave selects the lanes whose tiles will be computed in this iteration.
2. Each selected lane broadcasts the information necessary to compute the tile to the lanes responsible for computing the tile.
3. Each lane computes the texture coordinates where it must evaluate the material graph.
4. Each lane evaluates the material graph.
5. The lane that was assigned the top left texel performs the linear search to find a suitable index in the hash table and locks the entry.
6. Each lane writes its texel to the cache entry.
7. The lane that was assigned the top left texel overwrites the key.
8. The lanes that require one of the tiles computed in this iteration read the values they need from the corresponding lanes.

In the first step, lanes are selected one by one in ascending order of the lane index. Lanes that do not request a tile are skipped. A lane does not request a tile if it had a cache hit, if the tile it required has already been computed in a previous iteration, or if it has already been selected for this iteration. I use `WaveActiveBallot` to obtain a bit mask where each bit represents whether the corresponding lane requests a tile. The index of the first set bit is the index of the lane that is selected.

In the second step, the selected lanes broadcast the tile position, the mipmap level, the material ID and a set of flags that describe the inputs and outputs of the graph. I use `WaveReadLaneAt` for this. `WaveReadLaneAt` is called with a value and a lane index and returns the value passed to the function by the lane that has the given index.

In the third step, each lane first determines its assigned texel within the tile from its lane index. Then it computes the global texture coordinates using the tile position and mipmap level that it received in the previous step. To evaluate the material graph, the derivatives of the texture coordinates are needed as well to select mipmap levels when sampling textures in the graph. The only information I have about these derivatives when calculating a tile is the mipmap level m of the tile so I compute two fictional derivative vectors of equal length that would result in that mipmap level:

$$\frac{\partial t}{\partial x} = (2^{-m},\ 0) \tag{5.16}$$

$$\frac{\partial t}{\partial y} = (0,\ 2^{-m}) \tag{5.17}$$

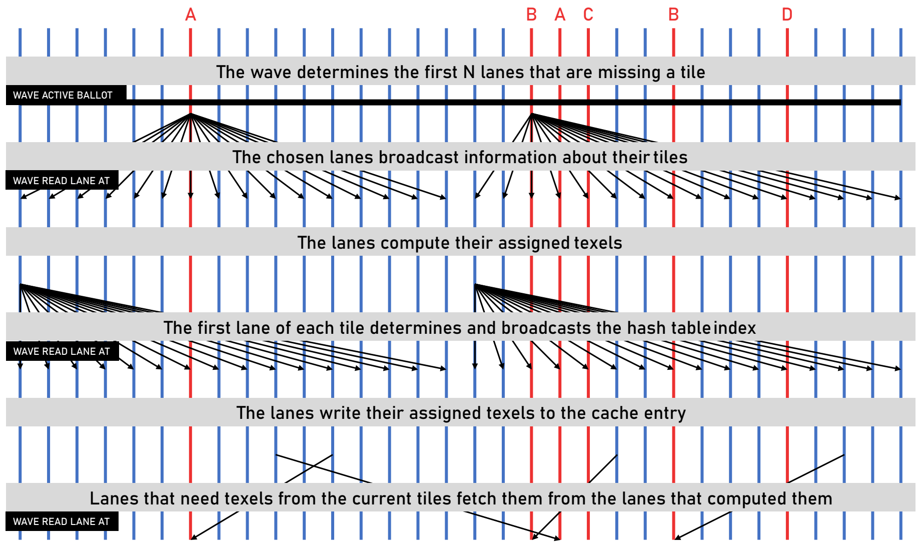

A
B
A
C
B
D
The wave determines the first N lanes that are missing a tile
WAVE ACTIVE BALLOT
The chosen lanes broadcast information about their tiles
WAVE READ LANE AT
The lanes compute their assigned texels
The first lane of each tile determines and broadcasts the hash table index
WAVE READ LANE AT
The lanes write their assigned texels to the cache entry
Lanes that need texels from the current tiles fetch them from the lanes that computed them
WAVE READ LANE AT

Figure 5.2: Communication between lanes in one iteration of the algorithm for computing missing tiles. Red lanes had a cache miss and request the computation of the tiles A, B, C, and D. Two 4×4 tiles (A and B) are computed in one iteration in this example.

Material graphs might also need other vertex attributes than just the texture coordinates but I do not support this in my implementation of the tiled variant. One possible solution would be to precompute textures containing the needed vertex attributes so that they can be sampled using the texture coordinates. Another one would be to extrapolate the vertex attributes from the ray hit of the lane that requested the tile. The lane that requested the tile knows the value and derivatives of the vertex attribute at its ray hit. From these, the value can be linearly extrapolated to other points on the tile. This results in the correct values if the tile lies entirely on one triangle. Otherwise, it is only an approximation but it may still work well in some cases.

Steps four, five, six and seven are very similar to the non-tiled variant. The lane that was assigned the top left texel of the tile performs the same steps as in the non-tiled variant, except that it also broadcasts the index of the cache entry and whether it obtained a lock on it to the other lanes.

The last step is necessary because it cannot be guaranteed that a tile is written to the cache. If no lock was obtained, the write operation is aborted. The lanes that requested the tile still need values but cannot read them from the cache. Aside from that problem, it is also faster to exchange the values with wave operations than via global memory. I exchange the values with `WaveReadLaneAt` as before. Nearest-neighbor sampling only requires reading from one lane while bilinear sampling requires reading from four lanes and interpolating the results.

5.4.4 Grouping Missing Tiles by Material

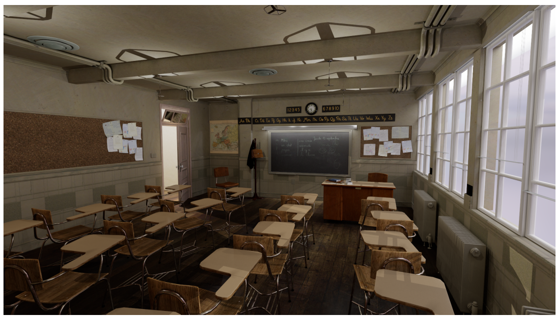
Even if multiple tiles are computed in one iteration, they are only computed in parallel if the material graph is the same. Otherwise, evaluating the material graph involves divergent branching and the tiles are effectively computed sequentially. To solve this problem, I also implemented an alternative, more complex version of the first step that attempts to sort tiles with the same material into the same iteration. The solution works only for the case where two tiles are computed in one iteration (e.g. two 4×4 tiles on a

GPU with 32 lanes per wave). Before I begin the iteration, I determine for each unique material in the wave whether the number of lanes requesting a tile of that material is even or odd using `WaveActiveCountBits`. I then add a high offset to the IDs of materials where the number is odd. This has the effect that sorting the tiles by material ID results in an order where tiles of the same material are optimally paired up for being computed in parallel. Tiles of the same material always appear next to each other in that order because they have the same material ID. The offset ensures that all materials with an even number of requested tiles appear first. Without the offset, a material with an odd number of requested tiles could break up possible pairings later in the list. For example, if one tile of material 1, two tiles of material 2 and two tiles of material 3 were requested, the resulting order and pairings would be $(1, 2), (2, 3), (3, -)$. With an offset of 1000, it would be $(2, 2), (3, 3), (1001, -)$, achieving two pairs that can be parallelized instead of none. I ensure that the tiles are processed in this order by implementing selection sort with wave operations. When selecting a lane, I first select the material ID for the next tile with `WaveActiveMin` and then select the first lane requesting a tile of that material with `WaveActiveBallot` as before.

6. Results

I evaluated the effectiveness of my techniques with regard to image quality and performance by testing them in an example scene with recorded camera motion. I also compared the different variants I described as well as many different configurations of their parameters. I rendered at 1920×1080 on an NVIDIA RTX 4070 Ti. I used two scenes for testing. The first one is the Blender Classroom example scene by Christophe Seux with slight modifications to make it work in my relatively basic path tracer. The second one is a landscape with a car model by user ROH3D on sketchfab.com. The two scenes are very different. The *Landscape* scene only uses four different materials, three of them on the car and a fourth, relatively complex one for the terrain. The *Classroom* scene in the version that I used has 69 different materials. Another important difference is that the *Classroom* scene is a mostly closed room while the *Landscape* scene is an open exterior scene. I recorded first-person-style camera motion for both scenes and played it back frame by frame in each test run to get comparable timings.

For the performance comparisons in the *Classroom* scene, I traced six paths of four rays per pixel with an additional shadow ray to a direct light source at each of the four ray hits. In the *Landscape* scene, I traced 18 such paths per pixel because the scene is much cheaper to render. For the quality comparisons, I traced significantly more rays because my goal was to quantify only the bias added by memoization.


(a) Classroom

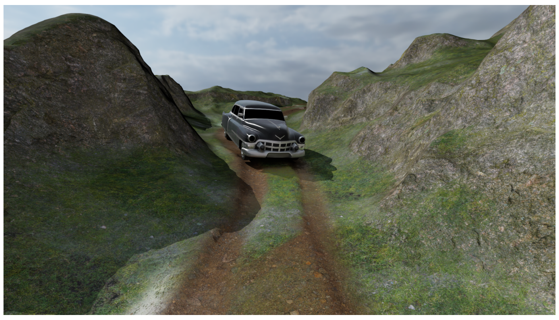

(b) Landscape

Figure 6.1: The two test scenes I used.

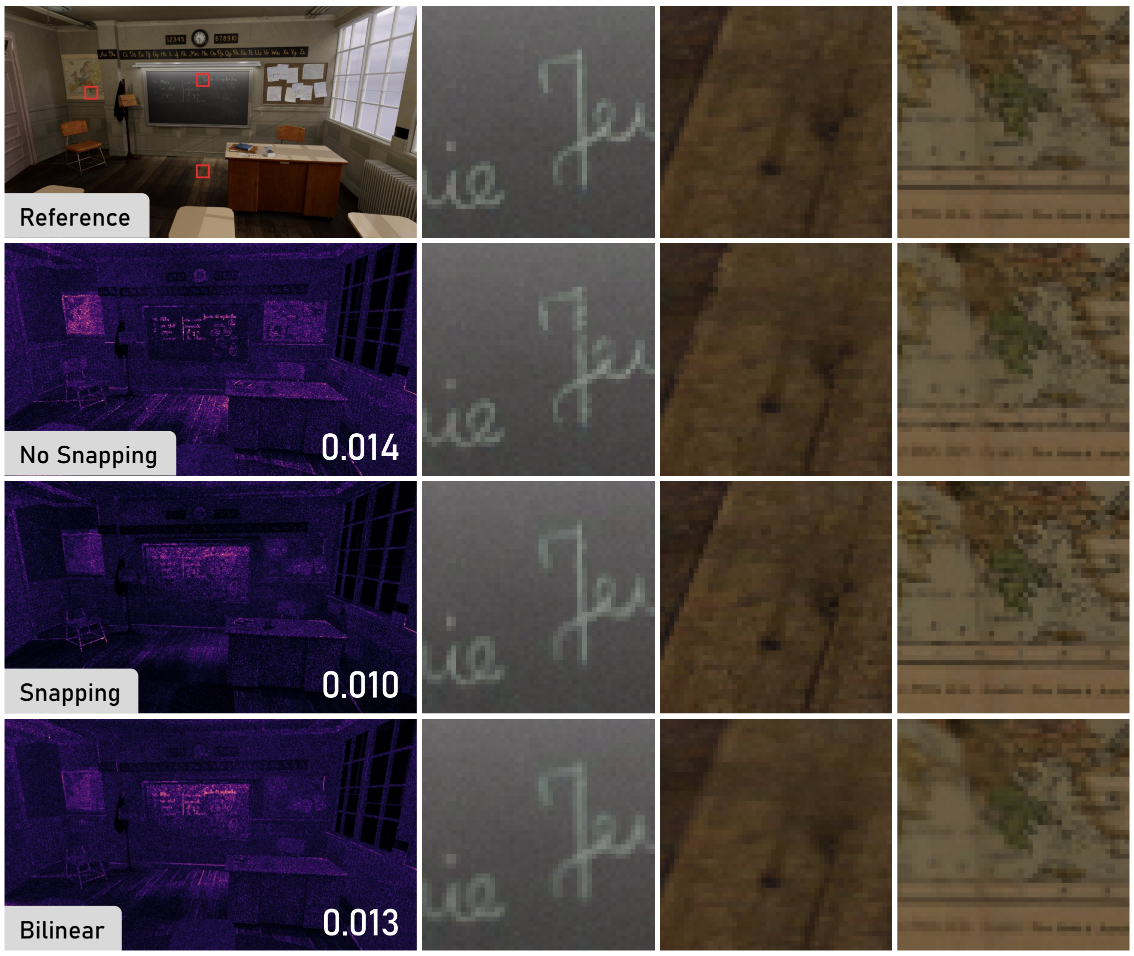


Figure 6.2: Quality comparison of converged (10, 000 paths per pixel) images. The left column is a ꟻLIP error map amplified by a factor of eight. *Snapping* refers to rounding the texture coordinates to the nearest cache texel before evaluating the material graph. This is implicitly done in the tiled variant but optional in the non-tiled variant. Note that *Reference* refers to an image rendered with the same path tracer without memoization, not with an unbiased reference path tracer.

6.1 Impact on Image Quality

Material graph outputs are not only reused when the inputs are identical but also when they are just similar enough. This is necessary for the cache to be effective but results in a slight loss of image quality when memoization is used. In the non-tiled variant, texture coordinates can either be passed to the material graph unchanged or snapped to the nearest cache texel. These two options result in different errors. In the tiled variant, the texture coordinates are always snapped to cache texels since all texels of a tile must be evaluated instead of just the one at the ray hit. The image quality is also affected by the interpolation method. The tiled variant supports nearest-neighbor and bilinear filtering while the non-tiled variant only supports nearest-neighbor filtering. This means that there are three different variants to compare regarding image quality: nearest-neighbor interpolation without snapping to the nearest texel, nearest-neighbor interpolation with snapping and bilinear interpolation.

I compared images rendered with $10,000$ paths per pixel to eliminate most of the noise from the Monte Carlo integration and focus on the additional bias caused by the memoization. The images were captured after 500 frames of camera motion. I quantified the error using the ꟻLIP metric [ANA$^+$20]. Figure 6.2 shows the example images that I used as well as the corresponding ꟻLIP error maps (amplified by a factor of eight).

While the images of all three variants look similar to the reference image, they are visibly different when zoomed in on specific details. As expected, the highest errors occur at high-frequent features in textures. Since the errors are the result of resampling and slightly misplaced texture information, they are not noticeable in low-frequency areas. Furthermore, since the geometry is not affected by the memoization, high-frequent geometric features do not result in significant errors either.

Both nearest-neighbor variants exhibit slight aliasing artifacts but they are more visible in the variant without snapping. The tiled variant with bilinear interpolation has no visible aliasing artifacts but results in slightly more blurry textures than the other two variants. I do not interpolate between mipmap levels in any variant. As a result, the border between mipmap levels is slightly visible in some areas, particularly in the variant with bilinear filtering due to the different blurriness. This is particularly visible in the rightmost column of figure 6.2.

The variant with nearest-neighbor interpolation and snapping has the best ꟻLIP score. The variant with bilinear interpolation scores slightly better than the variant without snapping. The blur caused by bilinear interpolation can be reduced by applying a bias to the mipmap level of the virtual textures of the cache (figure 6.3). However, this also results in worse performance as more tiles need to be evaluated.

By sampling random primary ray directions, the path tracer implicitly applies some anti-aliasing when many paths are accumulated. Turning this off by always tracing primary rays through pixel centers makes the differences in image quality much more visible. Figure 6.4 compares example images rendered without implicit anti-aliasing.

Aside from being different from the reference image, aliasing artifacts also result in a less temporally stable image. As the camera or objects move, high-frequent details in textures appear to change slightly instead of just moving across the screen. This is also much more visible when the implicit anti-aliasing of the path tracer is disabled. Figure 6.5 compares the temporal stability of bilinear interpolation and nearest-neighbor interpolation with snapping under sub-pixel camera motion.

It is worth noting that the errors are almost exclusively caused by sampling the cache at primary hits. A simple solution if better image quality is desired is to not cache primary

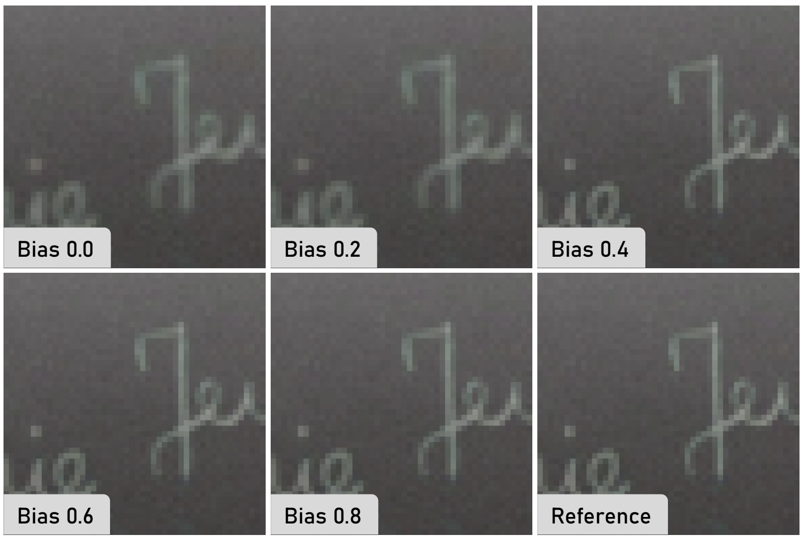


Figure 6.3: Applying a bias to the mipmap level of the virtual textures of the cache reduces the blur caused by bilinear interpolation.

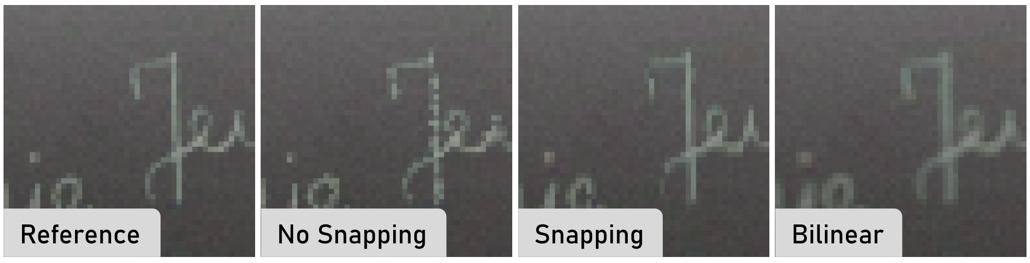


Figure 6.4: Usually the path tracer implicitly applies some anti-aliasing by sampling random directions for primary rays. Disabling this makes the differences more visible.

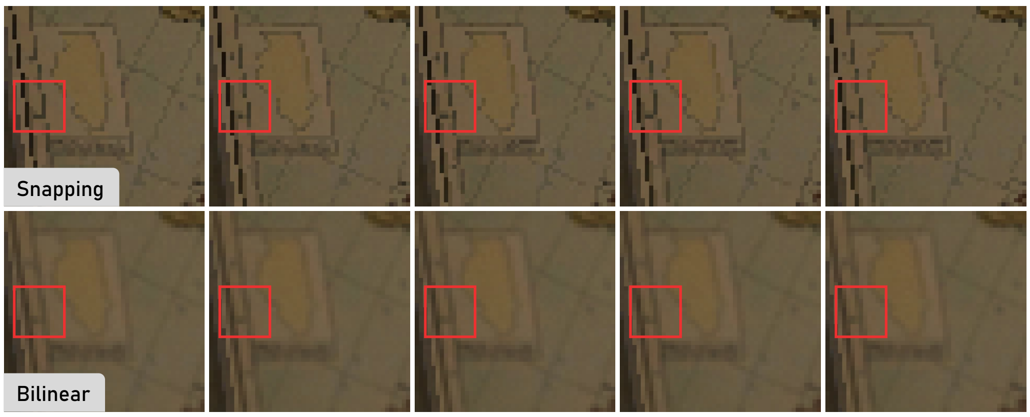


Figure 6.5: Comparison of the temporal stability of bilinear interpolation and nearest-neighbor interpolation with snapping. From left to right, the camera rotates one pixel to the right over five frames. Nearest neighbor interpolation causes visible changes in some texture details from one frame to another. Like in figure 6.4, the implicit anti-aliasing of the path tracer has been disabled here.

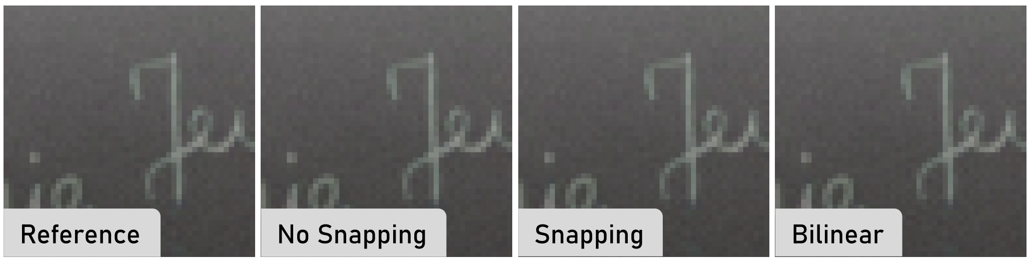


Figure 6.6: Quality comparison with memoization disabled for primary hits. Using the cache for secondary hits only results in near perfect images while still improving performance compared to using no cache at all.

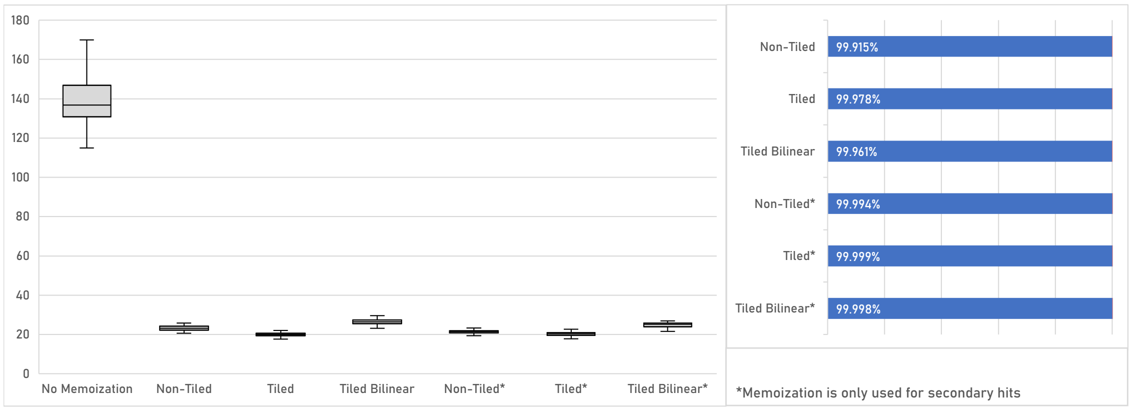


Figure 6.7: Frame times and hit rates in the *Classroom* scene. Times are in milliseconds per frame.

hits. Evaluating material graphs at primary hits is relatively fast compared to secondary hits. Slight aliasing or blur in secondary hits are usually not noticeable because the BRDF acts as a low-pass filter. Figure 6.6 compares the image quality with memoization disabled for primary hits.

6.2 Overall Performance

I tested the performance of the non-tiled and tiled memoization in the *Classroom* and *Landscape* scene. For the tiled variant, I compared nearest-neighbor and bilinear interpolation. I also tested how disabling memoization for primary hits affects performance. I used the least-recently-used eviction strategy with atomic counter. I compare the performance of different eviction strategies in section 6.5. I used a very large cache of 2^{24} cache entries in the non-tiled variant and 2^{20} 4×4 tiles in the tiled variant. I compare the performance of different cache sizes in section 6.6.1 and the performance of different tile sizes in section 6.6.3. The results are shown in figures 6.7 and 6.8 and listed in table 6.1.

Using any of the memoization variants improved the performance in both scenes. The difference between using memoization and not using it was much more significant in the classroom scene. One reason for this is that the classroom scene has more different materials. Another is that secondary rays often hit the skybox in the landscape scene and therefore do not need to evaluate a material graph. Caching only secondary hits still resulted in a noticeable performance improvement. The tiled variant with nearest-neighbor interpolation performed slightly better than the non-tiled variant. The tiled variant with bilinear interpolation performed worse than the non-tiled variant. Bilinear interpolation

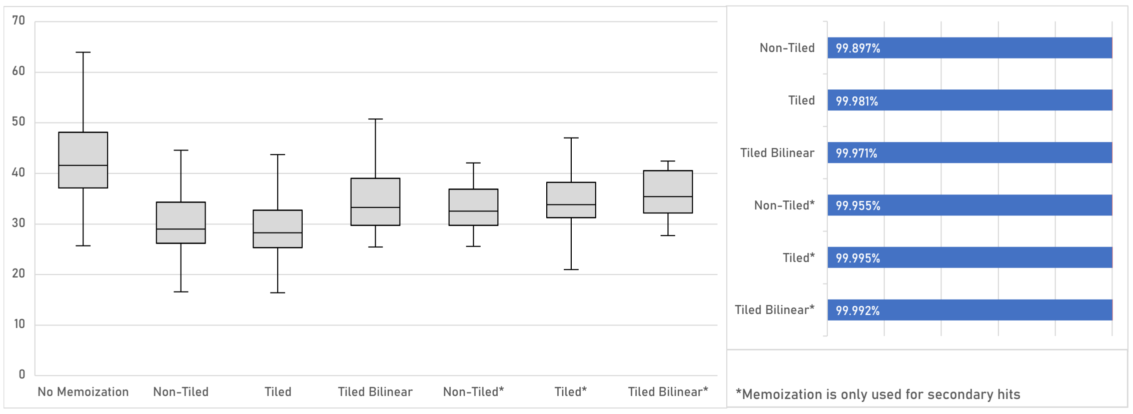


Figure 6.8: Frame times and hit rates in the *Landscape* scene. Times are in milliseconds per frame.

Table 6.1: Hit rates and median frame times in the tiled and non-tiled variant.

	Classroom		**Landscape**	
No memoization	136.9 ms	-	41.6 ms	-
Non-tiled	23.1 ms	99.915%	29.0 ms	99.897%
Tiled, nearest neighbor	**20.0 ms**	99.978%	**28.3 ms**	99.981%
Tiled, bilinear	26.5 ms	99.961%	33.3 ms	99.971%
Non-tiled*	21.4 ms	99.994%	32.5 ms	99.955%
Tiled, nearest neighbor*	20.3 ms	99.999%	33.8 ms	99.995%
Tiled, bilinear*	25.1 ms	99.998%	35.4 ms	99.992%

*Memoization is only used for secondary hits

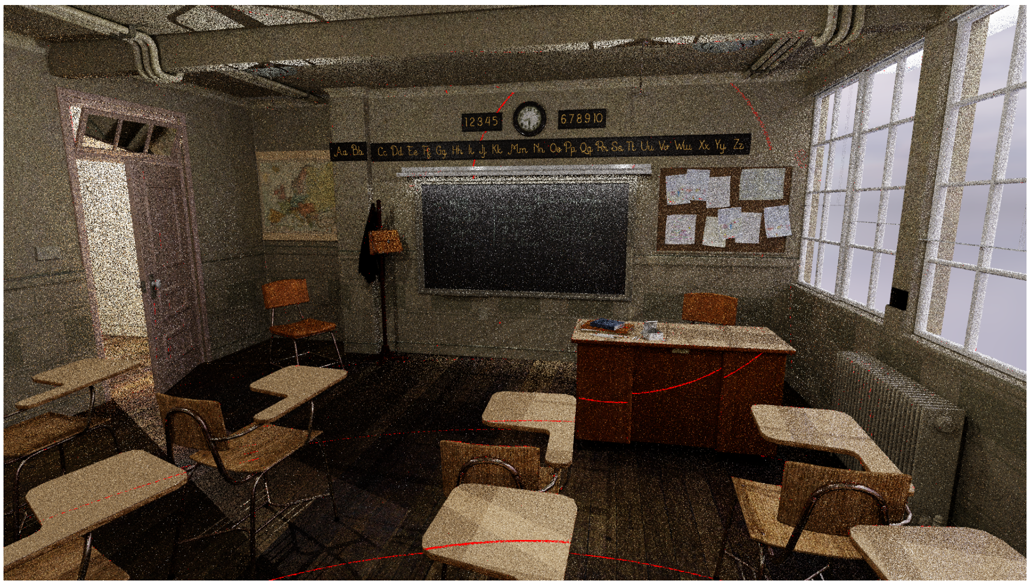


Figure 6.9: Cache misses (red) in the *Classroom* scene. The camera is moving forward, which results in concentric circles of cache misses where a new mipmap level becomes visible. Aside from this effect, very few cache lookups miss.

requires that tiles overlap by one texel, which means that more texels must be computed in total, resulting in worse performance. The hit rate was consistently very high in this comparison. In all but one run it exceeded 99.9%. Figure 6.9 shows a screenshot of the *Classroom* scene with cache misses visualized as red pixels. Cache misses mostly occur when new parts of the scene or new mipmap levels become visible.

6.3 Comparison of Memory- and Compute-Heavy Graphs

Compute operations and memory accesses such as sampling a texture are processed differently by GPUs. Which one of these a material graph uses more may change the effectiveness of the material graph memoization. To test this, I compared two different versions of the *Classroom* scene. The material graphs in the original scene mainly sample textures and combine or modify the results with relatively cheap operations. As a test case that uses exclusively compute operations, I replaced all material graphs in the scene with graphs that evaluate multiple octaves of simplex noise. By adjusting the number of octaves, I made the two versions of the scene perform similarly without memoization. I then tested the effect of memoization in the non-tiled and tiled variants on the performance. In the non-tiled variant, I used a cache size of $2^{24} = 16777216$ entries. In the tiled variant, I used a cache size of $2^{2}0 = 1048576$ 4×4 tiles (i.e. the same number of texels as in the non-tiled variant). The results are shown in figure 6.10.

In both versions of the scene, using memoization resulted in a significant speedup and tiled memoization was faster than non-tiled one. Interestingly, non-tiled memoization was slightly less effective in the compute scene than in the original one while tiled memoization was slightly more effective. This is likely because compute operations are more sensitive to divergent branching. When a lane evaluates a material graph, all other lanes in the wave that do not evaluate the same material graph are inactive and cannot perform any compute operations. In the non-tiled variant, a lane had on average 12.4 peers executing the same material graph (out of 32 lanes per wave), compared to 26.0 in the tiled variant.

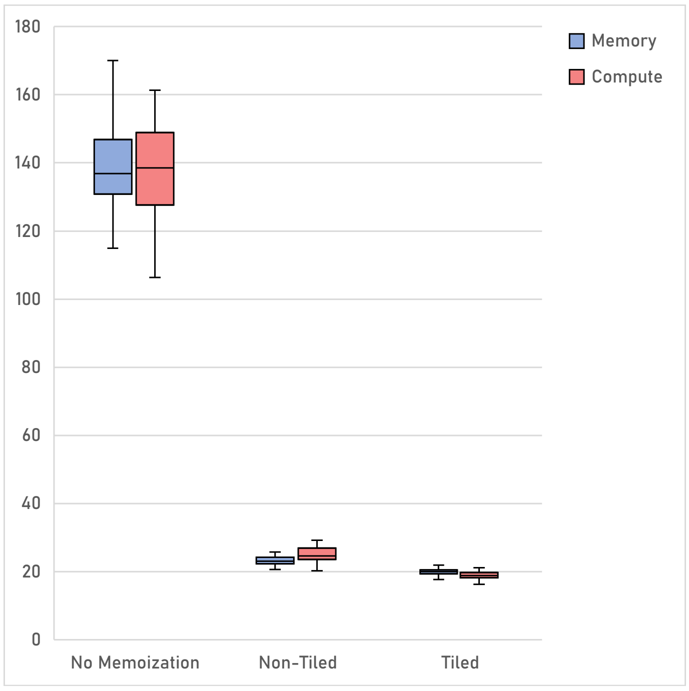


Figure 6.10: Performance comparison of memory-heavy and compute-heavy graphs. The chart shows milliseconds per frame.

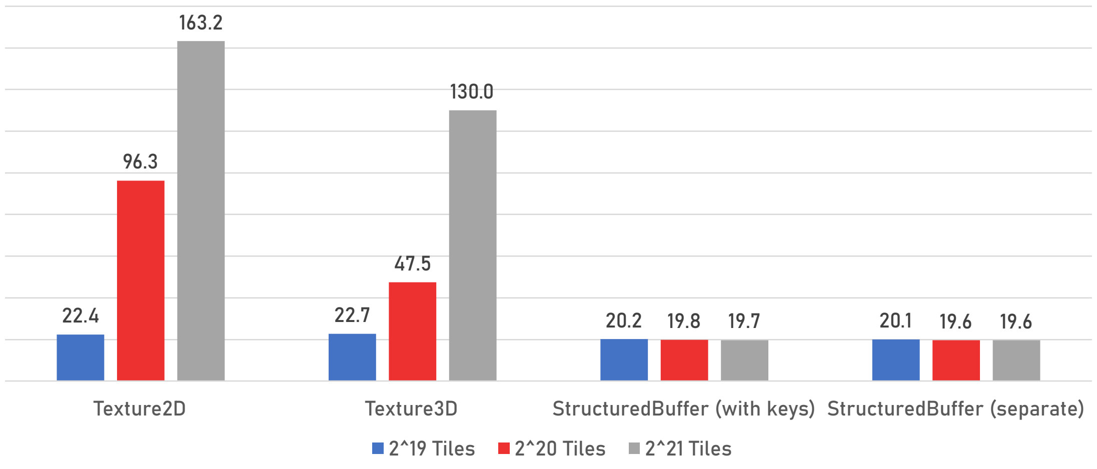


Figure 6.11: Performance comparison of different tile storage methods for different cache sizes. The chart shows milliseconds per frame. Nearest-neighbor interpolation was used in all runs. Notably, the performance is significantly worse for very large caches in the texture-based variants.

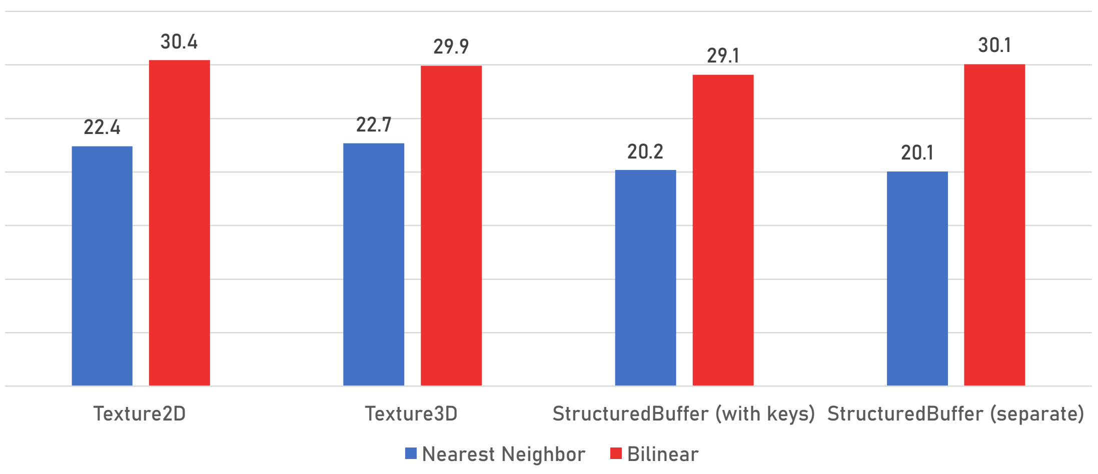


Figure 6.12: Performance comparison of different tile storage methods for nearest-neighbor and bilinear interpolation. The chart shows milliseconds per frame. The cache size was 2^{19} 4×4 tiles in all runs.

6.4 Performance of Different Tile Storage Methods

As explained in section 5.4.1, I implemented four different methods of storing tiles in the tiled memoization variant. The first is to store the tiles in a 2D texture. If more than four channels per tile are used, multiple tile-sized areas next to each other in X-direction are used to store a single tile. The second is to use a 3D texture so that the multiple tile-sized areas can be stored more compactly by stacking them in Z-direction. The third is to store the tiles in the same buffer as the keys and the fourth is to store them in a separate buffer. Storing tiles in a buffer instead of a texture has the advantage that all channels of a texel are stored compactly in memory. In theory, using a texture has the advantages that hardware bilinear interpolation can be used and that it enables the GPU to use a memory layout and caching strategy optimized for memory accesses that are local in 2D or 3D space. However, my results show that despite these advantages, the texture-based variants do not perform better than the buffer-based ones.

I compared the performance of the four methods with nearest-neighbor interpolation for three different cache sizes because the cache size significantly affects performance in the texture-based variants. I also compared their performance when using nearest-neighbor and bilinear interpolation for a cache size of 2^{19} 4×4 tiles. The results are shown in figures 6.11 and 6.12.

When nearest-neighbor interpolation is used, storing tiles in a buffer performs better than using textures. The performance of using a single buffer is nearly identical to the performance of separate buffers for keys and tiles. The performance of the texture-based methods is significantly worse for very large caches (2^{20} or 2^{21} 4×4 tiles). This effect is stronger for 2D textures. When bilinear interpolation is used at a cache size of 2^{19} tiles, the performance of all four methods is similar.

The results indicate that the memory layout of 2D and 3D textures provides little or no advantage in this use case, presumably because even the buffer-based variants preserve locality well due to the tile-based layout and locality beyond tile borders is irrelevant due to the hash function. Storing all channels of a single texel compactly appears to be more beneficial. The bad performance of large textures is surprising. To investigate this issue, I compared the variant using a 2D texture and the variant using a separate buffer with cache sizes of 2^{19} and 2^{21} tiles in the NVIDIA Nsight Graphics GPU Trace Profiler. In the texture-based variant, the profiler reports a much lower L2 hit rate for the larger cache (75% instead of 99.7%). In the buffer-based variant, the L2 hit rate is 99.4% for both cache sizes. The L1 hit rates are similar for all four runs (around 64%). The lower L2 hit rate is most likely the reason for the worse performance. It seems that the texture caching strategy used by the GPU is ineffective for this use case.

I recommend using either of the buffer-based variants. Even when bilinear filtering is used, the texture-based caches do not provide an advantage and they perform significantly worse for large caches.

6.5 Performance of Different Eviction Strategies

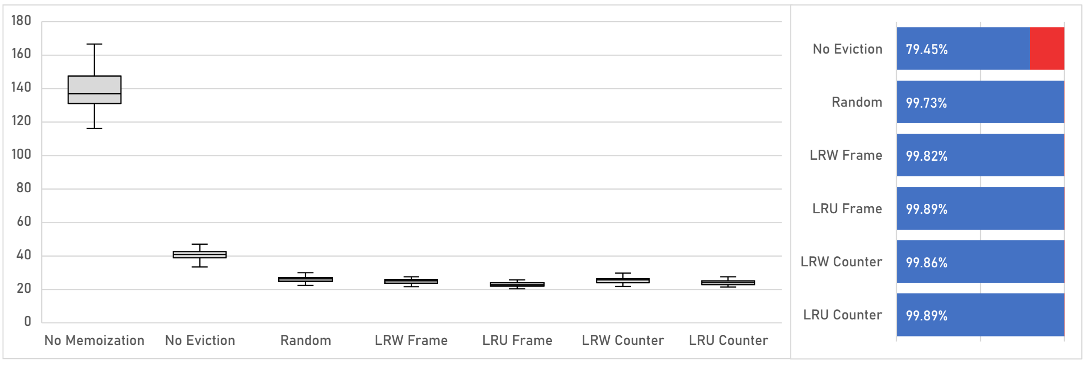
As explained in section 4.5, I implemented three different eviction strategies. The first is to evict a random entry in the linear probing range, the second one to evict the least recently written entry and the third one to evict the least recently used entry. The latter two strategies require comparing the age of cache entries, which I implemented in two different ways. The first is an approximation that uses the frame when a cache entry was written or read while the second one is exact by using an atomic counter that is incremented for each cache write or read. These eviction strategies differ in how well they predict the reuse of cache entries and in their computational overhead. I tested the eviction strategies by measuring frame times and hit rates in the *Classroom* scene with recorded camera motion. The hit rate indicates how well a strategy predicts reuse. The frame times indicate whether a higher hit rate justifies the higher overhead of a more expensive strategy.

The eviction strategies behave differently depending on the cache size. In the non-tiled variant, I compared them with a cache size of $2^{21} = 2097152$ and $2^{23} = 8388608$ entries. In the tiled variant, I used a tile size of 4×4 and the same cache size by the number of texels, which results in $2^{17} = 131072$ and $2^{19} = 524288$ cache entries respectively. The results are shown in figure 6.13 and listed in table 6.2.

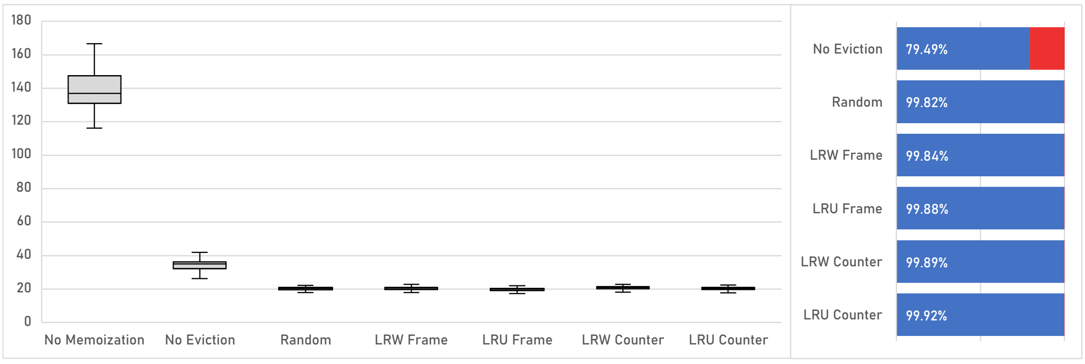
The results show that for a sufficiently large cache, using any of the three eviction strategies is a major improvement over not evicting cache entries but the differences between the strategies are relatively small. The importance of eviction also becomes evident when the cache hit rate is observed over time (figure 6.14). Without eviction, it declines quickly as the cache is full of cache entries that are no longer relevant.

All three strategies achieve hit rates above 99%. Generally speaking, the more expensive strategies did achieve higher hit rates although random eviction also performs surprisingly well. Interestingly, using the exact timing method based on an atomic counter instead of the frame counter approximation resulted in a slightly higher hit rate. This shows that not only motion in the scene is relevant for eviction but also the order in which pixels of the same frame are processed. However, the higher overhead of the atomic counter resulted in slightly worse performance despite the higher hit rate.

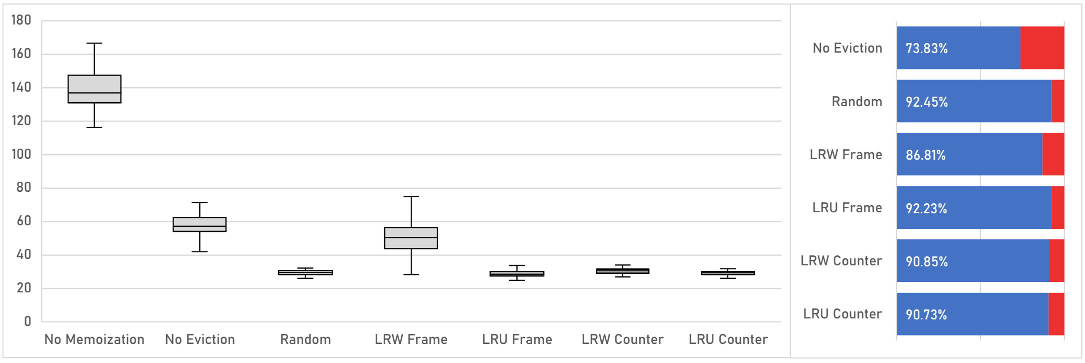
Reducing the cache size changes the results significantly. Especially in the tiled variant, using the frame counter approximation performs significantly worse than using an atomic counter. The key difference to the runs with the larger cache is that the cache is too small to contain many frames worth of cache entries. Counting the number of distinct keys used to query the cache in one frame reveals that a typical frame in the *Classroom* scene accesses around $1,300,000$ cache entries in the non-tiled variant and around $110,000$ tiles ($1,760,000$ texels) in the tiled variant. This is similar to the smaller cache size of $2^{21} = 2097152$ texels. Scene motion is therefore less relevant. Evicting older entries first can be counterproductive because the oldest entries are typically entries from the previous frame, which are particularly valuable for temporal reuse. The assumption that old entries



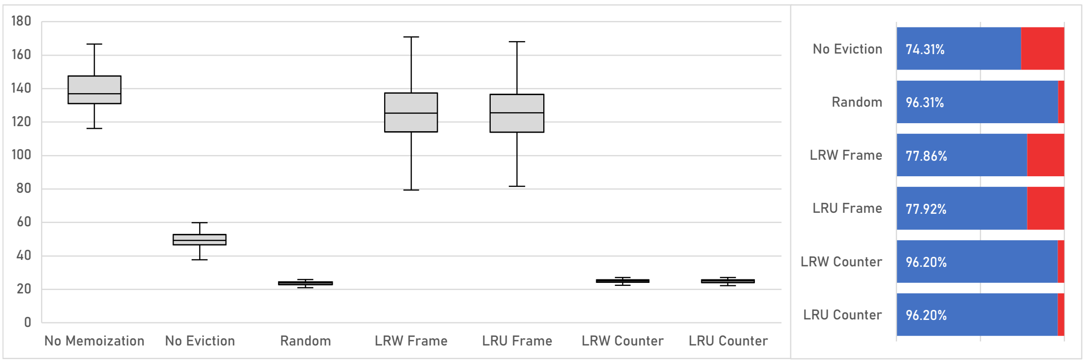
(a) Non-tiled, large cache ($2^{23} = 8388608$ entries)



(b) Tiled, large cache ($2^{23} = 8388608$ texels, $2^{19} = 524288$ tiles)



(c) Non-tiled, small cache ($2^{21} = 2097152$ entries)



(d) Tiled, small cache ($2^{21} = 2097152$ texels, $2^{17} = 131072$ tiles)

Figure 6.13: Frame times (left) and hit rate (right) of different eviction strategies for different cache sizes. Times are in milliseconds per frame.

Table 6.2: Median frame times and hit rates of different eviction strategies for different cache sizes. Times are in milliseconds per frame. The larger cache size is $2^{23} = 8388608$ texels or $2^{19} = 524288$ tiles, the smaller one is $2^{21} = 2097152$ texels or $2^{17} = 131072$ tiles.

Large Cache				
	Non-tiled		**Tiled**	
No memoization	136.9 ms	0.000%	136.9 ms	0.000%
No eviction	41.0 ms	79.449%	35.0 ms	79.486%
Random	26.3 ms	99.733%	20.5 ms	99.824%
LRW frame	25.0 ms	99.820%	20.6 ms	99.841%
LRU frame	22.9 ms	99.892%	19.9 ms	99.877%
LRW counter	25.7 ms	99.861%	21.0 ms	99.894%
LRU counter	24.0 ms	99.894%	20.4 ms	99.918%
Small Cache				
	Non-tiled		**Tiled**	
No memoization	136.9 ms	0.000%	136.9 ms	0.000%
No eviction	57.3 ms	73.834%	49.2 ms	74.311%
Random	29.6 ms	92.447%	23.8 ms	96.313%
LRW frame	50.5 ms	86.808%	125.4 ms	77.864%
LRU frame	28.5 ms	92.229%	125.6 ms	77.922%
LRW counter	30.6 ms	90.851%	25.2 ms	96.205%
LRU counter	29.3 ms	90.729%	25.1 ms	96.205%

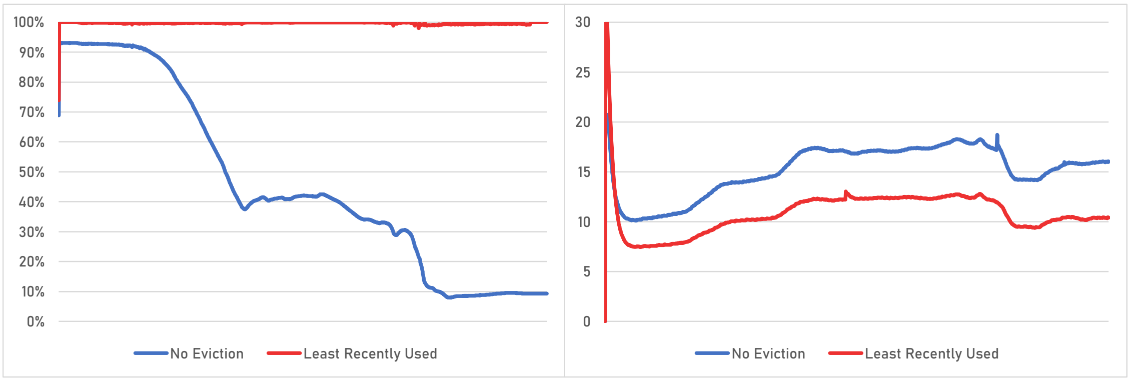


Figure 6.14: Hit rate (left) and frame times (right) over time in the *Landscape* scene with and without eviction. Times are in milliseconds per frame. Without eviction, the hit rate declines quickly as the cache is filled with entries that are no longer relevant.
Note: The frame times were smoothed slightly to reduce random fluctuation and show the overall trend better.

Table 6.3: Median frame times and hit rates for different cache sizes in two different scenes and in the tiled and non-tiled variant. Times are in milliseconds per frame. The cache size is the number of cache entries. In the tiled variant, each cache entry is a 4×4 tile.

	Non-Tiled				**Tiled**			
Size	**Classroom**		**Landscape**		**Classroom**		**Landscape**	
None	136.9 ms	0.00%	42.0 ms	0.00%	136.9 ms	0.000%	42.0 ms	0.00%
2^{10}	212.1 ms	48.70%	48.1 ms	12.53%	286.1 ms	57.28%	69.3 ms	30.61%
2^{11}	159.0 ms	58.80%	44.7 ms	15.56%	184.4 ms	66.76%	59.0 ms	48.17%
2^{12}	96.1 ms	66.66%	44.1 ms	17.12%	108.6 ms	76.76%	42.8 ms	80.47%
2^{13}	60.7 ms	73.03%	44.5 ms	20.57%	51.0 ms	91.39%	31.4 ms	95.39%
2^{14}	42.4 ms	81.68%	43.9 ms	36.28%	27.2 ms	95.78%	30.7 ms	96.26%
2^{15}	32.7 ms	88.10%	41.3 ms	62.91%	25.1 ms	96.20%	30.5 ms	96.31%
2^{16}	29.6 ms	89.88%	37.4 ms	84.25%	24.4 ms	96.30%	30.7 ms	96.32%
2^{17}	28.4 ms	90.21%	34.7 ms	89.52%	24.1 ms	96.34%	30.6 ms	96.38%
2^{18}	27.8 ms	90.36%	34.4 ms	89.90%	23.4 ms	96.98%	28.8 ms	98.41%
2^{19}	28.3 ms	90.44%	34.9 ms	89.94%	20.3 ms	99.92%	28.2 ms	99.97%
2^{20}	28.5 ms	90.49%	35.4 ms	89.96%	20.0 ms	99.98%	28.2 ms	99.98%
2^{21}	28.8 ms	90.73%	35.5 ms	90.71%	-	-	-	-
2^{22}	26.9 ms	97.50%	33.4 ms	97.58%	-	-	-	-
2^{23}	23.5 ms	99.89%	31.3 ms	99.88%	-	-	-	-
2^{24}	23.5 ms	99.91%	31.0 ms	99.90%	-	-	-	-

are most likely to have become irrelevant does not hold if only such a small timespan is represented in the cache. If all entries in the linear probing range were written in the same frame, the eviction strategy will always overwrite the first one.

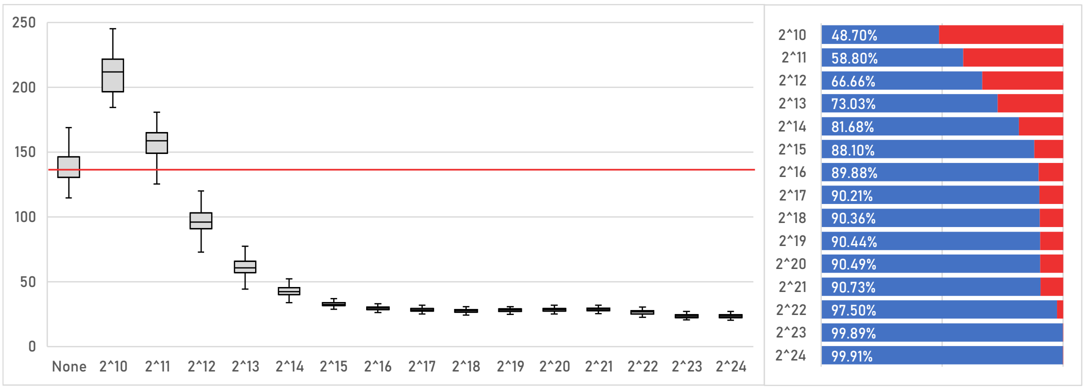
Overall, I recommend using the least recently used strategy with an atomic counter. If the cache is known to be sufficiently large (which depends on a variety of factors), one may choose to use the frame counter approximation instead but the very small performance improvement is hardly worth the risk of significantly worse performance if the cache is too small. Random eviction is a valid option as well as it performs similarly to the other strategies, is the easiest to implement, and doesn't require an additional field in the cache entry.

6.6 Effects of Cache Parameters on Performance

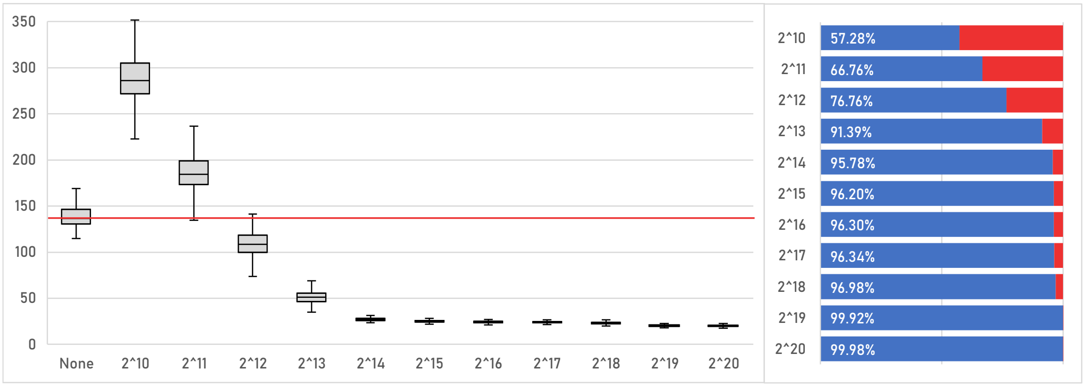
6.6.1 Cache Size

Choosing the cache size is a tradeoff between memory usage and performance. A bigger cache allows cache entries to stay in the cache longer and to be reused more often. However, graphics memory is limited and also needed for models, textures and the ray tracing acceleration structure. I tested the effect of the cache size on the performance and hit rate by rendering the same sequence of frames with cache sizes ranging from $2^{10} = 1024$ entries to $2^{24} = 16777216$ entries for the non-tiled variant and from $2^{10} = 1024$ 4×4 tiles to $2^{20} = 1048576$ 4×4 tiles for the tiled variant. The results are shown in figure 6.15 and listed in table 6.3.

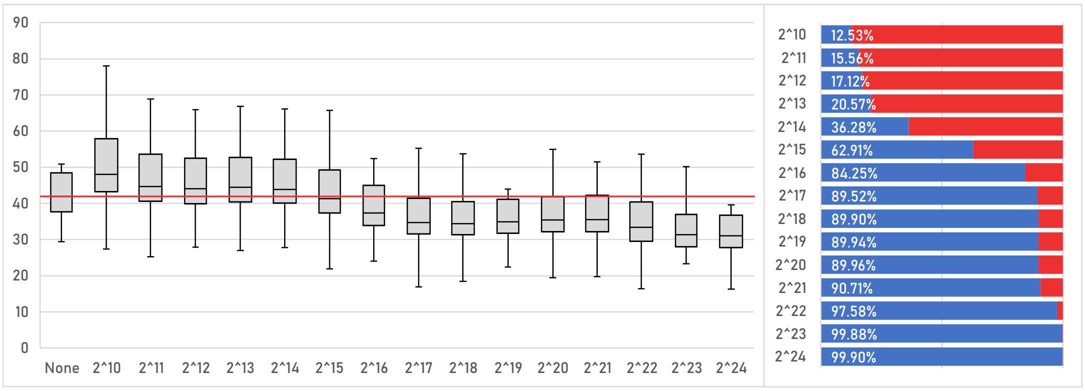
Generally speaking, larger caches result in better performance as expected. In the *Classroom* scene, a cache with only $2^{12} = 4096$ entries or tiles already improves the performance compared to using no cache at all. In the *Landscape* scene, a larger cache of $2^{15} = 32768$



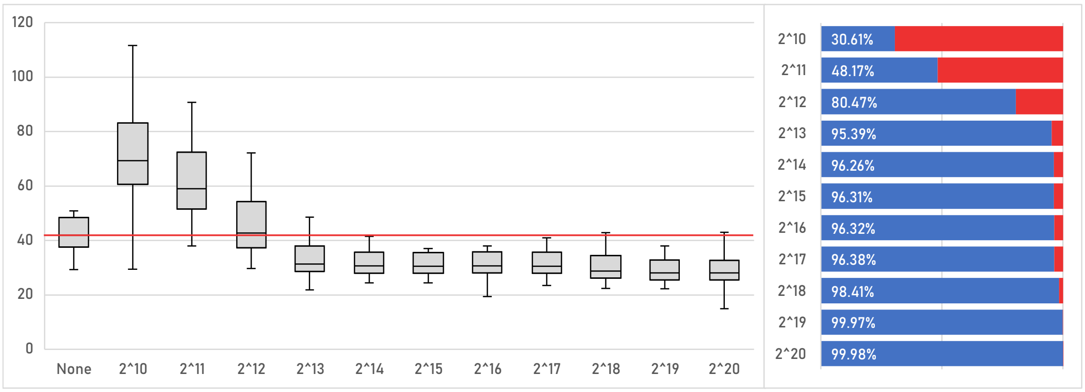
(a) Classroom, Non-Tiled



(b) Classroom, Tiled



(c) Landscape, Non-Tiled



(d) Landscape, Tiled

Figure 6.15: Frame times and hit rate for different cache sizes in two different scenes and in the tiled and non-tiled variant. Times are in milliseconds per frame. The cache size is the number of cache entries. The red lines are the median frame times without memoization. In the tiled variant, each cache entry is a 4×4 tile.

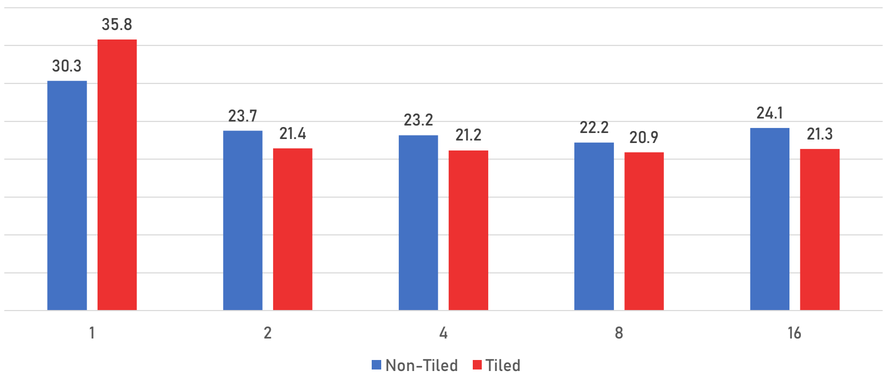


Figure 6.16: Median frame times of different linear probing distances. Times are milliseconds per frame.

entries or $2^{13} = 8192$ tiles was necessary. Performance and hit rate improve significantly with larger caches up to a certain point, then remain similar for a while and eventually start improving again slightly for very large caches. Qualitatively, the curves are similar in both scenes and variants. However, there are significant quantitative differences. The *Classroom* scene benefits much more from the memoization than the *Landscape* scene. The *Landscape* scene requires a larger cache than the *Classroom* scene to achieve similar hit rates. This is likely because the *Classroom* scene is a relatively small closed room, which limits the area covered by secondary hits, resulting in more reuse. Independently from this effect, the same hit rate results in a significantly higher performance improvement in the *Classroom* scene. One reason for this is that the *Classroom* scene has many different materials while the *Landscape* scene has very few. Divergent branching into different material graphs is a major problem in the classroom scene that is alleviated by memoization. Another reason is that in an outdoor scene, many secondary rays hit the skybox. This means that material graphs are less often evaluated at secondary hits where the evaluation is more divergent than at primary hits.

6.6.2 Linear Probing Distance

When reading a cache entry, I search a limited section of the cache starting at the location given by the hash of the key for an entry with a matching key. When writing an entry, I iterate over the same section and use an eviction strategy to decide which entry will be overwritten. The linear probing distance, i.e. the number of entries that I test for each cache access affects the performance of the cache. A high linear probing distance results in a lot of time being spent iterating over cache entries while a low one prevents the eviction strategy from working effectively. I compared the performance of the linear probing distances 1, 2, 4, 8 and 16 in the tiled and non-tiled variant. I used the least recently used eviction strategy. The results are shown in figure 6.16.

In both variants, the best performance is achieved with a linear probing distance of eight cache entries. Using a linear probing distance of one (i.e. no linear probing at all) is significantly worse than any of the other values. The differences between the others are relatively small.

6.6.3 Tile Size

The tile size in the tile-based variant affects performance in several ways. First, larger tiles have the advantage that fewer tiles are used within a wave, which improves hardware

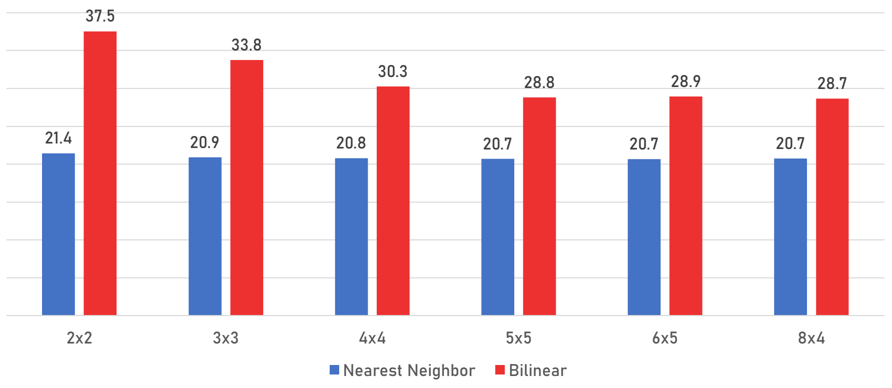


Figure 6.17: Median frame times of different tile sizes. Times are milliseconds per frame.

cache locality. Second, calculating and writing a tile requires some per-tile operations. Therefore, a few large tiles have less overhead than many small ones. Third, larger tiles result in more unnecessarily computed and cached texels. Even when only a single texel is needed, the entire tile must be computed and uses up space in the cache. Finally, when bilinear interpolation is used, tiles must overlap by one texel. Larger tiles result in less overlap. For example, a 3×3 tile only represents a 2×2 area in texture space when bilinear interpolation is used, while a 5×5 tile represents a 4×4 area. This means that 3×3 tiles require $\frac{9}{4} = 2.25$ times the texels to cover a large area in texture space while 5×5 tiles only require $\frac{25}{16} = 1.5625$ times the texels.

My implementation supports any tile size below the number of lanes per wave (32 on my RTX 4070 Ti). Larger tiles would require each lane to calculate multiple texels of a single tile, which would further complicate the process of calculating and writing a tile. I compared the performance of the tile sizes 2×2, 3×3, 4×4, 5×5, 6×5 and 8×4 using both nearest-neighbor and bilinear interpolation. I set the cache size to approximately 2^{24} texels for each run, i.e., I lowered the number of cache entries accordingly for bigger tiles (for example, I used 2^{22} 2×2 tiles but only 2^{20} 4×4 tiles). The results are shown in figure 6.17.

Bigger tiles perform better with both interpolation methods. However, the differences are small when nearest-neighbor interpolation is used and significantly bigger when bilinear interpolation is used. 5×5, 6×5 and 8×4 tiles perform nearly identically.

6.7 Evaluation of Various Modifications of the Tiled Cache

I tried various small modifications of the tiled variant to further improve its performance. These attempts were not successful. I documented the results in this section along with a brief discussion of why they did not have the desired effect.

6.7.1 Grouping Missing Tiles by Material

I implemented an alternative way of calculating missing tiles that attempts to group missing tiles within a wave by material (section 5.4.4). If the tile size is small enough, my system can calculate multiple tiles simultaneously by assigning their texels to different lanes within the same wave. However, this only has an effect if the tiles have the same material because the material graph evaluation would be divergent otherwise and would therefore effectively still be performed sequentially. The modified algorithm changes the order in which tiles are evaluated to ensure that as many of them as possible are evaluated simultaneously.

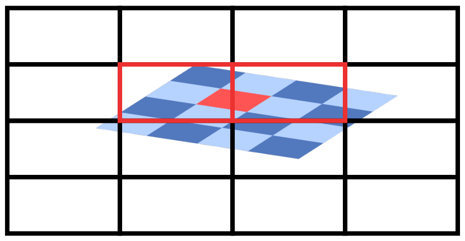


Figure 6.18: The red tile intersects the pixel rectangles of two waves. If these two waves are executed in parallel and the tile is not already cached, they might both have a cache miss, calculate the tile, and attempt to write it.

Using this modified algorithm did not improve performance. Testing it in the *Classroom* scene with a tile size of 4×4 resulted in a very slight increase in frame times from 20.1 ms to 20.3 ms, rather than a decrease. To investigate this, I logged the number of different tiles per wave as well as the number of different materials per wave. Grouping 4×4 tiles by material on a GPU with 32 lanes per wave can only have an effect for waves that compute more than two tiles and more than one material. Around 6.6% of all waves had any missing tiles to compute. Of these waves, 1.3% (0.086% of all waves) had more than two tiles and 0.050% (0.0033% of all waves) had more than one distinct material. The potential performance gain is therefore negligible and not worth the overhead of the alternative algorithm.

6.7.2 Distributing Frame Rendering Over Multiple Passes

When multiple waves that are executed in parallel need the same tile and that tile is not already cached, they all compute the tile but only one of them writes the result to the cache. This can happen randomly and is difficult to avoid for secondary hits. Primary rays are much less random. Each wave is responsible for calculating a small rectangle of pixels. Waves whose rectangles are next to each other are more likely to be executed in parallel but are also likely to compute the same missing tiles. Tile boundaries form a grid in texture space that is usually not perfectly aligned with the screen-space grid of waves. As a result, tiles often intersect multiple adjacent wave rectangles in screen space (figure 6.18). These redundant calculations are a potential performance problem.

To investigate how big of a problem it is in practice, I logged all tile calculations and then counted the distinct tiles per frame. If the same tile is calculated multiple times in the same frame, it was likely calculated redundantly by waves that were executed in parallel. It is also possible that a tile was written, then evicted and then written again in a single frame but that is unlikely because I used a large cache of 2^{20} tiles and the least-recently-used eviction strategy for this test. From the number of tiles computed in a frame and the number of distinct tiles computed in a frame, I calculated the duplicate ratio, i.e. what percentage of tiles has already been computed in the same frame. The median duplicate ratio was 13.4%, which is concerningly high.

In an attempt to reduce this duplicate ratio, I created a variant of my path tracer that calculates an image in multiple passes by dispatching the compute shader multiple times per frame. Each pass calculates a subset of the pixels. I spread out the pixel rectangles of the waves in such a way that they either only touch at corners or not at all (figure 6.19). The former requires two passes, the latter four.

The two-pass variant reduced the median duplicate ratio drastically to 1.48%. The four-pass variant reduced it further to 0.23%. Surprisingly, this had almost no effect on performance and the small effect that it had was that rendering in a single pass was the fastest

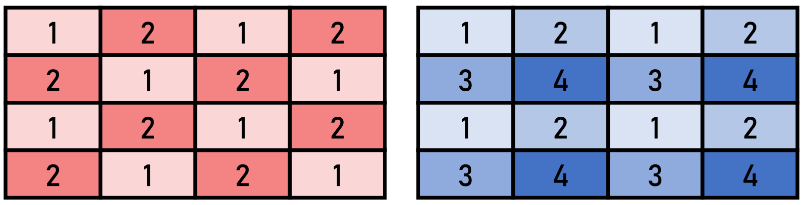


Figure 6.19: Distributing the calculation of a frame over two (left) or four (right) passes. The rectangles represent waves (8 × 4 pixels on my GPU). The numbers indicate in which pass a wave is executed.

and rendering in four passes was the slowest. The median time went up from 20.2 ms in the single-pass variant to 20.5 ms in the two-pass variant and 20.7 ms in the four-pass variant. At first, I suspected that rendering in multiple passes also has significant disadvantages that simply cancel out the benefits. For example, spreading out the primary rays could have reduced hardware cache hit rates. However, comparing the variants in the NVIDIA Nsight Trace Profiler showed no major differences. It seems more likely that preventing the redundant computations had almost no effect and that the slightly worse performance is a consequence of cutting the problem into smaller units, which slightly hampers parallel execution. The reason why the multi-pass variants do not perform better may be that they only reduce redundant computations for primary hits. Tiles for primary hits are cheaper to compute than for secondary hits because of a high degree of hardware cache locality and because the tiles computed for the primary hits in a wave are likely to have the same material, meaning that the graph evaluation is unlikely to be divergent.

Another attempt to solve the same problem was to do a “pre-pass” before rendering each frame that traces a small subset of the paths traced for the entire frame. The idea is that these paths already hit most of the relevant tiles needed to render the frame. Because the paths are less dense, duplicate tiles are less likely in the pre-pass. This approach was not successful either. I tested three different path densities for the pre-pass: one path for every 2 × 2 pixels, one for every 4 × 4 pixels and one for every 8 × 8 pixels. The median frame times went up from 20.2 ms to 24.5 ms, 23.9 ms and 23.6 ms respectively. The reason for the worse performance is that more paths are traced and that reducing the number of duplicate tiles does not have the expected performance benefit.

7. Conclusion

In this thesis, I explored material graph memoization in real-time path tracing in two main variants and with various parameters and smaller variations. I showed that the presented techniques significantly reduce the cost of using complex material graphs or many different ones in a scene. This gives artists more freedom in how they build scenes and design materials.

The tiled variant typically performs better than the non-tiled one. Despite this, I recommend considering both variants in practice because the tiled variant has multiple limitations compared to the non-tiled one. In the non-tiled variant, all vertex attributes are available in the material graphs. In the tiled one, only the texture coordinates are directly available while others are challenging to provide. This limits the creative freedom of artists to some extent. Another problem is that the tiled variant requires a certain number of lanes in a wave to be active when material graphs are evaluated so that all texels of a tile can be computed. This may not be the case if memoization is added to an existing path tracer. Even in a new path tracer written from the ground up, keeping lanes active is not always an easy task and limits certain design decisions. In particular, the implicit branching of assigning different hit shaders to different instances cannot be used. Reactivating inactive lanes for a section of a shader is technically conceivable but shading languages do not yet provide this functionality. Finally, another minor disadvantage of the tiled variant is that it is more complicated to implement while the non-tiled one is relatively simple.

When using the presented techniques in practice, I also recommend considering whether caching both primary and secondary hits or only secondary hits is more appropriate for the specific use case. Without mipmap bias, caching primary hits has a slight impact on image quality. For secondary hits, the difference is usually unnoticeable. Caching only secondary hits still provides relevant performance benefits compared to not using memoization at all. Caching only secondary hits is also more convenient if rasterization is used instead of primary rays. In that case, one would only implement memoization in the path tracer but use classic material-specific fragment shaders for rasterization.

While I believe the presented techniques to already be useful in their current form, I see multiple possible paths for further improvements. One limitation of my implementation is that cached values must not depend on the view direction or the current time. On the one hand, this makes sense because both limit reuse. On the other hand, when view direction or time dependency do occur in material graphs, they are often used to modify the texture coordinates before the rest of the graph is evaluated. The remaining graph then uses these modified coordinates. For example, parallax mapping applies a slight view-dependent offset to create an illusion of depth and UV scrolling applies a time-dependent offset for simple animations. The memoization system could be extended to perform such computations before the cache lookup and only memoize the remaining graph. More generally, applying memoization to well-suited subgraphs instead of the entire graph would make the system more flexible.

I implemented generic eviction strategies like least-recently-used and demonstrated that these work well in practice. However, even better eviction may be possible by using additional knowledge such as camera and object motion. One could either extrapolate the position of cameras and objects from the current frame or render with a delay of one frame so that the exact positions for the next frame are already known. This could in theory be used to predict the reuse of cache entries more accurately. However, especially for secondary rays, it may be too difficult or expensive.

I explained and implemented bilinear interpolation for the tiled variant. However, trilinear interpolation, i.e. interpolating between two mipmap levels would be desirable and is not supported in my implementation. One possible implementation is to query the cache for two tiles at different mipmap levels. Another is to always include two mipmap levels in each tile, similar to the one texel overlap for bilinear interpolation.

Acknowledgements

I thank my advisor Vincent Schüßler for his valuable feedback and advice in writing this thesis. I thank Unity Technologies for the Unity Engine that I based my implementation on. I also thank the artists whose models, scenes and textures I used in the example scenes as attributed below. Finally, I thank my family for their continuous support throughout my life and in my studies.

Attributions

Some modifications to the 3D models were made due to the limitations of my path tracer, such as the lack of support for transparent materials and area lights.

Bibliography

[AMCB$^+$21] T. Akenine-Möller, C. Crassin, J. Boksansky, L. Belcour, A. Panteleev, and O. Wright, “Improved shader and texture level of detail using ray cones,” *Journal of Computer Graphics Techniques (JCGT)*, vol. 10, no. 1, pp. 1–24, January 2021. [Online]. Available: http://jcgt.org/published/0010/01/01/

[ANA$^+$20] P. Andersson, J. Nilsson, T. Akenine-Möller, M. Oskarsson, K. Åström, and M. D. Fairchild, “ꟻLIP: A Difference Evaluator for Alternating Images,” *Proceedings of the ACM on Computer Graphics and Interactive Techniques*, vol. 3, no. 2, pp. 15:1–15:23, 2020.

[APX14] J.-M. Arnau, J.-M. Parcerisa, and P. Xekalakis, “Eliminating redundant fragment shader executions on a mobile gpu via hardware memoization,” in *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, 2014, pp. 529–540.

[Ble23] Blender Online Community, *Shader Editor*, Blender Foundation, 2023, last accessed on 2023-04-03. [Online]. Available: https://docs.blender.org/manual/en/latest/editors/shader_editor.html

[Cor23] M. Corporation, “DirectX raytracing (DXR) functional spec,” https://microsoft.github.io/DirectX-Specs/d3d/Raytracing.html, 2023.

[Epi23] Epic Games, “Material editor reference,” https://docs.unrealengine.com/4.27/en-US/RenderingAndGraphics/Materials/Editor/, 2023, last accessed on 2023-05-12.

[FH22] S. Fujieda and T. Harada, “Progressive material caching,” in *SIGGRAPH Asia 2022 Technical Communications*, ser. SA ’22. New York, NY, USA: Association for Computing Machinery, 2022. [Online]. Available: https://doi.org/10.1145/3550340.3564223

[Hei14] E. Heitz, “Understanding the masking-shadowing function in microfacet-based brdfs,” *Journal of Computer Graphics Techniques (JCGT)*, vol. 3, 06 2014.

[Hen18] N. Henning, “Vulkan subgroup tutorial,” 2018, last accessed on 2023-03-20. [Online]. Available: https://www.khronos.org/blog/vulkan-subgroup-tutorial

[JPC00] T. R. Jones, R. N. Perry, and M. Callahan, “Shadermaps: A method for accelerating procedural shading,” Mitsubishi Electric Research Laboratory, Cambridge, Massachusetts, Technical Report TR2000-25, June 2000. [Online]. Available: https://www.merl.com/publications/docs/TR2000-25.pdf

[Kaj86] J. T. Kajiya, “The rendering equation,” *SIGGRAPH Comput. Graph.*, vol. 20, no. 4, pp. 143–150, aug 1986. [Online]. Available: https://doi.org/10.1145/15886.15902

[KGPB05] J. Krivanek, P. Gautron, S. Pattanaik, and K. Bouatouch, “Radiance caching for efficient global illumination computation,” *IEEE Transactions on Visualization and Computer Graphics*, vol. 11, no. 5, pp. 550–561, 2005.

[LDN04] S. Lefebvre, J. Darbon, and F. Neyret, “Unified Texture Management for Arbitrary Meshes,” INRIA, Research Report RR-5210, 2004. [Online]. Available: https://hal.inria.fr/inria-00070783

[MRNK21] T. Müller, F. Rousselle, J. Novák, and A. Keller, “Real-time neural radiance caching for path tracing,” *CoRR*, vol. abs/2106.12372, 2021. [Online]. Available: https://arxiv.org/abs/2106.12372

[NVI09] NVIDIA Corporation, “Nvidia’s next generation cuda compute architecture: Fermi,” 2009, last accessed on 2023-03-20. [Online]. Available: https://www.nvidia.com/content/pdf/fermi_white_papers/nvidia_fermi_compute_architecture_whitepaper.pdf

[O’N14] M. E. O’Neill, “Pcg: A family of simple fast space-efficient statistically good algorithms for random number generation,” Harvey Mudd College, Claremont, CA, Tech. Rep. HMC-CS-2014-0905, Sep. 2014.

[PGSD13] S. Popov, I. Georgiev, P. Slusallek, and C. Dachsbacher, “Adaptive quantization visibility caching.” *Comput. Graph. Forum*, vol. 32, no. 2, pp. 399–408, 2013. [Online]. Available: http://dblp.uni-trier.de/db/journals/cgf/cgf32.html#PopovGSD13

[PJH16] M. Pharr, W. Jakob, and G. Humphreys, *Physically Based Rendering: From Theory to Implementation*, 3rd ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2016.

[RKLC$^+$11] J. Ragan-Kelley, J. Lehtinen, J. Chen, M. Doggett, and F. Durand, “Decoupled sampling for graphics pipelines,” *ACM Trans. Graph.*, vol. 30, no. 3, may 2011. [Online]. Available: https://doi.org/10.1145/1966394.1966396

[Sch94] C. Schlick, “An inexpensive brdf model for physically-based rendering,” *Computer Graphics Forum*, vol. 13, no. 3, pp. 233–246, 1994. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1111/1467-8659.1330233

[SM09] P. Shirley and S. Marschner, *Fundamentals of Computer Graphics*, 3rd ed. USA: A. K. Peters, Ltd., 2009.

[Uni23] Unity Technologies, “Using shader graph,” https://docs.unity3d.com/Packages/com.unity.shadergraph@14.0/manual/index.html, 2023, last accessed on 2023-04-03.

[WBC$^+$21] S. White, D. Batchelor, D. Coulter, A. Miles, M. Jacobs, and M. Satran, *HLSL Shader Model 6.0*, Microsoft Corporation, 2021. [Online]. Available: https://learn.microsoft.com/en-us/windows/win32/direct3dhlsl/hlsl-shader-model-6-0-features-for-direct3d-12

[WCJS20] S. White, D. Coulter, M. Jacobs, and M. Satran, *Atomic Functions*, Microsoft Corporation, 2020. [Online]. Available: https://learn.microsoft.com/en-us/windows/win32/direct3d11/direct3d-11-advanced-stages-cs-atomic-functions

[Wil83] L. Williams, “Pyramidal parametrics,” *SIGGRAPH Comput. Graph.*, vol. 17, no. 3, pp. 1–11, jul 1983. [Online]. Available: https://doi.org/10.1145/964967.801126

[WMLT07] B. Walter, S. R. Marschner, H. Li, and K. E. Torrance, “Microfacet models for refraction through rough surfaces,” in *Proceedings of the 18th Eurographics Conference on Rendering Techniques*, ser. EGSR’07. Goslar, DEU: Eurographics Association, 2007, pp. 195–206.

[WRC88] G. J. Ward, F. M. Rubinstein, and R. D. Clear, “A ray tracing solution for diffuse interreflection,” *SIGGRAPH Comput. Graph.*, vol. 22, no. 4, pp. 85–92, jun 1988. [Online]. Available: https://doi.org/10.1145/378456.378490

[ZCJE22] A. Zhang, K. Chen, H. Johan, and M. Erdt, “High-performance adaptive texture streaming and rendering of large 3d cities,” *Vis. Comput.*, vol. 38, no. 4, pp. 1245–1262, apr 2022. [Online]. Available: https://doi.org/10.1007/s00371-021-02152-z

Erklärung

Ich versichere, dass ich die Arbeit selbstständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht und die Satzung des KIT zur Sicherung guter wissenschaftlicher Praxis in der jeweils gültigen Fassung beachtet habe. Die Arbeit wurde in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt und von dieser als Teil einer Prüfungsleistung angenommen.

Karlsruhe, den 22. Mai 2023

(Bastian Urbach)