DOI: 10.1111/cgf.70338

EUROGRAPHICS 2026 / B. Masia and J. Thies
(Guest Editors)

COMPUTER GRAPHICS *forum*
Volume 45 (2026), Number 2

Variable-Rate Texture Compression: Real-Time Rendering with JPEG

Elias Kristmann[1], Michael Wimmer[1], Markus Schütz[1]

[1]TU Wien, Institute of Visual Computing & Human-Centered Technology

Figure 1: *Rendering Sponza with JPEG-compressed textures at 1500+ frames per second.*

Abstract

Although variable-rate compressed image formats such as JPEG are widely used to efficiently encode images, they have not found their way into real-time rendering due to special requirements such as random access to individual texels. In this paper, we investigate the feasibility of variable-rate texture compression on modern GPUs using the JPEG format, and how it compares to the GPU-friendly fixed-rate compression approaches BC1 and ASTC. Using a deferred rendering pipeline, we are able to identify the subset of blocks that are needed for a given frame, decode these, and colorize the framebuffer's pixels. Despite the additional ∼0.17 bit per pixel that we require for our approach, JPEG maintains significantly better quality and compression rates compared to BC1, and depending on the type of image, outperforms or competes with ASTC. The JPEG rendering pipeline increases rendering duration by less than 0.3 ms on an NVIDIA RTX 4090, demonstrating that sophisticated variable-rate compression schemes are feasible on modern GPUs, even in VR.

Source code and data sets are available at: https://github.com/elias1518693/jpeg_textures

CCS Concepts

• ***Computing methodologies*** → *Image compression; Rendering;*

1. Introduction

In the rapidly evolving landscape of digital media, few technologies have endured with as much prominence as JPEG [Wal92]. How-

ever, although it remains a popular image *storage* format, JPEG – and other variable-rate compressed image formats – are considered unsuitable for real-time *rendering* since they do not allow efficient access to individual texels. Due to the variable bit rates of different sections of the image, we cannot easily map a texel's index or uv-coordinates to its location in memory without creating additional indexing tables that raise the texture's memory usage. Furthermore, JPEG is based on Huffman-coded [Huf52] data that must be decoded sequentially, and therefore limits our ability to parallelize the decompression. Finally, pixels in JPEG are encoded block-wise in the frequency domain via coefficients of discrete cosine transformations, requiring numerous multiplications and additions to retrieve a single pixel's color.

In contrast, GPU-friendly fixed-rate compression algorithms encode texels in blocks with equal bit rates, allowing us to directly map a texel's coordinate to its location memory. We can then use the per-block encoded data such as color gradients and per-texel encoded data such as the weight to efficiently decode the color value with little computational effort. However, this comes at a cost of reduced compression efficiency. With limited VRAM capacity and the substantial increase in texture data associated with modern photorealistic rendering, these fixed-rate compression techniques are approaching the point where they can no longer meet the memory requirements.

JPEG and modern counterparts such as AVIF, WebP, and JPEG XL achieve higher compression ratios, enabling larger amounts of high-resolution textures to be stored in GPU memory while also reducing disk-to-memory streaming times. In this paper, we investigate whether the principles of variable-rate image compression can be applied effectively in real-time rendering. We focus on JPEG, arguably the most widely used variable-rate format. By demonstrating its feasibility for this purpose, we aim to challenge prevailing assumptions about the unsuitability of variable-rate compression for GPU textures and to lay the groundwork for adopting more advanced variable-rate compression schemes in real-time rendering.

In particular, our contributions to the state-of-the-art are:

- An efficient deferred rendering pipeline capable of real-time rendering JPEG-compressed textures with an overhead of less than 0.3ms on an NVIDIA RTX 4090.
- A texture block cache that retains previously decoded JPEG blocks, reducing the amount of data that needs to be decoded in each frame.
- A performance study that demonstrates massive benefits of mipmapping for this approach due to reduced decoding workload; the efficiency of caching; and the suitability to VR-rendering since stereo viewpoints share most of the required texels.
- A quality study that illustrates the benefits of variable-rate over fixed-rate texture compression, particularly AVIF and JPEG XL.

2. Related Work

In the context of real-time-rendering, we can categorize texture compression algorithms into fixed-rate compression, variable-rate compression, and the emerging neural texture compression approaches.

2.1. Fixed-rate compression

Fixed-rate compression refers to the fact that all segments of the image are encoded with the same bit rate, which allows us to compute the memory location of a texel from its coordinate. GPU-based algorithms also make an effort to keep the decoding computationally inexpensive.

Despite being over two decades old, S3 Texture Compression (S3TC) [INH99], also known as DXT or more recently BC1 through BC7, remains one of the most widely used texture compression algorithms today. BC1, for example, compresses images by dividing them into 4×4 pixel blocks and computing two color endpoints in RGB space for each block. The algorithm then encodes all pixels within the block as interpolations between these endpoints. BC1 and BC7 are tailored for color images, with BC1 offering lower quality but achieving twice the compression rate at 4 Bits per Texel (bpt) compared to the more modern and higher-fidelity BC7 at 8 bpt. Similarly, Adaptive scalable texture compression (ASTC) [NLP*12] builds on this idea but makes the block sizes adaptive. This allows the method to adjust better to the characteristics of a texture, achieving higher and more controllable compression, from 0.89 to 8.0 bpt, with even better visual quality. However, this improved compression comes with an increased complexity. Ericsson Texture Compression (ETC1/ETC2), available in OpenGL ES, targets with its low complexity lower-end devices and mobile phones [SA05]. More recently, Neural Texture Block Compression [FH24] uses a neural network that learns to optimize BC1 compression and therefore achieves better results for the same storage format than the handcrafted compression algorithms. Chen et al. propose a form of fixed-rate JPEG where they compress each JPEG block with different quantization tables until it fits into the fixed block size [CL02]. Hollemeersch et al. transform a texture with the discrete cosine transform (DCT) and then only store a fixed and limited number of coefficients for each block [HPLV12]. While this only slightly improves the compression rate compared to BC1, they also show that texture filtering can be done in the frequency domain before decoding, greatly increasing its efficiency.

2.2. Variable-rate compression

Variable-rate compression formats adjust the bit rate to the content, making some sections of an image compress better than others. Since they do not cater to real-time rendering, they also employ complex and computationally expensive algorithms that sacrifice decoding performance for higher compression rates.

The most famous lossy compression standard for images is JPEG [Wal92] defined in 1992. Since then, many follow-ups have emerged, offering new features and improved compression, such as JPEG2000 [CES00] and most recently JPEG XL [AvAB*19; SAV*25]. However, despite their advancements, these newer algorithms have not yet achieved the widespread support and adoption of the original JPEG standard.

Random access to JPEG blocks has been successfully applied to offline rendering, as shown by Radziszewski et al. [RA08]. Because the algorithm is specifically designed for CPU-based processing, its decoding efficiency is relatively limited. However, it provides a straightforward method for accessing individual texels

in JPEG-compressed images, by maintaining a list of offsets to the start of each Minimum Coded Unit (MCU) in the compressed data. Olano et al. introduce an online variable-rate texture compression technique that also stores an index list to each MCU and uses it to parallelize and therefore speed up the decoding process [OBGB11]. They avoid problems related to the missing random access by completely decompressing the textures on the GPU before rendering. Consequently, this approach provides storage savings primarily in scenarios where only a subset of textures is required at any given time, allowing for more textures to be stored in memory, but it does not reduce VRAM usage for an individual texture while it is used. Although far from a real-time application, Fichet et al. demonstrate a method for compressing spectral images used in spectral rendering by converting them to the JPEG XL format, and then decoding them while rendering, achieving file size reductions of 10 to 60 times compared to ZIP compressed files [FP25].

2.3. Neural texture compression

Recently, the trend has shifted to neural texture compression methods, utilizing neural networks to learn efficient and perceptually optimized compression schemes from feature vectors. These methods often achieve higher compression ratios and improved visual quality compared to conventional block-based compression algorithms, but have reduced decoding efficiency as a tradeoff. In Random-Access Neural Compression of Material Textures (NTC) [VSW*23], the textures are stored as a feature pyramid, with multiple channels compressed together. Therefore, this method excels in scenarios with various channels that have a strong correlation. Farhadzadeh et al. [FHL*24] promise even greater compression rates with their neural compression method, but offer no indication of the decoding efficiency of their method. Weinreich et al. [WdOHN24] leverage existing hardware support for S3TC block compression to encode and store their neural texture compression network, thereby reducing overall memory requirements. Building on this approach, Belcour et al. [LA25] enhance the performance by utilising cooperative vectors.

2.4. Other

Schuster et al. [STS*21] introduce a sparse coding technique to efficiently represent textured splats. For each splat data set, they construct a dictionary of "atoms" – small images (e.g. 32x32 pixels) that capture frequently occurring patterns in a data set. Each splat holds an average color, a list of indices to atoms, and a set of weights with which the atoms are multiplied to recover the texture. Luo et al. reduce redundancy by merging all the textures in a scene and remapping the texture coordinates to eliminate repeated content [LJP*23]. Zhang et al. [ZGX*24] propose using 2D Gaussian splats — a representation recently popularized in photorealistic 3D rendering [KKLD23] — for image representation and compression. To further improve compression they apply entropy encoding to the Gaussians. Zhang et al. [ZLK*25] propose a similar approach but manage to improve the results even without entropy coding. Their method adaptively distributes Gaussians based on local image complexity, allocating more splats to regions with fine structural detail. This leads to higher reconstruction fidelity in edge-rich areas while maintaining efficient compression overall. The representation supports random access, making it a possible candidate for texture compression.

Another way to reduce texture costs is to first compress them using a GPU-native format (such as BCn) and then apply a secondary, lossless and variable-rate compression algorithm. This "supercompression" achieves higher ratios for CPU storage and CPU–GPU transfer bandwidth. However, in contrast to our variable-rate compression method, they do not offer additional VRAM savings beyond the underlying GPU format, as they must be transcoded into that format before use. Ström et al. [SW11] propose an image-domain, prediction-based scheme for supercompressing textures. Krajcevski et al. [KPM16] apply a wavelet transform followed by Asymmetric Numeral Systems (ANS) [Dud09] entropy coding to BCn textures, a process conceptually similar to JPEG compression. Basis Universal [Bin25] is an open-source format that can transcode to most modern GPU texture formats at runtime, ensuring cross-platform compatibility. Oodle Texture [RAD25] has become an industry standard by rearranging bits within block-compressed textures to reduce entropy, thereby increasing the efficiency of subsequent lossless supercompression.

Beyond further texture compression, Virtual Texturing (VT) [LDN04; MG08] offers an alternative for managing VRAM constraints by uploading only the specific texels required for the current frame. In this scheme, a pre-pass identifies visible texture regions, which are then streamed into a large, pre-allocated physical texture or page cache. Because this content is view-dependent and updates dynamically, the technique draws a direct analogy to virtual memory in operating systems [May10]. Consequently, VRAM usage is bounded by the fixed size of the physical cache rather than the total footprint of the source assets. This allows shaders to sample textures through a uniform indirection mechanism, eliminating the need for frequent rebinding of individual resources.

However, VT introduces significant complexity; the system must constantly determine page visibility and stream missing data into the cache. This management overhead can become a bottleneck in scenes with rapid camera movement or high-frequency detail, where page churn and streaming latency lead to visible "pop-in" artifacts. Our approach shares the two-pass structure of VT by analyzing visibility before fetching texels, but we retain the entire compressed texture set in VRAM. By doing so, we remove data transfer costs and simplify the addressing logic. Despite these differences, our method is highly compatible with a VT pipeline, as we already have a tiled format. In addition, some virtual texturing already use DCT based compression on the CPU side, and transcode them similar to supercompression, which would be no longer required with our approach [Van06; vWH10].

3. Background

This section presents an overview of the concepts and algorithms behind JPEG and BC1, providing the fundamentals to understand our JPEG texture mapping approach, as well as reasons why standard JPEG is generally considered unsuitable for GPUs, and why BC1 is deemed suitable.

(a) *Image* **(b)** *MCU* **(c)** *Data Units (DU)*

Figure 2: *The building blocks of JPEG compression. An image using 4:2:0 chroma subsampling is partitioned into Minimum Coded Units (MCUs) of size 16 × 16 pixels. Each MCU is encoded as six Data Units (DUs): four 8 × 8 luminance blocks and two 8 × 8 chrominance blocks (one each for Cb and Cr).*

3.1. JPEG

JPEG is built on several concepts such as Huffman coding, difference coding, encoding in the frequency-domain, human perception, quantization, etc., each of which contributes to the compression efficiency. The YCbCr color space is used instead of RGB as it allows exploiting the higher sensitivity of humans towards differences in luminance. This color space separates luminance (Y) from chrominance components: blue-difference (Cb) and red-difference (Cr). In this paper, we specifically focus on JPEGs with 4:2:0 chroma subsampling – a commonly used configuration that samples luminance at full resolution but color information at half resolution (one Cb and Cr value per 2×2 pixels).

Structure: JPEG compresses images in blocks called minimum coded units (MCUs, see Figure 2) – aptly named after the fact that accessing a single pixel inside an MCU requires decoding the entire MCU. The pixel size of an MCU varies depending on the configuration. Using 4:2:0 chroma subsampling, each MCU represents a block of 16×16 pixels, and each MCU is further subdivided into 6 Data Units (DUs) that store 8×8 values. Since we sample luminance at full resolution and colors at half resolution, 4 of these 6 DUs are for luminance (one value per pixel). The remaining two are for Cb and Cr, respectively, each of which stores one value per 2×2 pixels. However, instead of storing colors per-pixel in the spatial domain, JPEG encodes colors in the frequency domain in form of a weighted sum of DCT components. The first weight/coefficient of each DU is called the DC coefficient, from which the average pixel value is derived. The remaining 63 coefficients are the AC coefficients. Compressing a 4096×4096 pixel image results in 65,536 MCUs and a total of 393,216 DUs.

Pipeline: To compress an image using baseline JPEG (Figure 3), the image is first converted from the RGB color space to YCbCr. Next, chroma subsampling is applied using a 4:2:0 scheme. In this process, the Cb and Cr channels are sampled at half the horizontal and vertical resolution compared to the Y channel. The image is then divided into MCUs (16×16 pixel) and DUs (six 8×8 blocks). Each DU undergoes a Discrete Cosine Transform (DCT), representing the block as a weighted sum of 64 cosine waves with varying frequencies. The coefficient at zero frequency in both dimensions, known as the DC coefficient, corresponds to the average of all samples. To further reduce the data size, the DC coefficients are encoded differentially, meaning that each DC coefficient is stored as the difference relative to the previous one inside the same channel, rather than its absolute value. The remaining 63 non-zero frequency coefficients are called AC coefficients. The AC coefficients are then quantized, typically using a standard quantization matrix, which discards many high-frequency components, resulting in numerous zero values. The different quality settings of JPEG modify this quantization matrix, where a low quality setting discards more frequencies. Finally, the quantized coefficients are compressed using run-length encoding followed by Huffman encoding. The decoding process follows the same steps, but in reverse order.

The major reasons why JPEG is considered unsuitable for real-time rendering on GPUs are:

1. No random access to individual texels. Due to the variable bit size of each MCU, we cannot directly map uv-coordinates to the memory location of MCUs and its texels without indexing tables that raise memory usage.
2. Even knowing the memory location of an MCU, fetching a single pixel during rasterization is expensive, as it requires decoding the entire MCU. This is because JPEG encodes in the frequency domain, where each coefficient affects the value of all pixels in the MCU. In other words, each individual pixel is a computationally expensive dot product of 384 elements – one dot product of $64 coefficients \times 64$ DCT components per DU.
3. AC coefficients are encoded sequentially, requiring us to decode 63 ACs per DU for a total of up to 378 ACs per MCU in a single-threaded fashion.
4. ACs are Huffman coded – decoding a single AC requires a loop to find a match between the next n bits in the stream and tens of codes in the Huffman table.

Our approach, as described in Section 4, addresses these issues through a combination of solutions such as a compact indexing table; warp-level parallel Huffman decoding; an MCU cache that allows reusing 16×16 blocks of pixels that were decoded in previous frames; and a deferred rendering pipeline as standard forward-renderers would run into unproductive stalls while waiting until texels are decoded.

3.2. BC1

The BC1 to BC7 compression formats all follow similar ideas and we will specifically focus on the BC1 format. BC1 stores textures in blocks of 4×4 pixels, using 64 bit per block for a total of 4 bit per pixel. 32 of a block's bits are used to store two 16 bit color values c_0 and c_1. The remaining 32 bit are distributed over the 16 pixels, encoding 2-bit interpolation weights between c_0 and c_1. As two bits grant us four possible states, each pixel can be either c_0; $\frac{2}{3}c_0 + \frac{1}{3}c_1$; $\frac{1}{3}c_0 + \frac{2}{3}c_1$; or c_1.

Encoding and finding suitable color endpoints that minimize the compression loss can be computationally expensive, but decoding is near-trivial. In contrast to the heavy and complex decoding pipeline of JPEG, BC1 only requires fetching 8 bytes, unpacking the two 16 bit color values, and interpolating the final color using each texel's 2-bit weight.

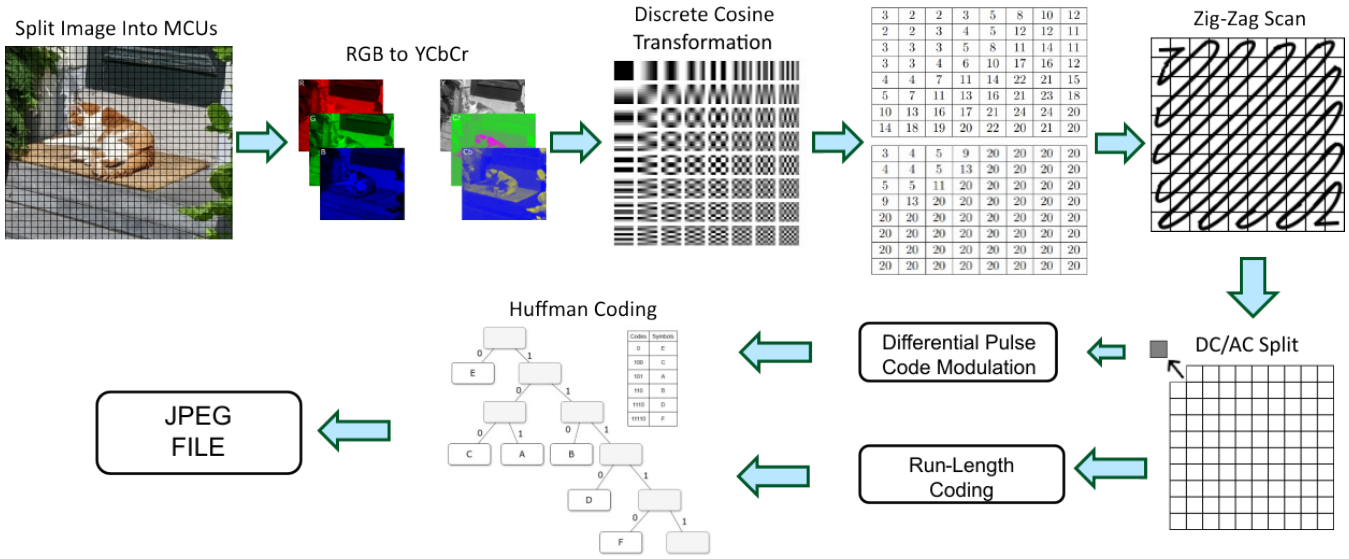


Figure 3: *Overview of the JPEG compression pipeline. The input image is converted from RGB to YCbCr and optionally chroma-subsampled, then divided into 8 × 8 blocks. Each block undergoes a discrete cosine transform (DCT), followed by quantization and zigzag reordering. The DC coefficient is encoded using differential pulse-code modulation (DPCM), while AC coefficients are compressed using run-length coding. Finally, all symbols are entropy-coded using Huffman coding to produce the JPEG bitstream.*

4. Method

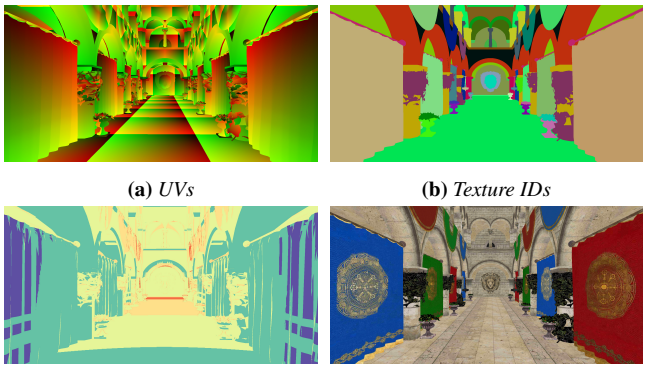
This section provides an algorithmic overview. Implementation details are provided in Section 5. We begin by determining which regions of each texture are actually required for rendering. Only these relevant portions are then decoded, enabling the scene to be rendered as usual without the need to decompress entire textures. Furthermore, any decoded texture regions that are needed in the subsequent frame are retained in a cache, reducing redundant decoding work and improving overall rendering efficiency. The overall process, as illustrated in Figure 5, is integrated into a deferred rendering pipeline, which consists of the following passes:

1. **Geometry Pass**: Draws all geometry into a G-Buffer comprising uv-coordinates, texture ID, and mipmap level.
2. **Mark**: Screen pass that identifies the MCUs that need decoding and reserves slots in the texture block cache.
3. **Decode**: Decodes queued MCUs and writes texels into the texture block cache.
4. **Resolve**: A screen pass that maps the G-Buffer's uv-coordinates, texture IDs and mip levels to decoded texel colors in cache.
5. **Update Cache**: Evict MCUs from cache that were not used in this frame.

Figure 4 shows the G-Buffer components from the geometry pass, and the final textured rendering after the resolve.

4.1. Texture Block Cache

To avoid re-decoding all visible MCUs each frame, we implement a texture block cache. It consists of a hash map that maps the index of an MCU to the block of 16 × 16 decoded texels, and a pool of such blocks from which entries can be acquired or released as needed.



(a) *UVs* **(b)** *Texture IDs* **(c)** *Mip Level* **(d)** *Rendering*

Figure 4: *(a,b,c) Visualisation of the additional G-Buffer components our method requires for deferred texturing. (d) Textured Rendering.*

4.2. Mark

This pass analyzes each pixel of the G-Buffer to identify the MCUs that we need to decode, but also to flag all previously decoded and still visible MCUs to keep them in cache. From the UV coordinate, we calculate the ID of the MCU:

$$\text{MCU} = \frac{(\lfloor u \cdot \text{width} \rfloor \bmod \text{width})}{16} + \frac{(\lfloor v \cdot \text{height} \rfloor \bmod \text{height})}{16} \cdot \left\lfloor \frac{\text{width}}{16} \right\rfloor$$

We then attempt to atomically reserve a spot in the cache's hash map using atomicCAS. If the MCU was not yet cached, the thread that was able to reserve a new spot in the hash map continues to add the MCU to a queue. If the MCU was already present in the cache, the thread instead sets that MCU's visibility flag inside the cache to true, ensuring that it will not get evicted from cache at the

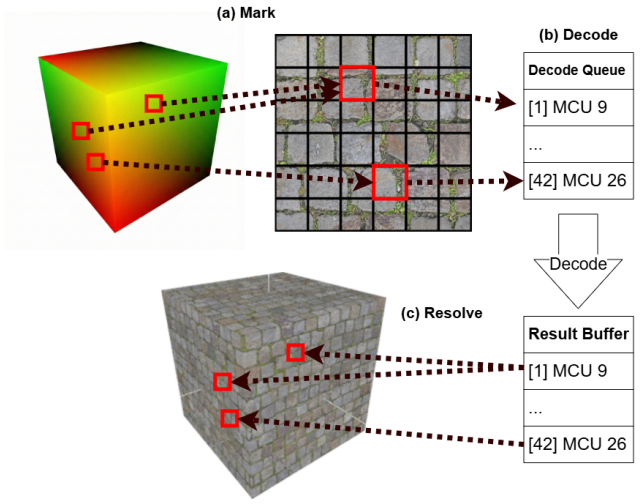


Figure 5: *Overview of our method. (a) For every pixel's UV coordinate, we compute the corresponding MCU and add it to the decoding queue. If an MCU is required multiple times, it is still added to the queue only once. (b) MCUs in the decoding queue are processed in parallel by GPU blocks. Each block decodes its assigned MCU and writes the resulting data to a buffer, preserving the order of insertion in the queue. (c) The decoded MCUs are retrieved from the buffer and sampled to compute the color for each pixel, completing the rendering process.*

end of the frame. For hash collisions, we implement linear probing with a maximum number of ten tries to place an entry, after which insertion would simply fail. However, it is mathematically near impossible that no empty slot is found within ten steps with a good hash function and a reasonably sized hash map. The size of the hash map should be chosen depending on how many MCUs a scene is expected to have, which depends on the screen resolution. We will discuss this in more detail in Subsection 5.3.

4.3. Decode

Next, we launch a kernel with 64 threads per block (two warps) for each queued MCU. Since the byte offset of each MCU is stored in the indexing table, all MCUs can be decoded independently and in parallel. Each block retrieves an MCU from the decoding queue, reads its byte offset in the compressed stream from the indexing table, and loads the subsequent 384 bytes into local memory to optimize access during decoding. The next step, and the main performance bottleneck during MCU decoding, is the Huffman decoding of AC coefficients, which must be executed sequentially. However, subsequent steps, including dequantization, inverse discrete cosine transform (IDCT), and color space conversion, are highly parallelizable and well-suited for efficient GPU computation. The decoded RGB values are stored in the texture block cache by first acquiring a free 16×16 block from the cache pool, writing the texels into it, and updating the corresponding hash map entry with a handle to the block.

4.4. Resolve

This pass launches one thread per pixel, converting the G-Buffer's uv-coordinate and texture index to the decoded texels. Using the MCU ID, we query the location of the decoded 16×16 block of texels from the cache via a hash map lookup, and retrieve the texel that corresponds to the uv index.

Linear interpolation is supported by fetching the four adjacent texels and weighting them according to the G-Buffer's uv-coordinate. A potential issue may be that adjacent texels could belong to different MCUs, which may or may not be decoded. In practice we found that in the vast majority of cases, adjacent MCUs are typically also visible and decoded. If not, we clamp and duplicate the bordering texel value with little to no observable artifacts. Figure 6c shows an example of a scene where the required neighboring MCUs are not decoded. For resolution of 1920×1080 with 2.1 million pixels, at most 5000 pixels (0.24%) are incorrectly filtered this way, and less than 1k MCUs are missing. While our scenes have an advantage in that the spatial locality translates directly to neighboring pixels in the texture, this is also true for most scenarios. Alternatively, we can also mark all neighboring MCUs to be decoded in the Mark step, resulting in perfect filtering with only minimal overhead of about 0.05 ms in the marking process. This can be achieved by first identifying, at the warp level, which threads refer to the same MCU key and allowing only a single representative thread to process each unique key. That thread then determines the required neighboring MCUs for linear filtering and adds them to the decode queue, ensuring that each MCU is enqueued only once while avoiding redundant work across threads. For trilinear interpolation, we can no longer clamp the filtering to a single MCU and need to decode all required neighbors. This increases the number of MCUs that need to be decoded by 20-25%.

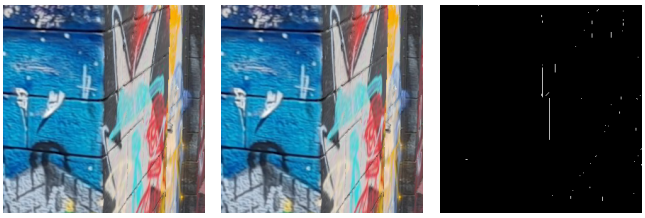
In its current state, our method does not perform well with anisotropic filtering [Mal25]. For anisotropic filtering, the texels in a higher mipmap level are averaged based on the gradient of the distortion in the UV space. Depending on the level of anisotropy, we need to decode an MCU that is up to three levels higher. Since this increases the number of texels by the power of two, each texel maps to an 8×8 region of the higher mipmap. Thus, even with perfectly aligned pixels, we would have to decode up to four times more MCUs. However, similar to modern neural methods, we can apply stochastic texture filtering [PWSF24] and collaborative texture filtering [AEPW25] that allows for filtering after texturing, requiring no additional MCUs to be decoded.

4.5. Update Cache

After the frame is rendered, we evict all MCUs from the cache that were not visible, i.e., MCUs whose visibility flag inside the cache was not set to true during the mark pass. The corresponding 16×16 blocks of decoded texels are returned to the pool for future allocations by the mark pass. The visibility flag of all retained MCUs is set to false, making them eligible for eviction in the next frame.

4.6. Multiple Textures and Mipmapping

Multiple textures are supported by combining the MCU ID with a texture ID into a single key for caching throughout the pipeline.



(a) *No Error* **(b)** *Filtering Error* **(c)** *Error Map*

Figure 6: *Comparison of filtering artifacts. (a) The clean reference image. (b) The image exhibiting artifacts due to missing MCUs for filtering. (c) An error map highlighting the pixel differences in white.*

Mipmapping is supported by creating seven downscaled versions of a texture and storing them as additional textures alongside the originals. The mipmap level is then also included in the cache key.

In our implementation we use 32 bit cache keys made of 16 bit for the MCU ID (supporting 4k textures with up to 256×256 MCUs), 13 bit for the texture ID (up to 8k unique textures), and 3 bit for the mipmap level (up to 8 levels, including original).

5. Implementation Details

This section provides additional details that are relevant for performance, overhead-reduction, and implementation.

5.1. Indexing Table

Random access to an MCU's data requires an indexing table mapping MCU indices to their corresponding byte offsets. This auxiliary structure adds memory overhead compared to standard JPEG. To minimize the overhead, we store offsets as follows: the byte offset of every ninth MCU is stored absolutely in a 32-bit integer, while the offsets of the following eight MCUs are stored as 16-bit values relative to that absolute offset.

5.2. Warp-Level Parallel Huffman Decoding

Decoding AC coefficients is a major bottleneck of the MCU decoding stage. Each AC is Huffman-coded with varying bit length, which requires us to sequentially decode up to $6 \cdot 63 = 378$ components per MCU in a single-threaded fashion. For each AC, we then need an additional inner loop in order to match the incoming bits with codes of the Huffman table.

We optimize this inner loop as follows: First, we prefetch the next 16 bit of the compressed bit stream. We then use all 32 threads of the warp to compare to 32 Huffman codes at a time. Since each Huffman code may have a different bit length, the threads trim the prefetched bits as needed. During each iteration, all threads run a ballot to communicate whether they found a match. If a match is found, the winning thread broadcasts the results to the others.

5.3. Memory Overhead

Enabling efficient random access requires adaptations of the JPEG format that increase the memory usage. This overhead consists of following components:

- **Indexing table**: $\frac{32+16 \cdot 8}{9} = 17.777$ bit per MCU. The overhead of the indexing table is a result of storing the offset to every ninth MCU as an absolute 32 bit value, and the offsets to the following 8 MCUs using 16 bit values relative to the absolute offset.
- **DC Coefficients**: $3 \cdot 12 = 36$ bit per MCU. Each MCU has 6 DC coefficients – 4 for luminance and 2 for chroma. JPEG encodes the DCs of the Y, Cr and Cb channels as the difference to the previous value of the respective channel, resulting in a sequential dependency to prior MCUs that we need to break to enable random access. We therefore store the DCs of Cr, Cb and the first DC of the Y channel in absolute terms as 12 bit values. The remaining 3 DCs of the Y component remain difference and Huffman coded, and thus do not add overhead.

This gives us an upper bound of 53.777 bit per MCU, or $\frac{53.777}{16 \cdot 16} = 0.21$ bit per pixel. However, in practice the overhead is closer to 0.17 bit per pixel since we need to subtract the number of bits that the original JPEG allocated for the re-encoded DC components to obtain the actual overhead.

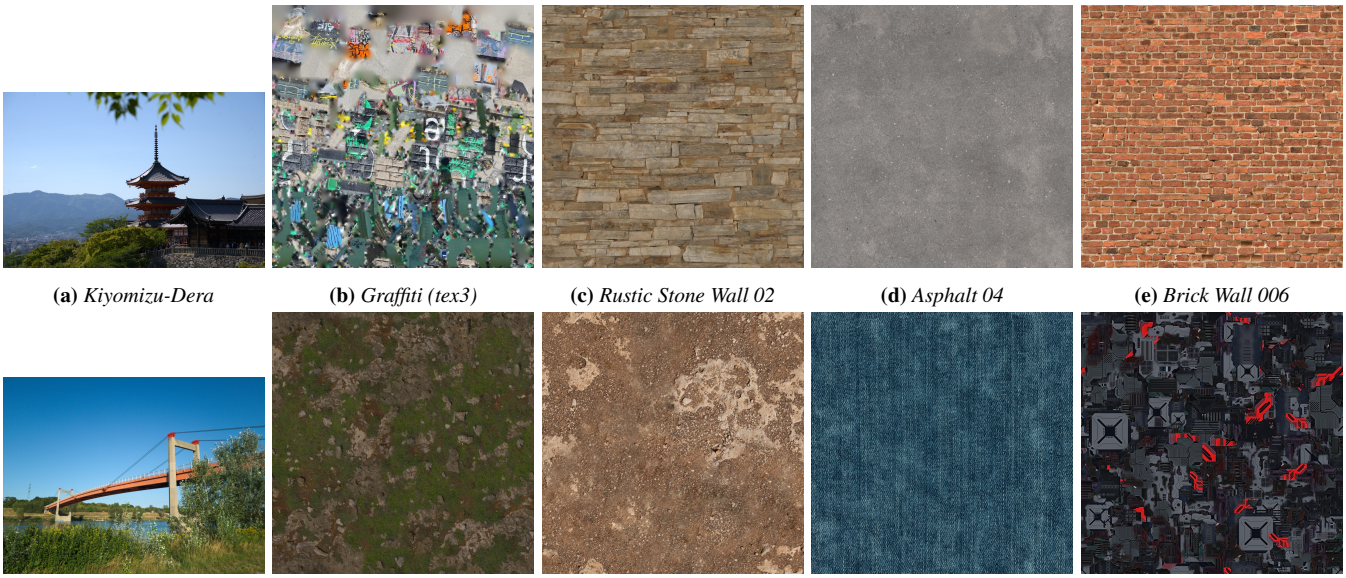
In addition, we need some resources to handle the caching of the decoded textures. This overhead is static and does not depend on the number of textures, but higher screen resolutions generally require a larger cache, as there will be more MCUs visible. The cache's memory requirements are mainly governed by the hash map capacity and the pool of decoded texture blocks. The hash map needs 8 bytes per element: 4 for the key and 4 for the index into the pool of decoded blocks. Due to the relatively low element size, the size of the hash map was set to a large static capacity of 1 million elements, resulting in 8 MB. For each decoded/visible MCU, we then need $16 \cdot 16 \cdot 4 bytes = 1 KiB$ of storage in the pool. In worst-case scenarios without mipmapping, the number of visible MCUs can get up to 700k MCUs, resulting in 700 MiB pool size, but with mipmapping enabled, 25 MB to at most 50 MB are sufficient.

6. Evaluation

To motivate the use of variable-rate compression, we evaluate both quality and bitrate across a range of image compression methods, including BC, ASTC, and JPEG, as well as newer formats such as AVIF and JPEG XL. We also include the Neural Texture Compression (NTC) method by Vaidyanathan et al. [VSW*23]. We then evaluate their feasibility for real-time rendering by measuring the performance of our JPEG-based rendering pipeline in 3D scenes.

6.1. Quality

This section evaluates the trade-off between quality and size of various compression algorithms, allowing us to estimate the potential gains of using variable-rate compression in real-time-rendering. Although we currently only support JPEG with 4:2:0 chroma subsampling, we also compare with JPEG XL and AVIF as our work is intended to function as a stepping stone towards these more sophisticated formats.



(a) *Kiyomizu-Dera* **(b)** *Graffiti (tex3)* **(c)** *Rustic Stone Wall 02* **(d)** *Asphalt 04* **(e)** *Brick Wall 006*

(f) *Bridge* **(g)** *Coast Sand* **(h)** *Coral Gravel* **(i)** *Denim Fabric 02* **(j)** *Chip005*

Figure 7: *Overview of our test textures: (a,f) Raw photos. (b) One of twenty 4k textures created by RealityScan during mesh construction. A part of the test scene "Graffiti". (rest) 4k textures from https://ambientcg.com/ and https://polyhaven.com/.*

The test data (Figure 7) comprises a set of lossless compressed textures from https://ambientcg.com/ and https://polyhaven.com/; photos in raw format (to avoid a-priori compression artifacts); and a texture of a photogrammetry-based 3D model that was created with RealityScan.

For encoding BC and ASTC, we use the NVIDIA texture tools exporter with compression quality set to "production". For JPEG, we use two encoders: libjpeg_turbo and Google's JPEGLI. The former is a faster variation of libjpeg, while the latter also improves the quality of the constructed JPEG images. In both cases we use 4:2:0 chroma subsampling, since this is the configuration we support in our renderer. JPEG XL was encoded with Python's Pillow library. For AVIF, we use ffmpeg using libaom-av1 and for NTC, the command line tool provided at https://github.com/NVIDIA-RTX/RTXNTC. We used version v0.5.0 BETA, as the newer version v0.9.1 BETA would generate corrupted files for some of our textures. However, judging from the release notes, the newer version should only improve rendering performance and actually reduce the image quality slightly as a tradeoff.

The compression performance of Google's JPEG encoder "JPEGLI", plus 0.17 bit memory overhead, is representative of our method, and indicated by a dashed red line in Table 2 and Table A2 (Appendix).

The quality of the encoded images is evaluated with the metrics PSNR, FLIP [ANA*20], SSIM [WBSS04], and LPIPS [ZIE*18]. PSNR scores are commonly used but also known to be flawed as they are not modeled after human perception. The others provide various solutions for scoring image quality based on perception, with FLIP being the most recent approach.

Across the scores shown in Table 2 and Table A2 (Appendix), we can observe that BC1 typically takes the last place, followed by ASTC. AVIF dominates the PSNR, SSIM and LPIPS metrics, while JPEG XL leads with FLIP. JPEG lies in between, with Jpegli performing slightly better than libjpeg_turbo. We can also observe several unintuitive results and disagreements between metrics. For example, LPIPS is the only metric that consistently scores Jpegli-encoded images worse than libjpeg_turbo-encoded images; and SSIM most often scores ASTC as good as or sometimes better than JPEG.

Overall, JPEG performs far better than BC, but surprisingly only modestly better than ASTC. JPEG XL and AVIF, on the other hand, are clearly superior, regularly encoding images in half the size for the same quality according to PSNR and FLIP, enabling sub-1bit-per-texel compression rates at sufficient quality. NTC performs better than JPEG except for the FLIP error metric, but worse than AVIF and JPEG XL. We note, however, that NTC is optimized for multi-channel textures beyond just RGB, and therefore does not fully demonstrate its strengths in scenarios limited to RGB textures. Table 1 and Table A1 (Appendix) provide impressions of the artifacts at a target bitrate of 0.89 bit per pixel.

6.2. Performance

Performance is evaluated in a CUDA- and OpenGL-based rendering engine. OpenGL renders a G-Buffer comprising uv-coordinates, texture-IDs and mipmap levels. CUDA decodes the required JPEG blocks and subsequently converts the G-Buffer into a textured rendering. OpenGL draw call durations are computed with timestamp queries, and CUDA kernel launch durations with CUDA events. All durations are computed as the median over 60 frames. Two test scenes (Table 3) were evaluated: The popular "Crytek Sponza" [McG17], and a custom 3D scan "Graffiti" that consists of high-resolution, non-repeating textures.

Table 1: *Artifacts of various compression algorithms at a target bitrate of 0.89 bit per pixel (bpp), or the quality level that comes closest. BC1 for comparison (always 4bpp). Each cell shows quality and bpp in the top row, and FLIP and PSNR in the bottom row. Best and worst FLIP and PSNR values in a row are highlighted, with the exception of BC1 which is not considered due to its high memory usage.*

Reference	BC1	ASTC 12 × 12	JPEG	JPEG XL	AVIF	NTC
Kiyomizu-Dera	4.00bpp	0.89bpp	Q90 0.64bpp	Q90 0.74bpp	Q85 0.84bpp	1.01bpp
	0.031 38.4	0.038 36.5	0.037 37.4	0.027 39.4	0.035 42.8	0.037 39.7
Graffiti (Tex3)	4.00bpp	0.89bpp	Q80 0.90bpp	Q80 0.77bpp	Q75 0.89bpp	1.01bpp
	0.033 38.3	0.051 33.2	0.042 37.3	0.035 37.9	0.038 41.7	0.049 34.5
Rustic Stone Wall 02	4.00bpp	0.89bpp	Q80 0.77bpp	Q80 0.74bpp	Q80 0.96bpp	1.01bpp
	0.032 41.6	0.048 37.4	0.047 37.8	0.039 38.7	0.039 40.4	0.043 39.0

Table 2: *Quality metric plots for the test data sets in Table 1. "JPEGLI + 0.17 bit" is representative of our work, which adapts JPEG but adds about 0.17 bit per pixel to enable random access.*

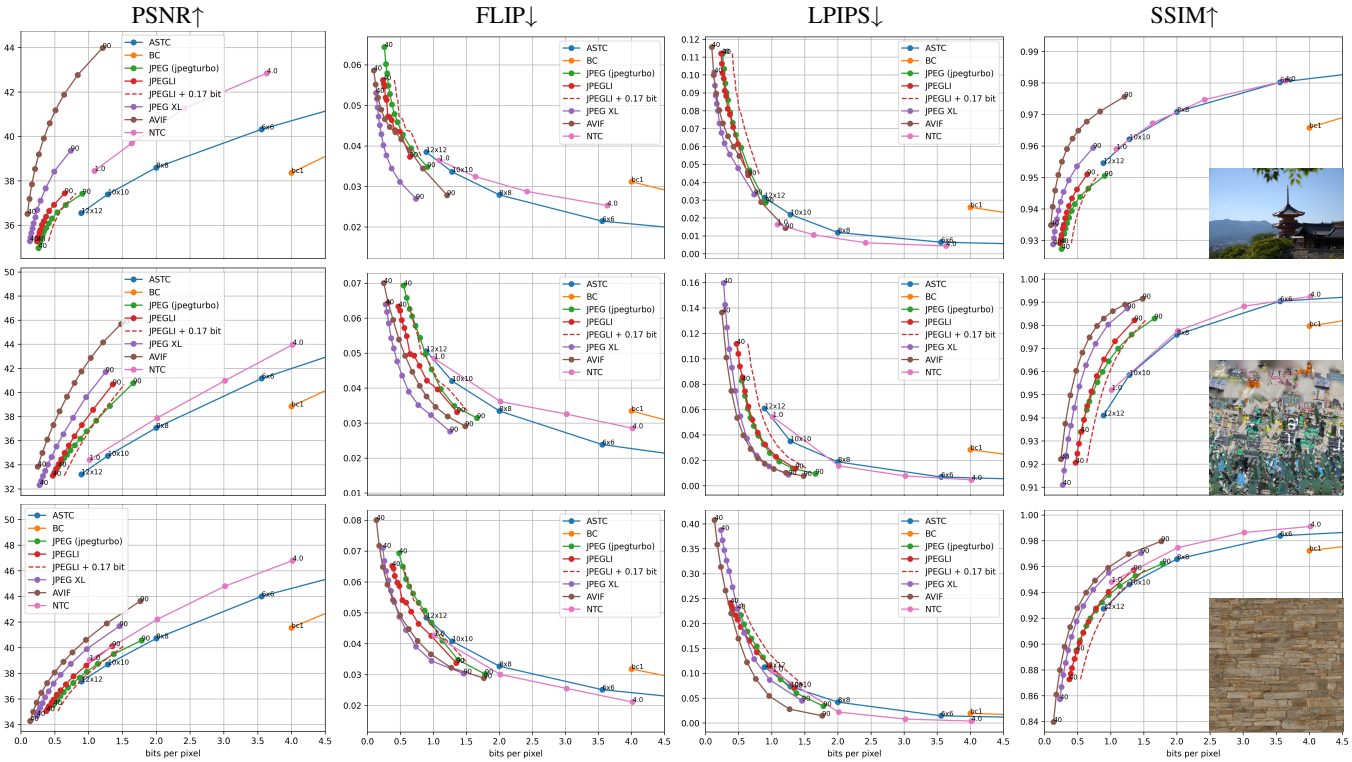
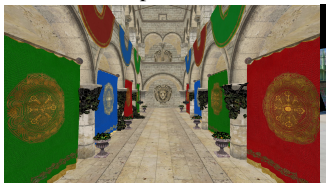
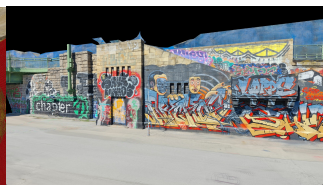


Table 3: *Test Scenes.*

Sponza	Graffiti
	
262k Triangles	966k Triangles
$92 \times 1k$ textures (96M texel)	$20 \times 4k$ textures (335M texel)
Repeated UVs (e.g. floor)	Non-Repetitive Texture Mapping

Performance was measured on a test system with an AMD Ryzen 9 7950X CPU, an NVIDIA RTX 4090 GPU, and a Valve Index HMD. Framebuffer sizes are 1920×1080 (2MP) on desktop, and 2468×2740 per eye in VR ($2 \times 6.8MP$).

6.2.1. Baseline Rendering

Table 4 reports the rendering performance across two test scenes and multiple JPEG quality levels with texture block caching disabled and linear interpolation enabled. Two key observations can be made: (1) higher JPEG quality levels significantly increase decoding time, and (2) mipmapping reduces decoding time to a fraction. The quality-dependent slowdown is explained by the number of AC coefficients: a higher quality implies more coefficients to decode. In contrast, mipmapping boosts performance by reducing the number of MCUs that are visible, as numerous small ones are replaced by fewer larger ones that better match their projected size. It also provides substantial performance improvements to the resolve stage as it increases the likelihood that adjacent pixels fetch texels from the same MCU.

Table 4: *Performance comparison. (a) Baseline (No mipmapping). (b) With mipmapping enabled. Mipmapping significantly reduces the number of MCUs and decode times.*

(a) Without Mipmapping (Baseline)

Scene	geometry	mark	decode	resolve	MCUs
Sponza Q50	0.05	0.06	0.80	0.09	81k
Sponza Q70	0.05	0.06	1.08	0.09	81k
Sponza Q90	0.05	0.06	1.90	0.09	81k
Graffiti Q80	0.12	0.12	2.63	0.19	224k

(b) With Mipmapping Enabled

Scene	geometry	mark	decode	resolve	MCUs
Sponza Q50	0.05	0.03	0.23	0.05	12k
Sponza Q70	0.05	0.03	0.26	0.05	12k
Sponza Q90	0.05	0.03	0.31	0.05	12k
Graffiti Q80	0.12	0.04	0.46	0.05	22k

6.2.2. Caching

Caching must be evaluated under motion, since rendering the same viewpoint twice reduces the number of MCUs to decode to zero. To simulate fast-paced motion, we rotate the camera by 6 degrees to the right each frame, for a full rotation every 60 frames. Since we are mainly interested in the worst-performing frame to ensure we do not suffer from stutters, we compute and report the max-of-medians: (1) We repeat the 60 rotations 100 times, i.e., we sample a total of 6000 frames, (2) We compute the median of the 100 samples of a view direction, for all view directions, (3) we compute the maximum over the 60 median values to obtain the median of the worst-performing viewpoint. The reason for not using max by itself is that frame times and kernel launches are subject to fluctuations (hardware, scheduling, OS interrupts, ...), which is why the median is typically used.

Table 5: *JPEG pipeline (mark+decode+resolve) duration under motion. Reporting the max-of-medians, i.e., the median of multiple repetitions of the worst-performing viewpoint.*

	Baseline	w. Mipmap	w. Mipmap & Cache
Sponza Q50	1.01 ms	0.35 ms	0.19 ms
Sponza Q70	1.27 ms	0.36 ms	0.20 ms
Sponza Q90	2.14 ms	0.43 ms	0.21 ms
Graffiti Q80	3.90 ms	0.59 ms	0.30 ms

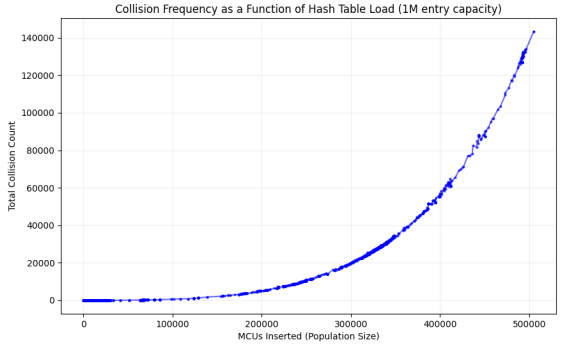


Figure 8: *Number of collisions for the number of MCUs inserted into our hash map with one million slots. If a slot was taken, each linear probe was counted as an additional collision.*

The results of the JPEG rendering pipeline in motion, with linear interpolation enabled, are shown in Table 5. From this we can conclude that the JPEG pipeline adds ≤ 0.3 ms to the frame on an NVIDIA RTX 4090, rendering into a framebuffer with a size of 1920×1080pixels.

Using a hash map with one million slots, we observed fewer than 20 collisions during the insertion of 30,000 MCUs. Even after halving the table size to 500,000 slots, collisions remained below 100, with all instances resolved in fewer than four linear probes. Figure 8 illustrates the relationship between the number of MCUs and hash collisions; while the collision rate grows exponentially, it remains negligible for the MCU counts typically found in a rendered scene. Because the collision frequency was so low, the change in table size had no measurable impact on rendering performance. We did not test performance with an even smaller hash table, as the 4 MB overhead for 500,000 slots is well within acceptable limits for most applications.

6.2.3. Linear Interpolation

Linear Interpolation between magnified texels has only a minor impact on overall rendering performance. In Sponza, the resolve kernel requires 25 µs with nearest-neighbor interpolation and 46 µs with linear interpolation. Most of this overhead arises when the interpolated texels lie in different MCUs, since each texel fetch triggers a hash map lookup into the texture block cache. When they reside in the same MCU, the result can be shared, reducing the cost. If we clamp texel fetches during linear interpolation to a single MCU, the cost is reduced to 32 µs, but this leads to artifacts along the borders of a 16×16 block of texels.

6.2.4. Virtual Reality

Figure 9b demonstrates that our approach scales well to virtual reality applications, as both viewpoints share the majority of MCUs. During rapid, unsophisticated head movements around Sponza, a minimum of 71% and an average of 86% of MCUs were shared between both eyes. In Graffiti, a minimum of 61% and an average of 90% of MCUs were shared.

Applying the methodology of Section 6.2.2 to VR – rotate by 6 degrees each frame, 60 times for a full rotation, with 100 repetitions – we obtain a max-of-medians value of 0.65ms for the JPEG rendering pipeline in Sponza Q70, compared to the 0.20ms of the non-VR benchmark. This increase is due to the 6.5 times higher amount of pixels that are processed by the mark and resolve kernels in VR, as well as the higher number of MCUs that need decoding. Due to the larger resolution, more detailed mipmap levels are needed, i.e., a larger amount of texels and corresponding MCUs are decoded.

6.2.5. A Note Regarding CUDA-OpenGL Interop

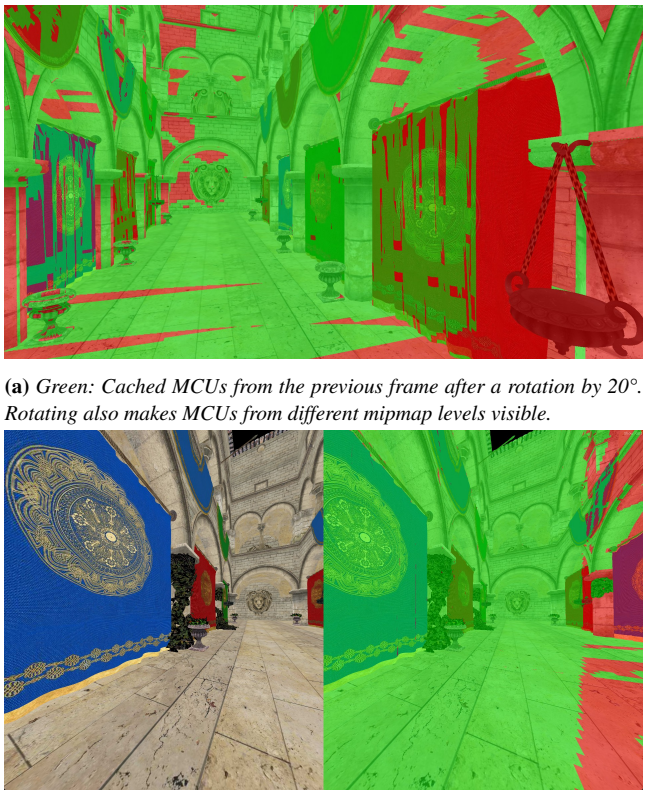
We found that the first kernel launch after switching from an OpenGL context to a CUDA is unpredictably slow, adding 0.03 to 0.3 ms to the frame. Since the *mark* kernel is launched first, this delay was attributed to it. To avoid this, we added a dummy kernel that is launched first and does nothing but still takes up to 0.3ms per frame. This interop cost is excluded from the JPEG pipeline timings but is inevitably reflected in the FPS reported in the teaser.

6.3. Memory Overhead

A calculation of the maximum memory overhead is provided in Section 5.3 (0.21 bits per texel). In practice, however, the actual overhead is lower because we counted the bits for DCs in our adaptation (12 bit) without subtracting the bits used in the original JPEG (variable). By subtracting the bits used in the original JPEG from the fixed 12-bit allocation, we obtain the following effective overheads: Sponza Q50: **0.177 bpp**; Sponza Q90: **0.154 bpp**; Graffiti Q80: **0.161 bpp**.

7. Discussion and Future Work

An unexpected finding of the quality evaluation (see Appendix A) is that despite JPEG's reputation as a highly efficient compression algorithm, it only modestly outperforms ASTC on tileable textures commonly used in games, according to several metrics. For photographs and non-tileable textures derived from photogrammetric mesh reconstruction, however, the quality gap is much larger and aligns with our expectations.



(a) *Green: Cached MCUs from the previous frame after a rotation by 20°. Rotating also makes MCUs from different mipmap levels visible.*

(b) *VR: The right eye shares most MCUs with the left. The shared frustum contains a small amount of MCUs that became visible after disocclusion.*

Figure 9: *Texture Block Cache: Only MCUs that were not visible in the previous frame or other VR viewpoints need decoding (red).*

Recent advances like JPEG XL and AVIF demonstrate a distinctively superior trade-off between quality and size, thus we consider this JPEG-based work the first step towards adapting these latest state-of-the-art algorithms for real-time rendering. JPEG XL [SAV*25], in particular, extends many concepts of JPEG and appears well-suited for this purpose. Its advances include a more appropriate color space, predictive coding of coefficient counts, variable block sizes, and several additional features that enhance efficiency. However, some of these techniques, such as predictive coding of coefficient counts, may hinder random access because they depend on data from multiple blocks. In such cases, real-time adaptations may need to revert to alternative options, like JPEG's end-of-block signaling in the case of coefficients.

As an alternative to mipmapping, future work could explore reconstructing lower-resolution versions of a JPEG by decoding only a subset of the AC coefficients, or exclusively the DC coefficient. This approach would allow significant reductions in computational effort at reduced resolutions without the need for explicitly stored mip levels.

In addition, due to JPEG's limitations, we currently only deal with three-channel diffuse textures without alpha. In a follow-up

work, we are interested in looking at more modern standards such as JPEG XL [AvAB*19], which promises far better compression, supports near unlimited channels and can deal with transparency.

Virtual Texture Mapping [MG08] could benefit f rom higher compression rates, enabling faster streaming from disk to GPU. Due to minimum page sizes for disk access, latency will not be lower but larger blocks of pixels could be fetched at once.

Neural textures may benefit from a texture block cache similar to ours, which could enable the use of more computationally intensive neural methods while maintaining real-time rendering performance.

8. Conclusion

In this paper we have shown that variable-rate image compression formats such as JPEG are a viable choice for efficient real-time rendering, given a deferred rendering pipeline, a compact indexing table, mipmapping, and the use of a cache. The deferred pipeline allows us to identify the visible JPEG blocks that need decoding; the indexing table enables random access to the compressed blocks; and mipmaps and cache significantly reduce the number of blocks we need to decode each frame. Although auxiliary data structures such as the indexing table add an overhead of about 0.17 bit per pixel, the resulting size is still well below the ASTC and BC formats.

At this stage, the reductions in file size of JPEG (+required overhead for random access) may not yet justify the additional computational and implementation effort in practice, but we are confident that with further development towards GPU-friendly adaptations of more recent advances like JPEG XL and AVIF, variable-rate textures will transform from a merely viable to a highly attractive choice for real-time rendering.

Source code and data sets are available at: https://github.com/elias1518693/jpeg_textures

9. Acknowledgements

The authors wish to thank following data set providers: Crytek Sponza 3D Model by Frank Meinl at Crytek, based on an older model by Marko Dabrović; Photograph of the Kiyomizu-Dera temple by Thomas Rausch; Textures provided by https://ambientcg.com/ and https://polyhaven.com/.

This research has been funded by WWTF project *ICT22-055 - Instant Visualization and Interaction for Large Point Clouds* and WWTF project *ICT25-084 - Instant Visualization and Editing of Arbitrarily Large 3D Data Sets*. Open Access funding provided by Technische Universitat Wien/KEMÖ.

References

[AEPW25] Akenine-Möller, Tomas, Ebelin, Pontus, Pharr, Matt, and Wronski, Bartlomiej. "Collaborative Texture Filtering". *High-Performance Graphics - Symposium Papers* (2025). DOI: 10.2312/HPG.20251174. URL: https://diglib.eg.org/handle/10.2312/hpg20251174 6.

[ANA*20] Andersson, Pontus, Nilsson, Jim, Akenine-Möller, Tomas, et al. "FLIP: A Difference Evaluator for Alternating Images." *Proc. ACM Comput. Graph. Interact. Tech.* 3.2 (2020), 15–1 8.

[AvAB*19] Alakuijala, Jyrki, van Asseldonk, Ruud, Boukortt, Sami, et al. "JPEG XL next-generation image compression architecture and coding tools". 2019. URL: https://www.spiedigitallibrary.org/conference-proceedings-of-spie/11137/111370K/JPEG-XL-next-generation-image-compression-architecture-and-coding-tools/10.1117/12.2529237.full 2, 12.

[Bin25] Binomial LLC. *Basis Universal GPU Texture Codec*. https://github.com/BinomialLLC/basis_universal. Accessed: 2025-12-29. 2025 3.

[CES00] Christopoulos, C. A., Ebrahimi, T., and Skodras, A. N. "JPEG2000: the new still picture compression standard". *Proceedings of the 2000 ACM Workshops on Multimedia*. MULTIMEDIA '00. Los Angeles, California, USA: Association for Computing Machinery, 2000, 45–49. ISBN: 1581133111. DOI: 10.1145/357744.357757 2.

[CL02] Chen, C.-H. and Lee, C.-Y. "A JPEG-like texture compression with adaptive quantization for 3D graphics application". *The Visual Computer* 18 (2002), 29–40. URL: https://api.semanticscholar.org/CorpusID:8072089 2.

[Dud09] Duda, Jarek. "Asymmetric numeral systems". *arXiv preprint arXiv:0902.0271* (2009) 3.

[FH24] Fujieda, Shin and Harada, Takahiro. "Neural Texture Block Compression". (2024). DOI: 10.2312/MAM.20241178 2.

[FHL*24] Farhadzadeh, Farzad, Hou, Qiqi, Le, Hoang, et al. *Neural Graphics Texture Compression Supporting Random Access*. 2024. arXiv: 2407.00021 [cs.CV]. URL: https://arxiv.org/abs/2407.00021 3.

[FP25] Fichet, Alban and Peters, Christoph. "Compression of Spectral Images Using Spectral JPEG XL". *Journal of Computer Graphics Techniques Vol* 14.1 (2025) 3.

[HPLV12] Hollemeersch, Charles-Frederik, Pieters, Bart, Lambert, Peter, and Van de Walle, Rik. "A new approach to combine texture compression and filtering". *Vis. Comput.* 28.4 (Apr. 2012), 371–385. ISSN: 0178-2789. DOI: 10.1007/s00371-011-0621-8 2.

[Huf52] Huffman, David A. "A Method for the Construction of Minimum-Redundancy Codes". *Proceedings of the IRE* 40.9 (1952), 1098–1101. DOI: 10.1109/JRPROC.1952.273898 2.

[INH99] Iourcha, Konstantine, Nayak, Krishna S., and Hong, Zhou. "System and method for fixed-rate block-based image compression with inferred pixel values". Patent US5956431. Incorporated, S3. Sept. 1999. URL: https://patents.google.com/patent/US5956431 2.

[KKLD23] Kerbl, Bernhard, Kopanas, Georgios, Leimkühler, Thomas, and Drettakis, George. "3D Gaussian Splatting for Real-Time Radiance Field Rendering". *ACM Transactions on Graphics* 42.4 (July 2023). URL: https://repo-sam.inria.fr/fungraph/3d-gaussian-splatting/ 3.

[KPM16] Krajcevski, Pavel, Pratapa, Srihari, and Manocha, Dinesh. "GST: GPU-decodable supercompressed textures". 35.6 (Dec. 2016). ISSN: 0730-0301. DOI: 10.1145/2980179.2982439. URL: https://doi.org/10.1145/2980179.2982439 3.

[LA25] Laurent, Belcour and Anis, Benyoub. *Hardware Accelerated Neural Block Texture Compression with Cooperative Vectors*. 2025. arXiv: 2506.06040 [cs.GR] 3.

[LDN04] Lefebvre, Sylvain, Darbon, Jérome, and Neyret, Fabrice. "Unified texture management for arbitrary meshes". PhD thesis. INRIA, 2004 3.

[LJP*23] Luo, Yuzhe, Jin, Xiaogang, Pan, Zherong, et al. "Texture Atlas Compression Based on Repeated Content Removal". *SIGGRAPH Asia 2023 Conference Papers*. SA '23. Sydney, NSW, Australia: Association for Computing Machinery, 2023. ISBN: 9798400703157. DOI: 10.1145/3610548.3618150 3.

[Mal25] MALLING, PEMA. *Mipmap selection in too much detail*. Accessed: 2026-01-11. May 2025. URL: https://pema.dev/2025/05/09/mipmaps-too-much-detail/ 6.

[May10] MAYER, ALBERT JULIAN. "Virtual texturing". PhD thesis. TU Wien, 2010 3.

[McG17] MCGUIRE, MORGAN. *Computer Graphics Archive*. July 2017. URL: https://casual-effects.com/data 8.

[MG08] MITTRING, MARTIN and GMBH, CRYTEK. "Advanced virtual texture topics". *ACM SIGGRAPH 2008 Games*. SIGGRAPH '08. Los Angeles, California: Association for Computing Machinery, 2008, 23–51. ISBN: 9781450378499. DOI: 10.1145/1404435.1404438. URL: https://doi.org/10.1145/1404435.1404438 3, 12.

[NLP*12] NYSTAD, J., LASSEN, A., POMIANOWSKI, A., et al. "Adaptive scalable texture compression". *Proceedings of the Fourth ACM SIGGRAPH / Eurographics Conference on High-Performance Graphics*. EGGH-HPG'12. Paris, France: Eurographics Association, 2012, 105–114. ISBN: 9783905674415 2.

[OBGB11] OLANO, MARC, BAKER, DAN, GRIFFIN, WESLEY, and BARCZAK, JOSHUA. "Variable Bit Rate GPU Texture Decompression". *Computer Graphics Forum* (2011). ISSN: 1467-8659. DOI: 10.1111/j.1467-8659.2011.01989.x 3.

[PWSF24] PHARR, MATT, WRONSKI, BARTLOMIEJ, SALVI, MARCO, and FAJARDO, MARCOS. "Filtering After Shading With Stochastic Texture Filtering". *Proc. ACM Comput. Graph. Interact. Tech.* 7.1 (May 2024). DOI: 10.1145/3651293. URL: https://doi.org/10.1145/3651293 6.

[RA08] RADZISZEWSKI, MICHAL and ALDA, WITOLD. "Optimization of frequency filtering in random access JPEG library". *Computer Science* 9 (2008), 109–120 2.

[RAD25] RAD GAME TOOLS. *Oodle Texture Compression*. https://www.radgametools.com/oodletexture.htm. Accessed: 2026-01-07. 2025. URL: https://www.radgametools.com/oodletexture.htm 3.

[SA05] STRÖM, JACOB and AKENINE-MÖLLER, TOMAS. "iPACKMAN: high-quality, low-complexity texture compression for mobile phones". *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*. HWWS '05. Los Angeles, California: Association for Computing Machinery, 2005, 63–70. ISBN: 1595930868. DOI: 10.1145/1071866.1071877 2.

[SAV*25] SNEYERS, JON, ALAKUIJALA, JYRKI, VERSARI, LUCA, et al. *The JPEG XL Image Coding System: History, Features, Coding Tools, Design Rationale, and Future*. 2025. arXiv: 2506.05987 [cs.MM] 2, 11.

[STS*21] SCHUSTER, KERSTEN, TRETTNER, PHILIP, SCHMITZ, PATRIC, et al. "Compression and Rendering of Textured Point Clouds via Sparse Coding". *High-Performance Graphics - Symposium Papers*. Ed. by BINDER, NIKOLAUS and RITSCHEL, TOBIAS. The Eurographics Association, 2021. ISBN: 978-3-03868-156-4. DOI: 10.2312/hpg.20211284 3.

[SW11] STROM, JACOB and WENNERSTEN, PER. "Lossless compression of already compressed textures". *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*. 2011, 177–182 3.

[Van06] VAN WAVEREN, JMP. "Real-time texture streaming & decompression". *ID Software, Inc., Shreveport, LO, USA, Tech. Rep* (2006) 3.

[VSW*23] VAIDYANATHAN, KARTHIK, SALVI, MARCO, WRONSKI, BARTLOMIEJ, et al. "Random-Access Neural Compression of Material Textures". *Proceedings of SIGGRAPH*. 2023 3, 7.

[vWH10] Van WAVEREN, J. M. P. and HART, EVAN. "Using Virtual Texturing to Handle Massive Texture Data". *GPU Technology Conference (GTC 2010)*. Presentation, Session 2152. San Jose, CA, USA, Sept. 2010. URL: https://www.nvidia.com/content/GTC-2010/pdfs/2152_GTC2010.pdf 3.

[Wal92] WALLACE, G.K. "The JPEG still picture compression standard". *IEEE Transactions on Consumer Electronics* 38.1 (1992), xviii–xxxiv. DOI: 10.1109/30.125072 1, 2.

[WBSS04] WANG, ZHOU, BOVIK, A.C., SHEIKH, H.R., and SIMONCELLI, E.P. "Image quality assessment: from error visibility to structural similarity". *IEEE Transactions on Image Processing* 13.4 (2004), 600–612. DOI: 10.1109/TIP.2003.819861 8.

[WdOHN24] WEINREICH, CLÉMENT, de OLIVEIRA, LOUIS, HOUDARD, ANTOINE, and NADER, GEORGES. *Real-Time Neural Materials using Block-Compressed Features*. 2024. arXiv: 2311.16121 [cs.CV]. URL: https://arxiv.org/abs/2311.16121 3.

[ZGX*24] ZHANG, XINJIE, GE, XINGTONG, XU, TONGDA, et al. *GaussianImage: 1000 FPS Image Representation and Compression by 2D Gaussian Splatting*. 2024. arXiv: 2403.08551 [eess.IV]. URL: https://arxiv.org/abs/2403.08551 3.

[ZIE*18] ZHANG, RICHARD, ISOLA, PHILLIP, EFROS, ALEXEI A, et al. "The Unreasonable Effectiveness of Deep Features as a Perceptual Metric". *CVPR*. 2018 8.

[ZLK*25] ZHANG, YUNXIANG, LI, BINGXUAN, KUZNETSOV, ALEXANDR, et al. "Image-GS: Content-Adaptive Image Representation via 2D Gaussians". *Proceedings of the Special Interest Group on Computer Graphics and Interactive Techniques Conference Conference Papers*. 2025, 1–11 3.