TU WIEN Informatics

Real-Time Rendering mit JPEG komprimierten Texturen

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Visual Computing

eingereicht von

Elias Imre Kristmann, BSc
Matrikelnummer 01519693

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ.Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer
Mitwirkung: Dipl.-Ing. Dr.techn. Markus Schütz, BSc.

Wien, 1. September 2025

Elias Imre Kristmann	Michael Wimmer

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

TU WIEN Informatics

Real-Time Rendering with JPEG-Compressed Textures

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Visual Computing

by

Elias Imre Kristmann, BSc
Registration Number 01519693

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer
Assistance: Dipl.-Ing. Dr.techn. Markus Schütz, BSc.

Vienna, September 1, 2025

Elias Imre Kristmann	Michael Wimmer

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Elias Imre Kristmann, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.
Ich erkläre weiters, dass ich mich generativer KI-Tools lediglich als Hilfsmittel bedient habe und in der vorliegenden Arbeit mein gestalterischer Einfluss überwiegt. Im Anhang „Übersicht verwendeter Hilfsmittel“ habe ich alle generativen KI-Tools gelistet, die verwendet wurden, und angegeben, wo und wie sie verwendet wurden. Für Textpassagen, die ohne substantielle Änderungen übernommen wurden, haben ich jeweils die von mir formulierten Eingaben (Prompts) und die verwendete IT- Anwendung mit ihrem Produktnamen und Versionsnummer/Datum angegeben.

Wien, 1. September 2025

Elias Imre Kristmann

Danksagung

An dieser Stelle möchte ich mich bei meiner Familie und Freund*innen bedanken, welche mich stets unterstützt und motiviert haben. Einen besonderen Dank gilt meiner Partnerin Irina, ohne unsere täglichen Spaziergänge wäre ich nie ans Ziel gekommen. Außerdem danke ich Markus, von dem ich dieses Jahr so viel lernen durfte, für die unglaublich gute und freundschaftliche Betreuung sowie die großartige Unterstützung. Zuletzt bedanke ich mich bei Michael Wimmer für die Flexibilität und das wertvolle Feedback, die es mir ermöglicht haben, diese Arbeit bestmöglich abzuschließen.

Acknowledgements

I would like to thank my friends and family for always supporting and motivating me during my studies. A special thanks goes to my partner Irina; without our daily walks, I would never have reached this goal. I am also very grateful to Markus, from whom I was able to learn so much this year, for his excellent and friendly supervision as well as his great support. Finally, I would also like to thank Michael Wimmer for his flexibility and the valuable feedback, which enabled me to finish my studies in the best possible way.

Kurzfassung

Obwohl mit variabler Bitrate komprimierte Bildformate wie JPEG weit verbreitet sind und eine effiziente Komprimierung ermöglichen, spielen sie im Bereich des Echtzeit-Renderings bislang kaum eine Rolle. Der Hauptgrund dafür sind spezielle Anforderungen wie direkter Zugriff auf einzelne Texel. In dieser Arbeit untersuchen wir, ob und wie sich JPEG als Format für Texture Kompression mit variabler Bitrate auf modernen GPUs einsetzen lässt – und wie es im Vergleich zu etablierten, GPU-optimierten Verfahren mit konstanter Bitrate wie BC1 und ASTC abschneidet.

Unser Ansatz basiert auf einer Deferred-Rendering-Pipeline und einer Pointer-Liste auf separat kodierten JPEG-Blöcke. So können wir gezielt nur die Blöcke identifizieren und dekodieren, die für das aktuelle Frame tatsächlich benötigt werden – und anschließend die entsprechenden Pixel im Framebuffer einfärben. Trotz eines Overheads von ca. 0,5 Bit pro Texel liefert JPEG deutlich bessere Qualität und Kompressionsraten als BC1 und kann sogar mit ASTC mithalten. Beim Dekodieren der Textur erreichen wir zwar nicht ganz das Niveau von klassischen GPU-Codecs, brauchen aber auf einer RTX 4090 trotzdem weniger als eine Millisekunde pro Frame. Damit zeigen wir, dass mit variabler Bitrate kodierte Formate auch im Kontext von Deferred Rendering oder Visibility Buffers durchaus praktikabel sind.

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 thesis, 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 and a list of pointers to individually encoded JPEG blocks, 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 0.5 bit per texel that we require for our approach, JPEG maintains significantly better quality and compression rates compared to BC1, and is able to compete with ASTC. Although we can not fully compete performance-wise, decoding the required texels of a JPEG texture requires less than 1ms per frame on an RTX 4090, thus demonstrating that variable-rate encoded image formats are feasible for rendering pipelines that are based on deferred rendering or visibility buffers.

Contents

CHAPTER 1

Introduction

Although JPEG [Wal92] is over 30 years old, it remains one of the most widely adopted image compression formats. Its compression efficiency—especially in terms of perceptual quality versus file size—has ensured its continued relevance across countless applications. Despite this success, JPEG has historically been considered unsuitable for real-time graphics applications. The prevailing view in the computer graphics community has been that "it simply does not work" for this purpose, primarily due to limitations such as poor random access due to sequential encoding, lack of parallel decode support, and variable-rate block coding structures that do not map well to GPU architectures.

At the same time, modern texture compression formats such as BCn [INH99] and ASTC [STS+21] have evolved specifically to meet the needs of real-time rendering—trading variable compression rates and perceptual flexibility for tightly controlled access patterns and hardware decode support.

In this thesis, we challenge the long-standing assumption that JPEG is inherently incompatible with real-time graphics. We present a novel method that overcomes the key limitations of JPEG, enabling it to be decoded in real-time directly on the GPU. Our goal is not to suggest that JPEG surpasses traditional texture compression methods, but to encourage the exploration of new GPU-friendly variable-rate compression strategies. By revisiting and rethinking a format once considered unfit for graphics, we hope to lay the groundwork for further innovation in texture compression, particularly in scenarios where memory efficiency and perceptual quality must be balanced dynamically.

1.1 Problem Statement

The ever-increasing quality of photorealistic rendering is inherently accompanied by a significant rise in texture data volume, which in turn places growing demands on storage and memory bandwidth. However, Video Random Access Memory (VRAM) remains

both costly and limited, particularly for users on lower-end or ageing hardware. As illustrated by a hardware survey in Table 1.1, more than 50% of Steam [1] users, one of the biggest online marketplaces for video games, currently operate with 8 GB of VRAM or less, underscoring a widening disparity between modern software requirements and the capabilities of mainstream hardware. Notably, the largest growth in market share is observed among devices equipped with as little as 512 MB of VRAM, indicating that increasing VRAM capacity is not always a viable solution, whether due to economic, technical, or platform constraints.

A promising solution lies in compressing textures, reducing their size, and decoding only the necessary portions in real-time when needed. This approach necessitates that compressed textures support random access–the ability to retrieve individual texels without decoding the entire texture. GPU-friendly compression algorithms address this challenge by encoding texels in blocks with a uniform number of bits. These so-called fixed-rate algorithms enable efficient mapping of texel indices to memory locations, allowing real-time decoding. However, this approach often results in suboptimal utilisation of the available storage, as Figure 1.1 demonstrates.

Variable-rate image formats, such as JPEG, achieve significantly higher compression by allocating fewer bits to low-detail regions, thereby avoiding inefficient uniform bit distribution. Despite this strength, such formats are generally regarded as unsuitable for real-time rendering, primarily due to several technical limitations:

- Inefficient Texel Access: Variable bit rates mean that texel indices or UV coordinates cannot be easily mapped to memory locations without creating additional indexing tables, which increases memory overhead.
- Sequential Decoding: JPEG relies on Huffman coding [Huf52], requiring sequential decoding and hindering parallelism, which is not GPU-friendly.
- Differential coding: components within JPEG are encoded as differences relative to preceding values, introducing data dependencies that further obstruct random-access.

If these constraints can be effectively addressed, it would enable the development of novel texture compression schemes that leverage the compression efficiency of variable-rate formats while preserving the random-access capability and decoding performance necessary for real-time GPU rendering.

1.2 Aim of the Work

This thesis explores the potential of variable-rate compressed image formats for real-time rendering on modern hardware. The primary objective is to investigate the feasibility of

[1] https://store.steampowered.com/

VRAM Size	Market Share (%)	Change (%)
512 MB	5.86	+0.37
1 GB	3.94	-0.02
2 GB	4.99	-0.04
3 GB	1.08	-0.05
4 GB	7.11	-0.16
6 GB	11.48	-0.04
8 GB	33.67	-0.58
10 GB	2.67	+0.01
11 GB	1.23	-0.03
12 GB	18.83	+0.25
16 GB	5.90	+0.34
24 GB	2.40	-0.06
Other	0.86	+0.04

Table 1.1: Market Share and Change by VRAM Memory Size of Steam Users Mai 2025 [Val25]

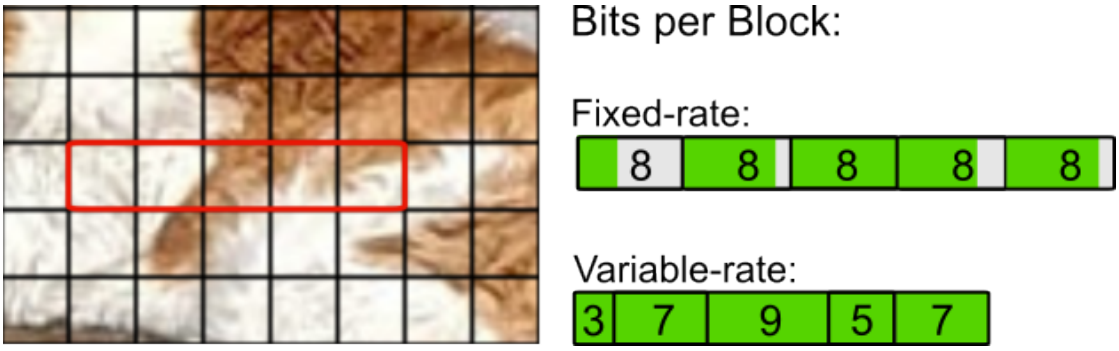


Figure 1.1: Comparison of fixed-rate and variable-rate storage for image compression. The green colour indicates the information density of a block, while the grey parts demonstrate excess storage usage that does not improve quality. Fixed-rate encoding uses 8 bits uniformly for each block, leading to inefficiencies such as unused storage (e.g., the first block) and loss of information when the required storage exceeds 8 bits (e.g., the third block). Variable-rate encoding adjusts the bits per block based on content complexity, achieving higher compression and better image quality by minimising storage waste and preserving details in complex regions.

variable-rate texture compression methods, focusing on decoding efficiency and memory consumption to highlight their strengths and trade-offs. JPEG was specifically chosen because it is one of, if not the most popular, variable-rate-compressed image format. By demonstrating the feasibility of a format that was conceived without GPU-friendly optimisations in mind, this work aims to challenge conventional assumptions about the viability of variable-rate compression for GPU textures.

Using CUDA, the method will be implemented and integrated into a deferred rendering framework. This framework will be extended to support JPEG textures and enable partial on-the-fly decoding of regions of interest, optimising the implementation to further assess the feasibility of this approach.

1.3 Contribution

The following list details the individual contributions:

1. A method to enable efficient real-time rendering of triangle meshes with JPEG-compressed textures.
2. A compact indexing structure that enables random access to individual texels within a variable-rate compressed texture with little memory overhead. Specifically, we design an index list that maps the starting positions of compressed blocks to their corresponding texels. This index list is optimised for GPU memory, allowing textures to be decompressed on the fly directly on the GPU.
3. We present a modified deferred rendering pipeline that seamlessly incorporates rendering from variable-rate compressed textures while minimising computational overhead.
4. We provide detailed statistics and visual comparisons against existing state-of-the-art texture compression methods. These analyses provide valuable insights into the strengths and challenges of our approach, thereby enhancing our understanding of the practical benefits and trade-offs associated with variable-rate texture compression algorithms.
5. Based on our results, we evaluate the general applicability and usefulness of variable-rate texture compression algorithms, and provide guidelines on how to design and optimise future, more GPU-tailored, methods.

More generally, through this work, we demonstrate how traditional storage-focused compression formats can be re-engineered to meet the performance demands of modern real-time rendering, ultimately contributing to advancements in texture compression techniques and GPU rendering workflows.

CHAPTER 2

Related Work

Since the debut of the first dedicated consumer graphics card, the 3dfx Voodoo Graphics in 1996, with only 2 MB of texture memory, GPU memory capacity has expanded dramatically. Modern GPUs now feature up to 32 GB of VRAM–a significant improvement over early architectures. However, even this capacity falls short of meeting the needs of modern graphic applications, where the increasing fidelity of textures and the sheer scale of graphical assets regularly outstrip available memory.

Therefore, to maximise the utilisation of available storage, researchers have continually sought to develop ever-improving texture compression algorithms. In the context of real-time rendering, texture compression algorithms can be broadly categorised into three groups based on how they process and store data: fixed-rate compression, variable-rate compression, and the emerging class of neural texture compression techniques. The following sections will review the most significant works within each of these categories, including hybrid approaches that fall between them or propose alternative strategies for addressing VRAM storage limitations.

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, making individual texels or blocks of texels randomly accessible. GPU-based algorithms also make an effort to keep the decoding computationally inexpensive.

One of the earliest references to a fixed-versus-variable compression strategy is found in the work of Beers et al., who pioneered an approach using vector quantisation to compress textures and render directly from the compressed data [BAC96]. Although developed decades ago, this method continues to hold significance as a foundational concept, shaping the evolution of modern texture compression techniques and influencing advancements in graphics rendering and hardware-accelerated compression technologies.

Building on these early concepts, one of the most enduring and widely adopted texture compression algorithms is S3 Texture Compression (S3TC) [INH99]. Despite being over two decades old, S3TC—also known as DXT or, in its more recent iterations, BC1 through BC7—continues to play a central role in modern graphics applications. It compresses images by dividing them into 4×4 pixel blocks and computing two colour 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 colour 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.

Adaptive scalable texture compression (ASTC) [STS+21] builds on this idea but makes the block sizes adaptive in the sense that we can chose larger block sizes compared to the fixed size of 4x4 that is used by BC1-7. This enables the method to adapt more effectively to the characteristics of a texture, resulting in higher and more controllable compression, ranging 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 [SAM05]. More recently, Neural Texture Block Compression [FH24] uses a neural network that learns to optimise 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 quantisation 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 [HPLVdW12]. 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]. However, despite their advancements, these newer algorithms have not yet achieved the widespread support and adoption of the original JPEG standard. All of these formats utilise transform coding, which involves converting the image into the frequency domain before compression. Since these methods are lossy (with the exception of JPEG XL,

Table 2.1: Usage of Image File Formats on Websites [W3T25].

Format	**Usage (%)**
None	3.5
PNG	79.2
JPEG	74.2
SVG	62.1
GIF	17.1
WebP	16.7
AVIF	0.7
ICO	0.2
BMP	0.1

Other formats (TIFF, APNG, JPEG XL): <0.1% usage

which also supports lossless compression, though that feature is beyond the scope of this thesis), the loss of information primarily affects high-frequency components of the image, as these are less perceptible to the human eye.

Early innovations included the adoption of a DCT-based variable compression scheme within Microsoft's Talisman graphics architecture [TK96, Ran97]. To achieve random access, the image is split into chunks that can be addressed and decoded individually. Despite its potential, this concept was discontinued following the termination of the Talisman project.

JPEG 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 also 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 parallelise 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.

As illustrated in Table 2.1, the only image format with a higher usage rate than JPEG is the lossless Portable Network Graphics (PNG) format [RK99]. Although it is a lossless algorithm, meaning that no information is lost during the compression and decompression process, it remains a variable-rate algorithm, as different parts of the image are encoded with varying bitrates. Despite its widespread adoption for websites due to its lossless nature, which ensures that the image is presented in the best possible quality, the relatively low compression efficiency of PNG makes it less suitable for real-time rendering applications, in addition to the challenges posed by variable-rate formats.

WebP [Goo25], developed by Google with web use in mind, shares its core functionality with JPEG, as both use transform coding for compression. However, a key distinction lies in WebP's use of predictive coding: image blocks are predicted based on the values of neighbouring blocks, and the residuals (the differences between the predicted and actual values) are encoded and later used to reconstruct the original blocks. This approach achieves improved compression efficiency compared to JPEG, but it also increases the complexity of decoding individual blocks due to the reliance on inter-block dependencies.

The AV1 Image File Format (AVIF) [HLM+21] is based on the AV1 codec, developed by the Alliance for Open Media. In recent years, AVIF has gained notable traction, particularly with the introduction of hardware support for encoding and decoding in APIs such as Vulkan [AG24]. Despite its compression efficiency and growing adoption, the format presents a key limitation: it requires full image decoding, as partial decoding or progressive rendering is not natively supported. While AVIF can reduce upload latency during texture transfer, its ability to minimise VRAM usage is restricted to textures that are not actively accessed during rendering.

Similar to WebP, the High Efficiency Video Coding (HEVC) format, also known as H.265 [LHVA16], employs predictive coding to enhance compression. Originally designed to support diverse applications, HEVC offers advanced features such as high bit-depth support (up to 16 bits), wide colour gamuts (e.g., HDR), and frame-based optimisations for video. However, like WebP, it faces challenges for texture usage, particularly due to its reliance on inter-block dependencies and the complexity of partial decoding.

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 in research has shifted to neural texture compression methods, utilising neural networks to learn efficient and perceptually optimised 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 trade-off. 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] propose a neural material model using Block-Compressed features (BCf) that emulate BC6 decompression. This allows for high-resolution features with a low memory footprint and a simple, efficient decoder. The features are designed to

be decoded continuously in space and scale, allowing for smooth transitions and random UV sampling without the need for post-filtering.

This method was further improved by [LA25] by using BC1 compression instead of BC6 for even higher compression ratio. They also improve runtime performance by using a tile-based rendering architecture that leverages hardware matrix multiplication engines and the cooperative vectors extension.

All these methods benefit from the new tensor cores of modern consumer graphics cards. However, they always operate under the assumption that these cores are unused for rendering. With the rise of neural methods in every aspect of rendering, the cost of these operations on the tensor cores might have to be reevaluated.

2.4 Other

Some unique compression techniques fall outside the main categories discussed above. Schuster et al.[STS+21] focus on textured point clouds, where textures are significantly larger than traditional ones. Their method learns a dictionary from the texture and represents each splat using an average colour along with a variable number of atom indices and corresponding weights, significantly reducing memory usage. Luo et al. reduce redundancy by merging all the textures in a scene and remapping the texture coordinates to eliminate repeated content [LJP+23].

Huffman Decoding is not only used in JPEG, and is often the bottleneck even in other applications due to its sequential nature. Goel et al. [GSNK24] use a clipped Huffman encoding for compressing huge point clouds. They limit the maximum codeword length to a predefined level, while the less common symbols are stored in a lookup table instead. This allows for faster and more efficient parallel decompression. Other approaches have looked into different ways to parallelise the decoding. Johnston et al. [JM17] present a parallel Huffman decoding algorithm that operates by first performing a concurrent decoding of individual symbols from every potential starting bit position within the compressed bitstream, populating an intermediate buffer. Subsequently, the algorithm leverages a pre-computed data structure, derived from the Huffman tree, to efficiently determine the correct sequence and placement of valid decoded symbols within the final uncompressed output. Similarly, [WS18] leverages the self-synchronising properties inherent to most generated Huffman codes. This approach ensures that even if a thread begins decoding at an incorrect position and produces erroneous symbols, it will eventually synchronise and decode correctly after a limited number of errors. By overlapping the decoded sections, the incorrectly decoded segments can be discarded, retaining only valid codewords. However, these techniques rely on large file sizes to achieve better performance than sequential decoding, which is impractical in our case due to the need to decode only a small number of codes at a time.

There is also the field of super-compressed textures [SW11, KPM16], which involves applying an additional layer of compression to textures that have already been compressed

using standard GPU-friendly formats (such as BCn or ASTC). These supercompressed textures are stored in a highly compact form and later decompressed at runtime—either on the CPU or GPU—back into their original compressed state before being sampled by the rendering pipeline. This technique allows developers to benefit from reduced storage and bandwidth usage, particularly for distribution and streaming, without sacrificing the compatibility and efficiency of hardware-accelerated texture formats.

CHAPTER 3

JPEG

Our proposed method enables real-time rendering of JPEG-compressed textures. To fully understand our texture compression approach, it is necessary to first review how the JPEG standard is defined. Accordingly, this chapter presents the fundamental design principles of the JPEG algorithm, followed by a detailed explanation of the key components that make up a standard JPEG file.

JPEG is a standardised image compression algorithm developed by the Joint Photographic Experts Group. It supports multiple operating modes, each tailored to specific use cases and varying in their compression strategies. Among these, the most widely adopted variant is baseline JPEG, which serves as the de facto standard in numerous applications due to its broad compatibility and balance between compression efficiency and image quality. An example of a JPEG-compressed image is shown in Figure 3.1.

Figure 3.1: Example JPEGS at quality levels 90, 50 and 5. When the quality becomes too low, artifacts become noticeable.

3.1 Algorithm

An overview of the JPEG algorithm can be seen in Figure 3.2. To compress an image

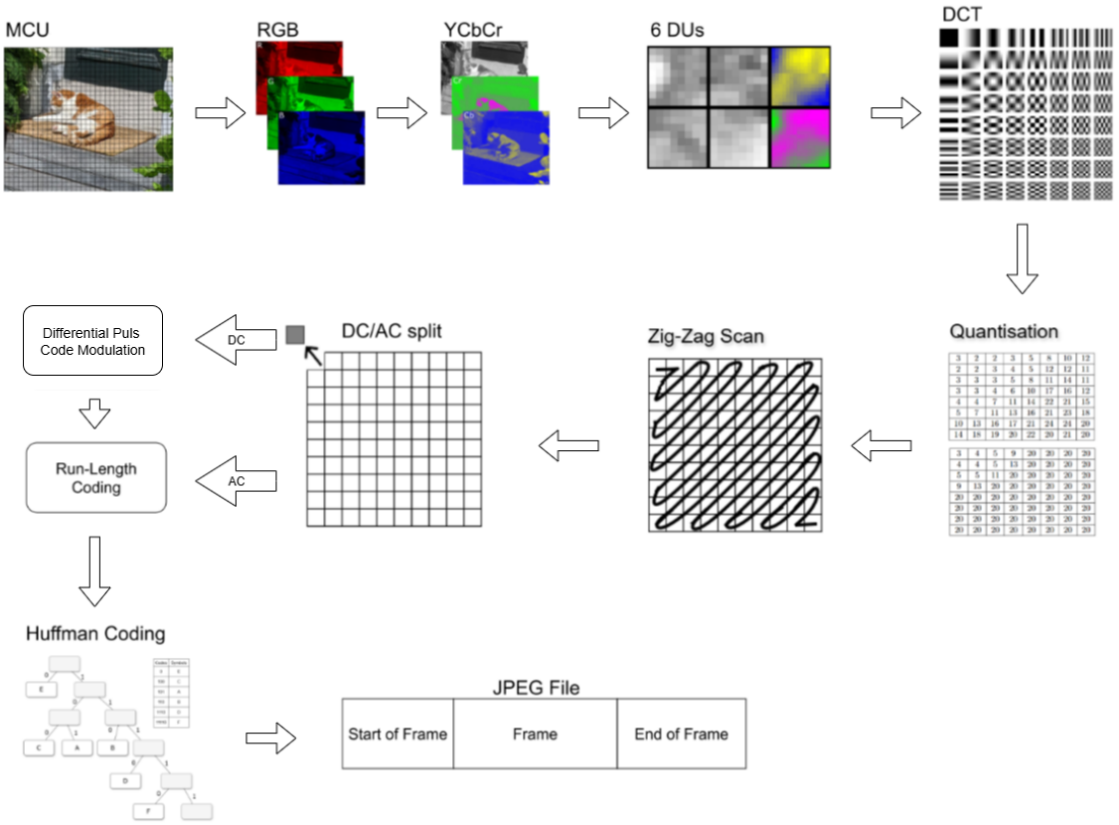


Figure 3.2: JPEG encoding pipeline: The image is divided into 16×16 pixel blocks, converted to luminance and chrominance channels, transformed using the cosine transform, quantised, scanned in zigzag order, then compressed using differential coding, run-length encoding, and Huffman coding.

using baseline JPEG, the image is first converted from the RGB colour space to YCbCr using Equation 3.1.

$$\begin{bmatrix} Y \\ Cb \\ Cr \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ -0.168736 & -0.331264 & 0.5 \\ 0.5 & -0.418688 & -0.081312 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix} + \begin{bmatrix} 0 \\ 128 \\ 128 \end{bmatrix} \tag{3.1}$$

This colour space separates luminance (Y) from chrominance components: blue-difference (Cb) and red-difference (Cr). An example can be seen in Figure 3.3. The reason for this is that it allows for more efficient compression by exploiting the human visual system's higher sensitivity to luminance detail [PB13]. This is done in two ways: the first is chroma subsampling. In this thesis, we are specifically focusing on the 4:2:0 sampling scheme. In this scheme, the Cb and Cr channels are sampled at half the horizontal and vertical resolution compared to the Y channel, as illustrated in Figure 3.4. These subsampled channels are later upscaled during the decompression process. The image is then divided

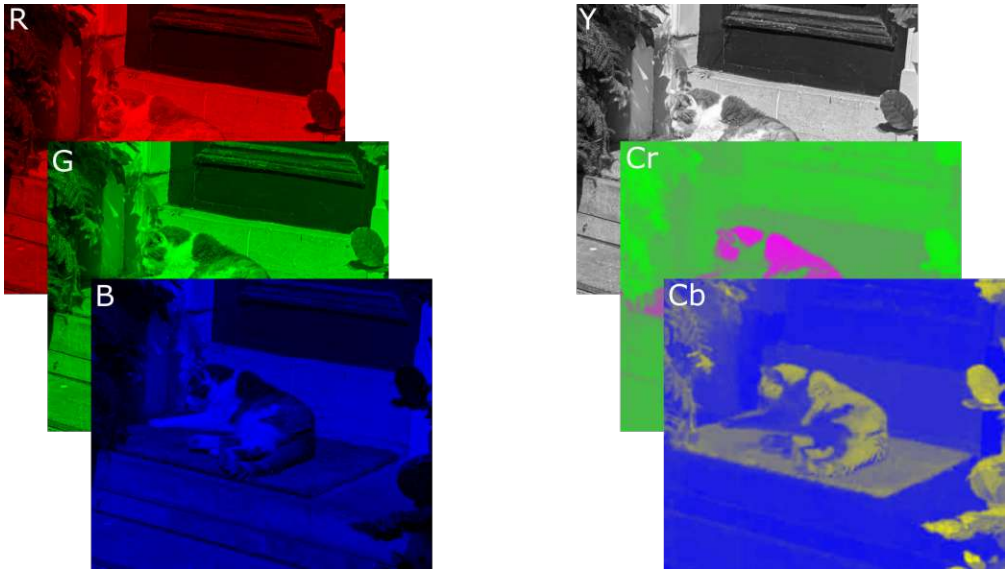


Figure 3.3: The Red (R), Green (G) and Blue (B) colour channels are converted to luminance (Y), Cb (blue difference) and Cr (red difference) channels.

Figure 3.4: The Y, CB and Cr colour channels are subsampled. Top: The original 4:4:4 retains full chroma information for every pixel, resulting in higher colour fidelity, while the 4:2:0 subsampling reduces the chroma resolution by encoding one value per 2x2 pixels, conserving bandwidth and storage at the cost of some colour detail.

(a) JPEG (b) Minimum Coded Unit (c) Data Unit

Figure 3.5: Baseline JPEG is divided into 16×16 pixel blocks, referred to as Minimum Coded Units (MCUs). Each MCU consists of 6 Data Units (DUs): 4 for the luminance channel (storing one value per pixel) and 1 each for the blue-difference and red-difference chrominance channels (storing one value per 2x2 pixels).

into Minimum Coded Units (MCUs) blocks. In the case of 4:2:0 subsampling, each MCU represents a 16x16 pixel region that is encoded via six 8x8 sub-blocks called Data Units (DUs): four DUs for the luminance channel (encoding one value per pixel) and one DU each for the Cb and Cr channels (encoding one value per 2x2 pixels). This subdivision is illustrated in Figure 3.5.

Each DU undergoes a Discrete Cosine Transform (DCT), representing the block as a weighted sum of 64 cosine waves with varying frequencies, that can be seen in Figure 3.6. The coefficient at zero frequency in both dimensions (located in the top-left of each block) is known as the direct current (DC) coefficient. It represents the average value of all samples in the block. To further reduce the data size, DC coefficients are encoded using differential coding: instead of storing absolute values, each DC coefficient is stored as the difference from the previous one. This is done separately for each color channel to maximise the correlation between consecutive DC values. The remaining 63 coefficients, corresponding to higher spatial frequencies, are referred to as alternating current (AC) coefficients.

The DCT coefficients are then quantised using a quantisation matrix, which assigns a divisor between 1 and 255 to each frequency component. Each coefficient is then divided by the corresponding value in the quantisation matrix and rounded to the nearest integer. This step is lossy and plays a central role in JPEG compression. The design of the quantization matrix is meticulously crafted based on psychovisual experiments, which aim to determine the Just Noticeable Thresholds (JNTs) of various spatial frequencies for the human visual system (HVS). Seminal research, such as that by Campbell and Robson [CR68] on the Contrast Sensitivity Function (CSF), has demonstrated that the HVS exhibits a band-pass characteristic, meaning it is most sensitive to mid-range spatial frequencies and significantly less sensitive to very high and very low frequencies. Consequently, the quantisation matrix is designed with larger divisors for high-frequency

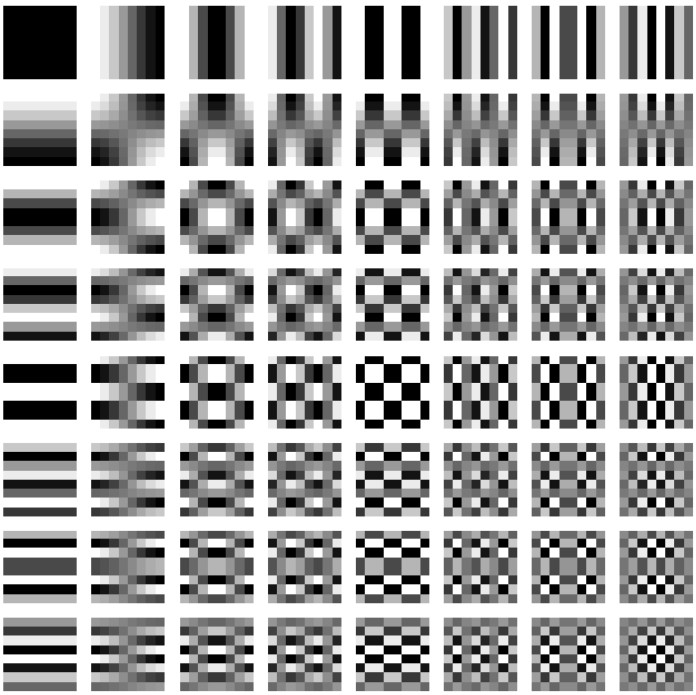


Figure 3.6: 8x8 dct basis functions: each square represents a unique 2D frequency component used in the Discrete Cosine Transform (DCT). Low-frequency components (top-left) capture coarse image details, while high-frequency components (bottom-right) represent finer details and edges.

components and smaller divisors for low and mid-frequency components. This strategic weighting allows for more aggressive removal of high-frequency information, as the HVS is less likely to perceive its loss, thereby achieving substantial file size reduction with minimal perceived quality degradation.

In addition, the HVS is more sensitive to changes in contrast than colour. This is why there are separate matrices for luminance and chrominance channels, where the chrominance quantisation is much more aggressive.

There is no standardised quantisation matrix in the JPEG specification–it is left to the implementer to define or adjust the matrix based on desired quality or application needs. Example quantisation tables for quality setting of 90 and 50 can be seen in Table 3.1.

To maximise the efficiency of run-length encoding, the coefficients are read in a zigzag pattern, beginning from the top-left corner. This traversal ensures that low-

Table 3.1: Quantization Tables

(a) Quantization Table luminance (q90)

3	2	2	3	5	8	10	12
2	2	3	4	5	12	12	11
3	3	3	5	8	11	14	11
3	3	4	6	10	17	16	12
4	4	7	11	14	22	21	15
5	7	11	13	16	21	23	18
10	13	16	17	21	24	24	20
14	18	19	20	22	20	21	20

(b) Quantization Table chrominance (q90)

3	4	5	9	20	20	20	20
4	4	5	13	20	20	20	20
5	5	11	20	20	20	20	20
9	13	20	20	20	20	20	20
20	20	20	20	20	20	20	20
20	20	20	20	20	20	20	20
20	20	20	20	20	20	20	20
20	20	20	20	20	20	20	20

(c) Quantization Table luminance (q50)

16	11	10	16	24	40	51	61
12	12	14	19	26	58	60	55
14	13	16	24	40	57	69	56
14	17	22	29	51	87	80	62
18	22	37	56	68	109	103	77
24	35	55	64	81	104	113	92
49	64	78	87	103	121	120	101
72	92	95	98	112	100	103	99

(d) Quantization Table chrominance (q50)

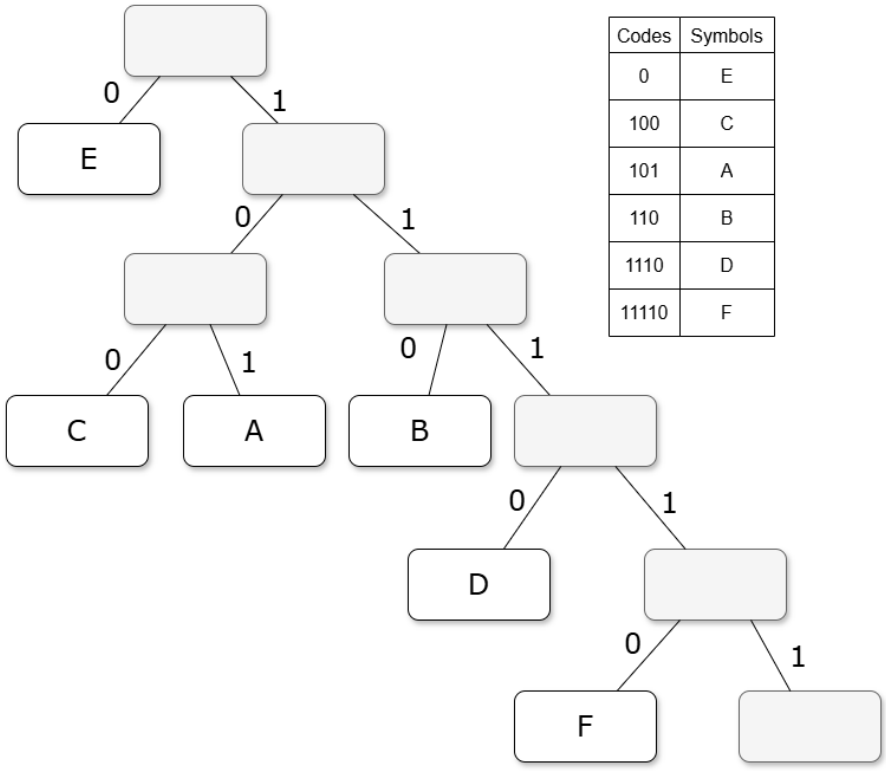
17	18	24	47	99	99	99	99
18	21	26	66	99	99	99	99
24	26	56	99	99	99	99	99
47	66	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99

frequency coefficients, which are less likely to be zero, are read first, while higher-frequency coefficients—more likely to be quantised to zero—are grouped together. This arrangement significantly enhances the overall compression rate.

Huffman encoding then assigns each symbol a unique codeword, with more frequently occurring symbols assigned shorter codewords to minimise overall storage. Each codeword is prefix-free (no code is a prefix of another), so that the encoded bitstream can be uniquely decoded. This forms a frequency-sorted binary tree that can be seen in Figure 3.7.

The decoding process in JPEG mirrors the encoding steps in reverse order. Consequently, to reconstruct the original image accurately, the decoder must have access not only to the compressed image data, but also to the Huffman tables and quantisation matrices used during encoding. These components are embedded within the JPEG file itself, ensuring that the decoder has all necessary information to perform a complete and correct reconstruction. A detailed description of the file structure will be presented in the following section.

Compressing a 4K texture with dimensions of 4096×4096 texels across three channels and a 4:2:0 subsampling scheme results in 65,536 MCUs and a total of 393,216 DUs.



Codes	Symbols
0	E
100	C
101	A
110	B
1110	D
11110	F

Figure 3.7: In this example, the symbol E occurs much more frequently than all the others. As a result, it is assigned the shortest possible codeword: 0. Because Huffman codes must be prefix-free, no other codeword can begin with 0. If the symbol probabilities were more evenly distributed, the resulting Huffman tree would be more balanced.

3.2 File format

The JPEG standard describes four different settings for the compression format:

- Baseline Sequential
- Extended Sequential
- Progressive
- Lossless

Depending on the compression setting, the layout of the JPEG file differs. In baseline sequential JPEG, often abbreviated to just baseline JPEG, the MCUs are stored in sequential order. Each MCU contains an 8×8 block of the luminance data, followed by the chrominance DUs in this specific order. Extended Sequential JPEG builds on this by

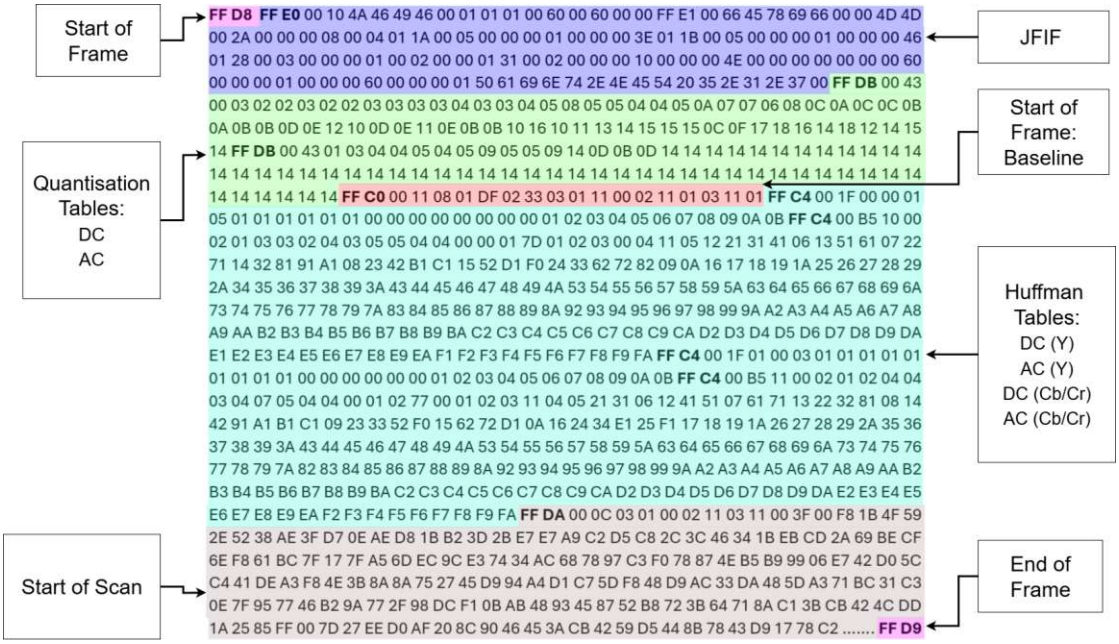


Figure 3.8: Example of a baseline JPEG file conforming to the JFIF standard. Key segments such as the Start of Image (FFD8), JFIF header, Quantization Tables (FFDB), Start of Frame (FFC0), Huffman Tables (FFC4), Start of Scan (FFDA), and End of Image (FFD9) are annotated. All markers are shown in bold, while portions of the scan data are omitted for brevity.

allowing the use of up to two additional Huffman and quantisation tables, offering more flexibility for compression, but is rarely used.

Progressive JPEG, however, uses a fundamentally different approach. Instead of processing MCU after MCU, it decodes the image in stages. Initially, the highest-frequency components of each DU are decoded, followed by progressively lower frequencies. This method enables progressive rendering: a coarse, blurry version of the image appears first, which becomes increasingly detailed as more coefficients are decoded. This feature makes Progressive JPEG particularly useful for web applications where users benefit from quickly previewing images while the full data is still loading.

Lossless JPEG does not use the DCT for image encoding. Instead, it relies on a predictive model based on neighbouring pixels. Because no information is discarded, the compression ratio is significantly lower compared to its lossy counterpart. As a result, Lossless JPEG has seen limited adoption and is supported by only a few applications.

The JPEG file format is structured around a sequence of markers, which define how the subsequent data should be interpreted. Each marker begins with a prefix byte of all 1s—represented in hexadecimal as 0xFF—to distinguish it from image data. The byte that immediately follows determines the type and function of the marker. While the ordering

of markers is not strictly defined by the JPEG specification, most implementations conform to the JPEG File Interchange Format (JFIF) standard [Ham04], which was introduced to ensure greater consistency across applications.

Below is a description of the most essential JPEG markers, presented in the order they typically appear in JFIF-compliant files.

- **0*xFFD*8 Start of Image**: Marks the beginning of a JPEG file. This marker verifies that the file is a JPEG file but does not contain any information for decoding.
- **0*xFFE*0 Application-Specific (APPn) Markers**: These markers are reserved for application-specific data, such as metadata. For example, **0*xFFE*1** is commonly used for EXIF metadata, while **0*xFFE*0** is typically used by the JFIF. Each APPn marker begins with a 2-byte length field, followed by application-defined data. In the case of JFIF, this includes the ASCII string `"JFIF\0"`, version information, resolution units, pixel density values, and optionally, thumbnail image data that is used in web applications or file explorers.
- **0*xFFDB* Define Quantization Table**: Defines the quantisation tables used for image data compression. This marker is immediately followed by two bytes specifying the table's length. The next 4 bits indicate the target channel (e.g., luminance or chrominance), while the following 4 bits denote the precision of the table values—either 8-bit or 16-bit. The quantisation values themselves are then listed, ordered from higher to lower frequencies, following the zig-zag scan pattern that is also applied to DCT coefficients.
- **0*xFFC*0 Start of Frame (Baseline DCT)**: Marks the beginning of the frame header, which defines the image dimensions, colour components, and sampling factors. It starts with 2 bytes indicating the length of the frame header, followed by 2 bytes for the image height and 2 bytes for the image width. The next byte declares the number of colour components. For each component, three bytes follow: the first byte is the component ID, the second byte contains the sampling factors where bits 0–3 represent the vertical sampling factor and bits 4–7 represent the horizontal sampling factor, and the third byte specifies the quantisation table number.
- **0*xFFC*4 Define Huffman Table**: Defines the Huffman tables used for entropy encoding of the image data. This marker is followed by two bytes indicating the length of the segment. The next three bits specify how many Huffman tables this marker defines, and the 4th-bit acts as a flag to indicate whether it is an AC or DC Huffman table. The following 4 bits are unused and must be set to 0. Finally, the values of the Huffman table are listed.
- **0*xFFDA* Start of Scan**: Signals the start of the image data stream encoded with entropy coding. The order of the data is dependent on whether it is a sequential or progressive JPEG.

- **0$xFFD9$ End of Image**: Marks the end of a JPEG file. It is optional, so a decoder can not rely on it, but most encoders do use it.

There are a few additional markers that are not listed as they are not important for this application and are rarely used. If the entropy-encoded image data contains a sequence that would result in $0xFF$, it is followed by an empty byte to indicate that is, in fact, data and not a marker. An example of a JPEG file in hexadecimal can be seen in Figure 3.8.

CHAPTER 4

Method

The proposed method employs baseline JPEG compression for textures due to its widespread adoption and practical efficiency. Specifically, it uses baseline sequential encoded files and implements 4:2:0 chroma subsampling. Supporting multiple JPEG settings would introduce significant implementation overhead without yielding substantial benefits, as the variants are not inherently superior but rather optimised for specialised tasks. However, the method is flexible and can be adapted to support other JPEG settings, such as 4:4:4 subsampling if needed.

4.1 Overview

In this section, we briefly outline the core steps of our algorithm; a more detailed explanation of each step is provided in the subsequent sections. The main motivation behind our approach is to enable random access to JPEG-compressed textures, which offer a superior quality-to-memory trade-off compared to many state-of-the-art compression algorithms. However, a naive implementation of JPEG decoding is unsuitable for real-time rendering as it provides no random-access property and is computationally expensive. To overcome these limitations, we introduce a series of optimisations that reduce decoding overhead while exploiting the inherent parallelism of the GPU.

To render from JPEG-compressed textures, the data is first preprocessed to compute an AC-offsets table and extract the Huffman and the quantisation tables. During rendering, textures are then partially decompressed using a three-step algorithm consisting of a Mark, Decode and Resolve step, which is integrated into a deferred rendering pipeline to minimise decoding overhead, as illustrated in Figure 4.1.

In the first step, the G-Buffer that holds all the data required for rendering the scene has to be created. However, instead of sampling the texture immediately, our approach only stores the UV coordinates. This enables a more GPU-friendly texture decoding process:

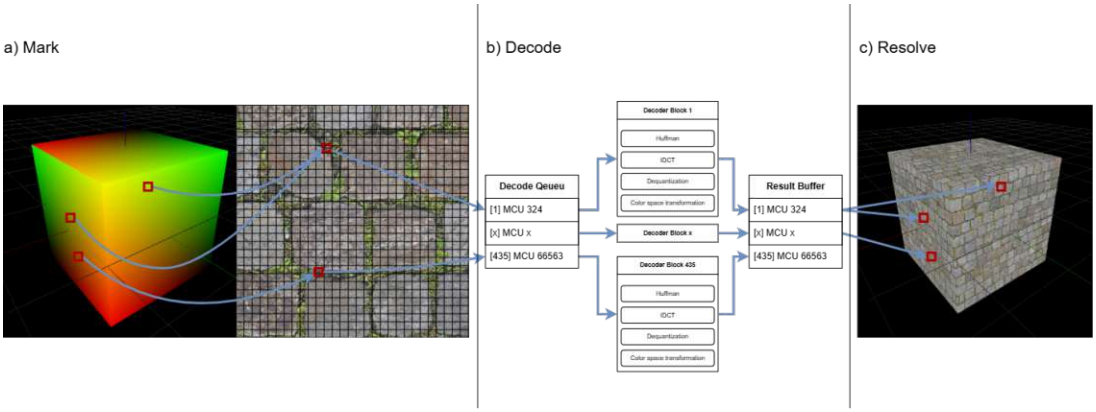


Figure 4.1: Overview of our method, following the construction of a G-Buffer in a deferred rendering pipeline. a) For every pixel's UV coordinate, we compute the corresponding MCU and add it to the decoding queue. If an MCU is required multiple times, it is still added to the queue only once. b) MCUs in the decoding queue are processed in parallel by GPU thread 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 colour for each pixel, completing the rendering process.

by delaying the texture lookup, all textures can be decoded at once while making optimal use of the available computing resources, which is the key factor behind our method achieving real-time performance.

Once the G-Buffer is constructed, we check every pixel to determine which Minimum Coded Units of a texture need to be decoded for correct texturing. This step effectively marks all MCUs that must be decoded to render the scene. By doing so, only the necessary MCUs are decoded, while the rest of the texture remains compressed. This not only reduces storage requirements but also improves performance, since fewer computations need to be performed.

Next, the marked MCUs of the previous step are decoded. This process is optimised by exploiting the GPU's parallelism to accelerate Huffman decoding and colour space transformations, and by leveraging the separability of the inverse discrete cosine transform.

Finally, the Resolve step gathers the required and now decoded MCUs from the first step and computes the pixel's final colour. At this point, scene shading proceeds identically to a standard deferred rendering pipeline.

4.2 Preprocessing for random access

Although we aim to remain faithful to the original JPEG specification as much as possible, some preprocessing is necessary before sending the data to the GPU. In addition, certain

steps—such as parsing the Huffman and quantisation tables—would be redundant at runtime and would unnecessarily degrade performance.

To mitigate this, JPEG markers are parsed at startup. The Huffman and quantisation tables are extracted and uploaded to a constant buffer for efficient GPU access. Following this, the entropy-coded image data is scanned to construct an index list that enables random access during decoding. To support this design, each data unit (DU) is split into its AC and DC components: the AC coefficients remain Huffman-encoded, while the DC coefficients are decoded and stored explicitly, each using a fixed number of bits. From the buffer now containing only AC components, we record the start position of each Minimum Coded Unit (every six DUs) and store it in a per-texture AC-offsets table. The entries to this table follow the natural raster order of the JPEG file: the top-left MCU is stored at position 0, increasing from left to right and top to bottom. As a result, given the order of an MCU in the stream, which in the following will be called the MCU's ID, we can directly locate the corresponding AC data and, since DC coefficients are stored at predictable offsets, retrieve the DC value as well—enabling true random access. Finally, we create a buffer that has one 32-bit entry per MCU. This buffer, referred to as the decoded-MCU-indices buffer in the following, is used to flag which MCUs should be decoded in the current frame and to store the index of the decoded colour data for each MCU. While this layout introduces some overhead compared to the standard JPEG format, the trade-offs and implications are further discussed in Section 4.7.

4.3 G-Buffer creation

Unlike traditional forward rendering, deferred rendering begins by rendering all geometric information from the scene – such as positions, normals, diffuse colours, and other material properties – into a series of textures collectively referred to as the Geometry Buffer (G-Buffer). Shading is performed only after the entire geometry has been rendered. This approach offers the advantage of computing the lighting model and other effects exclusively for objects visible in the final scene, eliminating redundant calculations caused by overdraw when one object obscures another in a subsequent draw call. This means that the diffuse colour values from textures are typically stored directly in the G-Buffer.

Our approach deviates from the standard deferred rendering pipeline by storing only the UV coordinates and the texture ID for each pixel instead of directly sampling the diffuse texture. This design allows texture lookups – and more importantly, the texture decoding – to be deferred to a separate stage as well. The key advantage of this strategy is that multiple threads can collaboratively decode the required MCUs in parallel, rather than each individual thread performing its own decoding operations, making the process viable for real-time applications. The details of this optimisation are discussed in the following section. Figure 4.2 illustrates an example of UV coordinates visualised as colour.

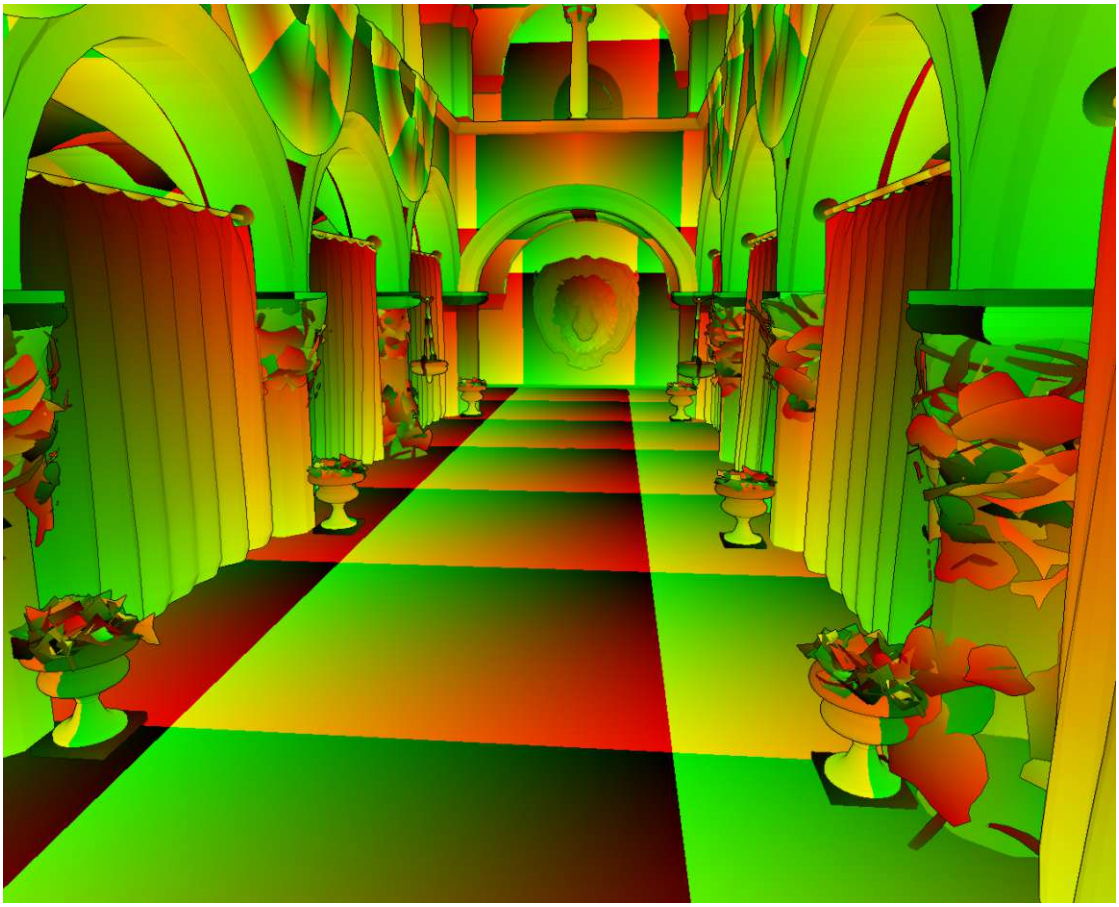


Figure 4.2: The UV texture of the G-Buffer. The quads on the floor indicate the repeating texture.

4.4 Mark

We initiate a CUDA kernel with one thread per pixel, where each thread reads its UV coordinate and calculates the ID of the MCU containing this pixel. The formula for this can be seen in Equation 4.1.

$$\text{MCU} = \frac{(\lfloor u \cdot \text{width} \rfloor \bmod \text{width})}{16} + \frac{(\lfloor v \cdot \text{height} \rfloor \bmod \text{height})}{16} \cdot \frac{\text{width}}{16} \tag{4.1}$$

We then check its corresponding location in the decoded-MCU-indices buffer of the specific texture. We use the highest bit of that 32-bit value as an indication if another thread has already added this MCU to the decoding queue: If the bit is already set, no further action is taken. Otherwise, the MCU's ID is added to the decoding queue, and the bit is set. This approach ensures that if multiple pixels request texels from the same MCU, the MCU is decoded only once. Additionally, the position of the MCU in the

decoding queue is written to the decoded-MCU-indices buffer, allowing efficient retrieval of decoded texture data later.

The queue size for MCUs requiring decoding must be estimated in advance, but it is highly dependent on the scene and camera perspective. In the worst case, each pixel on screen maps to a unique 16x16 pixel MCU. In our tests, allocating a queue size of 80k consistently proved sufficient, even in demanding scenarios.

Algorithm 4.1 illustrates the kernel and how each MCU is marked. The g_buffer is the geometry buffer from the deferred rendering stage. The actual kernel is a bit more complex due to managing parallel reads and writes for the bit that flags if an MCU should be decoded, but the overall structure remains the same.

Algorithm 4.1: Mark algorithm

```
1 pixel_ID ← Compute the global thread rank;
2 texture_ID ← g_buffer.texture_IDs[pixel_ID];
3 uv ← g_buffer.uvs[pixel_ID];
4 texture_ID ← g_buffer.texture_IDs[pixel_ID];
5 mcu_ID ← uv_To_MCU_Index(uv, texture_ID);
6 if mcu not in decode_queue then
7 |   decode_queue.add(mcu_ID);
8 |   decoded_MCU_indices[mcu] ← pointer to mcu in decode_queue;
9 end
```

4.5 Decode

In this step, we decode all the MCUs identified in the previous step as necessary for rendering the image. Therefore, we launch a kernel with one block of 64 threads for each MCU in the decoding queue, allowing us to process its six Data Units. While an MCU contains 256 texels, we found that using 64 threads provides a good balance for enabling efficient parallel execution for steps where all 64 coefficients of each Data Unit can be processed simultaneously, while minimising the number of idle threads during sequential decoding stages. Increasing the thread count to 256 would speed up the operations that need to be applied to all texels, but loses performance due to non-optimal usage of the GPU in other stages, which outweighs the benefits.

Since the starting position of each MCU AC data is stored in the AC-offset Table, and the DC coefficients have predictable offsets, all MCUs can be decoded independently and in parallel. Each block retrieves an MCU ID from the decoding queue at the position of its block ID, reads the starting position in the compressed stream from the AC-offsets table, and loads the subsequent 384 bytes into local memory to optimise access during decoding. This reserves one byte per coefficient. While Huffman codes can span more than one byte, making it theoretically possible that not all necessary bytes are loaded for

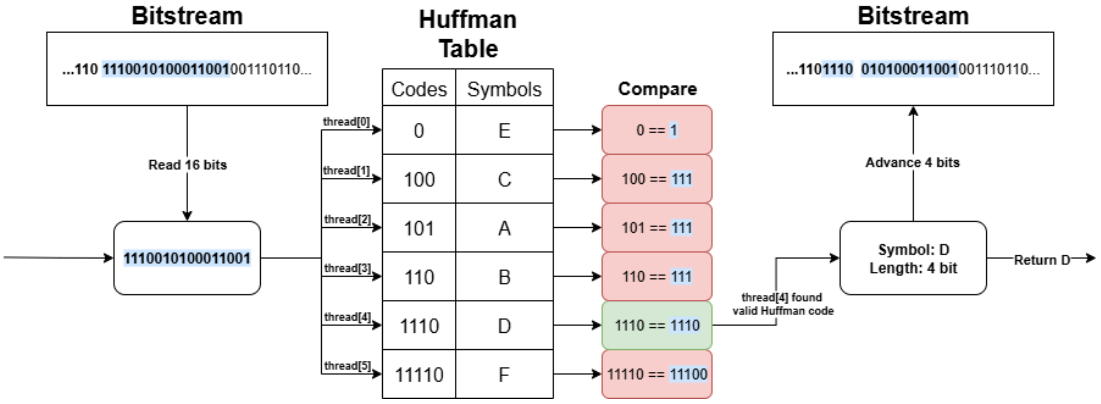


Figure 4.3: A single step of our parallel Huffman decoding process. Sixteen bits are first loaded from the current position in the bitstream. Each thread then compares the leading bits against a predefined codeword of the same length. Once a match is found, the corresponding symbol is returned, and the bitstream position is advanced to decode the next symbol.

decoding, this scenario is extremely unlikely, to the point that an image would need to be constructed specifically for this to occur. In practice, it was never an issue in any of the textures we tested.

The next step, and the main performance bottleneck in MCU decoding, lies in the Huffman decoding of the AC coefficients. The decoding process is built on two nested loops: An outer loop that iterates over the AC coefficients one after the other, and an inner loop that compares the next bits in the stream until it detects a valid Huffman code for the current AC. The outer loop is inherently sequential, but we can optimise the inner loop by comparing the next bits in the stream to multiple Huffman codes simultaneously: Instead of performing traditional Huffman tree traversal – reading one bit at a time and checking for a match at each step – we prefetch the next 16 bits from the bitstream. We then parallelise the comparison of bits to codes by assigning each thread of the leading warp a different code, allowing us to evaluate 32 codes at the same time. Using the length of its assigned code, the thread extracts the corresponding number of bits from the prefetched 16 bits and compares it against its code. If it matches, the thread marks the code as a candidate. A warp-wide reduction is then used to identify the shortest matching code among all 32 threads, which corresponds to the correct Huffman symbol and is selected as the decoded result. An illustration of this algorithm can be seen in Figure 4.3.

The next step, quantisation, is combined with de-zigzagging of the coefficients. A lookup table determines the order in which coefficients are read from shared memory, mapping the zigzag scan order back to the original 8×8 block layout while also multiplying the value by the corresponding quantisation value.

The IDCT implementation is based on the optimised version from NVIDIA for efficient

execution on parallel GPU architectures by Obukhov et al. [OK08]. To reduce memory access latency, intermediate data is stored and manipulated in fast shared memory rather than slower global memory. In addition, the transform is applied in two passes–first across rows and then across columns–taking advantage of the separability property of the 2D IDCT. The 2D IDCT is defined in Equation 4.2 from Sung et al. [SSYH06]:

$$f(x,y) = \frac{2}{\sqrt{MN}} \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} C(m)C(n)F(m,n) \cos\left[\frac{(2x+1)m\pi}{2M}\right] \cos\left[\frac{(2y+1)n\pi}{2N}\right] \quad (4.2)$$

with the normalisation term:

$$C(u) = \begin{cases} \frac{1}{\sqrt{2}}, & \text{if } u = 0 \\ 1, & \text{if } u > 0 \end{cases} \quad (4.3)$$

This expression is separable, as shown by rewriting it as nested sums in Equation 4.4:

$$f(x,y) = \sum_{n=0}^{N-1} \left[\sum_{m=0}^{M-1} \left(\frac{2}{\sqrt{MN}} C(m)C(n)F(m,n) \cos\left(\frac{(2x+1)m\pi}{2M}\right)\right) \cos\left(\frac{(2y+1)n\pi}{2N}\right)\right] \quad (4.4)$$

Defining an intermediate result as in Equation 4.5:

$$g(x,n) = \sum_{m=0}^{M-1} \frac{2}{\sqrt{MN}} C(m)C(n)F(m,n) \cos\left(\frac{(2x+1)m\pi}{2M}\right) \quad (4.5)$$

leads to the final column-wise computation step in Equation 4.6:

$$f(x,y) = \sum_{n=0}^{N-1} g(x,n) \cos\left(\frac{(2y+1)n\pi}{2N}\right) \quad (4.6)$$

This reduces the number of calculations performed from $O(N^4)$ to $O(N^3)$. In addition, in separating the transformation, we only need 8 threads to compute the IDCT for a DU. Since each MCU is processed by 64 threads, this allows us to compute the IDCT for all 6 DUs in parallel with 48 threads. Precomputed constants are used to eliminate expensive runtime calculations, and loop unrolling is employed to enhance instruction-level parallelism. Additionally, memory access patterns are carefully designed to avoid conflicts and ensure that data can be read and written efficiently by multiple threads in parallel. The resulting decoded YCbCr colour values are then stored in a buffer in the same order as the MCUs were added to the decoding queue.

Finally, we need to convert the image from the YCbCr colour space to RGB. This is done by applying the transformation formula described in Section 3.1 to each YCbCr triple. Since the image uses 4:2:0 chroma subsampling, the chrominance channels must be upscaled in this step. A simple approach is to advance the Cb and Cr channels at half the rate of the Y channel, effectively duplicating each chroma value for a 2×2 block of luma pixels. Since a single MCU contains 16×16 texels, each thread needs to decode four values. The final RGB values are then written to a Buffer at the position of the ID of the MCU.

Algorithm 4.2 shows the decoding. For simplicity, we ignore the upsampling of the chroma channels in this representation.

4.6 Resolve

This final pass closely mirrors the first and is responsible for mapping uv-coordinates to the now decoded MCUs in order to texture the scene. As in the initial pass, we compute the corresponding MCU for each pixel using its UV coordinates. The decoded-MCU-indices buffer is then referenced to determine the MCUs decoded data location in the decoded buffer. Since each MCU contains 256 texels, we multiply this index by 256 to compute the starting position of the MCU. To locate the specific texel within the MCU, we calculate the intra-MCU offset based on the pixel's position. Using this offset, we sample the corresponding colour value for each pixel. Once the colour information is reconstructed, the standard deferred rendering pipeline proceeds to compute the shading and final appearance of the scene. This procedure is implemented in the simple kernel shown in Algorithm 4.3.

4.7 Memory overhead

To enable random access within JPEG-compressed images, we store an index marking the start position of each MCU block. This introduces a memory overhead, but it remains minimal, requiring only a single 32-bit index per MCU. To further reduce this overhead, we store only every ninth index as an absolute 32 bit index, while the subsequent eight indices store 16 bit offsets relative to this absolute index, reducing the size of MCU offsets to 17.7 bits each. Additionally, because differential encoding conflicts with random access, the DC coefficients of each DU are not encoded and are instead stored as 12-bit values. Since each MCU requires 6 DC coefficients, this results in a total of 72 bits. Furthermore, 32 bits per MCU are allocated for the decoded-MCU-indices buffer. In total, this results in 121.7 bits of overhead per MCU, as shown in Figure 4.4. As each MCU comprises $16 \times 16 = 256$ RGB texels, our approach introduces an overhead of approximately $\frac{121.7}{256} = 0.4757$ bits per texel. However, this estimate assumes that the DC coefficients contribute 0 bits in the original JPEG-encoded image—an assumption that does not hold in practice. Therefore, this value represents an upper bound on the

Algorithm 4.2: Decode

```
1  sh_coefficients[384];
2  block_ID ← Compute the global block rank;
3  thread_ID ← Compute the global thread rank;
4  mcu_ID ← decode_queue[block_ID];
   // Load compressed data into shared memory
5  mcu_starting_position ← mcu_indices_buffer[mcu_ID];
6  for i ← 0 to 5 do
7  |  block_id ← thread_ID + 64 · i;
8  |  DC_coefficients[i · 64] ← DC_coefficients_buffer[mcu_ID · 6 + i];
9  |  sh_coefficients[block_id] ←
   |    AC_coefficients_buffer[mcu_starting_position + block_id];
10 end
   // Dequantize and de-zigzag
11 for i ← 0 to 5 do
12 |  block_id ← thread_ID + 64 · i;
13 |  sh_coefficients[block_id] ← Dezigzag(sh_coefficients[block_id]);
14 |  sh_coefficients[block_id] ← sh_coefficients[block_id ] × QuantTable;
15 end
   // Inverse DCT
16 for i ← 0 to 5 do
17 |  block_id ← thread_ID + 64 · i;
18 |  pixels[block_id] ← IDCT8x8(sh_coefficients[block_id]);
19 end
   // YUV to RGB conversion
20 for i ← 0 to 3 do
21 |  block_id ← thread_ID + 64 · i;
22 |  Y ← pixels[block_id] + 128;
23 |  Cb ← pixels[192 + thread_ID];
24 |  Cr ← pixels[256 + thread_ID];
25 |  R ← clamp(Y + 1.402 · Cr);
26 |  G ← clamp(Y − 0.344 · Cb − 0.714 · Cr);
27 |  B ← clamp(Y + 1.772 · Cb);
28 |  decoded_buffer[block_ID · 256 + block_id] ← (R, G, B, 255);
29 end
```

Algorithm 4.3: Resolve

$pixel\_ID \leftarrow$ Compute the global thread rank;
$uv \leftarrow g\_buffer.uvs[pixel\_ID]$;
$texture\_ID \leftarrow g\_buffer.texture\_IDs[pixel\_ID]$;
$width \leftarrow texturesData[texture\_ID].width$;
$height \leftarrow texturesData[texture\_ID].height$;
$mcu\_ID \leftarrow uv\_To\_MCU\_Index(uv, texture\_ID)$;
$mcu\_offset \leftarrow decoded\_MCU\_indices[mcu\_ID] \cdot 256$;
$tx \leftarrow (\text{int}(uv.x \cdot width)) \mod 16$;
$ty \leftarrow (\text{int}(uv.y \cdot height)) \mod 16$;
$row\_offset \leftarrow \left\lfloor \frac{ty}{8} \right\rfloor \cdot 128$;
$col\_offset \leftarrow \lfloor \frac{tx}{8} \rfloor \cdot 64$;
$sub\_tile\_offset \leftarrow (ty \mod 8) \cdot 8 + (tx \mod 8)$;
$offset \leftarrow mcu\_offset + row\_offset + col\_offset + sub\_tile\_offset$;
$color = decoded\_buffer[offset]$;

overhead, with the actual overhead likely being lower due to the existing bit cost of DC coefficients in standard JPEG compression. We exclude the quantisation and Huffman tables from the overhead calculation for texture decoding, as these tables remain constant across the entire texture or even across multiple textures, making their contribution to the overall overhead negligible.

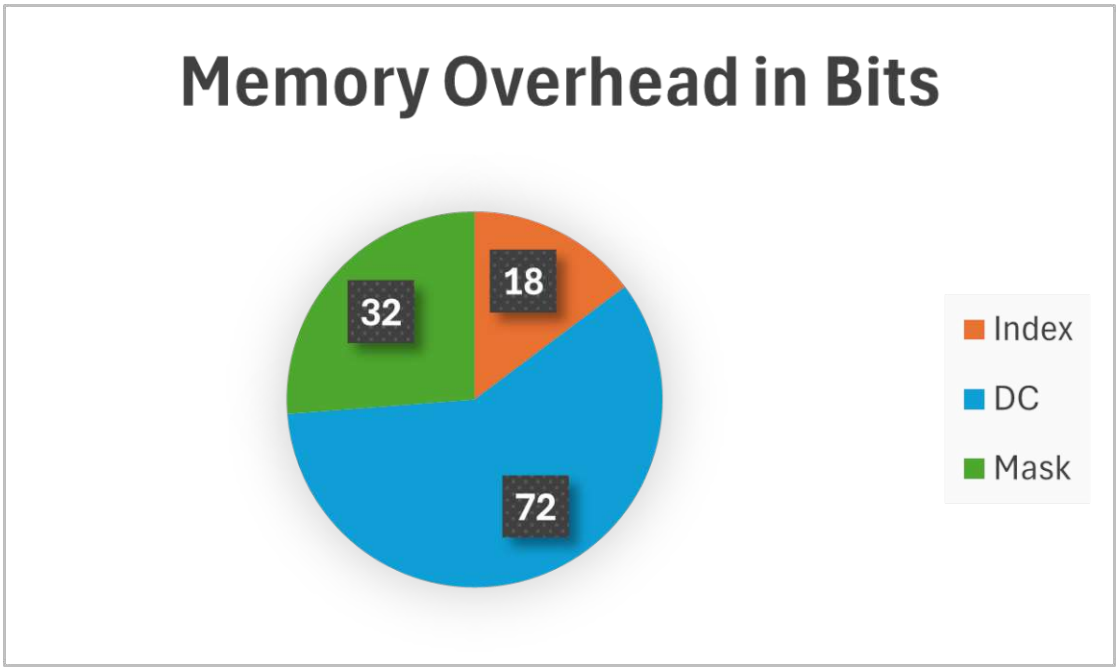


Figure 4.4: Breakdown of memory overhead per MCU to enable our GPU JPEG rendering. The primary contributor is the storage of 6 DC coefficients, which constitutes more than half of the overhead.

CHAPTER 5

Evaluation

To assess our method, we evaluate both the quality of the compression and its decoding efficiency. Quality is assessed by comparing visual fidelity and quantitative metrics across a range of test textures, while decoding efficiency is evaluated through performance benchmarks using GPU-based random-access decompression. By analysing these two aspects, we aim to demonstrate the strengths and limitations of our approach in comparison to widely used texture compression methods like BC1 and ASTC.

5.1 Quality

For the quality evaluation, we select 14 diverse RGB diffuse textures from Poly Haven [1] and ambientCG [2], a high-quality image of Pluto by NASA [3], as well as a few created on

[1] https://polyhaven.com/
[2] https://ambientcg.com/
[3] https://pluto.jhuapl.edu/Galleries/Featured-Images/

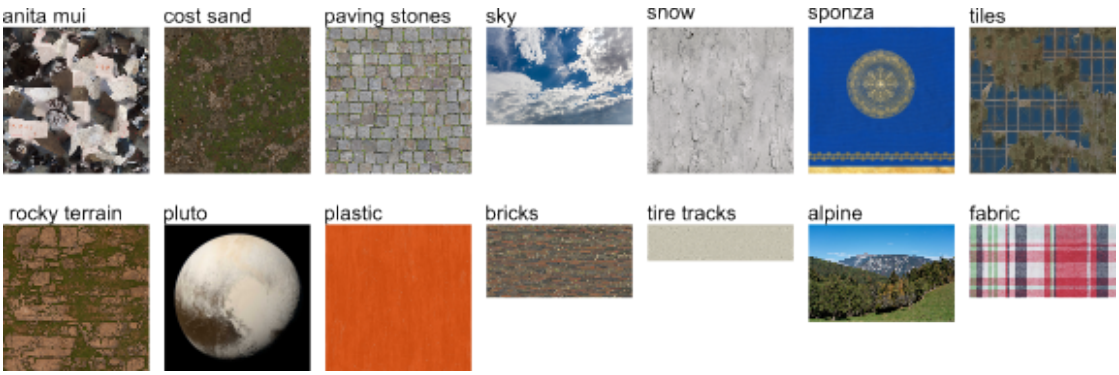


Figure 5.1: Overview of all textures used in our evaluation.

our own. The textures range from natural scenes, such as wood and rocky terrain, to more structured patterns, including brick walls, environmental photography, and texture atlases typically created by photogrammetry. An overview of all textures can be seen in Figure 5.1. The textures span resolutions from 1024×1024 up to 8192×8192. To accurately represent the overall memory footprint, we compute the memory overhead introduced by the pointer structures described in the previous section and include this in the final file size. In this Section we will first detail the tools that were used to compress these textures for the comparison, after which we explain the four different quality metrics, namely Peak signal-to-noise ratio (PSNR), structural similarity index measure (SSIM), FLIP [ANAM+20] and Learned Perceptual Image Patch Similarity (LPIPS) [ZIE+18], that were used in our evaluation.

5.1.1 Compression tools

Given the widespread adoption of the JPEG format, there exists a wide range of JPEG compression tools. As discussed in Section 3, several critical steps in the JPEG encoding pipeline—particularly the selection of the quantisation matrix—are left to the discretion of the implementer. To ensure consistency in our evaluation, we conducted a brief comparison of commonly used JPEG encoders before running our tests. Since our method requires JPEG images at various quality settings, we specifically focused on tools that support fine-grained quality control and enforce 4:2:0 chroma subsampling. Our investigation revealed that the majority of these tools rely on libjpeg [4] as their underlying encoding library. As a result, there were no notable differences in the quantisation strategies or output image quality among them, with all producing identical JPEG files for equivalent settings. Therefore, we are also using libjpeg for our image compression.

We compared the JPEG-compressed textures to two of the state-of-the-art compression formats. First, we chose BC1 because it remains the most widely used format. To compress images to BC1 we used NVIDIA's Texture-Tools-Exporter [5] version 2024.1.1. We disabled the option to generate mipmaps for a fair comparison and set the compression effort to "highest". All the other settings were left at their default level.

The second compression format used for our comparison is ASTC. While the hardware support for it is very sparse on desktop GPUs, it is one of the most advanced algorithms that generally achieves very good results for block-based compression methods. To compress it, we used the astc-encoder [6] version 5.3.0 with the "-thorough" quality preset as well as the low dynamic range colour profile for all our tests.

5.1.2 PSNR

Peak Signal-to-Noise Ratio (PSNR) is a metric used to quantify the similarity between two images. It is expressed in decibels (dB) and calculated based on the Mean Squared

[4] https://github.com/winlibs/libjpeg
[5] https://developer.nvidia.com/texture-tools-exporter
[6] https://github.com/ARM-software/astc-encoder

Error (MSE) relative to the maximum possible signal value, as shown in Equations 5.2 and 5.1.

$$\text{PSNR} = 10 \cdot \log_{10} \left(\frac{\text{MAX}^2}{\text{MSE}} \right) \tag{5.1}$$

$$\text{MSE} = \frac{1}{MN} \sum_{i=1}^{M} \sum_{j=1}^{N} (I(i,j) - K(i,j))^2 \tag{5.2}$$

A higher PSNR value typically indicates less loss of information and better visual quality in the processed image. However, while PSNR is a widely used objective metric, it may not always reflect human perception, as it does not account for the complexities of the human visual system in interpreting distortions. To calculate the PSNR we used the scikit-image python library [7].

5.1.3 SSIM

In contrast, SSIM considers human perception and measures error differently based on three components: luminance, contrast, and structure. As a result, SSIM is often considered a more meaningful metric compared to PSNR. However, research has shown that for JPEG-compressed images, the two metrics behave similarly [SAU19, Mar25].

5.1.4 LPIPS

Despite its advantages, SSIM does not fully model the complex processes underlying human visual perception. To address this limitation, LPIPS utilises deep features extracted from pre-trained neural networks to measure perceptual image similarity. By focusing on high-level semantic features, LPIPS provides a more robust and perceptually aligned image quality assessment.

5.1.5 FLIP

FLIP, developed by NVIDIA, is a perceptual image quality metric specifically designed to assess differences between rendered images and their corresponding ground truths in a way that aligns more closely with human visual perception. Unlike traditional metrics such as PSNR or SSIM, which often fail to capture perceptually relevant differences, FLIP takes into account factors like contrast sensitivity and spatial frequency to produce a more perceptually meaningful evaluation. In addition to computing a global error score, FLIP generates a visual error map that indicates the severity and spatial distribution of perceptual differences. In these maps, brighter regions correspond to areas where deviations from the reference image are more noticeable to the human eye.

[7] https://scikit-image.org/

5.1.6 Comparison

In contrast to BC1, JPEG and ASTC provide flexibility in the compression rate applied to images. Therefore, to guarantee a fair comparison, the JPEG quality settings and the ASTC block size are adjusted to match the PSNR of the BC1-compressed textures as closely as possible. Figure 6.2 illustrates the visual differences between the methods. Additionally, the textures are compressed to utilise an equal number of bits per texel and are then evaluated a second time. The corresponding results are presented in Table 5.1. The bits per texel are calculated by dividing the number of bits of the compressed texture by the number of texels it contains and are therefore for all three channels combined. Moreover, we conducted a second comparison at only 1 bit per texel in Table 5.2. Since BC1 compression can only achieve four bits per texel, we did not include it in this comparison.

Furthermore, we plot the PSNR for each quality setting and compare the results with BC1 and ASTC. The examples include a favourable case in Figure 5.2, average performance in Figure 5.3, the worst-case scenario in Figure 5.3, and an extraordinary case, as illustrated in Figure 5.4. We also plot the FLIP and LPIPS comparison in Figure 5.5 and Figure 5.6.

In addition, we plot the FLIP error maps for a mix of high and low frequency textures for BC1, ASTC and our method. Since the error of FLIP is between 0 and 1, we multiplied it by 255 to map it to PNG. The error maps can be seen in Figure 5.8.

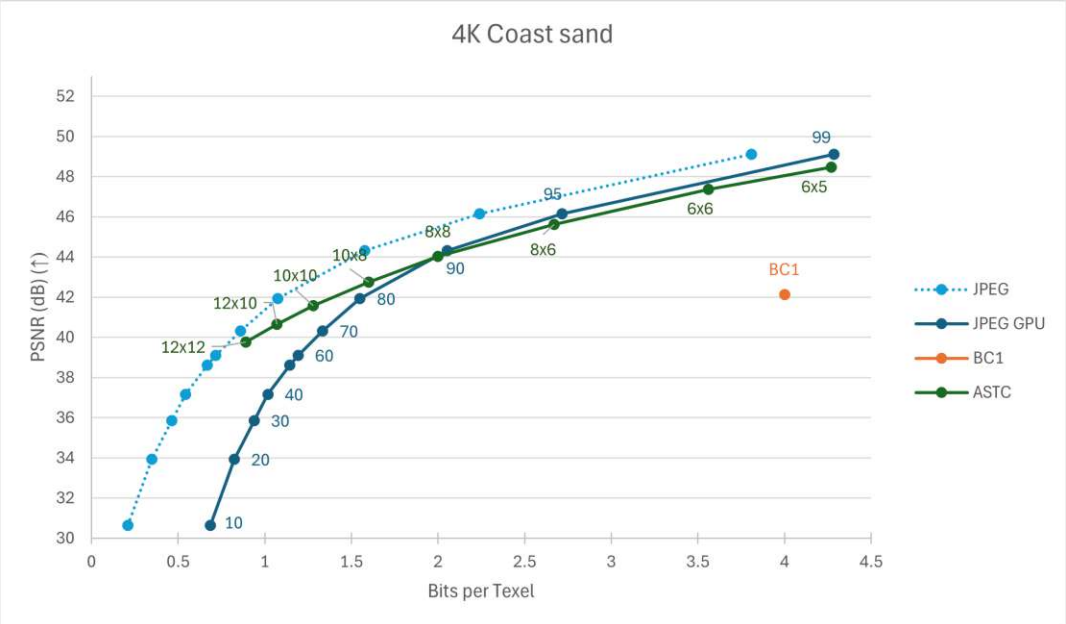


Figure 5.2: Comparison of the achieved PSNR at various bits per texel between JPEG, JPEG with the overhead for GPU decoding, BC1 and ASTC demonstrated using the 4096×4096 coast sand texture.

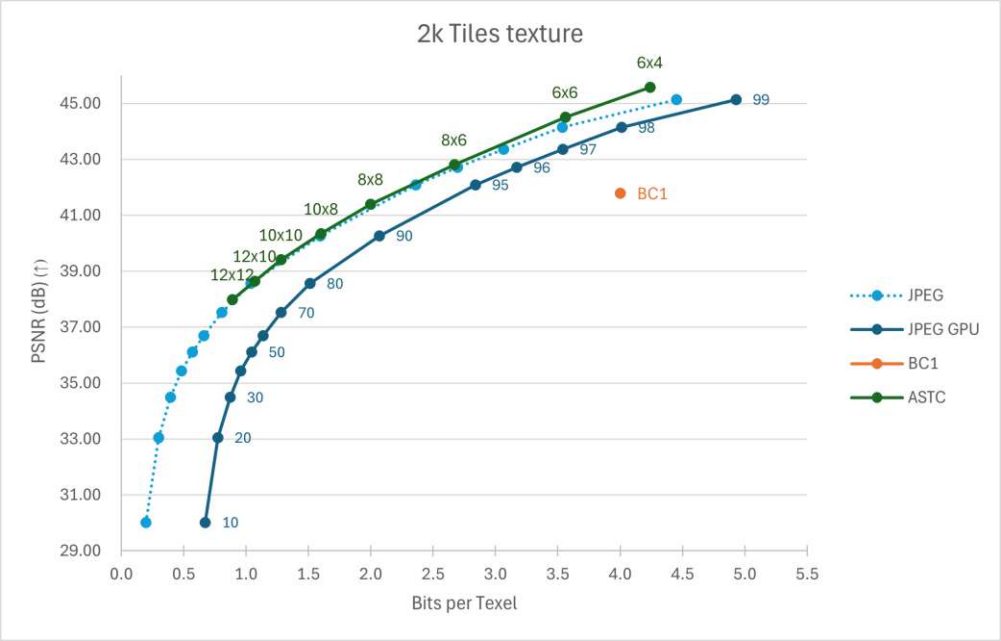


Figure 5.3: Comparison of the achieved PSNR at various bits per texel between JPEG, JPEG with the overhead for GPU decoding, BC1 and ASTC demonstrated using the 2048×2048 tiles texture.

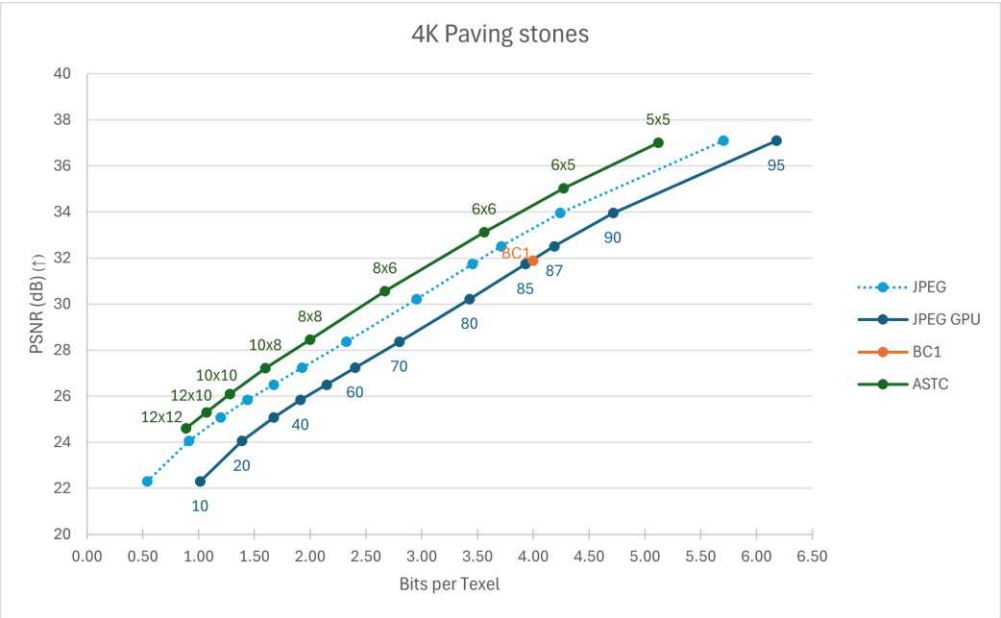


Figure 5.4: Comparison of the achieved PSNR at various bits per texel between JPEG, JPEG with the overhead for GPU decoding, BC1 and ASTC demonstrated using the 4096×4096 paving stone texture.

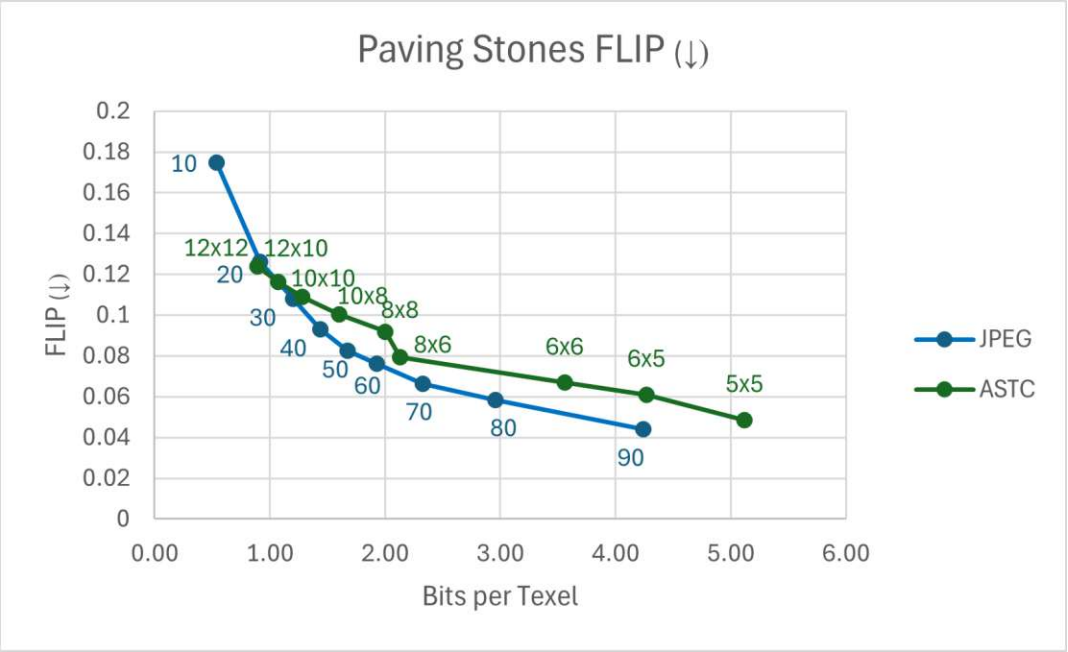


Figure 5.5: Comparison of the achieved FLIP at various bits per texel between JPEG and ASTC demonstrated using the 4096×4096 paving stone texture.

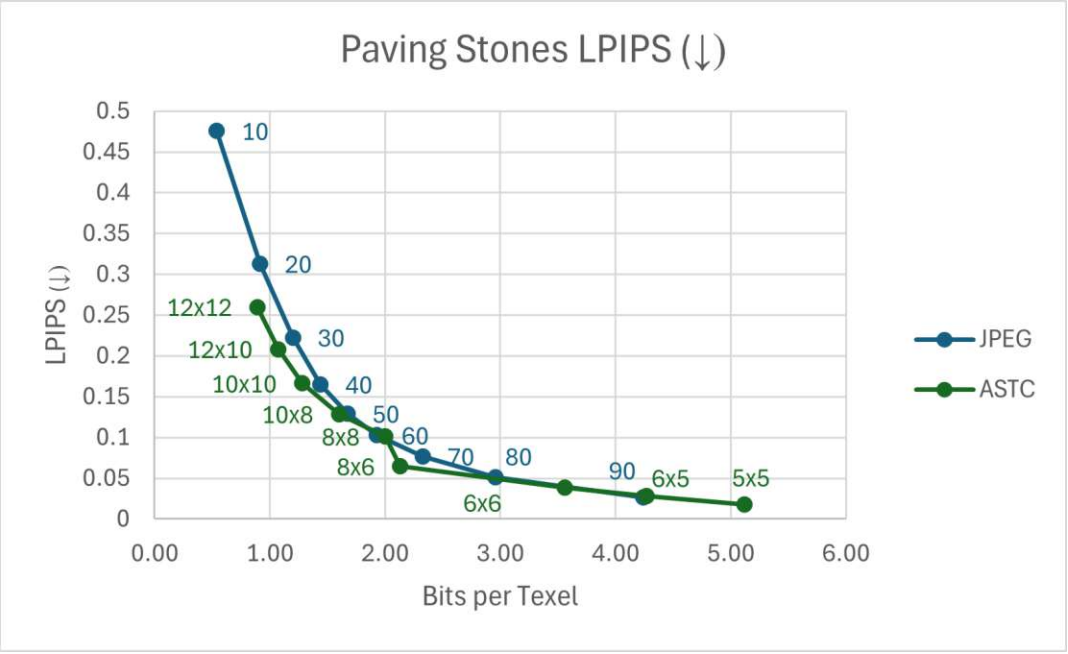


Figure 5.6: Comparison of the achieved LPIPS at various bits per texel between JPEG and ASTC demonstrated using the 4096×4096 paving stone texture.

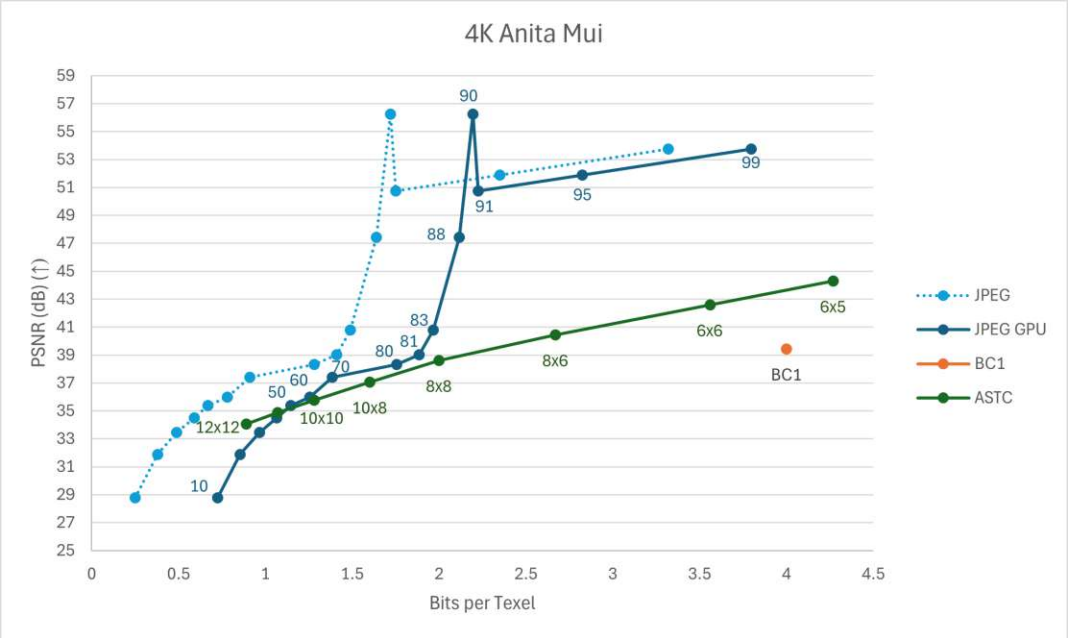
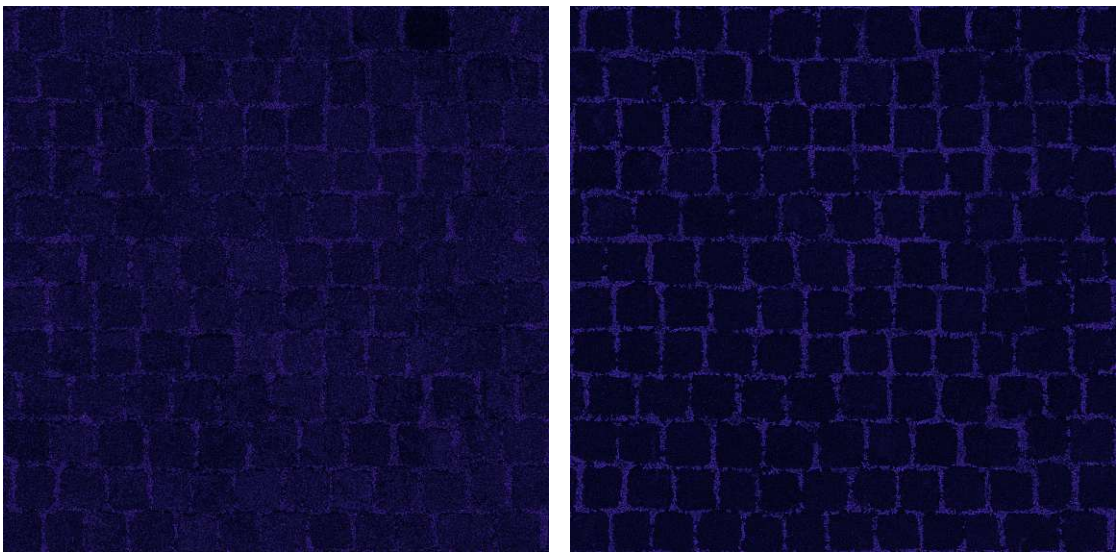


Figure 5.7: Comparison of the achieved PSNR at various bits per texel between JPEG, JPEG with the overhead for GPU decoding, BC1 and ASTC demonstrated using the 4096×4096 Anita Mui texture.



(a) ASTC FLIP (↓) (0.0609), 4.27 bpp (b) JPEG FLIP (↓) (0.0439), 4.72 bpp

Figure 5.8: Comparison between ASTC and JPEG using FLIP error maps for the paving stones texture. Brighter regions indicate areas where visual differences are more perceptible to the human eye. In the JPEG-compressed images, most of the error is concentrated in the high-frequency grass regions. In contrast, ASTC exhibits more uniformly distributed errors, but at an overall lower quality across the image.

Table 5.1: Comparison of average PSNR, SSIM, FLIP, and LPIPS metrics for all textures, evaluated at an approximate rate of four bits per texel. Results are shown for BC1, ASTC and JPEG.

		Average	anita mui	coast sand	paving stones	sky
BC1	PSNR (↑)	**38.75**	39.43	42.14	31.89	44.19
	SSIM (↑)	**0.95**	0.9781	0.9754	0.967	0.9747
	FLIP (↓)	**0.039**	0.0339	0.0308	0.0598	0.0343
	LPIPS (↓)	**0.027**	0.0316	0.0457	0.0278	0.0227
	bpp	**4.0**	4.0	4.0	4.0	4.0
JPEG GPU	PSNR (↑)	**45.31**	53.75	49.12	33.95	50.89
	SSIM (↑)	**0.989**	0.9985	0.9945	0.9776	0.9939
	FLIP (↓)	**0.024**	0.016	0.0171	0.0439	0.0142
	LPIPS (↓)	**0.008**	0.001	0.0048	0.0268	0.0025
	bpp	**4.33**	3.80	4.38	4.72	4.13
ASTC	PSNR (↑)	**42.973**	44.30	48.47	35.03	48.63
	SSIM (↑)	**0.987**	0.9939	0.9941	0.9842	0.9906
	FLIP (↓)	**0.029**	0.0210	0.0177	0.0609	0.0207
	LPIPS (↓)	**0.009**	0.0048	0.0048	0.0285	0.0086
	bpp	**4.27**	4.27	4.27	4.27	4.27

		snow	sponza	tiles	rocky terrain	pluto
BC1	PSNR (↑)	42.93	29.75	41.79	40.61	41.76
	SSIM (↑)	0.9711	0.9498	0.9719	0.9683	0.9831
	FLIP (↓)	0.0381	0.0579	0.0301	0.0343	0.0179
	LPIPS (↓)	0.077	0.0112	0.0206	0.0228	0.0246
	bpp	4.0	4.0	4.0	4.0	4.0
JPEG GPU	PSNR (↑)	47.31	38.38	44.14	45.79	48.44
	SSIM (↑)	0.9885	0.9939	0.9828	0.9886	0.9937
	FLIP (↓)	0.0311	0.0297	0.0169	0.0187	0.0101
	LPIPS (↓)	0.0208	0.0021	0.0066	0.0073	0.0046
	bpp	4.49	4.67	4.10	4.69	3.28
ASTC	PSNR (↑)	48.63	32.23	45.57	44.47	47.17
	SSIM (↑)	0.9917	0.9707	0.9873	0.9857	0.9944
	FLIP (↓)	0.0207	0.0557	0.0196	0.0255	0.0101
	LPIPS (↓)	0.0051	0.0109	0.0071	0.0087	0.006
	bpp	4.27	4.27	4.27	4.27	4.27

Table 5.1: Continuation from previous page.

		plastic	bricks	tire tracks	alpine	fabric
BC1	PSNR (↑)	41.89	35.60	40.45	35.92	34.240
	SSIM (↑)	0.9681	0.9594	0.9671	0.9739	0.697
	FLIP (↓)	0.0363	0.0425	0.0388	0.0388	0.054
	LPIPS (↓)	0.0142	0.0216	0.0242	0.0058	0.031
	bpp	4.0	4.0	4.0	4.0	4.0
JPEG GPU	PSNR (↑)	48.89	37.56	44.36	51.42	40.280
	SSIM (↑)	0.9945	0.963	0.986	0.9991	0.991
	FLIP (↓)	0.0275	0.0304	0.0296	0.0087	0.039
	LPIPS (↓)	0.0022	0.0178	0.007	0.0015	0.008
	bpp	4.04	4.84	4.35	4.82	4.36
ASTC	PSNR (↑)	46.97	37.79	43.18	40.50	38.68
	SSIM (↑)	0.9901	0.9744	0.9826	0.9917	0.989
	FLIP (↓)	0.0223	0.0369	0.0268	0.0261	0.0397
	LPIPS (↓)	0.0032	0.0155	0.0106	0.0114	0.007
	bpp	4.27	4.27	4.27	4.27	4.27

Table 5.2: Comparison of average PSNR, SSIM, FLIP, and LPIPS metrics for all textures, evaluated at an approximate rate of one bit per texel. Results are shown for ASTC texture compression and JPEG.

		Average	anita mui	coast sand	paving stones	sky
JPEG GPU	PSNR (↑)	**36.41**	37.85	41.93	25.07	42.65
	SSIM (↑)	**0.92**	0.9648	0.9748	0.8227	0.9589
	FLIP (↓)	**0.054**	0.0379	0.0339	0.108	0.0275
	LPIPS (↓)	**0.07**	0.039	0.0380	0.2225	0.0304
	bpp	**1.07**	0.96	1.10	1.2	1.2
ASTC	PSNR (↑)	**35.14**	34.89	40.65	25.30	42.08
	SSIM (↑)	**0.916**	0.9503	0.9663	0.8271	0.9539
	FLIP (↓)	**0.057**	0.0451	0.04	0.116	0.0323
	LPIPS (↓)	**0.09**	0.0553	0.0637	0.2083	0.0840
	bpp	**1.07**	1.07	1.07	1.07	1.07

		snow	sponza	tiles	rocky terrain	pluto
JPEG GPU	PSNR (↑)	42.2	24.52	38.56	38.17	44.09
	SSIM (↑)	0.9627	0.8015	0.9441	0.9432	0.9854
	FLIP (↓)	0.0499	0.0961	0.0338	0.0466	0.0164
	LPIPS (↓)	0.0424	0.105	0.0619	0.0846	0.0226
	bpp	1.06	1.08	1.04	1.06	1.05
ASTC	PSNR	41.56	23.78	38.65	37.91	40.36
	SSIM	0.9627	0.7824	0.9439	0.9388	0.9785
	FLIP	0.0338	0.1028	0.037	0.0497	0.021
	LPIPS	0.051	0.1749	0.069	0.0609	0.05
	bpp	1.07	1.07	1.07	1.07	1.07

		plastic	bricks	tire tracks	alpine	fabric
JPEG GPU	PSNR (↑)	37.99	30.92	34.47	39.05	32.29
	SSIM (↑)	0.9211	0.8572	0.8523	0.9867	0.9493
	FLIP (↓)	0.052	0.0632	0.0571	0.0468	0.091
	LPIPS (↓)	0.0391	0.1329	0.1189	0.03	0.0628
	bpp	0.93	1.2	0.94	0.96	1.21
ASTC	PSNR (↑)	38.68	30.13	34.65	32.64	30.63
	SSIM (↑)	0.9279	0.8494	0.8585	0.9488	0.93
	FLIP (↓)	0.0452	0.0728	0.0451	0.055	0.097
	LPIPS (↓)	0.0482	0.1468	0.1151	0.0767	0.0668
	bpp	1.07	1.07	1.07	1.07	1.07

5.2 Performance

To evaluate our method, we implemented the proposed random-access JPEG decompression algorithm in CUDA and measured kernel execution times on the GPU using two test scenes. Since the algorithm executes entirely on the GPU, our primary performance metric is kernel run time, excluding any CPU-side processing.

5.2.1 Scenes

We conducted experiments on two distinct scenes. The first, Crytek Sponza (Figure 5.9a) [McG17], features 25 textures, each with a resolution of 1024×1024. These textures cover a range of surface types, including patterned rugs, tiled floors, flower pots and stone columns, spanning both large, flat surfaces and thin or detailed geometry. To further increase the scene's complexity, we disabled object instancing to reuse textures, resulting in a total of 103 textures being uploaded to the GPU.

The second scene is a minimal test case intended to isolate decompression performance. It consists of a simple textured cube (Figure 5.9b) that uses a single large texture of resolution 4096×4096. Unlike the Sponza scene, which involves decoding smaller and larger parts of multiple textures, the cube scene forces the decoder to process an entire texture in full. The texture used for our measurements is the rocky terrain texture, but other textures compressed at similar bits per texel scored similar results.

5.2.2 Results

Both scenes were rendered at a resolution of 1920×1080 using an NVIDIA GeForce RTX 4090 GPU. To assess performance under different compression conditions, each scene was rendered four times using JPEG textures compressed at quality levels 30, 50, 70, and 90, all with 4:2:0 chroma subsampling. The resulting GPU kernel execution times for each configuration are summarised in Table 5.3. For 4:2:0 subsampling, a single MCU consists of 16×16 texels, amounting to 256 texels. This results in approximately 17.6 million texels decoded per frame for the Sponza scene and 16.9 million texels for the Cube scene.

To compare our method against state-of-the-art texture compression, we implemented BC1 decompression in software. While modern GPUs natively support BC1 decoding in hardware, this functionality is not exposed through CUDA. A software implementation, therefore, also provides a fairer basis for comparison. On the NVIDIA GeForce RTX 4070 Ti, rendering with either multiple textures in the Sponza scene or a single high-resolution texture in the cube scene took an average of 0.07 ms. We did not compare against ASTC, as it is unsupported on desktop GPUs and too complex to reimplement efficiently.

To further analyse the impact of MCU complexity on decoding performance, we rendered the Sponza scene at four JPEG quality levels: 90, 75, 50, and 30. This experiment was conducted on an NVIDIA GeForce RTX 4070 Ti GPU, which explains the slightly lower performance compared to the results reported above. For each quality level, we measured the average number of non-zero DCT coefficients per DU, which reflects the

decoding workload. At quality 90, the average was 14 coefficients per DU in the Cube scene and 19 in the Sponza scene, decreasing to only 3 coefficients for both scenes at quality 30. We then measured decoding time while varying the number of visible MCUs in the scene, allowing us to directly assess how performance scales with both MCU count and coefficient complexity across quality levels. The results are summarised in Figure 5.10 and Figure 5.11, where each curve corresponds to one of the tested quality settings.

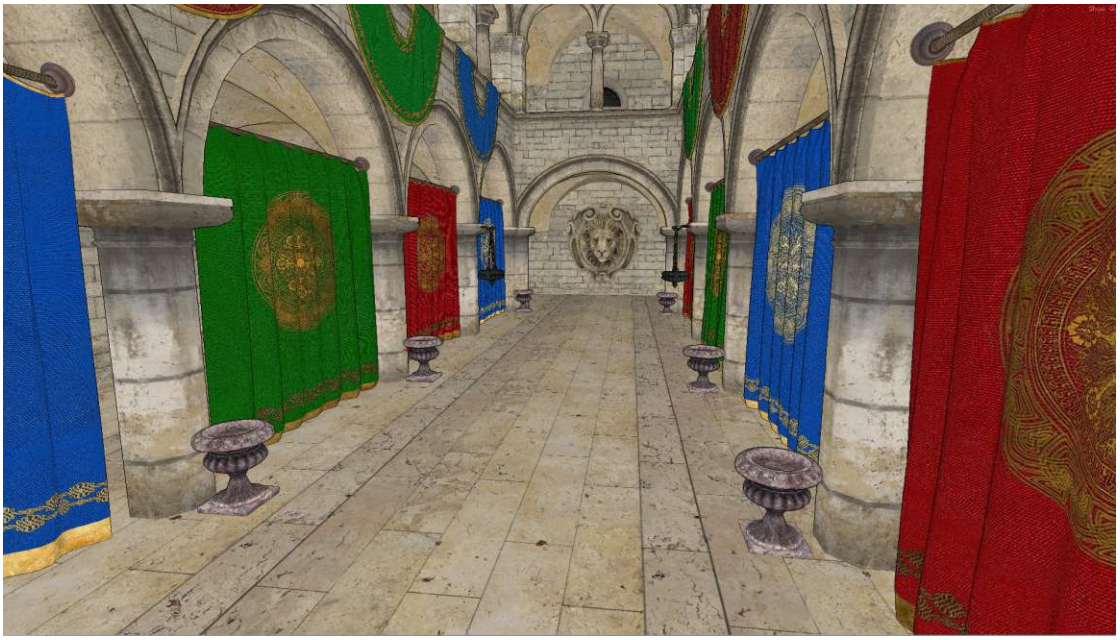
Additionally, we plotted the average time required to decode a single MCU for varying numbers of DCT coefficients in Figure 5.12. The average time was calculated by decoding between 100 and 60,000 MCUs, then dividing the total decoding time by the number of MCUs. By averaging across this range, we ensure that the reported cost reflects the actual decoding efficiency of an MCU across all scenarios.

To analyse potential overhead from rendering with multiple textures, we divided the Cube scene into equally sized regions, each assigned a copy of the same texture. Using identical textures ensures that any observed differences are solely due to the number of textures, without interference from texture content. The results for the rocky terrain texture at quality setting 70 are shown in Figure 5.13.

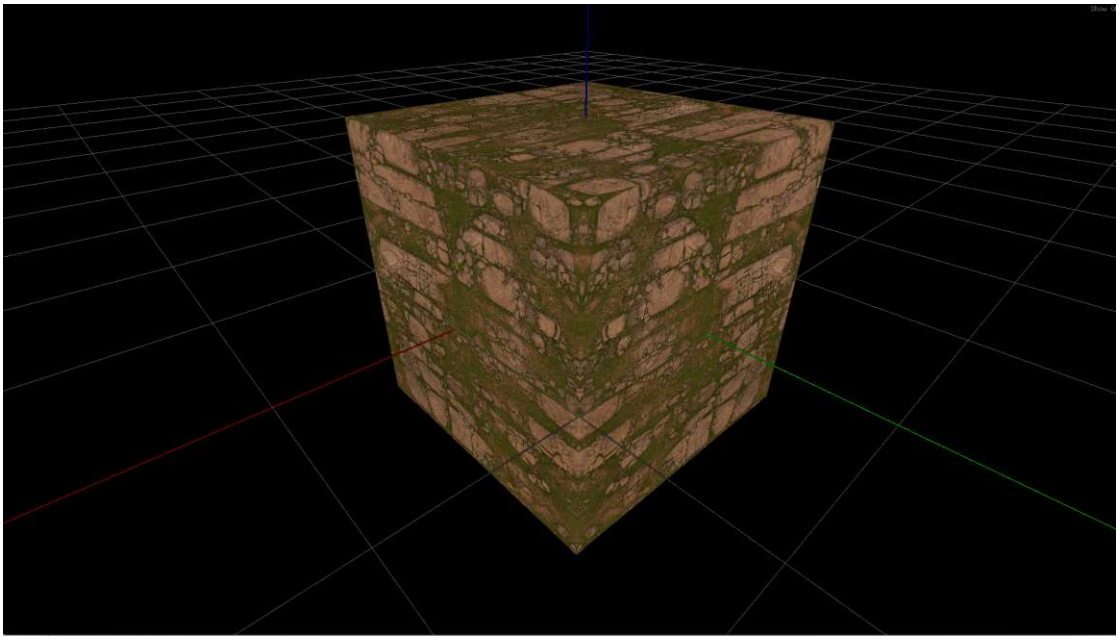
Table 5.3: Decompression performance and quality (PSNR) for 69,000 and 66,000 MCUs for the Sponza and Cube scenes with an NVIDIA GeForce RTX 4090.

Scene	**Compression**	**Quality**	**bpp**	**Size (MB)**	**Decode Time (ms)**	**PSNR (↑)**
Sponza (69k MCUs)	JPEG	90	4.00	51	1.54	38
	JPEG	70	2.10	27	0.8	28.4
	JPEG	50	1.52	19	0.6	26.3
	JPEG	30	1.08	14	0.33	24.5
	BC1	–	4.00	51	0.07	29.75
Cube (66k MCUs)	JPEG	90	1.87	3.72	0.92	41
	JPEG	70	0.98	1.95	0.56	38.8
	JPEG	50	0.70	1.37	0.43	35.9
	JPEG	30	0.48	0.96	0.34	34.1
	BC1	–	4.00	8.00	0.07	40.61

Figure 5.9: Sponza and cube scene rendered with JPEG compressed textures.



(a) Sponza



(b) Cube

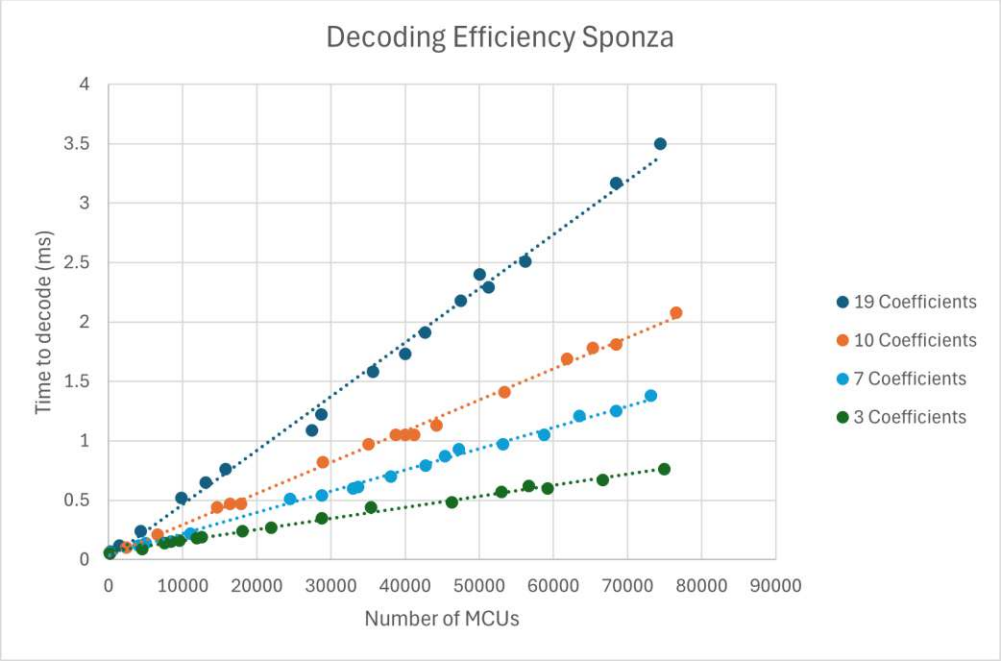


Figure 5.10: Coding efficiency as a function of the number of MCUs for different average number of Coefficients measured on an NVIDIA GeForce RTX 4070ti for the Sponza scene.

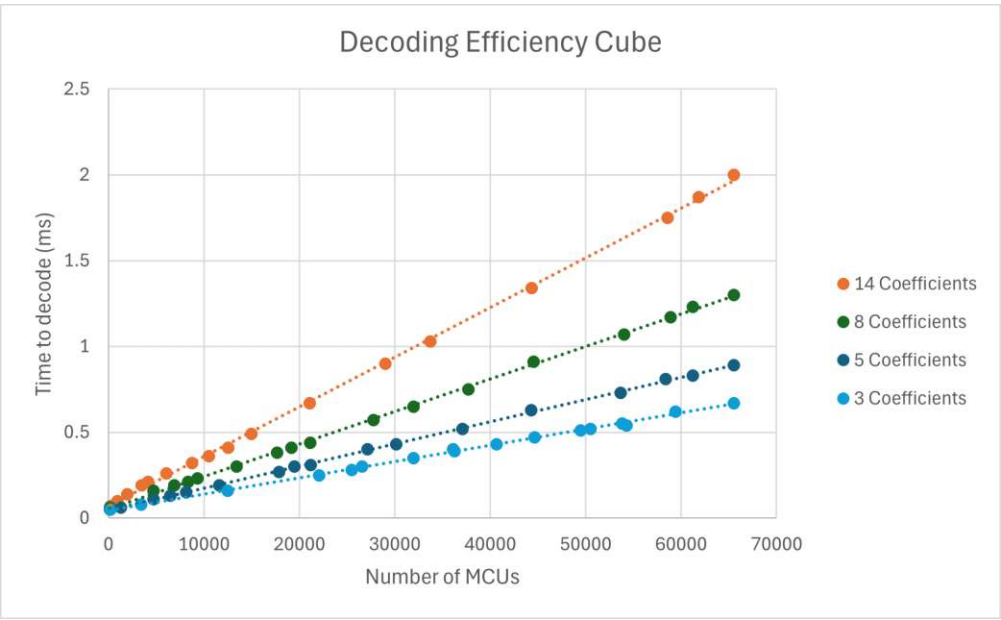


Figure 5.11: Coding efficiency as a function of the number of MCUs for different average number of Coefficients measured on an NVIDIA GeForce RTX 4070ti for the Cube scene.

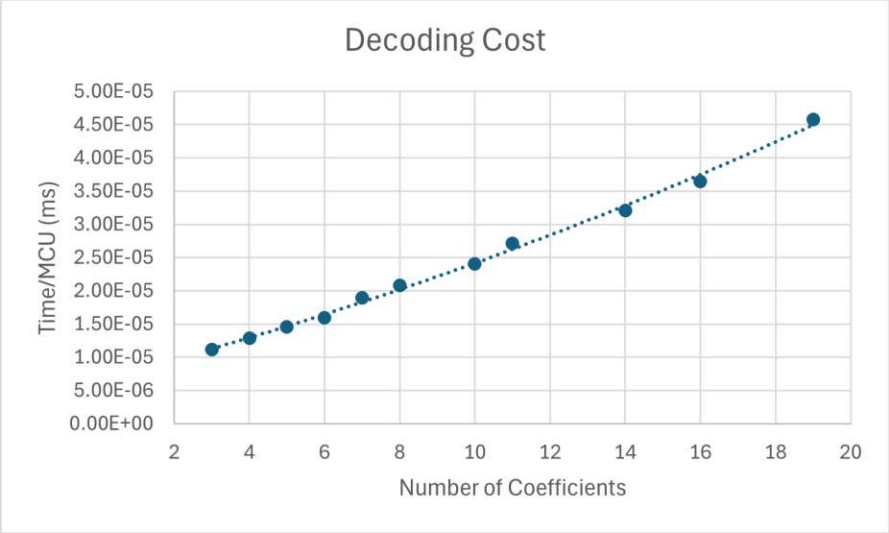


Figure 5.12: Average MCU decoding time on an NVIDIA GeForce RTX 4070ti as a function of the average number of DCT coefficients at different quality levels for the rocky terrain texture.

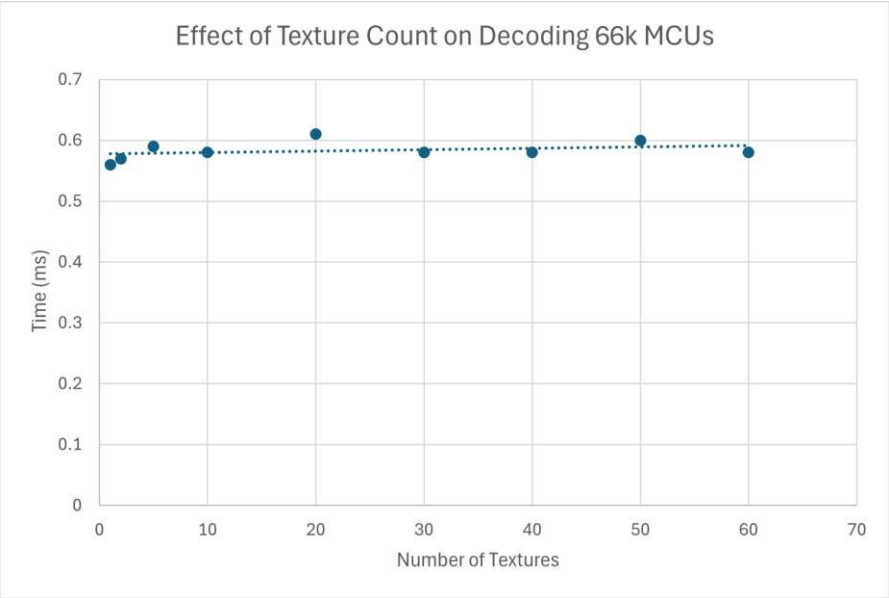


Figure 5.13: Rendering of the Cube scene on an NVIDIA GeForce RTX 4070 Ti, using multiple copies of the rocky terrain texture. The scene was divided into equally sized regions, each assigned to a different texture, while keeping the total number of MCUs constant.

CHAPTER 6

Discussion

In this chapter, we evaluate our results in terms of performance and visual quality. The first section analyses metrics of visual fidelity, comparing JPEG with state-of-the-art formats. We then explore specific cases where JPEG performs particularly well—so-called "sweet spots"—especially in the context of pre-compressed textures and photogrammetry-based assets. Following this, we present a detailed comparison of the visual error introduced by JPEG and BC1 across a variety of textures, using close-up examples to highlight qualitative differences. We then discuss the decoding efficiency of our method for our two test scenes, before concluding with an in-depth analysis of MCU-utilisation.

6.1 Quality

At a target rate of four bits per texel, JPEG consistently outperforms BC1 across all evaluated textures and quality metrics, as shown in Table 5.1. It also surpasses ASTC in terms of PSNR, and although the margin is narrower, JPEG maintains an advantage in perceptual metrics such as SSIM, FLIP, and LPIPS. There are, however, specific textures where ASTC yields higher PSNR and SSIM scores than JPEG, while still performing worse or even in perceptual evaluations. A notable example is the paving stones texture. As illustrated in Figure 5.4, JPEG struggles with this texture, even before factoring in any overhead introduced by our processing pipeline. Nonetheless, this disadvantage is less pronounced in LPIPS (Figure 5.6), and JPEG even outperforms ASTC in the FLIP metric (Figure 5.5).

A closer look at the error map for the paving stones texture (Figure 5.8) shows that JPEG compression introduces noticeable error in high-frequency regions such as grass. However, JPEG achieves significantly lower error across smoother areas of the image, leading to competitive overall perceptual quality. In contrast, ASTC distributes its error more uniformly across the entire texture, which cumulatively results in a higher perceptual error. In the more demanding scenario of one bit per texel, JPEG retains its edge over

ASTC, though the margin narrows. BC1 is excluded from this comparison, as it does not support compression below four bits per texel.

Moreover, JPEG offers the greatest flexibility in terms of quality tuning. Unlike BC1 or ASTC, JPEG's adjustable quantisation matrix and wide range of quality settings—typically up to 100 levels in standard tools—allow for fine-grained control over the trade-off between size and visual fidelity. This tunability makes JPEG especially appealing for applications where precise optimisation is required.

6.2 Quality metric anomalies

In general, JPEG's quality setting scales predictably with perceived image quality, as measured by standard evaluation metrics. However, for certain textures, the relationship between the quality setting and the actual measured quality exhibits unexpected behaviour, as the trend is non-monotonic. In these cases, quality metrics show a distinct peak at a specific JPEG quality value, with noticeably worse results at both lower and higher settings.

This phenomenon is clearly demonstrated in the Anita Mui texture (Figure 5.7), where a quality setting of 90 consistently outperforms all others. Surprisingly, settings both above and below 90 result in a rapid decline in quality across all four metrics evaluated. This behaviour is caused by double JPEG compression: the texture was previously saved using JPEG at quality level 90, and recompressing it at any other setting introduces additional quantisation errors. When recompressed at the same quality level as the original, JPEG can partially preserve the quantised coefficients, leading to a distinct spike in measured quality.

However, in the case of the Anita Mui texture, an additional anomaly emerges: despite being double-compressed at the same quality level, the resulting file size is still noticeably reduced. This is unexpected—typically, recompressing a JPEG at the same quality setting yields similar file sizes but reduced visual fidelity. One possible explanation is that the Anita Mui texture originates from photogrammetry, a process that involves stitching together multiple photographs to generate a panoramic or 3D surface texture. If the original source images used in this stitching process were themselves JPEG-compressed, their compression artifacts may have interacted with the second compression pass in a way that led to more aggressive quantisation and, consequently, smaller file sizes. This layered compression history may explain both the unusual quality spike and the reduction in file size, but needs further investigation.

Notably, this behaviour disappears when the texture is padded with pure white pixels along the left or top edge. However, once the padding reaches a multiple of 8 pixels—the size of JPEG's internal block structure—the quality spike reappears. This pattern highlights the sensitivity of JPEG's block-based architecture to image alignment during recompression. These findings complicate direct comparisons between JPEG and other compression algorithms, particularly when the input textures have already undergone

lossy compression. They also point toward the need for careful dataset curation and preprocessing when benchmarking texture compression methods.

6.3 Visual comparison

Figure 6.2 shows that typical JPEG artifacts are hardly noticeable at the quality levels commonly used for texture compression. Only for the tiles' texture, a slight artifact is visible where the blue hues from the tiles bleed into the separating lines. This occurs in areas where compression introduces minor colour leakage, but such issues are rare and typically confined to textures with sharp contrasts or highly detailed patterns.

6.4 Performance

While Table 5.3 shows that BC1 leads in raw decoding speed, our method still decodes the Cube scene in almost half a millisecond—using only a quarter of BC1's storage—while maintaining nearly identical visual quality. For the Sponza scene, our approach reduces storage by about half compared to BC1, yet remains well below the one-millisecond decoding threshold. Thus, even in scenes where JPEG offers less benefit from compression, our method sustains high performance.

Figure 5.10 and Figure 5.11 show that for all four cases, the decoding time increases approximately linearly with the number of MCUs. This indicates that the decoding process has a predictable, proportional cost per MCU, with minimal overhead. The slope of each line is directly related to the average number of coefficients per DU. With 19 coefficients, the decoding time grows the fastest, while with 3 coefficients, it grows the slowest. This demonstrates that decoding cost is dominated by the number of coefficients rather than by fixed per-MCU overhead. In the Cube scene at 65,000 MCUs, decoding with only 3 coefficients takes approximately 0.75 ms, while decoding with almost five times as many, at 14 coefficients, takes 2.0 ms, which is only 2.6 times slower.

When comparing the decoding efficiency plots of both scenes, we observe that decoding the same number of MCUs with a similar number of coefficients per DU takes approximately the same amount of time. This suggests that, in our method, rendering a scene with multiple textures does not introduce a performance penalty; decoding speed is determined solely by the number of coefficients per DU and the total number of MCUs to be decoded.

Figure 5.12 shows that the cost of each additional coefficient remains nearly constant. We attribute the slight polynomial increase to the fact that more coefficients require a larger Huffman table, since more symbols need to be encoded. As a result, even though our method can evaluate 32 Huffman codes in parallel, it may still require two or three extra iterations until it finds the matching codeword. However, besides having to decode additional coefficients sequentially, all other steps of our algorithm are unaffected by the number of coefficients, which explains why the plot appears almost linear.

Our multi-texture comparison shows that rendering from a single texture versus multiple textures introduces no measurable overhead. The minor fluctuations observed are likely attributable to GPU variability. This demonstrates that increasing memory usage from 1.95 MB for a single texture to 117 MB for multiple textures has no impact on performance.

Summarising, the performance of our method is governed primarily by the number of coefficients, which directly reflects the chosen compression quality. Higher compression yields fewer stored coefficients and, therefore, faster decoding. The second determining factor is the number of MCUs, with decoding time scaling linearly and predictably with both coefficients and MCUs. Importantly, neither the number of textures nor the resolution of the render target impacts performance. This is because once an MCU is decoded, it can be reused across all pixels that reference it with minimal overhead, ensuring consistent efficiency regardless of scene complexity.

6.5 MCU-Utilization

It is important to note that mipmapping was not implemented for the decoding efficiency evaluation. Consequently, the number of texels decoded per block is suboptimal in regions farther from the camera, as illustrated in Figure 6.1. This also explains why the number of MCUs that must be decoded is so high. For example, in the Sponza scene, we decode 66,000 MCUs (Table 5.3), which corresponds to 16.83 million texels. At a resolution of $1920 \times 1080 = 2{,}073{,}600$ pixels, this means that each MCU contributes, on average, to only 31.42 pixels. In other words, just 12.3% of the decoded texels are actually utilised per MCU. For the cube scene, utilisation is even worse, since the cube does not fill the full screen but still requires a similar number of MCUs to be decoded. Despite this low utilisation, we still achieve respectable decompression performance, particularly at lower quality settings, mainly due to the reduced number of coefficients requiring Huffman decoding.

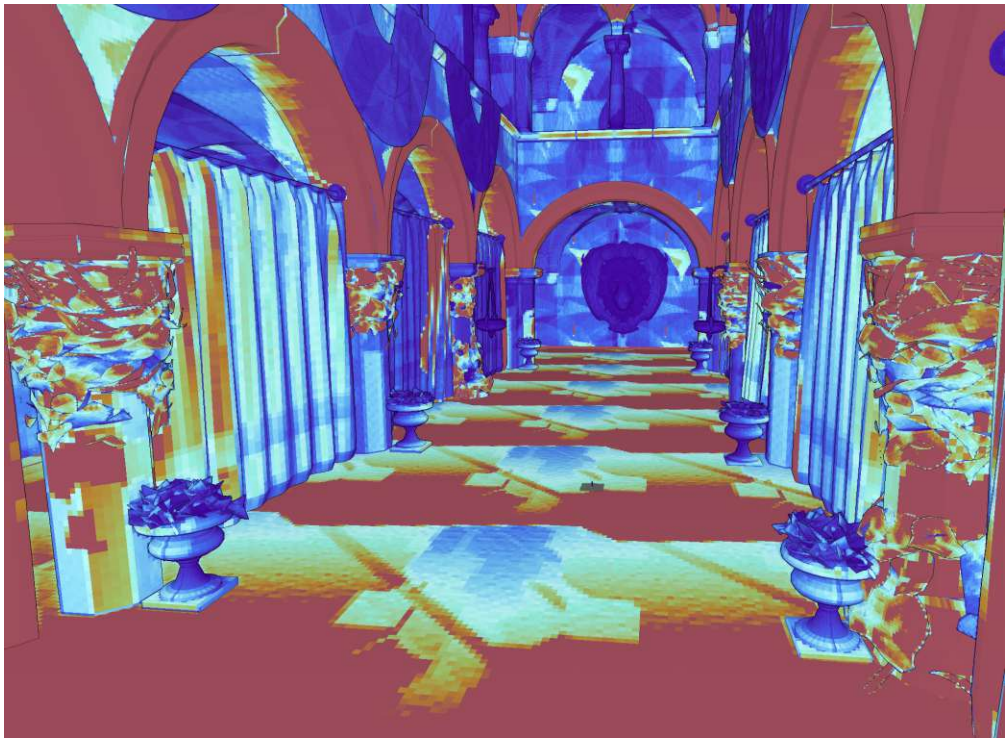


Figure 6.1: MCU-Utilisation: Each pixel is coloured by how many of the corresponding MCU's texels are actually used. Red: This pixel's colour is sourced from an MCU where all its texels are used. Blue: Only a single texel of the corresponding MCU made it into the final image.

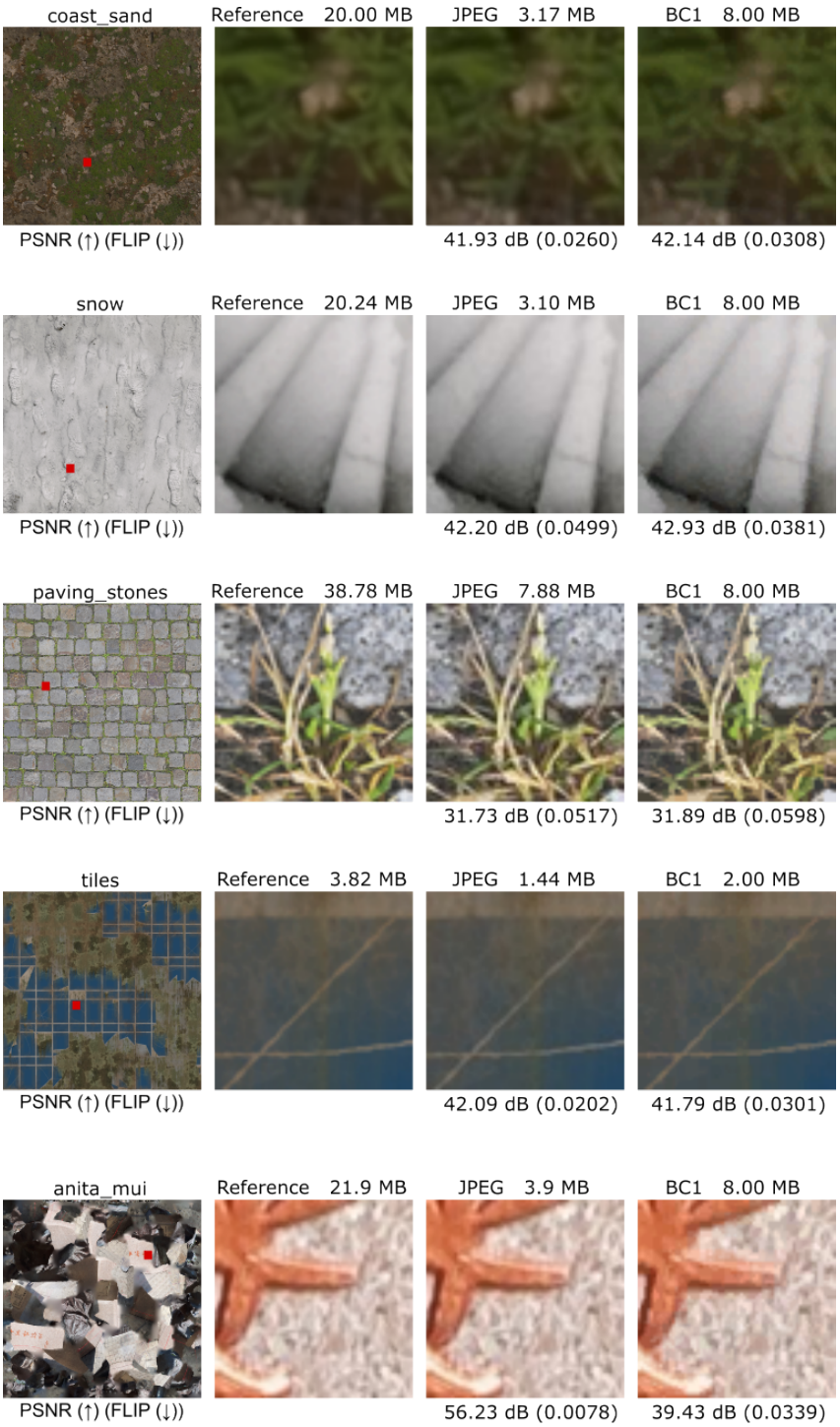


Figure 6.2: Close-ups of our method and BC1 compression with similar PSNR. In the best case, our compressed textures are only half the size of BC1.

CHAPTER 7

Future Work

While the current implementation does not explore every possible avenue—partly due to constraints of the chosen framework—it successfully lays a solid foundation for future development. This opens up exciting opportunities to build upon our work, particularly through the integration of variable-rate compression algorithms. With further refinements and extensions, both architectural and algorithmic, there is strong potential to achieve substantial improvements in performance, visual quality, and adaptability in real-time rendering scenarios.

7.1 Mipmapping

As discussed in the previous section, one of the most significant features currently missing from our implementation is mipmapping. Mipmapping serves two primary purposes in texture mapping. First, it helps mitigate aliasing artefacts that occur when rendering objects at a distance. At such distances, the rate of change in UV coordinates between neighbouring pixels increases, often leading to undersampling and visible high-frequency noise. Mipmapping addresses this by progressively downsampling the texture—typically averaging four texels at each level—until the texture is reduced to a minimal resolution, often as small as a single 2×2 block. This multilevel representation ensures that the appropriate texture resolution is sampled based on the object's screen-space size, effectively suppressing aliasing.

Secondly, mipmapping enhances rendering efficiency by allowing sampling from smaller texture levels, which improves the cache coherence of texture lookups. When objects are far away, the distance between texture lookups of neighbouring pixels can become so large that some texels are effectively skipped if sampling from the original resolution, leading to sparse and inefficient texture access. Using coarser levels of detail produces more localised and predictable memory access patterns, reducing cache misses and increasing throughput.

In the case of our JPEG rendering method, mipmapping can provide even more significant performance improvements, not only by enabling more coherent texture lookups, but also from more efficient utilisation of MCUs: Currently, an entire MCU comprising 16×16 texels may project to a single pixel (as shown by the blue pixels in Figure 6.1) so that only one of its texels will be visible, yet we still need to decode the entire MCU. With mipmapping, multiple MCUs projected to a few pixels or even a single pixel may be replaced by fewer MCUs covering a larger area of the framebuffer, thereby reducing the number of MCUs that need to be decoded.

For traditional texture compression schemes, the full mipmap chain typically adds about 33% additional memory overhead relative to the original texture. As an alternative to traditional mipmapping, future work could explore the possibility of generating lower-resolution representations of JPEG-compressed textures by decoding only a subset of the AC coefficients, or even just the DC coefficient. This approach would effectively discard high-frequency content, thereby reducing aliasing artifacts in a manner similar to mipmapping. However, the relationship between the degree of frequency reduction and the corresponding mip level would need to be carefully studied and formalised. If such a mapping can be established, this technique could offer a significant reduction in computational overhead at lower resolutions—without the need to explicitly store additional mipmap data.

7.2 Texture filtering

Texture lookups in real-time rendering rarely align perfectly with the centre of a texel. As a result, texture filtering techniques are employed to interpolate between neighbouring texels, thereby reducing aliasing artifacts and producing smoother visual results. This process, commonly referred to as texture filtering, is a fundamental feature that most texture compression formats must support to ensure high-quality rendering.

In our current implementation, which is based on a CUDA software rasterizer, sub-pixel precision for UV coordinates is not yet supported. Consequently, texture filtering is not currently possible within this framework. However, implementing it should be straightforward, as neighbouring pixels typically reside within the same MCU, avoiding the need for cross-block data access in most cases. Furthermore, Hollemeersch et al. [HPLVdW12] propose a technique for filtering directly in the frequency domain, which could provide an efficient solution for handling border texels, particularly when adjacent MCUs have not been decoded. This method may offer both performance and quality benefits in scenarios where spatial-domain filtering is impractical.

7.3 Texture specific quantisation tables

The quantisation tables used in JPEG compression are typically designed to provide a good balance between compression efficiency and image quality across a wide variety of images. However, these tables—originally developed decades ago—are now outdated

in light of modern hardware capabilities, the age of machine learning, and evolving application demands. Recent research has shown that redesigned quantisation matrices can improve compression quality by up to 2 dB for the same bitrate [AOS+17, SG25]. While these improvements are promising, we believe that real-time rendering offers opportunities to push this further.

Specifically, rather than designing quantisation matrices that generalise well across diverse images, tailoring the matrix to individual textures may yield significant quality gains, as demonstrated by Choi et al. [CH20] and Reich et al. [RDPC24]. In most graphics pipelines, texture compression is a one-time operation, making longer preprocessing times acceptable if they result in higher quality or better compression. Furthermore, since JPEG allows the quantisation matrix to be swapped independently of the rest of the encoding and decoding pipeline, this approach integrates naturally into existing workflows. During development, a standard matrix can be used for rapid iteration, while in the final production stage, a texture-specific matrix can be baked to maximise quality and efficiency.

7.4 Reduced memory overhead

The memory overhead introduced by our method could be further reduced. Currently, we store the DC coefficients using 12 bits each, resulting in up to 72 bits overall or 0.28 bits per texel. This is more than half of the overhead, as evident in Figure 4.7. One potential solution is to store only the first DC coefficient for each DU and entropy-encode the remaining ones, decoding them in parallel with similar to the AC coefficients. This would reduce the overhead to less than 0.2 bits compared to the standard JPEG compression.

7.5 Non-colour textures

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.

On a similar note, textures that do not store colour information but instead encode normals, height maps, or other forms of data typically have different requirements for compression. For example, in BC5, normal maps are stored in only two-channel textures that have twice the block size of BC1. These types of textures prioritise preserving the integrity of numerical data rather than perceptual visual quality. It would be interesting to explore how variable-rate compression techniques, such as those adapted in this work, can be extended or specialised to handle these non-colour textures.

7.6 JPEG XL

One promising direction is the JPEG XL [AvAB+19] standard, which builds on JPEG to not only support an alpha channel but also allows for up to 4099 additional channels. Moreover, it offers significantly better compression efficiency compared to baseline JPEG. However, this increased capability comes at the cost of greater format complexity, which may make an efficient GPU-based decoding implementation challenging. Nevertheless, the flexibility of JPEG XL makes it a strong candidate for future research on improving variable-rate texture compression and extending our approach to a broader range of textures.

CHAPTER 8

Conclusion

As real-time rendering advances toward photorealism, the size of texture data in graphic applications will only continue to rise. This creates a growing need for better compression algorithms that can balance image quality with high levels of compression. Current fixed-rate solutions do not fully utilise available storage, as they must adhere to the random-access constraint. In this thesis, we have shown that variable-rate texture compression is not only feasible but has the potential to challenge or even outperform state-of-the-art methods.

We have laid out the workings of JPEG—the most common variable-rate compression standard—and explained why it is typically considered unfit for texture compression. To address these challenges, we proposed a set of solutions: an indexing scheme to enable random access, a parallel lookup-based Huffman decoding method, and a modified deferred rendering pipeline. These solutions, drawing on both established literature and novel contributions, aim to enable efficient random access and high-performance decoding directly on the GPU.

Our GPU-optimised JPEG textures outperform BC1 and rival ASTC across several metrics in compression efficiency, while still maintaining real-time performance with decoding times under 1 ms per frame. Furthermore, JPEG's inherent flexibility in adjusting the compression rate remains unmatched by current state-of-the-art fixed-rate texture compression methods.

While our method already shows promising results, we have also presented several improvements that could further enhance the algorithm. The current implementation lacks support for mipmapping, texture filtering, and alpha channels, and the indexing structure introduces additional memory overhead. Nonetheless, these are not fundamental limitations but rather implementation gaps that can be addressed in future iterations.

Ultimately, this thesis challenges a longstanding assumption in computer graphics: that variable-rate formats, such as JPEG, are inherently unsuited for real-time applications. By

demonstrating real-time performance with a 33-year-old compression standard, this work opens the door to reimagining texture compression for modern demands. We encourage keeping an open mind and not ruling out options from the get-go, just because the consensus is that something is not possible. We hope this work provides critical insights into which aspects of JPEG are well-suited for GPU acceleration and which present ongoing challenges, thereby paving the way for the development of novel variable-rate formats specifically tailored to modern GPUs.

Overview of Generative AI Tools Used

Grammarly v1.2.189.1739 was used to correct spelling mistakes and refine the wording of some sentences. No paragraphs were fully rewritten; only the spelling and suggestion features were applied.

Chat GPT 4 and 5 were used to improve the formatting of tables and to convert CUDA code into pseudo-code snippets.

List of Figures

List of Tables

List of Algorithms

Bibliography

[AG24] Ahmed Abdelkhalek and The Khronos Group. Khronos announces vulkan video encode av1 & encode quantization map extensions. Blog post on The Khronos Group website, November 2024. Version 1.3.302 release.

[ANAM+20] Pontus Andersson, Jim Nilsson, Tomas Akenine-Möller, Magnus Oskarsson, Kalle Åström, and Mark D Fairchild. Flip: A difference evaluator for alternating images. *Proc. ACM Comput. Graph. Interact. Tech.*, 3(2):15–1, 2020.

[AOS+17] Jyrki Alakuijala, Robert Obryk, Ostap Stoliarchuk, Zoltan Szabadka, Lode Vandevenne, and Jan Wassenberg. Guetzli: Perceptually guided JPEG encoder. *CoRR*, abs/1703.04421, 2017.

[AvAB+19] Jyrki Alakuijala, Ruud van Asseldonk, Sami Boukortt, Martin Bruse, Iulia-Maria Comsa, Moritz Firsching, Thomas Fischbacher, Sebastian Gomez, Evgenii Kliuchnikov, Robert Obryk, Krzysztof Potempa, Alexander Rhatushnyak, Jon Sneyers, Zoltan Szabadka, Lode Vandevenne, Luca Versari, and Jan Wassenberg. Jpeg xl next-generation image compression architecture and coding tools. 2019.

[BAC96] Andrew C. Beers, Maneesh Agrawala, and Navin Chaddha. Rendering from compressed textures. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '96, page 373–378, New York, NY, USA, 1996. Association for Computing Machinery.

[CES00] C. A. Christopoulos, T. Ebrahimi, and A. N. Skodras. Jpeg2000: the new still picture compression standard. In *Proceedings of the 2000 ACM Workshops on Multimedia*, MULTIMEDIA '00, page 45–49, New York, NY, USA, 2000. Association for Computing Machinery.

[CH20] Jinyoung Choi and Bohyung Han. Task-aware quantization network for jpeg image compression. In *European Conference on Computer Vision*, pages 309–324. Springer, 2020.

[CL02] C.-H. Chen and C.-Y. Lee. A jpeg-like texture compression with adaptive quantization for 3d graphics application. *The Visual Computer*, 18:29–40, 2002.

[CR68] Fergus W Campbell and John G Robson. Application of fourier analysis to the visibility of gratings. *The Journal of physiology*, 197(3):551, 1968.

[FH24] Shin Fujieda and Takahiro Harada. Neural texture block compression. 2024.

[FHL+24] Farzad Farhadzadeh, Qiqi Hou, Hoang Le, Amir Said, Randall Rauwendaal, Alex Bourd, and Fatih Porikli. Neural graphics texture compression supporting random access, 2024.

[FP25] Alban Fichet and Christoph Peters. Compression of spectral images using spectral jpeg xl. *Journal of Computer Graphics Techniques Vol*, 14(1), 2025.

[Goo25] Google. Webp developer guide, 2025. Accessed: 2025-06-30.

[GSNK24] Rahul Goel, Markus Schütz, P. J. Narayanan, and Bernhard Kerbl. Real-time decompression and rasterization of massive point clouds. *Proc. ACM Comput. Graph. Interact. Tech.*, 7(3), August 2024.

[Ham04] Eric Hamilton. Jpeg file interchange format. 2004.

[HLM+21] Jingning Han, Bohan Li, Debargha Mukherjee, Ching-Han Chiang, Adrian Grange, Cheng Chen, Hui Su, Sarah Parker, Sai Deng, Urvang Joshi, et al. A technical overview of av1. *Proceedings of the IEEE*, 109(9):1435–1462, 2021.

[HPLVdW12] Charles-Frederik Hollemeersch, Bart Pieters, Peter Lambert, and Rik Van de Walle. A new approach to combine texture compression and filtering. *Vis. Comput.*, 28(4):371–385, April 2012.

[Huf52] David A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

[INH99] Konstantine Iourcha, Krishna S. Nayak, and Zhou Hong. System and method for fixed-rate block-based image compression with inferred pixel values, Sep 1999.

[JM17] Beau Johnston and Eric McCreath. Parallel huffman decoding: Presenting a fast and scalable algorithm for increasingly multicore devices. In *2017 IEEE International Symposium on Parallel and Distributed Processing with Applications and 2017 IEEE International Conference on Ubiquitous Computing and Communications (ISPA/IUCC)*, pages 949–958, 2017.

[KPM16] Pavel Krajcevski, Srihari Pratapa, and Dinesh Manocha. Gst: Gpu-decodable supercompressed textures. 35(6), December 2016.

[LA25] Belcour Laurent and Benyoub Anis. Hardware accelerated neural block texture compression with cooperative vectors, 2025.

[LHVA16] Jani Lainema, Miska M. Hannuksela, Vinod K. Malamal Vadakital, and Emre B. Aksu. Hevc still image coding and high efficiency image file format. In *2016 IEEE International Conference on Image Processing (ICIP)*, pages 71–75, 2016.

[LJP+23] Yuzhe Luo, Xiaogang Jin, Zherong Pan, Kui Wu, Qilong Kou, Xiajun Yang, and Xifeng Gao. Texture atlas compression based on repeated content removal. In *SIGGRAPH Asia 2023 Conference Papers*, SA '23, New York, NY, USA, 2023. Association for Computing Machinery.

[Mar25] Maria G. Martini. Measuring objective image and video quality: On the relationship between ssim and psnr for dct-based compressed images. *IEEE Transactions on Instrumentation and Measurement*, 74:1–13, 2025.

[McG17] Morgan McGuire. Computer graphics archive, July 2017.

[OBGB11] Marc Olano, Dan Baker, Wesley Griffin, and Joshua Barczak. Variable Bit Rate GPU Texture Decompression. *Computer Graphics Forum*, 2011.

[OK08] Anton Obukhov and Alexander Kharlamov. Discrete cosine transform for 8x8 blocks with cuda. *NVIDIA white paper*, 2008.

[PB13] Denis G. Pelli and Peter Bex. Measuring contrast sensitivity. *Vision Research*, 90:10–14, 2013. Testing Vision: From Laboratory Psychophysical Tests to Clinical Evaluation.

[RA08] Michal Radziszewski and Witold Alda. Optimization of frequency filtering in random access jpeg library. *Computer Science*, 9:109–120, 2008.

[Ran97] M. Randall. Talisman: multimedia for the pc. *IEEE Micro*, 17(2):11–19, 1997.

[RDPC24] Christoph Reich, Biplob Debnath, Deep Patel, and Srimat Chakradhar. Differentiable jpeg: The devil is in the details. In *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision (WACV)*, pages 4126–4135, January 2024.

[RK99] Greg Roelofs and Richard Koman. *PNG: The Definitive Guide*. O'Reilly & Associates, Inc., USA, 1999.

[SAM05] Jacob Ström and Tomas Akenine-Möller. ipackman: high-quality, low-complexity texture compression for mobile phones. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, HWWS '05, page 63–70, New York, NY, USA, 2005. Association for Computing Machinery.

[SAU19] Umme Sara, Morium Akter, and Mohammad Shorif Uddin. Image quality assessment through fsim, ssim, mse and psnr—a comparative study. *Journal of Computer and Communications*, 07:8–18, 01 2019.

[SG25] Shahrzad Sabzavi and Reza Ghaderi. Enhancing image-based jpeg compression: Ml-driven quantization via dct feature clustering. *IEEE Access*, 13:9047–9063, 2025.

[SSYH06] Tze-yun Sung, Yaw-shih Shieh, Chun-wang Yu, and Hsi-chin Hsin. High-efficiency and low-power architectures for 2-d dct and idct based on cordic rotation. In *2006 Seventh International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT'06)*, pages 191–196, 2006.

[STS+21] Kersten Schuster, Philip Trettner, Patric Schmitz, Julian Schakib, and Leif Kobbelt. Compression and Rendering of Textured Point Clouds via Sparse Coding. In Nikolaus Binder and Tobias Ritschel, editors, *High-Performance Graphics - Symposium Papers*. The Eurographics Association, 2021.

[SW11] Jacob Strom and Per Wennersten. Lossless compression of already compressed textures. In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*, HPG '11, page 177–182, New York, NY, USA, 2011. Association for Computing Machinery.

[TK96] Jay Torborg and James T. Kajiya. Talisman: commodity realtime 3d graphics for the pc. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '96, page 353–363, New York, NY, USA, 1996. Association for Computing Machinery.

[Val25] Valve Corporation. Steam hardware & software survey. `https://store.steampowered.com/hwsurvey/Steam-Hardware-Software-Survey-Welcome-to-Steam`, Mai 2025. Accessed: 2025-06-01.

[VSW+23] Karthik Vaidyanathan, Marco Salvi, Bartlomiej Wronski, Tomas Akenine-Möller, Pontus Ebelin, and Aaron Lefohn. Random-Access Neural Compression of Material Textures. In *Proceedings of SIGGRAPH*, 2023.

[W3T25] W3Techs. Usage statistics of image file formats for websites. `https://w3techs.com/technologies/overview/image_format`, Jul 2025. Accessed: 2025-07-20.

[Wal92] G.K. Wallace. The jpeg still picture compression standard. *IEEE Transactions on Consumer Electronics*, 38(1):xviii–xxxiv, 1992.

[WdOHN24] Clément Weinreich, Louis de Oliveira, Antoine Houdard, and Georges Nader. Real-time neural materials using block-compressed features, 2024.

[WS18] André Weißenberger and Bertil Schmidt. Massively parallel huffman decoding on gpus. In *Proceedings of the 47th International Conference on Parallel Processing*, ICPP '18, New York, NY, USA, 2018. Association for Computing Machinery.

[ZIE+18] Richard Zhang, Phillip Isola, Alexei A. Efros, Eli Shechtman, and Oliver Wang. The unreasonable effectiveness of deep features as a perceptual metric, 2018.