# MPTC: Video Rendering for Virtual Screens using Compressed Textures

Srihari Pratapa*[1], Pavel Krajcevski†[1] and Dinesh Manocha‡[1]

[1]The University of North Carolina at Chapel Hill

http://gamma.cs.unc.edu/MPTC/

## Abstract

We present a new method, *Motion Picture Texture Compression* (MPTC), to compress a series of video frames into a compressed video-texture format, such that the decoded output can be rendered using commodity texture mapping hardware. Our approach reduces the file size of compressed textures by exploiting redundancies in both the temporal and spatial domains. Furthermore, we ensure that the overall rendering quality of the compressed video-textures is comparable to that of compressed image-textures. At runtime, we render each frame of the video decoded from the MPTC format using texture mapping hardware on GPUs. Overall, MPTC improves the bandwidth from CPU to GPU memory up to $4-6\times$ on a desktop and enables rendering of high-resolution videos (2K or higher) on current Head Mounted Displays (HMDs). We observe $3-4\times$ improvement in rendering speeds for rendering high-resolution videos on desktop. Furthermore, we observe $9-10\times$ improvement in frame rate on mobile platforms using a series of compressed-image textures for rendering high-resolution videos.

**Keywords:** texture compression, video compression, GPU decoding, video in VR

**Concepts:** •**Computing methodologies** → **Image compression;** *Graphics processors; Graphics file formats;*

## 1 Introduction

Virtual reality (VR) is increasingly used for immersive gaming and multimedia experiences. In order to properly render the scenes and videos, head movements are tracked and different images are generated for each eye. There is considerable interest in watching movies, sporting events, or $360^\circ$ videos in VR headsets [Neumann et al. 2000]. These virtual screens can be easily re-sized in terms of size or distance and can allow the user to make the screen larger for a cinematic experience. The recent availability of high-resolution videos (e.g. $2K$ or $4K$ videos) and $360^\circ$ videos, requires appropriate hardware and processing capabilities to render these videos at high frame rates on virtual screens (see Fig. 1).

An immersive VR experience that requires the rendering of a high number of images or video on arbitrarily shaped screens involves the transfer of a large amount of image or video data from the CPU memory to the GPU memory. Rendering the video in a virtual scene requires each frame to be copied into GPU memory before it can be viewed. With the increasing availability of high-resolution content, the CPU-GPU bandwidth can become a bottleneck, especially on mobile platforms. In addition, a large number of memory transfers between the CPU and GPU increases the power consumption, which can be a major issue on mobile devices. Our goal is to transform and compactly represent a series of images into an intermediate format such that it can be easily decoded and rendered on commodity GPUs.

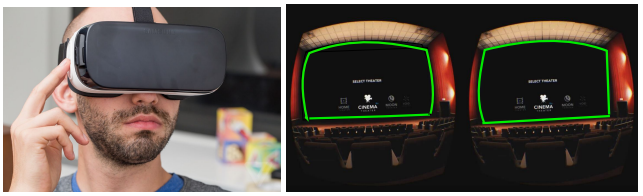

**Figure 1:** *(left) Person watching a movie in Galaxy Gear VR. (right) The different view points are rendered for each eye on the user's mobile screen. We use our novel MPTC algorithm to compactly represent the video frames and decode them using commodity GPUs. MPTC reduces CPU-GPU bandwidth requirements and can result in $3-4\times$ improvement in the frame rate over prior representations on HMDs using a desktop.*

There is considerable work on *compressed image-texture* (i.e., compressed texture representation for a single image) formats. Some of the widely used compressed image-texture representations include DXT, ETC, ASTC, etc. These formats support the random access property, which constrains the resulting schemes to fixed bit-rate encoding. On the other hand, standard image compression methods such as JPEG or PNG use entropy encoding techniques that use variable bit-rate compression and thereby achieve higher compression ratios as compared to compressed image-texture representations. Rendering applications can use standard image compression techniques but the streamed data has to be decompressed on the CPU and then uploaded to the GPU memory. In order to overcome these issues, supercompression methods have been proposed [Geldreich 2012; Krajcevski et al. 2016b] that further exploit the spatial coherence in a single compressed image-texture. These supercompression methods use standard entropy encoding techniques on top of the compressed textures, which reduces the bandwidth requirements. However, these methods do not exploit the temporal coherence between successive frames.

**Main results:** We present a novel technique (MPTC) that exploits the temporal coherence between successive frames of compressed image-textures, similar to standard video compression techniques like MPEG. Our approach is designed for endpoint-based compressed formats and uses their structure to split the video frames into different streams of data. In order to compress the index data for a $n \times m$ block of palette indices, we search for a reusable index

*psrihariv@cs.unc.edu

†pavel@cs.unc.edu

‡dm@cs.unc.edu



I3D '17,, March 04 - 05, 2017, San Francisco, CA, USA

ISBN: 978-1-4503-4886-7/17/03

DOI: http://dx.doi.org/10.1145/3023368.3023375

block in a search window surrounding the block. The search is performed using spatial and temporal methods to find the best match for the index data. We re-encode and re-optimize the endpoints of the block for each index block in the search window to find the best reusable index block. While re-encoding, we generate a dictionary of $n \times m$ blocks of index data that is used to reconstruct the entire index data. Next, we compress the re-optimized endpoints for each frame independently using standard image compression techniques. The resulting compressed video representation can be directly transcoded into compressed image-textures, which are rendered using the decoding hardware on a GPU. We observe significant benefits in terms of rendering speed using our *compressed video-texture* (MPTC) to display a video on virtual screens. We highlight the advantages of our approach to render $360^\circ$ videos on Oculus DK2. In practice, we observe $4-6\times$ reduction in CPU-GPU bandwidth and $3-4\times$ increase in rendering speed on a desktop. On mobile platforms, we observe an order of magnitude improvement in the frame rates.

The rest of this paper is organized as follows. Section 2 gives a briefoverview of prior work in texture and video compression. Section 3 describes our compression method in detail. We use our compression method to render videos for virtual screens in Section 4. We highlight the results and compare the performance with prior techniques in Section 5.

## 2 Background and Related Work

In this section, we give a brief overview of prior work in image, texture, and video compression schemes.

### 2.1 Image & Video Compression

Image codecs tend to find and exploit spatial redundancy in images by applying filters and transforming the pixel values to convert them into a different domain or color space. Entropy encoding [Huffman 1952; Rissanen and Langdon 1979] is used to losslessly compress the transformed pixel values, usually after a quantization step. The quantization step determines the quality of the original image that is then preserved in the compressed image. JPEG [Wallace 1992] uses the discrete cosine transform (DCT) on non-overlapping square blocks in the image to find coherence and separate out important values from unimportant values. The latest JPEG standard JPEG2000 [Skodras et al. 2001] uses a wavelet transform to transform pixel values and allows for lossless compression of images.

Video codecs additionally use temporal redundancy across frames to achieve higher compression. MPEG is a video codec [Le Gall 1991] that is widely used on current platforms. Over the years, many improvements to MPEG have been proposed. The latest version of MPEG is MPEG-4 Advanced Video Codec (AVC) or H.264 [Schwarz et al. 2007], which is widely used for live streaming of videos from the Internet to desktop and mobile devices. These video codecs use motion vectors to predict image blocks in the current frame from a previous frame. Frames encoded using motion vectors from previous frames are called predictive frames (P-frames) and, frames encoded independently are intra-frames (I-frames). For each block, the closest match is found by searching in a search proximity window; the difference between the closest match and the block is stored and the location of the match is stored as a motion vector for that block. Once the searching and transformation steps are performed, an entropy encoder is used to compress the transformed and predicted data. Some recent video codecs use different techniques, such as adaptive block sizing and bi-directional prediction for frames [Schwarz et al. 2007; Sullivan et al. 2012] to include new objects entering the scene in the future frames to achieve more redundancy.

Some of the latest mobile devices such as the Google Nexus 6P and the Samsung Galaxy S7 offer support for hardware H.264 encoders and decoders to support fast compression and decompression. The recent version of the standard under development is H.265 [Sullivan et al. 2012], which is being currently adopted to replace MPEG-4. VP9 is another new open-source video encoding standard that supports lossless compression. Daala [Valin et al. 2016] is an open-source experimental video codec that uses overlapping block transforms to minimize blocking artifacts encountered when using straightforward DCT. ORBX [Inc. 2015] is an experimental proprietary video codec that targets low-latency real-time video streaming.

### 2.2 Images & Texture Representations

In computer graphics, textures are used to add detail to 3D models and scenes. In addition, texture mapping can also be used to present information in different ways depending on the context. Applications like Google Earth use textures to provide satellite images projected onto arbitrary surfaces and shapes seamlessly. This is especially useful in 3D VR environments, where it is necessary to render image data onto arbitrary surfaces. Commodity GPUs support texture mapping capabilities in hardware to accelerate the rendering.

### 2.3 Texture Compression

Despite the ubiquity of texture mapping in computer graphics, video memory has been regarded as a scarce resource. This problem is even more evident in mobile computer graphics where data bandwidth is directly correlated to power consumption. In order to alleviate this problem, hardware manufacturers have developed dedicated GPU hardware for decompressing encoded chunks of texture data back into image pixels.

Modern texture compression formats are based on the four main properties established by Beers et al. [1996]. In particular, the two main properties of compressed texture formats are random access to pixels and fast decompression speed. The random-access requirement is necessary to properly allow the rendering pipeline to access the pixel data in parallel. This random access property necessitates a fixed-rate compression ratio for all images resulting in necessarily lossy compression. The quality of compression for each image varies based on the amount of detail in the image versus the fixed compression ratio. The texture compression formats are generally categorized into two sets *endpoint*-based schemes [Iourcha et al. 1999; Nystad et al. 2012; OpenGL 2010; Fenney 2003] and *table*-based schemes [Ström and Akenine-Möller 2005; Ström and Pettersson 2007].

The endpoint-based compression formats are all variations and improvements of Block Truncation Coding introduced by Delp and Mitchell [1979] and adapted to graphics by Kugler [1997]. In each instance, a block of pixels is represented by two low-precision colors. These two colors, known as *endpoints*, are used to generate a palette of full-precision colors by linearly interpolating between the endpoints. Additionally, each compressed block also contains per-pixel *palette indices* that select the final decompressed color. The first such format available on commodity hardware was DXTC (also known as S3TC and BC1) introduced by Iourcha et al. [1999]. In PVRTC [Fenney 2003] the endpoints are bilinearly interpolated across blocks before the palette generation. Recent methods such as BPTC and ASTC introduced multiple palette endpoints by allowing partitions in a single block [OpenGL 2010; Nystad et al. 2012].

Common texture compression formats, such as DXT1 [Iourcha et al. 1999], ETC1 [Ström and Akenine-Möller 2005], ETC2 [Ström and Pettersson 2007],and ASTC [Nystad et al. 2012],

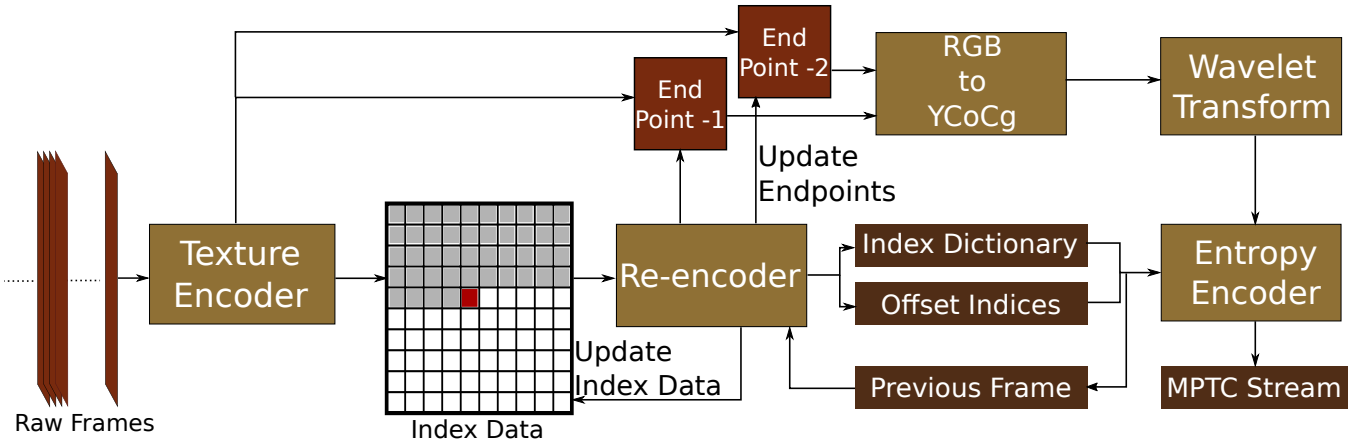


**Figure 2:** *Our compression pipeline: The input to our method is a sequence of uncompressed frames. The texture encoder converts the frames to compressed image-textures. The re-encoder searches in the corresponding search windows for suitable index data and optimizes the endpoints. The red block in the index data is re-encoded in this example. If the matching index data is found, the index data and the optimized endpoints for that block are updated in the current frame. After the re-encoding step is finished for a frame, the endpoint images are processed for further encoding. The index dictionary, offset indices, and wavelet coefficients are entropy encoded and appended to the MPTC stream.*

have become the de-facto standard on most commodity graphics hardware. DXT1 operates by storing $4 \times 4$ blocks of pixels into a fixed 64-bit representation, achieving a 6:1 compression ratio for 8-bit RGB data. ASTC, on the other hand, uses a 128-bit per-block representation but varies in block size from $4 \times 4$ to $12 \times 12$, corresponding to 3:1 to 27:1 compression ratios. In contrast, traditional image and video compression techniques, such as JPG and MPEG, use a variable number of bits to store a given image or frame of video. The final step of their encoding process usually requires a serial entropy encoding step that may not map well to GPU hardware. Encoding a texture into one the compressed texture formats can be slow. Many fast compression algorithms such as FasTC [Krajcevski et al. 2013] and SegTC [Krajcevski and Manocha 2014] have been proposed.

### 2.4 Supercompressed Textures

To overcome the problems of fixed bit-rate and bandwidth that manifest in streaming compressed textures, different methods have been proposed to add one more layer of compression on top of the fixed-size compressed texture formats (e.g., DXT, ASTC, ETC). Ström and Wennersten [2011] described a scheme to further compress ETC2 textures. They predict the per-pixel indices of a block by predicting the final color of the pixel from the previous pixel values. The Pixel index is computed from the predicted color and is used for updating the distribution model used for compression. Another scheme to compress DXT textures is *Crunch* which uses a dictionary and indices into the dictionary for the endpoints and the interpolation data. Huffman encoding is applied to the dictionary entries and the indices to achieve compression [Geldreich 2012]. VBTC [Krajcevski et al. 2016a] is a variable bit-rate texture compression scheme which uses an extra level of indirection to overcome the fixed bit-rate constraint. VBTC uses an adaptive block size instead of fixed block size to encode the texture. The block size is decided based on the local details. A recent supercompression technique, GST [Krajcevski et al. 2016b], is a GPU decodable scheme for endpoint-based texture compression methods. In *GST*, the endpoints and interpolation data are separated and compressed in different ways. The interpolation data is compressed based on a dictionary scheme, whereas the endpoints are compressed using a wavelet compression scheme, similar to JPEG2000.

## 3 Motion Picture Texture Compression

In this section, we present the details of our new approach and give an overview of our compression pipeline. The input to our method is a sequence of uncompressed frames of a video and the output is a compressed MPTC stream. The decoded output from the MPTC stream is a compressed image-texture that corresponds to one of the endpoint-based methods mentioned in Section 2.3.

Figure 2 shows our compression pipeline. MPTC decompression and rendering using GPUs is shown in Figure 6 (bottom). The first step of our compression scheme converts an input frame into a compressed image-texture using a texture encoder. For an endpoint compressed image-texture, our method separates the compressed image-texture into two different streams of data: endpoint data and index data. Figure 4 shows an example of the resulting separated streams. After the separation of the data, our method compresses the endpoints and index data in different ways as they exhibit different spatial and temporal patterns. Our approach has three major stages in encoding the compressed image-texture. First, we try to recompute the index data for each compressed image-texture block by looking at the surrounding blocks for any reusable indices. While the index data is recomputed, new optimal endpoints are computed for every recomputed index and the endpoint images are updated. In the second stage, the updated endpoint images are considered as the low resolution frames of the input image and processed for further compression. The final stage of our method uses an entropy encoding step to remove all redundancies that are formed after processing the index and endpoint data in the previous stages.

The rest of the section describes the first two stages of the compression pipeline in detail. We use the concept of Intra-frames (I-frames) and Predictive-frames (P-frames) used in standard video codecs (Section 2.1) to represent whether or not the compression of the current frame is dependent on a previous frame. I-frames are compressed independently of the previous frames and P-frames are compressed in tandem with the previous frames.

**Notation:** We are use the following notation in explaining our approach: For a block of $n \times m$ pixels that is compressed into an endpoint-based compressed texture, $I$ represents the index data of

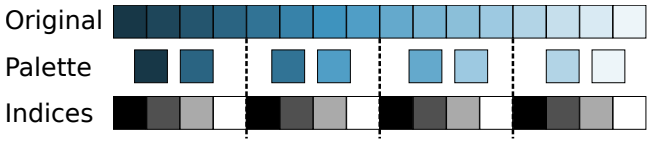


**Figure 3:** *Our method (MPTC) uses the block level coherency of the image features to find redundancies in index data. In this example, a gradient duplicates the same indices four times across the compressed data. Many images exhibit block-level coherence for image features that are larger than a block. This assumption comes from the per-block palette endpoints interpreted as a low-frequency approximation of the original image while the indices represent the high-frequency modulation values.*

a block; $\mathbf{ep}_A$ and $\mathbf{ep}_B$ represent the two endpoints; $E$ represents the mean error in the compressed block; $E_{threshold}$ represents the error threshold allowed in re-encoding the block.

### 3.1 Index Data Compression

Our goal is to reduce the amount of information needed to represent the index data in the frames of the video. The index data does not show any coherence at the pixel level, but at block level granularity, it exhibits coherency with the blocks in the previous frame and current frame. As shown in Figure 3, image features that span blocks generally lead to similar palette indices. This phenomenon can be attributed to the way that palette endpoints represent the low-frequency components of an image while the index values approximate the high-frequency components [Fenney 2003]. Due to the spatial coherence of the index data with adjacent blocks, we only need to consider blocks surrounding the current block in the previous frame and current frame while looking for redundancy. We employ a technique similar to vector quantization (VQ) to group all the minimal set of blocks that are required for reconstruction of the index data into a dictionary.

Endpoint compression formats compress $n \times m$ blocks of pixels separately. Instead of storing a global dictionary, as in the Crunch library [Geldreich 2012], we progressively construct our dictionary as we iterate over blocks in the compressed data. For every block in each frame, our re-encoding method performs the following steps:

1. Run an optimal texture encoder on a given block to produce indices $I$ and palette with error $E$.
2. Consider the *search window* for the current block.
3. For each block in the search window, find the new optimal endpoints $\mathbf{ep}'_A$ and $\mathbf{ep}'_B$ for the corresponding index data for that given block, as explained in Section 3.2.
4. If the new index data and their optimal endpoints produce error $E'$ such that $E' - E < E_{threshold}$, use selected index data $I'$ and optimized endpoints $\mathbf{ep}'_A$ and $\mathbf{ep}'_B$.
5. If the error is too large from all the entries in the search window, add the indices $I$ as a recent entry into the index dictionary for that frame.

The *search window* in step 2 differs depending on whether the current frame being encoded is an I-Frame or a P-frame. If it is an I-frame, the search window is defined within the current frame. If it is a P-frame we define two search windows, one in the previous frame centered around the current block position, and another one in the current frame. Figure 5 shows the search patterns for an intra-block search and a predictive-block search. For a P-frame we first search in the previous frame search window (i.e. predictive-block search). If there is no match for it within the error threshold,

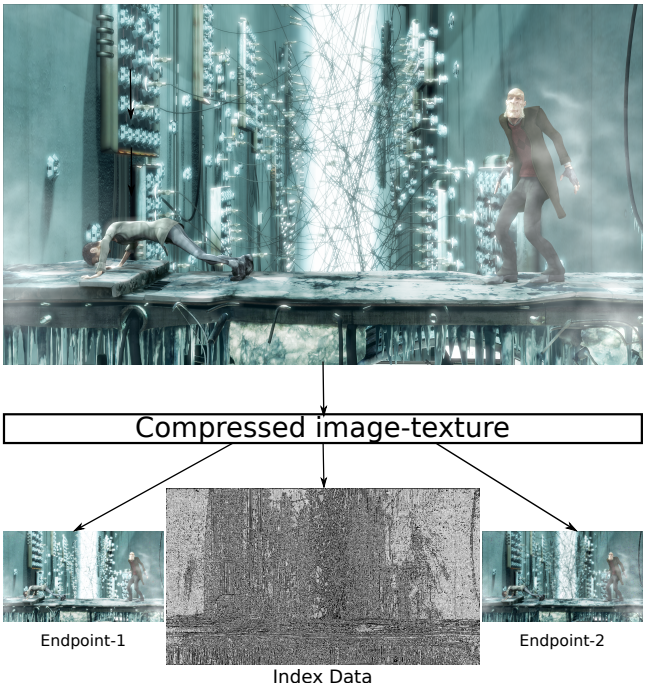


**Figure 4:** *In our approach (MPTC) we split all the frames of the video into separate constituents and encode all the constituents separately exploiting coherence in both the spatial and temporal dimensions. This example shows how we split an endpoint-based compressed image-texture into separate constituents, endpoint images, and index data. The endpoint data looks like a low resolution version of the original image. The index data looks like the high frequency data of the original image, having all the edge information. We handle these datasets separately in MPTC.*

we perform the search step in the current frame (i.e. intra-block search).

Unlike GST [Krajcevski et al. 2016b], where only the recent dictionary entries are searched, we search in a nearby window in order to find more blocks with similar index data, thus enabling more compression. Rather than storing a code word as an index into the dictionary entries like all VQ methods, we store a motion offset into the referring block. For example, for a block at position $(x, y)$ with the least-error index data match found at position $(x', y')$ within the search window, we store the offset vector as $(x' - x, y' - y)$ as the code word. If a matching index data is not found, a flag is used to mark that block as a direct entry in the dictionary. For a P-frame, for each block that has a matching index data, the offset vector can be either an intra-block offset or predictive-block offset. Additional data needs to be stored to distinguish between an I-frame offset and a P-frame offset.

In order to remove the additional mask data required to distinguish between an I-frame offset vector, a P-frame offset vector, and a direct dictionary entry, we use additional bits to represent the offsets. If the size of the search window is $w$, the minimum number of bits required to represent the window is $n = \lceil \log_2 w \rceil$. For window size $w$, all the offset values lie in the range $[0, w - 1]$ ($n$ bits) and we represent the offsets with $n + 2$ bits. We use the additional bits to differentiate between different offsets and direct dictionary entry. This bit manipulation helps to distinguish different offsets while decoding without additional information. In our implementation for MPTC, we restrict the maximum search window size to $w = 64$ (6 bits) such that the offset values can be represented with a byte. We

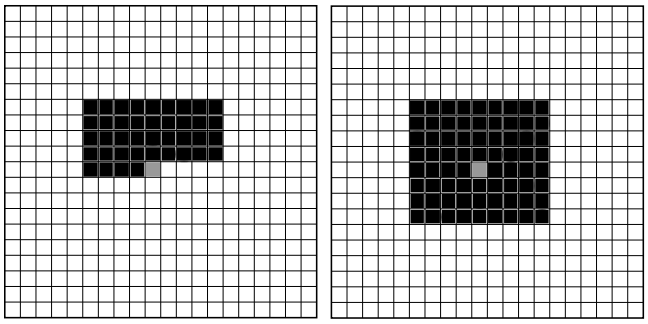

**Figure 5:** *The search windows used during recomputation of the index data in our approach (MPTC). (left) The gray block is the current block being re-encoded; the blocks marked in black are the search blocks for a possible index data match for the gray block in an intra-block search. (right) The blocks marked in the black are the search blocks for predictive-block search in the previous frame. The gray block shows the position of the block being re-encoded in the current frame.*

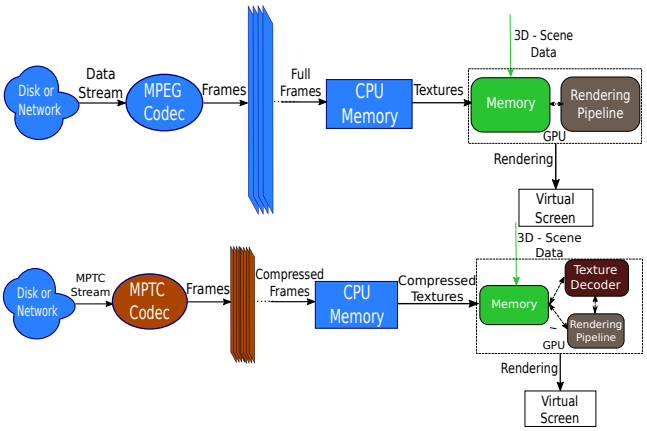


**Figure 6:** *Top figure shows the traditional video rendering pipeline for VR. Bottom figure shows the MPTC decompression and video rendering method using compressed textures for VR. Our pipeline provides various benefits over the current pipeline.*

utilize the two high order bits that are not used in the byte of the offset value (maximum value 63) to distinguish an I-frame offset and a P-frame offset.

The dictionary generated for each frame can be compressed further using entropy encoding. We observe that when dictionaries of multiple consecutive frames are combined together, the entropy of the combined dictionaries is smaller than the entropy of separate individual frames. The number of frames whose dictionaries can be combined is determined by an interval, which is a user input parameter for the compression. All of the frames in that interval have their dictionaries appended together for further encoding. The I-frames and P-frames are decided based on an error measure which quantifies the visual difference between the current and the previous frame.

### 3.2 Endpoint Optimization

The endpoint data still exhibits coherence across the block and can be treated as color images for compression purposes. For each block, if the index data is re-encoded, the optimal endpoints corresponding to that index are recomputed. Given possible index data for a block, the optimal endpoints for that block are calculated using the technique in [Castaño 2007]. We use the same method we use to re-encode the index data for the blocks to compute the error in determining a suitable index match in the search window. The recomputing of the endpoints for DXTC encoders is explained below.

DXTC encoders compress $4 \times 4$ blocks of pixels by storing two 565 RGB (5 bits Red, 6 bits Green, 5 bits Blue) endpoints to generate the full palette and 16 two-bit palette index values. In order to generate the palette, the two 565 RGB values $\mathbf{ep}_A$ and $\mathbf{ep}_B$ are expanded to full 888 RGB (8 bits Red, 8 bits Green, 8 bits Blue) by replicating high-order bits, and then linearly interpolating to two other colors at $\frac{1}{3}$ and $\frac{2}{3}$ the distance between $\mathbf{ep}_A$ and $\mathbf{ep}_B$ in RGB space. These four colors correspond to the full palette from which the two-bit index values are selected. Hence, the final reconstructed color for a pixel with index $i$ can be defined as

$$\frac{3-i}{3}\mathbf{ep}_A + \frac{i}{3}\mathbf{ep}_B$$

with $i \in \{0, 1, 2, 3\}$. From this formulation, we can set up an overdetermined system of sixteen equations whose solution provides the optimal endpoints for the given set of indices. In particular, for indices $i_0..i_{15}$ and image values $X_0..X_{15}$, the values for $\mathbf{ep}'_A$ and $\mathbf{ep}'_B$ that minimize the error

$$E' = \sum_{k=0}^{15} \left| \frac{3-i_k}{3}\mathbf{ep}'_A + \frac{i_k}{3}\mathbf{ep}'_B - X_k \right|$$

will give the optimal endpoints for the given set of indices $I'$. This error is compared to the original error, $E$, and if the difference between the errors is within a threshold, the indices $I'$ are used for the given block.

### 3.3 Endpoint Compression

Once the optimized endpoints are computed for each image, we extend the method described in [Krajcevski et al. 2016b] to compress both endpoint images independently. The image is divided into blocks and each block of pixels is transformed from a 565 RGB color space to a 667 YCoCg color space losslessly. Each of the Y, Co, and Cg planes are separately transformed to the wavelet domain. The wavelet coefficients are compressed using an entropy encoder to remove the redundancy. All of the transformations performed are lossless because any loss introduced in the precision of the endpoints is reflected across the 16 pixels in the 4x4 block corresponding to that block. Figure 2 outlines the endpoint compression method.

## 4 Video Rendering using MPTC

In this section, we present a video rendering algorithm that uses MPTC and highlight its benefits over prior methods. Standard video applications such as playing a movie or streaming video over the internet typically do not use a GPU for rendering. Instead, the video players and rendering applications use the hardware or software MPEG codecs to generate the image frames. The decoder generates full sized pixel images that are displayed on the screen at a given frame rate. Mapping these frames onto a 2D screen is straightforward: a one-to-one map exists from the pixels of the image to the pixels on the screen. Sometimes images need to be scaled depending on the screen resolution and other parameters. These steps can be performed in realtime on either a CPU or with a dedicated decompression hardware.

Rendering each frame of the video on a virtual screen requires the use of dedicated GPU hardware. This includes $360^\circ$ videos, where each frame corresponds to an equi-rectangular projection (spherical panorama) [Szeliski and Shum 1997] of multiple images of a scene from different viewpoints. To render such a $360^\circ$ video, every frame is mapped onto a spherical surface inside out by defining a mapping for each vertex to a 2D coordinate in the image. In order to map these images onto a sphere, each frame is treated as a texture and uploaded into GPU memory.

Our approach is to represent each video frame as a compressed texture and accelerate the data transfer and rendering using MPTC. We assume that the client receives an MPTC stream from either a disk or a server. The output of the MPTC decoder is a sequence of compressed texture frames. Figure 6 (bottom) shows a pipeline of our system. The three main benefits of our approach over the traditional way of rendering (shown in Figure 6 (top)) are:

1. Compressed data need not be decoded on CPU to a sequence of full, uncompressed frames;
2. The size of the compressed textures saves CPU-GPU bandwidth overhead;
3. Dedicated hardware texture decoders in a GPU are used to decode and render the texture.

Some of the desktop and mobile systems use dedicated MPEG decoders. Using hardware based decoders [NVIDIA 2016] for rendering video might improve the overall frame load times, but the resulting schemes still have some bottlenecks. The CPU-based hardware decoder improves the decoding speed significantly but it does not help reduce the CPU-GPU transfer bandwidth. GPU hardware decoders reduce CPU-GPU bandwidth significantly and improve the load times, but they do not help in saving video memory because fully uncompressed frames are stored while rendering. Our method ( Fig. 6) addresses both of these issues while using existing texture decoding hardware that is more common than hardware MPEG decoders on desktop as well as mobile platforms.

## 5 Results

| Name | Resolution | Frame Count |
|---|---|---|
| Elephant | $1024 \times 1024$ (1K) | 700 |
| 360 Coaster2K | $2560 \times 1280$ (2K) | 550 |
| 360 Coaster3K | $3584 \times 1792$ (3K) | 550 |
| Dongtan Day | $4096 \times 2048$ (4K) | 450 |
| Dongtan Night | $4096 \times 2048$ (4K) | 450 |

**Table 1:** *Benchmark Results: Name of the benchmark, resolution of each frame, and count of frames for each benchmark video. We have tested MPTC on these datasets and compared the performance with the existing techniques.*

In order to test our method against the standard process of video rendering, we chose MPEG as the choice of video codec, as it is widely used. We compare MPTC's performance with that of MPEG-2, MPEG-4, and H.265 codecs, as they correspond to previous, current, and future state-of-the-art video standards, respectively. Furthermore, different desktop and mobile systems support different codecs. In our implementation for MPTC, we have restricted the end point texture format to the DXT1 texture format, though techniques based on ASTC can lead to higher compression ratios. We compared the performance using several metrics, such as compressed file size, quality preserved, decoding speed, and rendering speed on a virtual screen (on an HMD). We gathered different uncompressed raw video frames with different resolutions as our benchmarks to test our method and system. These benchmarks are highlighted in Table 1. Videos showing the benchmark frames are included in the supplementary material.

| Benchmark | Format | CPU Load & Decode | GPU Load | Total (ms) |
|---|---|---|---|---|
| 360 Coaster2K | MPEG-2 | 0.51 | 2.00 | 2.51 |
| | MPEG-4 | 0.55 | 2.03 | 2.58 |
| | H.265 | 0.52 | 2.11 | 2.63 |
| | MPTC | 0.55 | 0.27 | 0.82 |
| 360 Coaster3K | MPEG-2 | 0.53 | 4.63 | 5.13 |
| | MPEG-4 | 0.51 | 4.81 | 5.32 |
| | H.265 | 0.50 | 4.64 | 5.14 |
| | MPTC | 0.55 | 0.58 | 1.13 |
| Dongtan Day | MPEG-2 | 0.59 | 7.31 | 7.90 |
| | MPEG-4 | 0.55 | 7.64 | 8.19 |
| | H.265 | 0.55 | 7.28 | 7.83 |
| | MPTC | 0.57 | 0.82 | 1.39 |

**Table 2:** *We compare the decoding and loading times for different formats in our rendering application in the Oculus HMD. We used benchmarks with high resolution frames to demonstrate the benefits of our MPTC representation. The GPU load times are $8-9\times$ faster when MPTC is used to render the videos. The overall load times are $3-4\times$ faster with MPTC as compared to other formats. All the timing data was collected on a desktop PC running Windows 10 with an Intel Xeon CPU and GeForce GTX TitanX GPU.*

| Benchmark | Format | CPU Load & Decode | GPU Load & Decode | Total (ms) |
|---|---|---|---|---|
| 360 Coaster2K | DXT1 | 0.96 | 0.19 | 1.15 |
| | CRN | 4.72 | 0.28 | 5.00 |
| | GST | 0.31 | 1.49 | 1.80 |
| | MPTC | 0.74 | 0.27 | 1.01 |
| 360 Coaster3K | DXT1 | 3.00 | 0.44 | 3.44 |
| | CRN | 8.12 | 0.44 | 8.56 |
| | GST | 0.56 | 2.83 | 3.39 |
| | MPTC | 0.76 | 0.58 | 1.34 |
| Dongtan Day | DXT1 | 4.12 | 0.70 | 4.82 |
| | CRN | 13.23 | 0.72 | 13.95 |
| | GST | 1.48 | 2.87 | 4.35 |
| | MPTC | 0.72 | 0.82 | 1.54 |

**Table 3:** *We compare the decoding and loading times with DXT1 and supercompression formats (CRN and GST) in our rendering application in the Oculus HMD. CPU We used benchmarks with high resolution frames to demonstrate the benefits of our MPTC. The overall load times are $2-3\times$ faster with MPTC as compared to other formats. All the timing data was collected on a desktop PC running Windows 7 with an Intel Xeon CPU and AMD Fury 9 GPU.*

In order to demonstrate the performance benefits of our method and implementation, we rendered $360^\circ$ videos from the benchmark datasets in an Oculus DK2 HMD. Our rendering application follows the pipeline shown in Figure 6 (bottom). In the render loop, a call to the frame decoder computes a new frame, which is uploaded onto the GPU for rendering on a spherical surface for $360^\circ$ viewing of the scene. Table 2 compares the performance of the rendering applications using different formats. Since the images being rendered are of a very high resolution, we implemented a buffered decode for both FFmpeg and MPTC decoders to reduce the high decode times of high resolution frames. The CPU load and decode

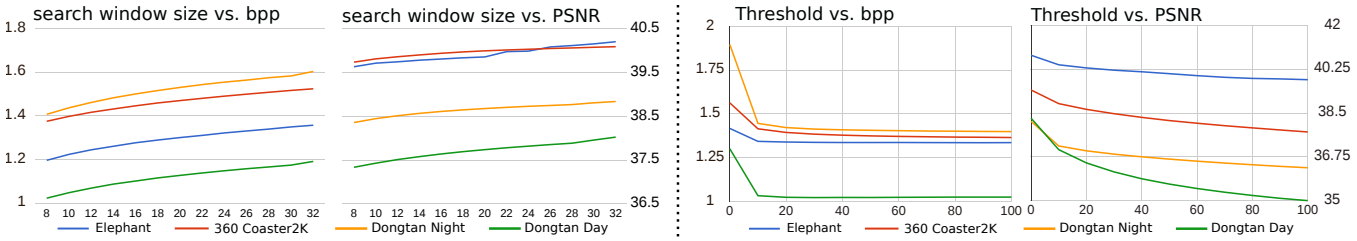


**Figure 7:** *We show the effect of varying certain parameters on the compression rate and image quality generated by our method (MPTC). (left) As we increase the window size, we find better-matching indices that results in better PSNR. However, an increase in the window size also results in the resulting motion indices to have higher entropy and that increases the file size. (right) As we increase the threshold, we notice a drop in the average file size and the average PSNR. The bits per pixel (bpp) and PSNR get saturated after a certain increase in the window size or the error threshold and that supports the assumption that the indices are coherent at a block level with respect to the surrounding blocks.*

| Format | FPS | |
|---|---|---|
| | 360 Coaster3K | 360 Coaster2K |
| JPG | 4.37 | 8.33 |
| DXT1 | 30.77 | 97.56 |
| ASTC 4x4 | 15.27 | 45.45 |
| ASTC 8x8 | 81.6 | 166.67 |

**Table 4:** *We highlight the benefits of using compressed image-textures to render the videos on a mobile device for 360 Coaster2K and 360 Coaster3K benchmark videos. The average FPS is calculated as the inverse of total time per frame. The time spent per frame is the average time to render a frame from the disk. The overall FPS is very high when compressed image-textures are used for rendering. The FPS is* $9-10\times$ *faster when video is rendered from compressed image-texture frames. This performance is measured on an HTC Nexus 9 using an NVIDIA Tegra-K1 GPU. The resulting compressed image-texture frames can be further compressed using MPTC, which exploits the temporal dimension.*

timings in Table 2 are the same for both MPEG and MPTC because of buffered decoding. The GPU load times of all MPEG-based formats are nearly the same as all of them load 24bpp uncompressed images onto the GPU.

The results from Table 2 highlight the benefits of using compressed image-textures and video-textures. The rendering speed increases by a factor $3-4\times$ when we use MPTC. The GPU load times are $8-9\times$ faster. CPU-GPU bandwidth reduction with MPTC is about $4-6\times$, as DXT1 requires 4 bits per pixel (bpp) and the uncompressed image is 24bpp. This significantly reduces the CPU-GPU memory footprint and also reduces the total number of memory accesses during video rendering.

In Table 3 we compare the rendering speeds of our method MPTC with the existing compression methods. For supercompressed formats such as GST [Krajcevski et al. 2016b] and Crunch (CRN) [Geldreich 2012] we use a motion JPEG based technique for rendering each frame separately. For CRN and GST we use the decoders provided in the public release of the code provided by the authors. The average rendering speeds are $2-3\times$ faster with MPTC as compared to these formats. We observe that total file sizes of the independently compressed frames for the benchmarks using GST or CRN are $50-60$ MB larger, as compared to MPTC.

Table 4 shows the average frames per second performance for loading and rendering a series of video frames in several formats on a mobile platform. The rendering application reads each frame from the disk and renders the frame onto a sphere for $360^\circ$ viewing. This is similar to the motion JPEG like technique described in Pohl et al. [2014]. The average rendering speeds are $9-10\times$ faster when compressed image-textures are used for rendering. This performance gain displays the benefit of using compressed image-textures or an MPTC like video-texture format for rendering videos on a mobile platform, where CPU-GPU bandwidth is not very high.

In order to perform bit-rate comparison with other formats, we compressed all the benchmark videos to MPEG and MPTC formats. Furthermore, we use compression options such that both formats have the same level of quality measure (PSNR). The encoding of the frames into MPTC is performed using our implementation with compression settings of $search\ window = 16$, $error\ threshold = 50$. The encoding and decoding of the frames into MPEG is performed using the open source library FFmpeg [ffmpeg 2016] API and FFmpeg command line tool. Table 5 shows the comparison between both formats for various benchmarks. The file size of MPTC is within a factor of $2-3\times$ of MPEG-2 for a comparable PSNR. Furthermore, the file size of MPTC is within a factor of $5-6\times$ when compared with advanced video codes (MPEG-4, H.265) for a comparable PSNR. Figure 8 shows a zoomed-in comparison for two formats. The visual quality of the decoded frames from MPTC format is comparable to the frames decoded from MPEG formats. We also analyze different compression parameters used in MPTC and described in Section 3. In particular, we show the effects of varying the error threshold and window size on the compression size and image quality. Figures 7 highlight the graphs generated using this analysis.

Overall, MPTC has relative benefits and disadvantages as compared to other formats in terms of file size and decode speeds. However, the benefits highlighted in (Table 2) outweigh the drawbacks by a large factor. Decoding into compressed image-textures and video-textures has several advantages while rendering on a virtual screen. Current GPUs are optimized for the image data being laid out in a specific format to provide fast access while rendering. Our method takes advantage of this characteristic during the decoding phase and thereby reduces the number of calls to memory copy in the video memory, by decoding directly into compressed image-textures. We believe further optimization of our method can result in better bit-rates, close to that of MPEG-2.

## 6 Conclusions, Limitations, and Future Work

We present a new method (MPTC) to encode a series of video frames as compressed textures and use it to render videos on virtual screens. Our approach exploits commodity texture mapping hardware on desktop and mobile platforms for fast rendering. MPTC can reduce the CPU-GPU bandwidth requirements and considerably improve the frame rates. Our current implementation uses the

| Benchmark | Format | Size (MB) | PSNR (dB) |
|---|---|---|---|
| Elephant (1K) | MPEG-2 | 43.00 | 41.70 |
| | MPEG-4 | 21.30 | 42.33 |
| | H.265 | 19.40 | 43.40 |
| | MPTC | 122.36 | 39.80 |
| 360 Coaster2K | MPEG-2 | 154.12 | 41.63 |
| | MPEG-4 | 94.30 | 42.10 |
| | H.265 | 83.70 | 42.50 |
| | MPTC | 298.56 | 40.20 |
| Dongtan Day (4K) | MPEG-2 | 226.46 | 40.30 |
| | MPEG-4 | 158.20 | 41.10 |
| | H.265 | 141.60 | 41.82 |
| | MPTC | 659.23 | 37.80 |
| Dongtan Night (4K) | MPEG-2 | 269.09 | 40.00 |
| | MPEG-4 | 157.10 | 41.06 |
| | H.265 | 140.30 | 40.42 |
| | MPTC | 669.71 | 38.93 |

**Table 5:** *A comparison of the resulting file size and average quality (PSNR) of our method (MPTC) with other formats for the benchmark videos. The file sizes are in Megabytes. The PSNR values are in dB. Although the resulting file sizes are* $2-6\times$ *larger, the rendering speeds are much better than other formats. As compared to other methods, MPTC can be easily decoded and rendered using GPUs.*

DXT1 texture format, though the approach can be extended to any endpoint compression format. We have compared the compression rates and results with MPEG video compression methods.

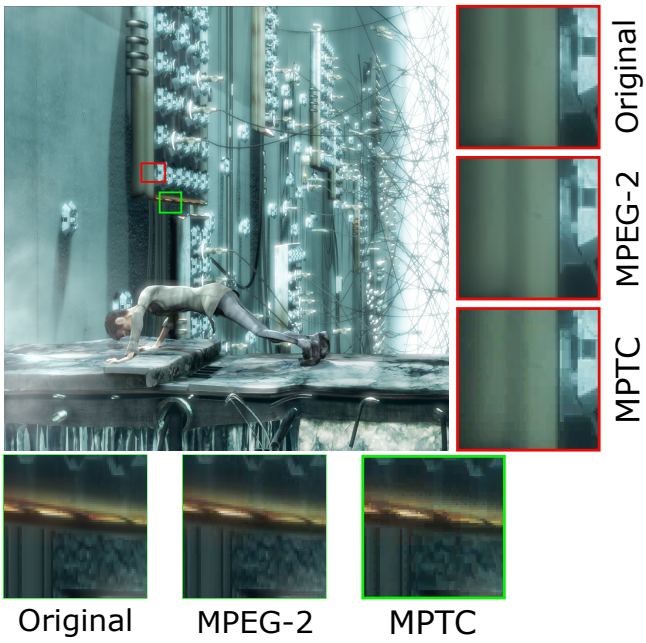


**Figure 8:** *Zoomed in visual comparison for a frame from the elephant benchmark video. An interesting part (*$32 \times 32$*) of a frame is cropped and zoomed to compare different formats and is compared with the original.*

**Limitations:** One limitation of our approach is the size of the compressed video file. This is due to the difficulty of compressing index data. A compression aware DXT1 texture encoder that can take into account the surrounding blocks may be able to achieve higher compression rates. Currently, we use a simple adaptive arithmetic encoder [Witten et al. 1987] as our entropy encoder. Using a context-based adaptive encoder or other advanced entropy encoders available as part of latest video codecs [Sullivan et al. 2012; Schwarz et al. 2007] can improve the compression efficiency. In our endpoint compression scheme, we compress the images independently without any link with previous frames. It would be useful to exploit the coherency between endpoint images in the temporal dimension.

**Future Work:** The compressed size of the endpoints can be reduced if we can use sophisticated video codecs such as H.265 or VP9 (see Section 2.1) to compress them. The challenge in implementing a motion based compression scheme for the endpoints would be achieved by designing a method similar to H.264/H.265 for 565 RGB, without increasing the bit-depth of the data upto 888 RGB. Moreover, that compression scheme has to be lossless, as the error in endpoints gets spread across the pixels in the block. Compared to DXT1, ASTC, and BPTC endpoint formats, the resulting schemes can be more complex due to use of partitioning, different color modes, and different bit depths for the palette data. We would like to extend MPTC to other texture compression formats such as ASTC and BPTC and evaluate their performance on high resolution videos. In this paper we have used PSNR to compare the quality. It would be useful to perform a perceptual evaluation of different compression schemes on a large number of videos.

## 7 Acknowledgements

This research is supported in part by ARO Contract W911NF-14-1-0437, Google, and NVIDIA. We thank Andrew Dickerson and John Furton at Samsung for the video benchmarks. We thank Rohan Prinja for his help in implementing decoders using ffmpeg and Nick Rewkowski for his help with video editing.

## References

BEERS, A. C., AGRAWALA, M., AND CHADDHA, N. 1996. Rendering from compressed textures. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, ACM, SIGGRAPH '96, 373–378.

CASTAÑO, I. 2007. High Quality DXT Compression using CUDA. *NVIDIA Developer Network*.

DELP, E., AND MITCHELL, O. 1979. Image compression using block truncation coding. *Communications, IEEE Transactions on 27*, 9 (sep), 1335–1342.

FENNEY, S. 2003. Texture compression using low-frequency signal modulation. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, Eurographics Association, HWWS '03, 84–91.

FFMPEG. 2016. Open source video codec library.

GELDREICH, R., 2012. Advanced dxtc texture compression library. https://github.com/richgel999/crunch.

HUFFMAN, D. A. 1952. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE 40*, 9 (Sept), 1098–1101.

INC., O., 2015. Highlighted features. https://home.otoy.com/stream/orbx/features/.

IOURCHA, K. I., NAYAK, K. S., AND HONG, Z., 1999. System and method for fixed-rate block-based image compression with inferred pixel values. U. S. Patent 5956431.

KRAJCEVSKI, P., AND MANOCHA, D. 2014. SegTC: Fast Texture Compression using Image Segmentation. In *Eurographics/*

*ACM SIGGRAPH Symposium on High Performance Graphics*, The Eurographics Association.

KRAJCEVSKI, P., LAKE, A., AND MANOCHA, D. 2013. Fastc: Accelerated fixed-rate texture encoding. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, ACM, New York, NY, USA, I3D '13, 137–144.

KRAJCEVSKI, P., GOLAS, A., RAMANI, K., SHEBANOW, M., AND MANOCHA, D. 2016. VBTC: GPU-Friendly Variable Block Size Texture Encoding. *Computer Graphics Forum*.

KRAJCEVSKI, P., PRATAPA, S., AND MANOCHA, D. 2016. Gst: Gpu-decodable supercompressed textures. *ACM Trans. Graph. 35*, 6 (Nov.), 230:1–230:10.

KUGLER, A. 1997. High-performance texture decompression hardware. *The Visual Computer 13*, 2, 51–63.

LE GALL, D. 1991. Mpeg: A video compression standard for multimedia applications. *Commun. ACM 34*, 4 (Apr.), 46–58.

NEUMANN, U., PINTARIC, T., AND RIZZO, A. 2000. Immersive panoramic video. In *Proceedings of the eighth ACM international conference on Multimedia*, ACM, 493–494.

NVIDIA, 2016. Nvidia video decoder (nvdec) interface. https://developer.nvidia.com/nvidia-video-codec-sdk.

NYSTAD, J., LASSEN, A., POMIANOWSKI, A., ELLIS, S., AND OLSON, T. 2012. Adaptive scalable texture compression. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on High Performance Graphics*, Eurographics Association, HPG '12, 105–114.

OPENGL, A. R. B., 2010. ARB_texture_compression_bptc. http://www.opengl.org/registry/specs/ARB/texture_compression_bptc.txt.

POHL, D., NICKELS, S., NALLA, R., AND GRAU, O. 2014. High quality, low latency in-home streaming of multimedia applications for mobile devices. In *Computer Science and Information Systems (FedCSIS), 2014 Federated Conference on*, 687–694.

RISSANEN, J., AND LANGDON, G. G. 1979. Arithmetic coding. *IBM J. Res. Dev. 23*, 2 (Mar.), 149–162.

SCHWARZ, H., MARPE, D., AND WIEGAND, T. 2007. Overview of the scalable video coding extension of the h.264/avc standard. *IEEE Transactions on Circuits and Systems for Video Technology 17*, 9 (Sept), 1103–1120.

SKODRAS, A., CHRISTOPOULOS, C., AND EBRAHIMI, T. 2001. The jpeg 2000 still image compression standard. *IEEE Signal Processing Magazine 18*, 5 (Sep), 36–58.

STRÖM, J., AND AKENINE-MÖLLER, T. 2005. iPACKMAN: high-quality, low-complexity texture compression for mobile phones. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, ACM, HWWS '05, 63–70.

STRÖM, J., AND PETTERSSON, M. 2007. ETC2: texture compression using invalid combinations. In *Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, Eurographics Association, GH '07, 49–54.

STRÖM, J., AND WENNERSTEN, P. 2011. Lossless compression of already compressed textures. In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*, ACM, New York, NY, USA, HPG '11, 177–182.

SULLIVAN, G. J., OHM, J. R., HAN, W. J., AND WIEGAND, T. 2012. Overview of the high efficiency video coding (hevc) standard. *IEEE Transactions on Circuits and Systems for Video Technology 22*, 12 (Dec), 1649–1668.

SZELISKI, R., AND SHUM, H.-Y. 1997. Creating full view panoramic image mosaics and environment maps. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, ACM Press/Addison-Wesley Publishing Co., 251–258.

VALIN, J.-M., TERRIBERRY, T. B., EGGE, N. E., DAEDE, T., CHO, Y., MONTGOMERY, C., AND BEBENITA, M. 2016. Daala: Building a next-generation video codec from unconventional technology. *arXiv preprint arXiv:1608.01947*.

WALLACE, G. K. 1992. The jpeg still picture compression standard. *IEEE Transactions on Consumer Electronics 38*, 1 (Feb), xviii–xxxiv.

WITTEN, I. H., NEAL, R. M., AND CLEARY, J. G. 1987. Arithmetic coding for data compression. *Communications of the ACM 30*, 6, 520–540.