GPU-accelerated depth codec for real-time, high-quality light field reconstruction

BABIS KONIARIS, MAGGIE KOSEK, DAVID SINCLAIR, AND KENNY MITCHELL,
Disney Research, University of Edinburgh and Edinburgh Napier University

(a) Color image (b) Updated quadtree nodes (c) Reconstruction error

Fig. 1. Approximation of a video frame using our codec. The image differences are encoded as quadtree nodes, where each node is approximated with one of the following modeling functions: raw, platelet, biquadratic and our novel BC4w block compression mode. Our codec allows for rapid decompression and is suitable for decoding multiple streams simultaneously.

Pre-calculated depth information is essential for efficient light field video rendering, due to the prohibitive cost of depth estimation from color when real-time performance is desired. Standard state-of-the-art video codecs fail to satisfy such performance requirements when the amount of data to be decoded becomes too large. In this paper, we propose a depth image and video codec based on block compression, that exploits typical characteristics of depth streams, drawing inspiration from S3TC texture compression and geometric wavelets. Our codec offers very fast hardware-accelerated decoding that also allows partial extraction for view-dependent decoding. We demonstrate the effectiveness of our codec in a number of multi-view 360 degree video datasets, with quantitative analysis of storage cost, reconstruction quality and decoding performance.

CCS Concepts: • **Computing methodologies** → **Rasterization**; **Image compression**; **Virtual reality**;

Additional Key Words and Phrases: depth image based rendering, compression, light field rendering

Author's address: Babis Koniaris, Maggie Kosek, David Sinclair, and Kenny Mitchell,
Disney Research, University of Edinburgh and Edinburgh Napier University, [babis.koniaris,maggie.kosek,
david.sinclair,kenny.mitchell]@disneyresearch.com.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2018 Copyright held by the owner/author(s). Publication rights licensed to ACM.
2577-6193/2018/5-ART3 $15.00
https://doi.org/10.1145/3203193

ACM Reference Format:
Babis Koniaris, Maggie Kosek, David Sinclair, and Kenny Mitchell. 2018. GPU-accelerated depth codec for real-time, high-quality light field reconstruction. *Proc. ACM Comput. Graph. Interact. Tech.* 1, 1, Article 3 (May 2018), 15 pages. https://doi.org/10.1145/3203193

1 INTRODUCTION

Recent advances in virtual and augmented reality software and hardware have sparked interest for high quality mixed reality experiences, seamlessly blending together characters and environments. For such immersive and interactive experiences that allow free user movement with six degrees of freedom, presented video content needs to be adapted for consumption from any point of view. Standard single-view or multiview videos are limited for this scope, as they lack high quality depth information, which is essential for interactive reconstruction of the scene from any vantage point.

High quality (and bit-depth) depth videos have had a far shorter history compared to color videos, and that is reflected in codec development, which is strongly focussed on color data. In this work, we target compression of high-quality depth videos with a particular focus on decoding performance, as it is essential for immersive real-time experiences. We develop a depth video codec that is aimed for high-quality reconstruction, as aggressive depth compression allowing high errors creates problems when geometric reconstruction of the scene is desired.

Our codec allows GPU-accelerated decompression of several high-resolution video streams simultaneously, allowing 3D reconstruction and real-time exploration of captured or offline-rendered video with the capability of compositing additional 3D elements and characters due to the existence of an accurately reconstructed depth video stream. A component of our codec, the BC4-Wedged (BC4w) block compression format can be used as a generic block compression method for depth data that exhibit discontinuities as well as surface variation. The applications of our codec are numerous and general, as it can be used in any application that requires high-performance depth streaming, for example light field probes with depth data [14] or any application that utilizes reconstruction from depth video where decoding is only *part* of a tight performance budget, for example interactive VR experiences.

Contributions. Our contributions form a compression and decompression pipeline for depth data:

- A novel lossy block-compression format, optimized for compression and fast decompression of high-resolution depth data, greater than 8 bits per pixel. The format offers fixed 4:1 compression ratio for 16-bit data input, and uses a fixed block size of 8×8 pixels.
- A novel depth video codec, designed for high quality reconstruction and optimized for high decoding speed. The codec utilizes the block-compression format as one of the used modeling functions, and it is based on fixed-size nodes for optimized data-parallel decoding.

2 RELATED WORK

Re-purposing h264. Several approaches for depth compression attempt to reuse the mature H.264 color compression infrastructure for compressing depth data. Pece et al. [19] convert the depth data to 3 8-bit channels before compressing them with h264, ensuring that the lossy nature of the compressor affects the depth values as little as possible. While this is a straightforward way to use the existing pipeline, the decoding costs are too high to

be used for multiple depth streams simultaneously for real-time reconstruction. Liu et al. [13] apply a hybrid scheme where, for 12-bit depth videos, the 2 most significant bits are losslessly compressed while the remaining 10 bits are encoded as a YUV stream using h264.

Piecewise-linear functions. A common approach to compress depth data that exhibit edge discontinuities and flat surfaces (such as indoor environments) is via subdivision of the image to a quadtree and approximating the nodes with piecewise-linear functions, such as *wedgelets* and platelets [16]. Alternatively to lines marking discontinuities, contours can also be used [6]. Such functions are fast to evaluate and are suitable for real-time decoding in multiview use-cases [22]. More recently, Kiani et al. [8] introduced *planelets*, using a linear fractional model to address the non-linearity of depth data as typically captured by sensors such as Kinect. Planelets are successful in improving approximation quality but are still limited to planar surfaces.

Texture compression. Another approach to compress depth video data is by applying any temporal compression method, and use GPU-accelerated texture compressed formats for the intra-frame coding. Such an approach is used by Koniaris et al. [9], using frame-to-frame differences to calculate update regions, and compressing the region data using 3Dc+ (BC5) format. This approach, while very efficient to decode, results in uncontrollable artifacts for depth video that exhibits high-frequency content, due to lack of any rate-distortion control.

Multiview plus depth. Several methods have been proposed for improving reconstruction of free viewpoint video by incorporating depth streams [15]. Some methods utilise similarities (such as motion and structure) between color and depth streams to improve depth compression [10, 12, 18, 23]. The high-efficiency video coding (HEVC) standard has also been extended to support multi-view video plus depth, applying intra-coding techniques that are suitable for the unique characteristics of depth maps [17]. Such methods use a fixed set of cameras and, while they achieve very good compression rates, they are expensive to decode [5].

Other. Other methods attempt the conversion of the depth map to a mesh, such as the work by Banno et al. [2] and Collet et al. [3]. While such approaches can work well for animated models such as moving humans, they have diffficulty capturing fine, high-frequency geometric details such as blades of grass and tree leaves.

Another recent approach by Wilson [24] uses lossless compression that can be efficiently decompressed. The reported decoding times are very fast, but due to the low compression rate, the method would not scale in terms of required bandwidth in scenarios with multiple parallel depth streams.

A more general approach for lossless compression of floating point buffers is described by Pool et al. [20]. It is based in the work of Ström et al. [21], but allows variable-precision compression, buffers of any layout, dynamic bucket selection and uses a Fibonacci encoder. The method is applicable to any type of data, but the algorithm performs best on depth data, although there are no performance figures in the paper.

Didyk et al. [4] apply a form of spatial subdivision based on the depth buffer, where they generate an adaptive grid based on the similarity of depth values. The method is used for synthesizing stereo views from a depth buffer. While not used for compression, the concept is similar to adaptive subdivision for compression of dynamically-sized image blocks. Their use-case is a warp from one eye to another, which is typically a very short distance, whereas our method is not limited in any way in terms of where the original and synthetic views are.

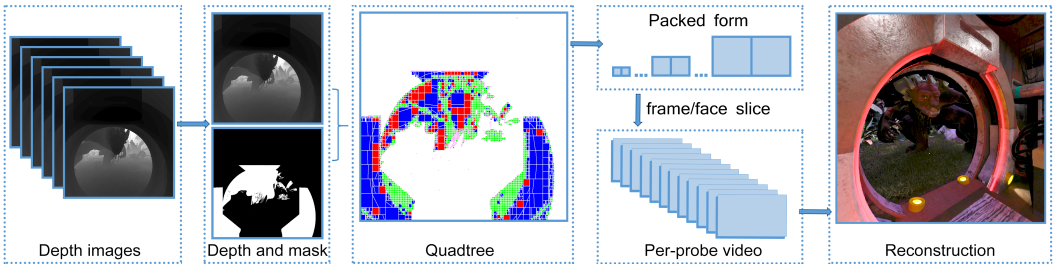


Fig. 2. Codec pipeline overview. We first determine the mask for depth image i from depth frames $[i-N, i]$. The mask and depth frame are then used to create the intermediate quadtree form. From the quadtree, we can generate runtime data given any maximum error threshold. The runtime data are packed with the rest of the frames/faces in buffers which are then loaded and decoded on GPU online.

2.1 Overview

Our proposed codec has distinct spatial and temporal components. In terms of the time domain, we use an approach similar to Koniaris et al. [9] by calculating differences to previous N frames and generating a mask of the pixels that need to be updated; the spatial component of our compressor only targets this subset of pixels.

The novelty of our work lies in the spatial compression and decompression of depth images of high bit-depth, and can be divided in the following stages. The first step is the quadtree generation, where we generate a quadtree data structure per image and associated difference mask (section 3). We proceed with *runtime packed form generation* given a desired maximum error for the final compressed stream (section 4). Finally, all the compressed images are assembled to a single stream and decoded in runtime (section 5).

Our pipeline supports decoding of a multitude of depth video streams simultaneously in real-time. Each video stream corresponds to the depth data as seen by a *light field probe* in the scene. A light field probe stores color data a single point in space, representing a single point in the 5D plenoptic function [1]. For these probes, we include depth data: distances to the closest surface at any direction. The depth video data for each of these probes are represented in our method using six separate streams, one per cubemap face.

3 QUADTREE GENERATION AND MODELING FUNCTIONS

The input to the first stage of the spatial compression is a depth image, a mask of the pixels that need to be compressed and a maximum reconstruction error $E_{\max}$. The output of this stage is a partial quadtree, with nodes covering all masked pixels. The quadtree is calculated bottom-to-top, where a node is approximated only if *all* of its existing children (i.e. in the mask) can be approximated by any modeling function successfully within the given error. The maximum reconstruction error is used as a bound on the optimisation time required; coarser nodes can take significantly more time to approximate due to the larger search space, which is also subject to the modeling function used.

The quadtree nodes approximate the underlying data using one of four modeling functions: *raw*, *platelet*, *biquadtratic* and *BC4w*. Each node stores the best approximating function and its coefficients. Some modeling functions can only be used at certain quadtree levels; this "modeling function type" property is used as an *implicit* parameter of the compressed datastream. We use modeling functions whose coefficients can be stored within 32 bytes, as it allows very efficient decoding by means of random access (more details in section 5). As

such, the finest level of the quadtree is comprised of 4×4 pixel nodes storing uncompressed quantized data, fully occupying all 32 bytes. Below, we discuss the error metrics and describe the modeling functions that we use in this work.

3.1 Error metrics

High depth errors can prove catastrophic for reconstruction quality: they can manifest as animated floating geometry or holes, leading to further errors in color reconstruction and user perception of the environment. To prevent this, we use two criteria for choosing a modeling function for a node. The first criterion is the maximum absolute distance in log-space per block of dimension k:

$$e_{\max} = \max_{x,y \in [1,k]} ||d_{xy} - \hat{d}_{xy}|| \tag{1}$$

For each block, we calculate this error for all modeling functions and keep the ones with acceptable error $e_{\max} \leq E_{\max}$. For the successfully approximated modeling functions, we calculate their mean-squared error (MSE) for the block:

$$e_{\text{MSE}} = \frac{1}{k^2} \sum_{y=1}^{k} \sum_{x=1}^{k} (d_{xy} - \hat{d}_{xy})^2 \tag{2}$$

and simply select the modeling function that approximates the block with the lowest MSE.

3.2 Raw, platelet and biquadratic modeling functions

The raw modeling function stores the raw pixel data directly as 16 16-bit values in a quantized logarithmic z space, similar to Koniaris et al. [9]. It is used only in the finest level (4×4 pixel tiles) and exists as a fail-safe when no good approximation can be found given an error threshold in coarser level nodes.

$$z_{xy} = D_{xy} \tag{3}$$

where D the quantized depth data and x,$y \in [1, 4]$.

The platelet modeling function approximates the floating point linear depth data of a node at any level using two planar surfaces separated by a line:

$$z_{xy} = \begin{cases} a_0 x + a_1 y + a_2 & \text{when } (i,j) \text{ left of line} \\ b_0 x + b_1 y + b_2 & \text{otherwise} \end{cases} \tag{4}$$

where x,$y \in [1, 2^{3+k}]$.

The biquadratic modeling function approximates the floating point linear depth data of a node at any level using a biquadratic surface:

$$z_{xy} = a_0 x + a_1 y + a_2 xy + a_3 x^2 + a_4 yy + a_5 \tag{5}$$

Platelets and biquadratic functions can be calculated by solving linear systems: for platelets, this is described by Morvan et al. [16], while for biquadratics is simpler, as the design matrix A and output vector b are calculated as follows:

$$X = [0, 1, 0, 1, 2, 0]$$
$$Y = [0, 0, 1, 1, 0, 2]$$
$$A_{ij} = \sum_{y=1}^{k} \sum_{x=1}^{k} x^{X_i+X_j} y^{Y_i+Y_j} \tag{6}$$
$$b_i = \sum_{y=1}^{k} \sum_{x=1}^{k} xy d_{xy} \tag{7}$$

where k is the block dimension and d_{xy} is the depth value at a pixel (x, y).

3.3 BC4w modeling function

Platelets and biquadratic functions capture structured surfaces (planar and smooth) very well, but fail to capture depth variation on such surfaces. Therefore, when a high quality approximation is required, these modeling functions fail and the raw modeling function is used at the finest level, which is considerably more expensive to store (4×).

To improve high quality approximations, we developed a modeling function that is based on the block compression format *3Dc+* (also known as BC4), adapted for 16 bits per pixel and 8×8 pixel tiles, but also augmented with a line and 2 pairs of endpoints, each pair chosen based on the halfspace with respect to the line. An example of such a lines and endpoint pairs are shown on the left of figure 3.

Our modeling function combines the high quality gradient approximation offered by BC4, with the discontinuity approximation offered by platelets. Each 8×8 pixel tile is approximated using 32 bytes, offering a fixed 4:1 compression ratio. The data block is comprised of 4 depth endpoints (2 pairs), palette indices per pixel and a line. The palette indices require 3 bits per pixel, as per BC4. We use 2 endpoint pairs instead of a single pair, where a pixel selects the pair that corresponds with the side of the line that it is on. The endpoints use the numerical order to break degeneracy, as in BC4: if $d_0 > d_1$, the palette indices correspond to the 8 evenly spaced values between (and including) the endpoints, where if $d_0 < d_1$, the palette indices correspond to the 6 evenly spaced values between (and including) the endpoints and additionally include the boundary values of the 16-bit space: 0 and 65535.

The line specification for an 8×8 tile requires a total of 12 bits, 3 per line point coordinate. As the palette indices require 192 bits ($3 \times 8 \times 8$), we need to pack each depth endpoint in 13 bits, in order for them to fit in the 32 byte memory block. This quantization introduces an insignificant error in the endpoint storage (1e-04) that we consider acceptable for a lossy codec. The bit allocation is shown on the right of figure 3.

The compression algorithm is exhaustive; it samples the parameter space as densely as possible. For each side of each potential line with discrete endpoints that are part of the 8×8 pixel block, we search for the depth endpoints that minimize reconstruction error of the pixels on the side of that line. The process is highly data-parallel and is implemented in the GPU. We describe the process in algorithm 1.

4 RUNTIME PACKED FORM GENERATION

The generated per-frame quadtrees are used as an intermediate format. A partial quadtree can be used to reconstruct an image at *any* error threshold $E_{\max}$, up to its maximum reconstruction error and down to lossless. To improve decoding performance that is paramount

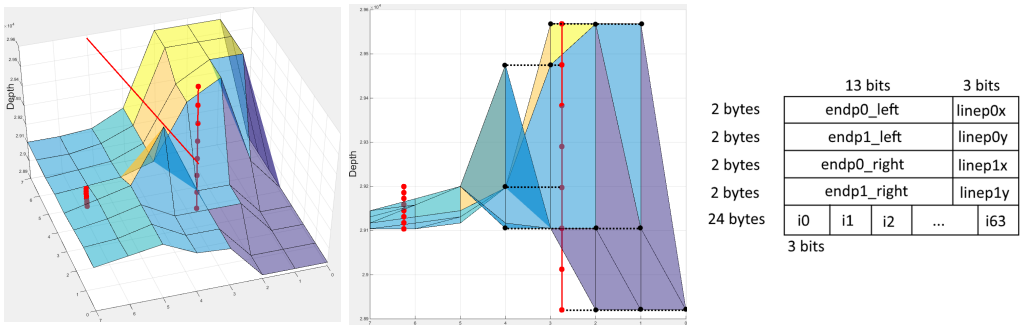
Algorithm 1 BC4w compression

Precondition: 8×8 depth blocks $D_i(x, y)$, $i \in [1, N]$, $x, y \in [1, 8]$. State S per block, storing best endpoint pairs and associated block approximation errors. Coarse stride s.
Output: N BC4w blocks B_i

function CalcTileErrorGpuKernel(S, d_0, d_1)
  if kernel executed at fine level **then**
    $d_0 \leftarrow d_0 + s(S_{d_0} - 0.5)$
    $d_1 \leftarrow d_1 + s(S_{d_1} - 0.5)$
  Reconstruct actual d_0, d_1 depending on execution as coarse or fine
  for each line ℓ **do** ▷ Executed in parallel from threads within a group
    Use d_0, d_1 as "left" endpoint pair and calculate error
    Update "left" endpoint pair and error state
    Use d_0, d_1 as "right" endpoint pair and calculate error
    Update "right" endpoint pair and error state

function PackToOutputGpuKernel(S)
  Calculate line that reconstructs both state depth endpoint pairs with least total error
  Calculate per-pixel palette indices based on line and state depth endpoint pairs
  Build BC4w block based on line, depth endpoint pairs and per-pixel palette indices

function Main
  Initialize state
  $d_{\min} \leftarrow \max(\min D - 255, 0)$ ▷ Padded minimum depth value
  $d_{\max} \leftarrow \min(\min D + 255, 65535)$ ▷ Padded maximum depth value
  for $d_0 \in [d_{\min}, d_{\max}]$ with step s **do** ▷ Coarse level
    for $d_1 \in [d_{\min}, d_{\max}]$ with step s **do**
      Launch kernel CalcTileErrorGpuKernel(S, d_0, d_1)
  for $d_0 \in [1, s]$ **do** ▷ Fine level
    for $d_1 \in [1, s]$ **do**
      Launch kernel CalcTileErrorGpuKernel(S, d_0, d_1)
  Launch kernel PackToOutputGpuKernel(S)



(a) Block, line and depth pairs (b) Quantization (Side view) (c) Bit allocation

Fig. 3. BC4w format overview. The 8×8 pixel tile is approximated using a line and two depth endpoint pairs. Pixels get assigned to a depth endpoint pair depending on the side of the line they are on. Pixels store a 3-bit palette index which is used to interpolate the depth endpoint pair they are assigned to. The depth endpoints are quantized to 13 bits in order for the block format to fit in 32 bytes. The depth endpoint order is used to determine interpolation parameters (section 3.3).

to high-throughput data such as light field video we transcode the quadtree data to a more lightweight form: a *flat* list of non-overlapping nodes that can be used to reconstruct the (partial, masked) image at a fixed maximum error threshold $E, E \leq E_{\max}$. To extract the flat nodes from the quadtree, we simply traverse it top-to-bottom, depth-first, stopping at nodes that can sufficiently approximate the underlying image region within error e and copying them to the output stream.

This flat form of a partial, masked image consists of a fixed header and variable payload data. The header stores the total number of flat nodes and the number of flat nodes per level, whereas the payload data stores the per-node offsets followed by the per-node coefficients. The 2D offsets are stored using 2 bytes in our implementation, exploiting the fixed power-of-two node sizes and the knowledge about what level a node represents. As each node starts at $2^{2+i}(x, y)$, where i the quatree level, we can therefore represent offsets for a 1024×1024 image. As at this stage we intend to minimize storage cost, for each node we store the coefficients and two bits for identification of the modeling function used.

When we load the video data in the application, we shift our attention from minimization of storage to maximisation of decoding efficiency. For each node, we allocate the same amount of memory (32 bytes for coefficients and 2 bytes for offsets) so that the node data can be efficiently random-accessed. Separating the payload to offset data and coefficient data also allows 128-bit aligned accesses[1] that are preferred in GPU architectures, improving decoder performance. This packed form does not contain information about the tile type, as it can be implicitly determined from the node level and certain bit values in the 32-byte coefficient block:

- A 4×4 pixel block always uses the raw modeling function.
- The biquadratic modeling function holds 24 bytes, so the last 8 bytes are set as `0xFF`.
- The platelet modeling function requires 28 bytes, so the last 4 bytes are set as `0xFF`.
- The 4-byte BC4w data block that describes endpoints and lines is stored at the end, as no such valid block would be filled with `0xFF`.

From the above, we apply the following logic to determine the tile type:

- If the 8 last coefficient bytes are `0xFF`, the coefficients represent a biquadratic function.
- Otherwise, if the 4 last coefficient bytes are `0xFF`, the coefficients represent a platelet.
- Otherwise the coefficients represent a BC4w function.

5 PER-PROBE STREAMS AND GPU DECODING

One of the main requirements for our codec is very high decoding performance. This is made possible by organizing, accessing and decoding the data stream effectively, utilizing the GPU as a massive data-parallel processor. As such, our runtime decoder is entirely implemented in GPU[2]

As mentioned in section 2.1, our input video data are six video streams per probe, one per cubemap face. Each cubemap-face video stream can be further subdivided into smaller *cells*, so that individual cells can be selectively decoded or not based on if their content is visible to the viewer (view-dependent decoding in [9]).

The compressed video for each probe is stored as a contiguous data buffer of runtime packed forms, organized by frame, cubemap face and cell[3]. We also store per-frame look-up tables to identify the buffer offset for a given combination of (frame,face,cell), as well as the data buffer memory range utilized per frame. The buffer and tables are stored in GPU memory. The decoding output is cubemap texture faces. For each probe, we maintain *state*: the frame that was last loaded in each cell of each face. At every application frame, we determine the video frame that needs to be loaded and the cells/frames that are visible (even partially) to the viewer. We compare this data with the state and identify the cells that need to be updated. Since both the buffer and lookup tables already reside in GPU

[1] The header and the offset are required to be stored with 128-bit alignment as well.
[2] A CPU implementation is trivial, but would not be as performant.
[3] Each face is split into 2×2 cells for more effective view-dependent decoding.

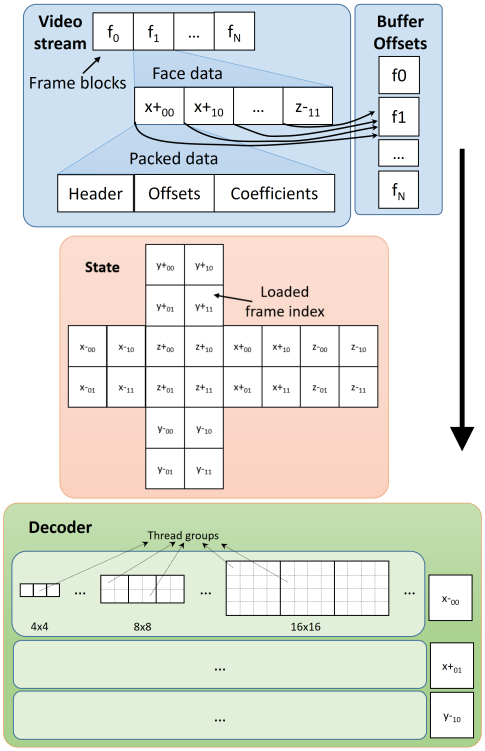


Fig. 4. Decoder overview. The video stream is organized in large buffer, with additional per-frame buffers storing per-cell offsets (top). In runtime, we determine which cells need to be decoded, we bound the buffer range of interest and the appropriate offset buffer and spawn decoding thread groups (middle). Each thread group decodes a fixed-size pixel tile in parallel, and all thread groups are executed in parallel (bottom).

memory, the only CPU-to-GPU communication required is binding the buffer using the buffer range, binding the look-up table for the given frame and uploading the list of cell indices that need to be updated. A decoding compute shader is then executed, dispatching a large number of thread groups, where each group is comprised of a 4×4 grid of threads that maps to a region of a node. Figure 4 shows the decoding logic, and the decompression process is described in algorithm 2. The majority of the decoding relies on bit operations, and is therefore inexpensive to calculate.

6 RESULTS

Our test system is an Intel Core i7 6700K with 32GB RAM and an NVIDIA Quadro P5000 GPU card with 16GB RAM. The input datasets were created using Pixar's RenderMan. The *Sponza* dataset consists of nine 360°cameras, 600 frames each. The *Pirate* dataset uses fifteen 360°cameras distributed in front of the face of the pirate, each consisting of 150 frames. This dataset demonstrates the capability for an end-to-end pipeline from real-world data to a real-time light field playback. The *Robot* dataset consists of sixteen 360°cameras, 78 frames each. This example poses several challenges for a faithful reconstruction, such as thin geometry (potted plant leaves) and highly specular and reflective surfaces (robot, floor,

Algorithm 2 Decompression of data for a single probe

Precondition: Coefficient data buffer B. Buffer offsets for current frame per face/cell O_B.
Output: Output depth cubemap d_{out} (6 slices)

```
1:  nodePartIndex ← workgroupID                                   ▷ Each part is an 4×4 block
2:  nodePartLocalCoords ← localThreadId                           ▷ values ∈ [1, 4]^2
3:  faceCellIndex, nodePartIndexInFaceCell ← CalculateFaceCell( nodePartIndex )   ▷ faceCellIndex ∈ [1, 24]
4:  bufferOffset ← O_B(faceCellIndex)
5:  (nodeCoeffs, nodeOffset, nodeLevel) ← ExtractBufferData(B, bufferOffset, nodePartIndexInFaceCell)
6:  nodeLocalCoords ← CalculateNodeLocalCoords( nodePartLocalCoords, nodePartIndexInFaceCell,nodeLevel)
7:  d ← 0
8:  if nodeLevel = 0 then
9:      d ← EvaluateRaw(nodeLocalCoords, nodeCoeffs)
10: else if IsBiquadratic(nodeCoeffs) then
11:     d ← EvaluateBiquadratic(nodeLocalCoords, nodeCoeffs)
12: else if IsPlatelet(nodeCoeffs) then
13:     d ← EvaluatePlatelet(nodeLocalCoords, nodeCoeffs)
14: else
15:     d ← EvaluateBC4w(nodeLocalCoords, nodeCoeffs)
16: d_out( nodeOffset + nodeLocalCoords, faceCellIndex/4 ) ← d
```

table). The *Horror* dataset consists of 9 360°cameras, 307 frames each. This dataset is the most challenging with regards to geometry, as the outside view includes millions of blades of grass and tree leaves, while it also includes specular and reflective surfaces (pipes). All dataset videos have a size of $1024 \times 1024 \times 6$ (cubemaps).

Our comparisons are against NVIDIA's HEVC implementation (table 1), Koniaris et al. [9] (U16 columns in figure 5 and table 1) and configuration variations of our codec.

6.1 Reconstruction quality and storage cost

The storage cost of the runtime data is calculated for our method using a set of quality settings versus temporal-only compression, utilizing the same lossless temporal compressor used in [9]. The results are shown in the right plot of figure 5. Using our codec, the storage cost lowers significantly compared to the original. There is less benefit for the Pirate dataset as the majority of the compression is temporal (large empty areas that do not update).

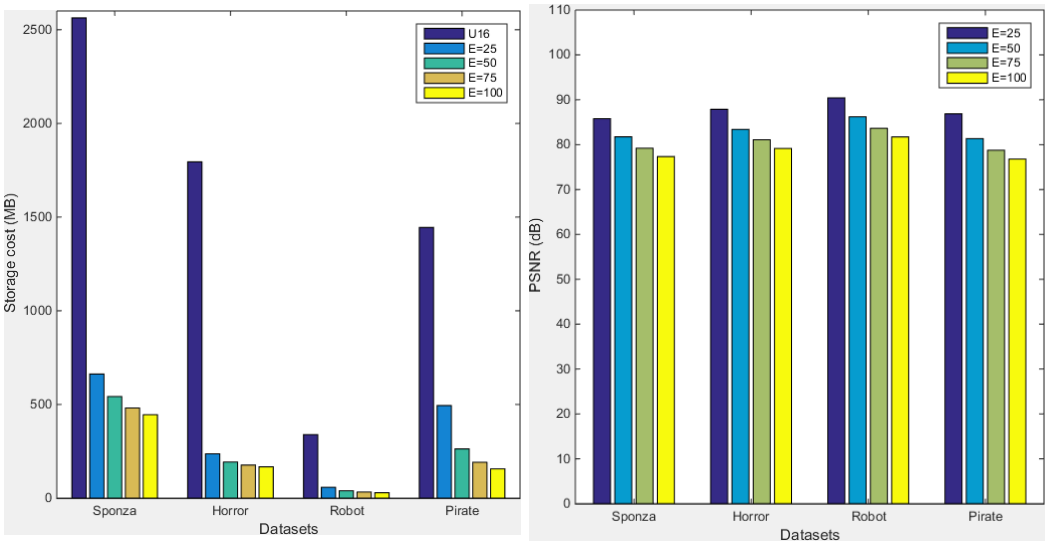
We measure the quality of the reconstruction by capturing reconstructed depth maps using our compressor versus temporal-only compression. The resulting depth maps are compared using PSNR. The results are shown in the left plot of figure 5. Even at the lowest quality setting, the PSNR remains high for all datasets. A video of the reconstruction quality using different error thresholds is included in the supplementary materials.

6.2 BC4w approximation quality

To measure the approximation quality of our BC4w format, we evaluated platelets, biquadratics and BC4w modeling functions in all 8×8 tiles of all frames of all datasets. The results are presented in figure 6. In the line plots, BC4w curves rise very quickly to good PSNRs, while only a few tiles reach near-perfect approximation. Using them in combination with platelets and biquadratic functions results in much improved approximations, as BC4w provides good approximations where others fail, while platelets and biquadratic provide near-perfect approximations that BC4w does not exhibit.

6.3 Decompression performance

We measure the decoding performance by recording and averaging timings and throughput over several thousand frames in a realistic scenario that includes user movement and looping video playback (example for Horror dataset shown in supplementary video). We additionally



(a) Storage costs

(b) Approximation errors

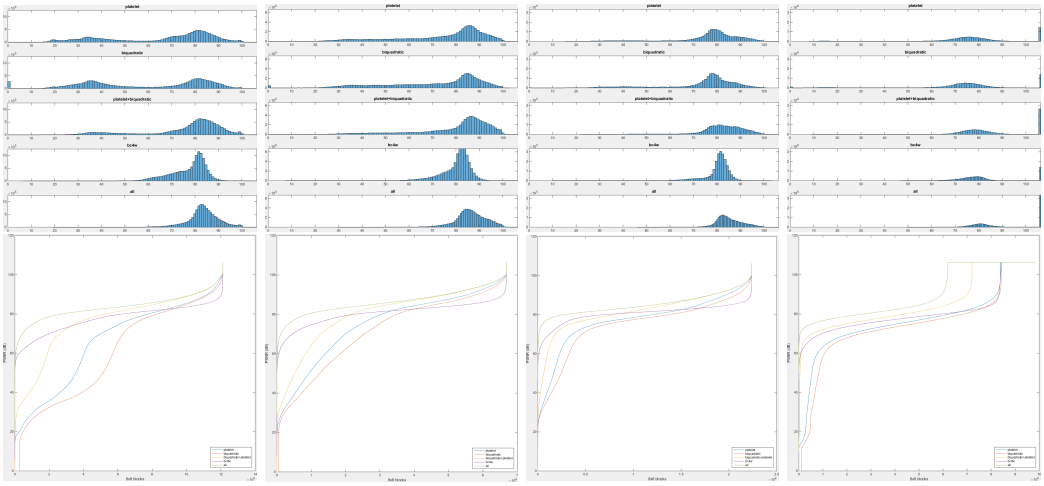
Fig. 5. Storage cost and approximation quality. In the left graph, we show the storage costs using just temporal compression (U16), versus the combined temporal and spatial compression using our codec with select quality settings.

	Size (GB)	Ratio (%)	Decode (ms)
HEVC-hq	0.009	0.031	57.5
HEVC-lossless	0.208	0.71	57.69
Koniaris [9]	0.159 (0.053)	0.54 (0.18)	1.62
Ours, E=100	0.029 (0.020)	0.099 (0.068)	0.42

Table 1. Depth compression comparison in Robot dataset. We compare compression and decoding performance against [9] and HEVC using high quality and lossless presets. Decoding measures decoding time for all faces from a subset of 9 views (54 streams in total). For [9] and our method, we include storage cost after lossless compression in parentheses. Our method has a clear advantage in decoding speed, which is paramount for the required data throughput. Losslessly compressed streams (as LZMA2 in [9]) are decompressed either at application startup or asynchronously, so decompression times are not included in the decoding times. While the decoding performance is more than 10× faster, the encoded dataset is marginally higher than 2× HEVC-hq.

capture the per-frame average pixel update rate, illustrating the decoding performance of the codec as a whole. The results are shown in figure 8.

We observe that the performance of our compression method is better compared to simple texture updates, even though the decompression complexity is *higher*. The improved decompression performance can be explained by the *batching* of block decompression tasks to as few shader calls as possible, and spawning a thread per block, exploiting the massively data parallel nature of the graphics processor (see section 5).



(a) Sponza (b) Horror (c) Robot (d) Pirate

Fig. 6. 8×8 tile approximation using various modeling functions. In the top row, we show histograms where the x axis corresponds to PSNR values and the y axis to the number of tiles approximated with such a PSNR. We show a histogram per modeling function, but also histograms for the minimum of platelet and biquadratic, and for the minimum of all three functions (raw not included in 8×8). In the bottom row, we show the distribution of approximation errors for each modeling function and for combinations as previously described. The x axis maps to tile index, where tiles are sorted by approximation PSNR, while the y axis shows PSNR values. From these graphs we can observe that BC4w has very few bad approximations, but also it rarely has near-perfect approximations. This means that, while it can be used as a good standalone lossy format, it is also really useful when included as another potential modeling function.

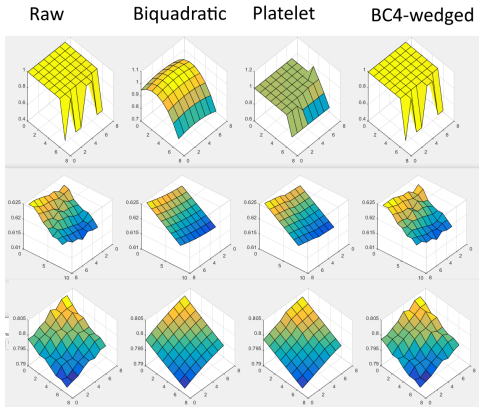


Fig. 7. 8×8 tile approximation comparison of 3 tiles (one per row) using our modeling functions (one per column). Our novel block compression format BC4w can very accurately capture high-frequency detail, in addition to discontinuities.

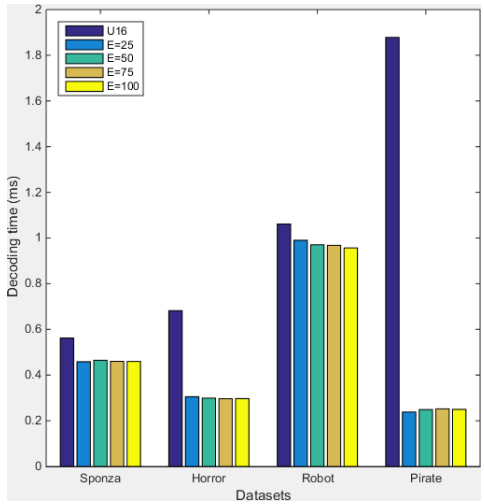


Fig. 8. Decoding performance comparison between uncompressed and ours with various error thresholds. Our compressed datasets are faster to decode in all cases, due to lower CPU-to-GPU communication overhead.

6.4 Implementation details

Compressor performance. Our compressor implementation is exhaustive, to guarantee near-optimal approximations using the modeling functions. The slowest modeling function approximator is BC4w, as we need to find an optimal 4D point in $[0, 65535]^4$ for each potential line that can intersect the tile. A standard optimisation that also exists in BC4 compressors is to reduce the search space by only considering coordinates near the maximum and minimum depth values of the input data. Our compressor is implemented in the GPU to exploit the massively parallel nature of the optimisation problem: a thread group is spawned for every 8×8 tile, and every thread in the group is assigned to an individual line. The kernel is executed for all 4D value combinations that we are interested in, and writes to a state the running optimal point. Due to the performance-intensive nature of the encoder and the occasional near-perfect approximations using platelets and biquadratics, we approximate 8×8 tiles first using other modeling functions, and if the PSNR is above 85 we don't approximate the tile, as most BC4w approximations result in PSNR 85 or less (see figure 6). Compressing a 1024×1024 image using our BC4w requires several minutes (5-10), and is an order of magnitude slower than our implementation of the platelet-based compression.

Depth mask and partial quadtree nodes. After the calculation of the depth mask, we need to determine the starting set of leaf 4×4 pixel nodes: if such a 4×4 tile contains even a single masked pixel, the node is added. In order to identify if a coarser node needs to be calculated, we typically require that 3 out of 4 children should exist: if we just require a single child to exist, then the updated area would be at least 75% redundant, therefore we require that the majority of the underlying area exists.

7 CONCLUSION

We have presented a codec that can be used to compress depth data, using quadtree subdivision and approximating nodes using modeling functions. We have introduced the modeling function BC4w that allows for depth variation while approximating tiles with

depth discontinuities. We have described a compressed depth data stream optimized for GPU decoding, suitable for high throughput data, as in the case of light field video. We have shown the improvement in compression rate and decoding performance compared to other methods.

The compression of BC4w tiles could be improved in the future by identifying the subspace where the optimal solutions are found in relation to the input depth extrema in order to further reduce the search space. Additionally, we would like to further research into extending or generalizing the format for use with 16×16 pixel tiles and above. Finally, we can further investigate the tiles that fail to get approximated with any of the stated modeling functions, and identify if a new modeling function can be introduced that can approximate this tile group.

ACKNOWLEDGMENTS

We would also like to thank the anonymous reviewers for the insightful comments that greatly helped improve this manuscript. This work has been supported in part by the InnovateUK funded project, OSCIR, grant number 102684.

REFERENCES

[1] Edward H Adelson, James R Bergen, et al. 1991. The plenoptic function and the elements of early vision. (1991).

[2] Filippo Bannò, Paolo Simone Gasparello, Franco Tecchia, and Massimo Bergamasco. 2012. Real-time Compression of Depth Streams Through Meshification and Valence-based Encoding. In *Proceedings of the 11th ACM SIGGRAPH International Conference on Virtual-Reality Continuum and Its Applications in Industry (VRCAI '12)*. ACM, New York, NY, USA, 263–270. https://doi.org/10.1145/2407516.2407579

[3] Alvaro Collet, Ming Chuang, Pat Sweeney, Don Gillett, Dennis Evseev, David Calabrese, Hugues Hoppe, Adam Kirk, and Steve Sullivan. 2015. High-quality Streamable Free-viewpoint Video. *ACM Trans. Graph.* 34, 4, Article 69 (July 2015), 13 pages. https://doi.org/10.1145/2766945

[4] Piotr Didyk, Tobias Ritschel, Elmar Eisemann, Karol Myszkowski, and Hans-Peter Seidel. 2010. Adaptive Image-space Stereo View Synthesis.. In *VMV*. 299–306.

[5] C Goktug Gurler, Anil Aksay, Gozde Bozdagi Akar, and A Murat Tekalp. 2009. Multi-threaded architectures and benchmark tests for real-time multi-view video decoding. In *Multimedia and Expo, 2009. ICME 2009. IEEE International Conference on*. IEEE, 237–240.

[6] Fabian Jäger. 2011. Contour-based segmentation and coding for depth map compression. In *Visual Communications and Image Processing (VCIP), 2011 IEEE*. IEEE, 1–4.

[7] Xiaoran Jiang, Mikaël Le Pendu, and Christine Guillemot. 2017. Light field compression using depth image based view synthesis. In *Multimedia & Expo Workshops (ICMEW), 2017 IEEE International Conference on*. IEEE, 19–24.

[8] Vahid Kiani, Ahad Harati, and Abedin Vahedian. 2017. Planelets - A Piecewise Linear Fractional Model for Preserving Scene Geometry in Intra-Coding of Indoor Depth Images. *IEEE Transactions on Image Processing* 26 (2017), 590–602.

[9] Babis Koniaris, Maggie Kosek, David Sinclair, and Kenny Mitchell. 2017. Real-time Rendering with Compressed Animated Light Fields. In *Proceedings of the 43rd Graphics Interface Conference (GI '17)*. Canadian Human-Computer Communications Society, School of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, 33–40. https://doi.org/10.20380/GI2017.05

[10] Jianjun Lei, Shuai Li, Ce Zhu, Ming-Ting Sun, and Chunping Hou. 2015. Depth coding based on depth-texture motion and structure similarities. *IEEE Transactions on Circuits and Systems for Video Technology* 25, 2 (2015), 275–286.

[11] Marc Levoy and Pat Hanrahan. 1996. Light field rendering. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*. ACM, 31–42.

[12] Shujie Liu, PoLin Lai, Dong Tian, and Chang Wen Chen. 2011. New depth coding techniques with utilization of corresponding video. *IEEE Transactions on broadcasting* 57, 2 (2011), 551–561.

[13] Yunpeng Liu, Stephan Beck, Renfang Wang, Jin Li, Huixia Xu, Shijie Yao, Xiaopeng Tong, and Bernd Froehlich. 2015. Hybrid Lossless-Lossy Compression for Real-Time Depth-Sensor Streams in 3D

Telepresence Applications.. In *PCM (1) (Lecture Notes in Computer Science)*, Yo-Sung Ho, Jitao Sang, Yong Man Ro, Junmo Kim, and Fei Wu (Eds.), Vol. 9314. Springer, 442–452.

[14] Morgan McGuire, Mike Mara, Derek Nowrouzezahrai, and David Luebke. 2017. Real-time global illumination using precomputed light field probes. In *Proceedings of the 21st ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*. ACM, 2.

[15] Philipp Merkle, Aljoscha Smolic, Karsten Muller, and Thomas Wiegand. 2007. Multi-view video plus depth representation and coding. In *Image Processing, 2007. ICIP 2007. IEEE International Conference on*, Vol. 1. IEEE, I–201.

[16] Yannick Morvan, Dirk Farin, et al. 2006. Platelet-based coding of depth maps for the transmission of multiview images. In *Proceedings of SPIE, Stereoscopic Displays and Applications*, Vol. 6055. 93–100.

[17] Karsten Muller, Heiko Schwarz, Detlev Marpe, Christian Bartnik, Sebastian Bosse, Heribert Brust, Tobias Hinz, Haricharan Lakshman, Philipp Merkle, Franz Hunn Rhee, Gerhard Tech, Martin Winken, and Thomas Wiegand. 2013. 3D High-Efficiency Video Coding for Multi-View Video and Depth Data. *Trans. Img. Proc.* 22, 9 (Sept. 2013), 3366–3378. https://doi.org/10.1109/TIP.2013.2264820

[18] Dawid Pająk, Robert Herzog, Radosław Mantiuk, Piotr Didyk, Elmar Eisemann, Karol Myszkowski, and Kari Pulli. 2014. Perceptual depth compression for stereo applications. In *Computer Graphics Forum*, Vol. 33. Wiley Online Library, 195–204.

[19] Fabrizio Pece, Jan Kautz, and Tim Weyrich. 2011. Adapting Standard Video Codecs for Depth Streaming. In *Joint Virtual Reality Conference of EGVE - EuroVR*, Sabine Coquillart, Anthony Steed, and Greg Welch (Eds.). The Eurographics Association. https://doi.org/10.2312/EGVE/JVRC11/059-066

[20] Jeff Pool, Anselmo Lastra, and Montek Singh. 2012. Lossless compression of variable-precision floating-point buffers on GPUs. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*. ACM, 47–54.

[21] Jacob Ström, Per Wennersten, Jim Rasmusson, Jon Hasselgren, Jacob Munkberg, Petrik Clarberg, and Tomas Akenine-Möller. 2008. Floating-point buffer compression in a unified codec architecture. In *Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*. Eurographics Association, 75–84.

[22] Krzysztof Wegner, Olgierd Stankiewicz, and Marek Domanski. 2014. Fast View Synthesis using platelet-based depth representation. In *Systems, Signals and Image Processing (IWSSIP), 2014 International Conference on*. IEEE, 55–58.

[23] M. O. Wildeboer, T. Yendo, M. P. Tehrani, T. Fujii, and M. Tanimoto. 2010. Color based depth up-sampling for depth compression. In *28th Picture Coding Symposium*. 170–173. https://doi.org/10.1109/PCS.2010.5702451

[24] Andrew D. Wilson. 2017. Fast Lossless Depth Image Compression. In *Proceedings of the 2017 ACM International Conference on Interactive Surfaces and Spaces (ISS '17)*. ACM, New York, NY, USA, 100–105. https://doi.org/10.1145/3132272.3134144