RANDM: Random Access Depth Map Compression Using Range-Partitioning and Global Dictionary

Srihari Pratapa and Dinesh Manocha

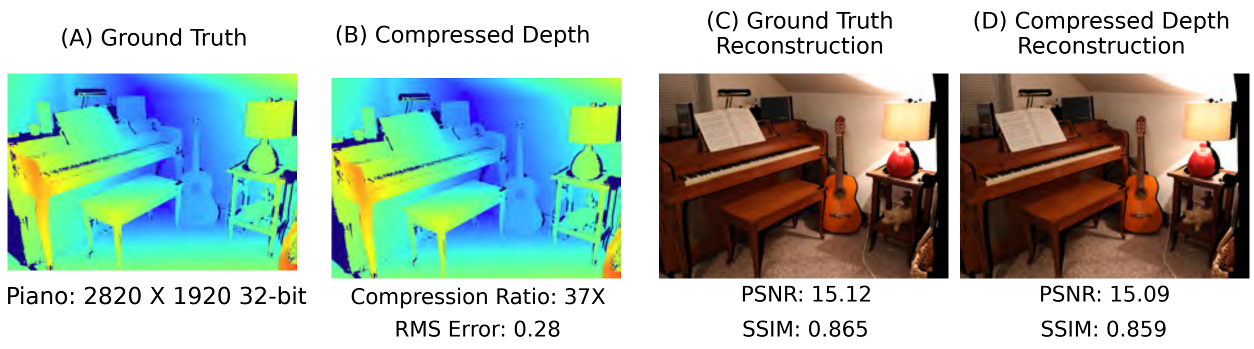


Fig. 1: We show a visual comparison between the ground truth depth maps and depth map compressed using our algorithm (RANDM). We do not see any artifacts or noticeable error We also highlight the error in visual reconstruction algorithms based on the ground truth and depth map compressed using RANDM. (A) Actual ground truth depth data; (B) Depth data compressed using RANDM; (C) Right-view reconstructed from ground truth; (D) Right-view reconstructed from the compressed depth map. The random access capability of our approach is used for interactive rendering and VR applications.

Abstract— We present a novel random-access depth map compression algorithm (RANDM)for interactive rendering. Our compressed representation provides random access to the depth values and enables realtime parallel decompression on commodity hardware. Our method partitions the depth range captured in a given scene into equal-sized intervals and uses this partition to generate three separate components that exhibit higher coherence. Each of these components is processed independently to generate the compressed stream. Our decompression algorithm is simple and performs prefix-sum computations while also decoding the entropy compressed blocks. We have evaluated the performance on large databases on depth maps and obtain a compression ratio of $20-100\times$ with a root-means-square (RMS) error of $0.05-2$ in the disparity values of the depth map. The decompression algorithm is fast and takes about 1 microsecond per block on a single thread on an Intel Xeon CPU. To the best of our knowledge, RANDM is the first depth map compression algorithm that provides random access capability for interactive applications.

1 Introduction

Virtual reality (VR) and augmented reality (AR) technologies are increasingly being used for immersive viewing experiences using head mounted displays (HMD) and 3D projection display systems. Current VR systems can display wide field-of-view content at high pixel densities, and this is combined with accurate tracking of the viewer's head position and orientation for interactive rendering. To increase the sense of immersion and make the viewing experience natural, there has been considerable work done on analyzing different factors corresponding to motion parallax, depth cues, and binocular disparity [55] and perform accurate rendering based on these factors.

Earlier work in VR mainly used synthetic content based on 3D geometric models of virtual environments. Over the last decade, there has been considerable work on capturing real-world scenes based on new camera systems and computer vision techniques to generate VR or AR experiences. These captured datasets are typically represented as depth maps. A *depth map* is an image that contains information about the distance between the surface of objects from a given viewpoint. Moreover, this map is merged with the RGB source image to create a "3D image". Depth maps are used in light field rendering and image-based rendering (IBR) [15, 16, 49] to provide depth cues for accurate rendering. In 360° immersive videos, panoramic depth maps are used to enable motion parallax. This results in translational shifts in the viewer's head to support 6-degrees-of-freedom (6-DoF) motion and to provide higher realism in an HMD [20, 29, 59]. Many real-time telepresence systems use depth sensors to capture a large sequence of depth maps and transmit them with the color data to reconstruct the 3D scene [39].

There is considerable work on capturing depth information using hardware sensors [24] or using computer vision techniques on stereo [37] and monocular images [35]. These include techniques based on time of flight (ToF) cameras [12], structural light cameras [19], or the use of multiple camera views [26]. These methods are increasingly being used in research and commercial systems to capture depth maps. Recently, high-end consumer mobile phones have started to support depth sensors to enable more extended/augmented reality (XR/VR) applications [1].

Generally, a single depth map is captured or computed and associated with each of the captured color images. In many IBR applications, each captured view has a per-view depth map associated with it. For 360° immersive videos, a per-frame depth map is stored and used for interactive rendering. Hence, the size of depth map data is directly proportional to the color or RGB data in terms of size and resolution. For a good-quality IBR, the amount of RGB data required is quite high and can vary from hundreds of MB [5] to hundreds of GB [31, 33]. Recently, devices for capturing very high-resolution (e.g., 2K or 4K or higher) 360° videos for rendering in VR have become widely available [1], and result in data sizes capture rates of 1.2 GB/s at

• *University of North Carolina at Chapel Hill E-mail: psrihariv@cs.unc.edu.*
• *University of Maryland at College Park E-mail: dm@cs.umd.edu.*

Manuscript received xx xxx. 201x; accepted xx xxx. 201x. Date of Publication xx xxx. 201x; date of current version xx xxx. 201x. For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org. Digital Object Identifier: xx.xxxx/TVCG.201x.xxxxxxx

[1]https://bit.ly/2x0Pz4f

60 fps. To support VR or AR applications, we need techniques for efficient compression, storage, transmission, and rendering of such high resolution depth maps.

Real-time rendering and VR/AR applications impose additional constraints on the design of compression schemes. These constraints include low-complexity decoding and utilizing fast parallel hardware on commodity processors. For interactive applications, only a portion of the RGB pixels and the corresponding depth values are typically required at run-time for further processing or rendering a scene [16, 30, 52]. As a result, we need to be able to perform selective decoding on a compressed depth map on the hardware, as storing of large amounts of uncompressed depth data can result in memory bandwidth bottlenecks [9]. Many of the current techniques to compress depth maps are similar to JPEG or MPEG are designed for high compression efficiency [25, 28]. They are not best suited for interactive applications, as these methods need to decode the entire depth map at run-time and store it in the memory. Instead, most interactive rendering algorithms tend to use random access compression schemes for selective decoding and to reduce the memory bandwidth requirement [8]. Random access compression schemes are widely used to compress textures [2, 21] and are supported in current GPUs. The idea of random access for compressing has been extended to other image-based representations [17], including videos and lightfields [30, 53, 72]. However, there is no work on developing random-access compression schemes for depth maps.

Main Results: We present a new and fast algorithm (RANDM) for compressing and decoding depth maps based on random access capabilities for interactive applications. The main idea behind our approach is reducing the range of depth values in a scene to a much smaller range and re-map all the depth values to the smaller range. We partition the depth range of a given depth map into several equal-sized intervals, and each depth pixel in the depth map is assigned to a specific interval. Given the partitioning, we decompose the overall depth map into three different parts. First, a new pixel-index image (PI) is computed for storing the interval index for all the depth pixels in the depth map. Second, we compute a global depth dictionary (GDD) to store all normalized depth values for the entire depth map. We use this representation to gather all the coherent depth values across the depth map into the global depth dictionary. Third, a new dictionary-index image (DI) is created to store the dictionary index (in GDD) for all the corresponding normalized depth values in the depth map. We process these three components, PI, DI, and GDD, independently and process them using entropy encoding (arithmetic encoding) to compute the final compressed stream. At runtime, we use additional buffers and arrange the final compressed stream to facilitate random access and selective decoding of depth pixels for interactive rendering.

The main contributions of our approach (RANDM) include:

1. First random-access lossy depth map compression scheme suitable for use in interactive applications;

2. New range-split global dictionary method for encoding depth maps; our compression scheme is highly parallelizable for fast real-time encoding of depth maps;

3. Low-complexity decoding scheme that is amenable for fast parallel decoding on hardware and interactive rendering.

We have evaluated our method on a large set of depth maps from Middlebury datasets [18, 56] and TUM RGBD datasets [65]. Our method obtains a compression ratio of $20-100\times$ for a root-means-square (RMS) error of $0.05-2.5$ in the disparity (inverse of depth) values of the depth map. The decompression times for decoding a block (block size: 8) of depth values are up to 1 microsecond using a single thread on an Intel Xeon CPU. We have evaluated the reconstruction error of pairs of stereo images using our compressed depth maps, and our method provides similar reconstruction quality as the original uncompressed depth maps.

2 Prior Work and Background

In this section, we give an overview of prior techniques used to compress image-based representations and depth maps.

2.1 Compression of Image-Based Representations

Different image-based representations are used for interactive rendering. Shum et al. [17] present a taxonomy of rendering approaches based on different image-based representations: textures, depth, videos, lumigraph, light fields, etc. We broadly classify the existing compression schemes for image-based representations into two categories:

2.1.1 High-Efficiency Compression Schemes

High-efficiency compression schemes for image-based representations are modifications or direct applications of the traditional image and video compression approaches, including static image compression methods (JPEG, PNG, etc.) and video compression schemes (MPEG-2, H.264, H.265, etc.). Many compression schemes are designed for light fields and lumigraph that extend techniques from standard image and video compression schemes such as discrete-cosine transform, wavelet transform, predictive block encoding, and motion-vector compensation. One set of these methods orders light field images (LFI) in a sequence and applies one or more of the mentioned techniques from video codes for compression [7, 34, 50]. Other sets of methods select fixed reference frames and predict the rest of the frames using predictive block coding then apply domain transformation techniques to separate the data into important and non-important parts [11, 22, 38]. All of these approaches achieve very high compression ratios from $100\times$ to $1000\times$. High-efficiency compression schemes for encoding depth maps have been developed, and are described below.

2.1.2 Random Access Compression Schemes

These techniques are designed to provide selective decoding for interactive rendering. Delp and Mitchell [8] introduce a fixed rate per block compression method for encoding grayscale images. Beers et al. [2] present a random access compression scheme for textures and list the main requirements for random access compression schemes to be feasible for interactive applications: random access, low-complexity decoding, and visual quality of decompressed data. In the last few years, many random access compression schemes for encoding textures has been developed [9, 21, 46, 48, 62, 63] and these schemes are widely supported on commodity GPUs. Several other super-compression schemes [10, 27, 64] have been introduced to provide additional compression. Different random access compression schemes for encoding video have been proposed for video rendering [13, 42, 52, 58, 67]. Some random access compression methods have been proposed for lightfields based on vector-quantization (VQ) [30], hierarchical representations [51, 53], motion-compensation [49, 72], etc.

2.2 Depth Map Compression

Several lossy schemes have been proposed to compress depth maps that are based on standard image compression schemes (JPEG & JPEG2000). In practice, depth maps have different properties than standard images (color images), and, they are not directly used for viewing. On the other hand, standard image compression schemes are designed to optimize the maximum perceived visual quality, and a direct application of those methods to depth maps may not be optimal. We categorize the existing compression schemes, for depth maps into two categories, *lossy compression schemes*, and *lossless compression schemes*, and none of the existing approaches provide random access capability.

2.2.1 Lossless Schemes

Mehrotra et al. [43] present a lossless entropy encoding scheme for Kinect like depth sensors. They store the depth values using inverse depth coding as integer values with a dynamic range of 16-bits. Run-length encoding with Golomb-Rice code [40] is used to encode the inverse depth values, after arranging the depth values in raster scan order. Differential pulse-code modulation (DPCM) is used as the predictive method to exploit the spatial coherence between the neighboring pixels in a raster scan. Wilson [69] present a lossless method similar to Mehrotra et al. [43], where the Golomb-Rice encoding is replaced with new variable-length encoding to code the depth values after raster scanning and to apply DPCM to the neighboring pixel values. This method achieves compression rates of around 1bb - 3bpp on 16-bit depth maps, and overall provides $\sim 4-16\times$ compression.

2.2.2 Lossy Schemes

There is considerable work on lossy schemes designed for a single depth image or a video.

Static Depth Maps: Sarkis and Diepold [54] present an approach based on compressive sensing to compress the depth image. To ensure sparsity, this method uses regularization techniques and preserves the properties of the depth maps, including depth discontinuities, using total variations constraints. They achieve compression ratios of up to $10\times$. Krishnamurthy et al. [28] use a region of interest (ROI) coding for depth maps based on JPEG2000 and achieve compression rates of $\sim 50\times$. Chai et al. [4] present mesh-based generation methods, where a mesh is generated from the depth map. The computed mesh geometry is encoded using a binary tree structure, and depth values from the pixels are stored in the tree nodes, resulting in compression ratios of $\sim 30X$.

Depth Map Videos: Kim and Ho [25] present a depth map compression scheme for rendering 3D videos. Each depth map for a frame of the video is hierarchically divided into one of four different regions, based on the edges in the depth map. Next, the regions are merged and the sequence of frames (video) is compressed using a video codec. Liu et al. [36] describe a new trilinear filtering scheme that is used as a replacement for the deblocking filter in H.264 to preserve the depth discontinuities after motion compensation. Wildeboer et al. [68] present a method where depth maps from a video sequence are down-sampled and compressed using the H.264 scheme. During up-sampling, a weight function similar to bilateral filtering that uses both color data and pixel distance to preserve the depth is used. This method achieves a compression ratio of around $100-500\times$.

2.2.3 Z-buffer Compression

A depth map referred to as Z-buffer is part of a standard graphics rasterization pipeline and is used to perform visibility tests [66]. A simple or naive implementation of Z-buffer for visibility testing consumes large amounts of memory bandwidth. Several methods have been proposed to compress the Z-buffer [14]. The primary goal of these approaches is to reduce the size of the Z-buffer while enabling real-time parallel visibility tests during rasterization. Z-buffer compression methods have to be lossless because any loss might lead to errors in the final rasterized image. These methods are mostly based on lossless tile-based, fixed bit-rate compression schemes and can enable parallel random access to the compressed Z-buffer. The compression rates achieved by these methods are typically $2-4\times$. Although these approaches enable random access, the compression rates are quite low due to fixed bit-rate and application-specific constraints.

3 Our Method: RANDM

In this section, we present our compression pipeline (Fig. 2) and describe our encoding method. Depth maps are sometimes represented as disparity maps, indicating the binocular disparity in terms of pixel shifts with respect to a reference camera position [41]. The disparity maps are considered to be the inverse of depth maps [3, 61]. The input to our method is a disparity map or a depth map of a given scene. If the input is a disparity map, we convert it into the corresponding depth map using the camera calibration parameters as the input. Depending on the precision required by an application or method of capture, different dynamic ranges (8-bit integer, 16-bit integer, 16-bit floating, or 32-bit floating) are used to represent the depth values. For the rest of the paper, we assume that each depth value is an integer, and the same approach can be easily extended to other data types. Our approach is general and can handle any dynamic range of depth values. The output of our RANDM algorithm is a compressed stream that enables random access and parallel decoding of the depth values from the compressed stream.

The main idea behind our approach is to remap a large range of depth values to a much smaller range by decomposing the values into three different components. Each of these components has integer values in a fixed range that is much smaller than the actual range of depth values in the input scene. For a fixed data size, a smaller range of bounded values exhibits less entropy and leads to higher compression. The smaller range for a given input is computed by partitioning the depth range into equal-sized intervals that are smaller than the actual range. The number of intervals is set using an encoding parameter. In order to compute the remapping, we compute a new image *pixel index image* (**PI**) for storing the interval information for all the depth pixels in the depth map. After remapping the depth values to a smaller range, we compute a *global depth dictionary* (*GDD*) to store only the unique depth values in the smaller range removing duplicates. All the similar depth values present across the entire depth map are gathered into one *GDD* to exploit the coherence in a global manner. We quantize and convert the values in *GDD* to integers. Next, we compute a *dictionary index image* (**DI**) to index into the *GDD* to store the remapped smaller range values for all the depth pixels in the depth map. The three components (**PI**, **DI**, and *GDD*) computed have bounded ranges of integer values, which reduces the overall entropy and makes them suitable for compression. The *GDD* is further processed, and entropy encoded. Finally, the **PI** and **DI** are divided into non-overlapping blocks, and the blocks are entropy encoded. We use additional offset arrays to store the lengths of compressed block stream to facilitate random access decoding for interactive applications.

3.1 Notation and Terminology

The 2D input depth map is represented by **Z** and has resolution $M \times N$. $Z_{(x,y)}$ denotes the depth value at a depth pixel indexed by $(x,\ y)$. Our approach is formulated based on the following representations and components.

Interval divisions: The depth range in the input **Z** is partitioned into intervals of equal size. The intervals are ordered serially ($Interval_0$, $Interval_1$, ... and so on) and marked with an interval index (an integer value). The *number of intervals* is set as an encoding parameter.

Interval size (*IS*): A single scalar value representing the size of the resulting interval after partitioning the depth range into equal intervals.

Pixel Index Image (PI): A 2D image to store the interval index for all the depth pixels in the input depth map **Z**. The resolution of **PI** is $M \times N$, which is the same as in the input depth map (**Z**). **PI** stores the interval index, which is an integer value and the range of values is constrained to the *number of intervals*.

Depth normalization: The depth values for all the depth pixels in the depth map are normalized based on the interval index stored in **PI**. After normalization, all the depth values in the depth map are reduced to a much smaller range, *IS*.

Split Depth Map (SDM): An intermediary 2D image computed to store the normalized depth values for all depth pixels. The resolution of the **SDM** is same as the input resolution $M \times N$.

Global Depth Dictionary (*GDD*): A global depth dictionary computed to store all the unique normalized depth values from the **SDM**. The *GDD* is represented as a hash map and implemented using a linear array. The values stored in the *GDD* can be referred using an index into the linear array.

Dictionary Index Image (DI): This is a 2D image that is used to store the dictionary index of all the pixels in the **SDM**. **DI** stores a dictionary index that is an integer value, and the range of values stored is constrained to the size of *GDD*. The resolution of **DI** is the same as the input resolution $M \times N$.

Our method uses the following encoding parameters in our compression scheme:

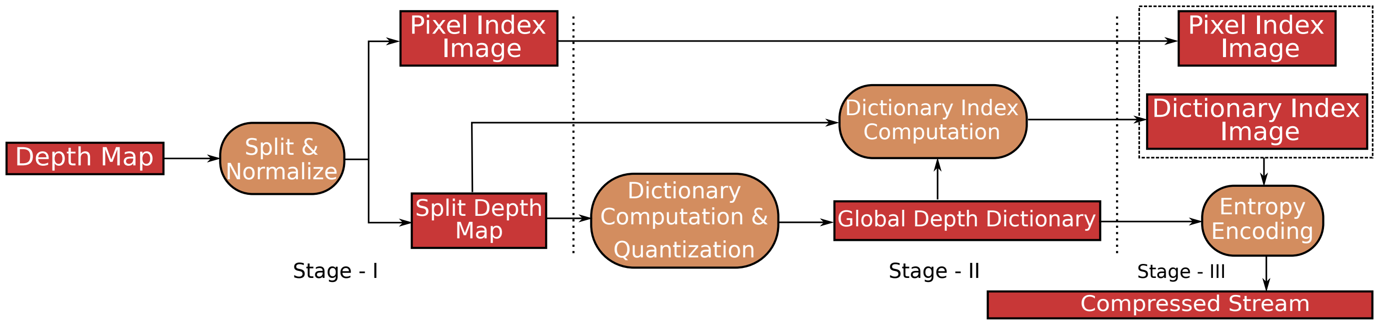


Fig. 2: *Our compression pipeline*: The red rectangles represent the data, the brown ovals the represent processing blocks, and the arrows indicate flow and data transfer operations. Our compression pipeline consists of three stages: (Stage-I) The depth map is partitioned into several equal sized intervals, and depth values in all the intervals are normalized to decrease the range of depth values to a smaller range. A pixel index image (*PI*) is computed to store the interval index of the pixels in the depth map; (Stage-II) We compute the global depth dictionary (*GDD*) is to capture the coherence in the depth map at a global scale; (Stage-III) The dictionary index image (*DI*) is computed for the depth map to store the index of the normalized depth values in *GDD*. Finally, *PI*, *GDD*, and *DI* are processed and entropy encoded to compute the final compressed bitstream along with offset arrays.

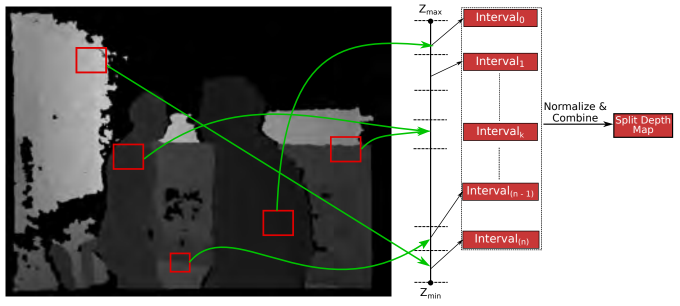


Fig. 3: We highlight the partitioning of the depth range into intervals and computation of a normalized split depth map. Z_{min} and Z_{max} indicate the minimum and maximum depth, respectively, for the input depth map. Each depth pixel in the depth map belongs to a particular interval depending on the depth value. We highlight a few depth values in the depth map with boxes and also point to the specific intervals ($Interval_i$) that they belong to. The depth values in all the intervals are normalized to reduce the range of the depth values in all splits to the same range.

1. *Number of intervals* (*NI*): Number of intervals into which a given depth range is partitioned;

2. *Precision* (*Pr*): The minimum error in precision allowed in the depth values in some units (e.g., 0.1 , 0.01, 0.001, etc.). This is only applicable for input depth maps with floating point precision.

3. *Dictionary error* (*DE*): The error allowed in the quantization of the normalized depth values stored in the global depth dictionary (GDD);

4. *Block size* (*B*) : **PI** and **DI** images are divided into smaller non-overlapping rectangular blocks (of a given *block size*) for further compression. These blocks are used for selective decoding at runtime.

3.2 Division of Depth Range

From the input depth values, the minimum depth (Z_{min}) and the maximum depth (Z_{max}) captured in the scene are computed. The depth range is partitioned into equal intervals, and the interval size (*IS*) is computed based on the *NI* set as encoding parameters:

$$IS = \frac{Z_{max} - Z_{min}}{NI}. \tag{1}$$

Each pixel in the depth map belongs to an interval depending on the depth value (Z) of the pixel, and the corresponding interval index is computed as:

$$PI_{(x,y)} = interval\ index_{(x,y)} = \lfloor \frac{Z_{(x,y)} - Z_{min}}{IS} \rfloor. \tag{2}$$

$Z_{(x,y)}$ denotes the depth value at pixel index (x,y) and $interval\ index_{(x,y)}$ is the corresponding interval index (integer value) of the pixel at index (x,y) in the depth map. The interval indices are computed for all the pixels and are stored in the pixel index image (**PI**). Once the **PI** is computed, we normalize the depth value of each pixel and compute a new intermediary split depth map (**SDM**) as follows:

$$SDM_{(x,y)} = Z_{(x,y)} - (PI_{(x,y)} * IS) - Z_{min}; \tag{3}$$

$SDM_{(x,y)}$ denotes the normalized depth value for a pixel at index (x,y) for a given interval index in $PI_{(x,y)}$. The depth values computed in the **SDM** are bound to lie in the range of $[0,\ IS]$. The values in the **PI** are integers and are bound between the range of $[0,\ NI-1]$. Using intervals and normalization, we compute two new 2D images with bounded range of values, increasing the overall coherency and resulting in data that is amenable to better compression. The original depth values of a pixel can be recomputed without loss using the newly computed **SDM** and **PI** images. Fig. 3 illustrates the splitting of the depth range.

3.3 Global Depth Dictionary

Once **SDM** is computed, a new linear *global depth dictionary* (*GDD*) is computed by gathering all the unique normalized depth values stored in **SDM**. *GDD* is stored as a hash map and implemented using a linear array. We refer to the values held in the *GDD* as the normalized depth values. Using the *precision* set as an encoding parameter we compute a precision factor (pf). If the input depth values are within floating point precision, the values in *GDD* are multiplied by the pf and rounded to the nearest integers. We compute the precision factor (pf) from the input *precision* (*Pr*) and convert the values of *GDD* into integers:

$$\begin{gathered} pf = 10^{|(\log_{10}(Pr))|}, \\ GDD_k = \lfloor GDD_k * pf \rceil \ \ \forall k \in [1, \#(GDD)], \\ \#(GDD) = \text{size of } GDD. \end{gathered} \tag{4}$$

GDD_k denotes the residual depth value at index k. After the *GDD* is converted to an integer value, it is further quantized to reduce the size based on a *dictionary error* set as the encoding parameter. The values of the *GDD* are divided into disjoint subsets such that no two depth values in the same subset have greater than input *dictionary error* (*DE*).

$$S_l = \{GDD_{k_1}, GDD_{k_2}, ...GDD_{k_n}\}, \quad (5)$$
$$\forall (k_u, k_v) \in \{k_1, k_2, ...k_n \ni |GDD_{k_u} - GDD_{k_v}| < DE\}.$$

For each subset S_l computed from *GDD*, the mean of all the normalized depth values from the subset S_l is computed. Next, we discard the old *GDD* and compute a new *GDD* storing only the mean values computed from all the sets (S_l).

$$GDD_l = \frac{\sum_{v_i \in S_l} v_i}{\#(S_l)}, \quad (6)$$
$$\#(S_l) = \text{size of subset } \ \& \ \forall l \in [1, M],$$
$$M = \text{number of subsets.}$$

Based on the value of *ED* set as an encoding parameter, the final size of the *GDD* is reduced by a factor that introduces loss in the depth values and is amenable to significant compression.

Algorithm 1 Compress depth map. We highlight the main steps of the pipeline. It is simple and parallel friendly

```
Input:
  Depth map or Disparity map: DM
  Encoding parameters: enc
Output:
  CompressedStream
  function COMPRESSDM(DM, enc)
    //Compute the split size based on depth range
    SplitSize ← $ComputeSplitSize(\mathbf{DM}, enc)$
    //Compute the pixel index iamge (PI) based on split size
    PI ← $ComputeDivisionIndicies(\mathbf{DM}, SplitSize)$
    //Compute the normalized split depth map (SDM)
    SDM ← $NormalizeDepthValues(\mathbf{DM}, SplitSize, \text{PI})$
    /*Gather all the unique normalized depth values from
    SDM into global depth dictionary (GDD)*/
    GDD ← $GatherUniqueValues(\text{SDM})$
    //Quantize the normalized depth values stored in GDD
    GDD ← $QuantizeGDD(\text{GDD}, enc)$
    //Compute the dictionary index image (DI) from GDD
    DI ← $ComputeDictionaryIndices(\text{SDM, GDD})$
    //Differential pulse code modulation applied to GDD
    $GDD_{dpcm}$ ← $DPCM(\text{GDD})$
    //Entropy Encode DPCM GDD
    GDDStream ← $EntropyEncode(GDD_{dpcm})$
    //Entropy Encode PI and DI
    PIStream ← $EntropyEncode(\text{PI})$
    DIStream ← $EntropyEncode(\text{DI})$
    //Append the streams and return the final stream
    CompressedStream ← (GDDStream: PIStream: DIStream)
```

3.4 Dictionary Index Image

For each residual pixel value in the **SDM**, we compute a closest matching (based on absolute error) normalized depth value from the *GDD*. The *dictionary index image* (**DI**) is computed to store the index of the closest matching normalized depth value from the *GDD*.

$$\min_k |SDM_{(x,y)} * pf - GDD_i| \forall i \in [1, M], \quad (7)$$
$$DI_{(x,y)} = k.$$

M is the size of the quantized *GDD*. The range of values stored in the *dictionary index image* are bound within the range of $[1, M]$, bounding the range to facilitate better compression. Once the **DI** is computed, the depth values of the image can be re-computed from the values of *GDD*, **PI**, and **DI**:

$$k = DI_{(x,y)} \quad (8)$$
$$\overline{Z}_{(x,y)} = PI_{(x,y)} * IS + Z_{min} + (GDD_k / pf)$$

$\overline{Z}_{(x,y)}$ is the re-computed depth value at pixel index (x, y).

3.5 Entropy Encoding

The normalized depth values in the *GDD* are sorted in increasing order before the **DI** is computed. We apply Differential-pulse code modulation (DPCM) [47] to the *GDD* values, reducing the range of the values and decreasing the entropy improving compression. We sort *GDD* as it guarantees an increasing order of values, which results in only a positive range of the values resulting after DPCM.

$$GDD_{dpcm} = DPCM(GDD)$$

After DPCM, the GDD_{dpcm} is entropy encoded using adaptive arithmetic encoding [70].

The **PI** is divided into non-overlapping rectangular blocks and each block is independently compressed using adaptive arithmetic encoding. An array of block offset values is used to store the entropy compressed length of all the compressed blocks. The block offsets are used to facilitate random access to the compressed data. The **DI** is processed and compressed similar to the **PI** and additional block offset array is used to store the compressed block lengths. The block offset arrays are further encoded using entropy encoding (arithmetic encoding) to reduce the final size of the compressed stream.

Algorithm 2 Decompress depth map block. It is simple, fast and runs at realtime rates on commodity hardware.

```
Input:
  LFI compressed stream: CompDM
  Block index: BlkIdx
Output:
  Depth values: DepVals
  // Load the stream into memory and separate
  Initialization:
  PIStream ← $ReadPIStream(\mathbf{CompDM})$
  DIStream ← $ReadDIStream(\mathbf{CompDM})$
  PIBlockoffsets ← $ReadPIBlockOffsets(\mathbf{CompDM})$
  DIBlockoffsets ← $ReadDIBlockOffsets(\mathbf{CompDM})$

  GDDStream ← $ReadGDDStream(\mathbf{CompDM})$
  $GDD_{dpcm}$ ← $EntropyDecode(\text{GDDStream})$
  GDD ← $PrefixSum(GDD_{dpcm})$

  function DECOMPRESSDMBLOCK(BlkIdx)
  // Get the start location of BlkIdx in bitstream
  PIBlockOffset ←PIBlockOffsets[BlkIdx]
  DIBlockOffset ←DIBlockOffsets[BlkIdx]
  // Read the PI and DI block compressed streams
  PIBlockStream ← $ReadBlocks(\text{PIStream, PIBlockoffsets})$
  DIBlockStream ← $ReadBlocks(\text{DIStream, DIBlockoffsets})$
  // Entropy decode PI and DI block streams
  PIBlockVals ← $EntropyDecode(\text{PIBlockStream})$
  DIBlockVals ← $EntropyDecode(\text{DIBlockStream})$
  // Recompute the block of depth values
  DepVals ← $ReComputeDepthVals(\text{PIBlockVals, DIBlockVals, GDD})$
```

3.6 Decompression

The input to our decompression scheme is the final compressed stream from our encoding approach. In the initialization stage of decompression, the entropy encoded GDD_{dpcm} is decompressed to retrieve the GDD_{dpcm}. A prefix-sum is performed on the GDD_{dpcm} to construct back the *GDD*. After that, the block offset values are computed by decoding the corresponding entropy compressed blocks of **PI** and **DI**.

For a given pixel location, we compute the corresponding block index for a given *block size*. Once the block index is computed, the start location of the current block's entropy compressed stream is located using the block offset values. We gather the block entropy compressed streams and decode them to retrieve the blocks of **PI** and **DI** values. After that, we re-compute the depth value using Eq. 8. Using the additional block offsets, only the required parts of the **PI** compressed stream and **DI** compressed stream.

3.7 Performance Analysis

Algorithm-1 highlights the high-level steps of our compression scheme. The steps to compute **PI** and **SDM** in the first stage are straight forward, as shown in Eq. 2 and Eq. 3. In the second stage, once the *GDD* is computed and sorted, the quantization step is performed in linear time in the size of the *GDD*. In the third stage, the dictionary index for all the depth pixels can be computed using binary search on the quantized sorted *GDD* to generate the *DI*. After the GDD_{dpcm} is entropy encoded, the **PI** and **DI** are entropy compressed. The entropy encoding of **PI** and **DI** can be parallelized at a block-level because all the blocks are independently entropy encoded.

The pseudo-code of our decompression scheme is presented in Algorithm-2. In the initialization stage, the entropy encoded block offset arrays for the **PI** and **DI** images are decompressed and loaded into the memory. Next, the entropy encoded GDD_{dpcm} is decoded and followed by a prefix-sum to compute *GDD*. To decode the required block of pixels, we perform the following steps: 1. Read the corresponding entropy compressed **PI** and **DI** block streams from memory based on the offset values; 2. Perform entropy decoding to retrieve the blocks of **PI** and **DI**; 3. Sum up the values from the three components using Eq. 8. Our decompression scheme is simple because it primarily requires only reading two small bit streams from memory and entropy decoding both the streams to re-compute the final depth values. Each step is parallel friendly.

3.8 Compression Analysis

We briefly examine the relationship between the primary encoding parameters that control the compression ratio and compression quality. The **PI** and **DI** are entropy encoded losslessly; therefore the compressed sizes of the entropy encoded **PI** and **DI** mainly depend on the frequency distribution of values in the corresponding blocks of **PI** and **DI**. The larger the range of values to be stored in **PI** and **DI**, the higher the resulting entropy; this leads to larger compressed sizes of **PI** and **DI** components.

- *Number of intervals* (*NI*): The interval size (*IS*) is computed based on the *NI* (Eq. 1), which affects the range of normalized depth values (Eq. 3). The *GDD* is computed from the normalized depth values in **SDM**. A variation in the values of **SDM** results in a change in the final dictionary size (*GDD*). The size of the *GDD* changes the distribution of values in **DI**, which affects the compression rate. As *NI* increases, it causes a change in the frequency distribution of values in **PI** (Eq. 2). This increases the size of the entropy compressed **PI**. A decrease in the size of *GDD* affects the frequency distribution of values in **DI**, decreasing the size of the entropy compressed **DI**. The exact pattern of variation in the final compression ratio with the *NI* depends on the depth values in the depth map. The *NI* does not affect the compression quality as there is no loss introduced in the first stage of computation.

- *Dictionary error* (*DE*): The *DE* introduces the loss in our compression approach. The increase in the *DE* introduces more quantization errors in the depth pixels, and the overall compression quality decreases. The size of *GDD* decreases as the *DE* increases because more values will be quantized. A change in the size of the *GDD* affects the frequency of the distribution of values in **DI**. As the size of *GDD* is reduced the total number of different values in **DI** decreases (Eq. 7), which decreases the entropy in the blocks of **DI** and thereby increases the total compression ratio.

- *Precision* (*Pr*): The variation in the compression ratio and compression quality with *precision* is simple and direct. As the *precision* increases, the size of the final *GDD* for a fixed dictionary error increases by a large factor; the compression quality also improves accordingly.

The minimum *NI* allowed in our approach is two. When *NI* is set to one, there is only one interval and computation of **PI** is not required (all pixels are in the same interval $Interval_0$), and the interval size (*IS*) would be a substantially large value. In this case, the normalization step (Eq. 3) reduces to just subtracting the minimum depth value (Z_{min}). Since no normalization is performed, the *GDD* only consists of large depth values, and the resulting size of *GDD* would be very large. In this case the quantization is directly applied to the depth values instead of the normalized depth values increasing the final error by a significant factor. Also, as the size of *GDD* increases, the range of values stored in **DI** increases, and the size of the entropy compressed *DI* increases. Even though there is no requirement to store **PI** when the *NI* is set to one, the final compression ratio gains might not be high for a much larger final error (i.e., low compression quality).

3.9 Interactive Rendering

During interactive rendering, a small portion of RGB-D pixels are requested by the renderer [20] to generate a new view for a given camera viewpoint. Given the location of the required depth pixels, we can compute the corresponding block indices of the pixels based on the block size. Only the required and corresponding entropy compressed blocks stream are loaded into the memory using the block offset arrays, and entropy decoded. Our method inherently supports parallel decoding of all the requested blocks of pixels. Hardware support for decoding entropy compressed streams are widely available [23, 44] and may commodity GPUs also support hardware entropy decoding. Therefore, our decompression scheme is hardware friendly as our methods primarily consist of only entropy decoding and simple operations (Eq. 8) to re-compute the final depth value.

4 Results

We have analyzed and evaluated our approach on Middlebury datasets [18, 56] and TUM RGB-D [65] datasets. In addition to the datasets, there are large number other depth datasets available [6,45,60,71] Middlebury datasets [18,56] are high-resolution (1K, 2K) depth maps computed using stereo matching algorithms. TUM RGB-D datasets have depth maps captured using a Microsoft Kinect camera. We present some interesting results from the Middlebury datasets [56] in this section. The results on the other Middlebury datasets [18,57] and TUM datasets are presented in the suppl. material (Sec-1).

In standard image compression, the errors introduced in compression directly affect the rendered view quality. Hence, metrics like Peak Signal-to-Noise Ratio (PSNR) and Structural Similarity Index (SSIM) are suitable for measuring the compression quality. In the case of depth maps, the errors lead to distortions in the geometry that indirectly affect the final rendered view. Therefore, different global statistical metrics on the disparity values are used for estimating the error or similarity between two depth maps [32,56]. The statistical metrics are evaluated on the disparity values of the depth map and are measured in terms of pixels. We convert the depth maps into disparity maps to perform the analysis. Let $\mathbf{Z}$ be the actual depth map and $\overline{\mathbf{Z}}$ be the compressed depth map with N total pixels. The statistical metrics widely used are:

Dataset (resolution : bit-depth)	Compression ratio	RMS error
Adirnodack (2880 × 1988 : 32)	54	0.36
Jade Plant (2632 × 1988 : 32)	40	0.89
Cable (2796 × 1984 : 32)	55	0.64
Sword2 (2856 × 2000 : 32)	48	0.74
Piano (2820 × 1920 : 32)	37	0.28
Backpack (2940 × 2016 : 32)	40	0.45
Couch (2300 × 1992 : 32)	43	0.58
Playroom (2800 × 1908 : 32)	36	0.38
Sword1 (2912 × 2020 : 32)	22	0.69

Table 1: The compression ratio and RMS error in disparity are shown for different Middlebury datasets [18, 56]. All the depth maps are floating point with 32-bit dynamic range. Our RANDM algorithm works well on these challenging depth map datasets.

Dataset	Metric	Block Size: 4	Block Size: 6	Block Size: 8	Block Size: 10
Adirnodack	ratio	18	16	14	13.5
	RMS	0.57	0.57	0.57	0.57
Jade Plant	ratio	20	17	16	15.5
	RMS	1.47	1.47	1.47	1.47
Piano	ratio	16	14.5	14.5	13
	RMS	0.44	0.44	0.44	0.44
Couch	ratio	22	19	17	15.5
	RMS	1.36	1.36	1.36	1.36

Table 2: The effect of varying block size on the compression ratio and RMS error is highlighted. The block size has no effect on the RMS error as the blocks of *PI*, and *DI* are compressed in a lossless manner. The entropy of each block increases as the block size increases causing a decrease in the effective compression ratio.

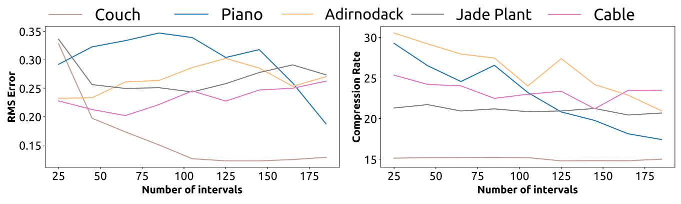


Fig. 4: The alterations in the RMS error and compression ratio with variation in the *number of intervals*. Increasing the *number of intervals* increases the entropy of the blocks in the *PI* and reduces the range of normalized depth values, reducing the size of *GDD* and decreasing entropy of the block in *DI*. The balance between the increase and decrease in the sizes of *PI* and *DI* is reflected in the plots for different datasets.

$$\textbf{Threshold}: \ Bad_{\sigma} = \text{percentage of } \ \bar{z}_{(x,y)} \text{ such that } |z_{(x,y)} - \bar{z}_{(x,y)}| > \sigma,$$

$$\textbf{Absolute Average Error}: AvgErr = \frac{1}{N}\Sigma_{x,y}|z_{(x,y)} - \bar{z}_{(x,y)}|,$$

$$\textbf{Root Means Squre}: RMS = \sqrt{\frac{1}{N}\Sigma_{x,y}|z_{(x,y)} - \bar{z}_{(x,y)}|^2},$$

$$\textbf{Maximum Error}: \text{MaxErr} = max(|z_{(x,y)} - \bar{z}_{(x,y)}|).$$

Table 1 shows the compression ratios, and RMS error for different depth maps in the Middlebury datasets. The compression ratio varies between $30-60\times$ and RMS error range is about $0.20-0.91$, based on the contents of the scene captured in the depth map. Additional metrics for the datasets in Table 1 are presented in the Supplementary material or the appendix (Sec-1).

We analyze the rate-distortion properties of our approach with respect to variation in different encoding parameters. Table 2 shows how the compression ratio and RMS error varies with changes in the block size. The blocks of *PI* and *DI* are encoded losslessly using arithmetic coding and varying *block size* has no effect on the RMS error. Increasing the block size affects the distribution of values in a given block of *PI* and *DI*, increasing the entropy and reducing the effective compression ratio.

Dataset	Metric	No. intervals: 5	No. intervals: 1
Adirnodack	ratio	11.8	16.84
	RMS	0.49	7.062
Jade Plant	ratio	8.94	19.2
	RMS	1.51	14.91
Cable	ratio	13	24.2
	RMS	0.62	9.45
Sword2	ratio	12.6	22.9
	RMS	0.4	6.95
Piano	ratio	6.64	12.9
	RMS	0.42	5.7
Backpack	ratio	8.5	16.08
	RMS	0.53	5.55
Couch	ratio	7.8	15.17
	RMS	1.4	28.81
Playroom	ratio	6.84	13.12
	RMS	0.68	11.025

Table 3: The compression ratio and compression quality when the *number of intervals* is set to one is highlighted. Without splitting, a large error is introduced in the *GDD*. The RMS error when the *number of intervals* is set to one is significantly higher than when the *number of intervals* is set to five.

We study the outcome of varying the *number of intervals* in Figure 4 with fixed *block size* and *dictionary error*. An increase in the *number of intervals* affects the distribution of values in the blocks of *PI*, increasing the entropy, which increases the size of the compressed *PI* stream. As the *number of intervals* increases, the value of the split size decreases, which effectively reduces the size of the *GDD*. A reduction in the size of *GDD* reduces the entropy in the blocks of *DI*, resulting in a decrease in the size of the *DI* stream. The trade-off between the increase in the size of the *PI* stream and the decrease in the size of the *DI* stream results in fluctuations (as shown in Figure 4) in the final compression ratio and RMS error.

Figure 5 and Figure 6 show the results of varying *dictionary error* for a fixed *number of intervals*. Introducing more error results in higher compression rate and increased errors (RMS), as expected. An increase in the *dictionary error* reduces the size of *GDD*, which increases the quantization errors and affects the distribution of values in *DI*. As the *dictionary error* increases, the size of the quantized *GDD* decreases, as do the resulting normalized depth values in the quantized dictionary. The entropy of the *DI* blocks might decrease as the values of *DI* are computed based on the best matching value (in the quantized dictionary), causing small local fluctuations in the compression ratio, and RMS error is noticed. The variation in the RMS error with a change in the compression ratio is plotted in Figure 7. The data in the plot (Figure 7) is collected by varying the *dictionary error* and keeping the *number of intervals* constant.

Table 3 highlights the case when the *number of intervals* is set to one. We notice a very significant increase in the RMS error as estimated in the compression analysis (Section - 3.7). Since there is no *PI* image present, the compression ratio improves, but the RMS error is too high to consider any gains.

Figure 9 shows a visual comparison between the ground truth depth map and a depth map compressed using our approach. We reconstruct the right-view of the stereo pair from the depth map and left-view. The reconstruction quality is measured between the right-view and the ground truth right-view in terms of PSNR and SSIM. We highlight

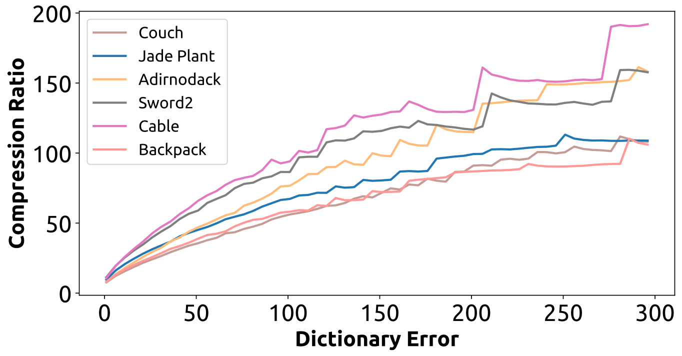


Fig. 5: The variation of compression ratio with increase in *dictionary error* is plotted. For a fixed *number of intervals*, an increase in the *dictionary error* reduces the final size of the quantized *GDD* decreasing the entropy of *DI* block and increasing the resulting compression ratio.

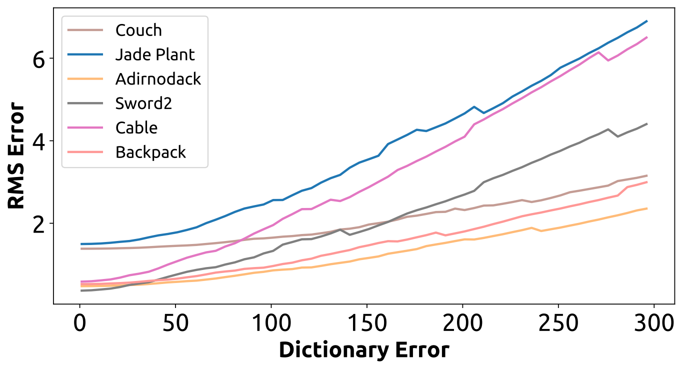


Fig. 6: The variation of RMS error in disparity with increase in *dictionary error* is highlighted. As the *dictionary error* increases, the quantization errors increase, increasing the resulting RMS error in the disparity.

the reconstruction error of the right-view computed from both uncompressed depth maps and compressed depth maps. This highlights the benefits of our approach for compressing and transmitting depth maps for telepresence applications, and reconstructing them at the other end. Our approach does not degrade the quality of reconstructions.

The existing schemes for compressing static depth maps achieve compression ratios from $10\times$ to $50\times$. Our approach provides compression ratios of $20-60\times$ for an RMS error of $0.5-2$ and provides good visual reconstruction in comparison with the ground truth (Fig. 9). Our method has an added benefit of random access decoding for interactive applications.

The most critical regions of a depth map are the sharp discontinuities in-depth (edges), and some of the static depth map compression schemes are designed to preserve the edges [28]. Our method also preserves the edges of the depth map. We provide a zoomed-in comparison for different depth maps on the edges in the Figure 8. Although other small quantization errors are visible in the zoomed-in compressed areas, we notice that our method preserves the depth discontinuities and edges.

In order to measure the decompression time, we have implemented a CPU decoder to decode blocks of depth maps from the compressed stream. The decompression time for decoding a block of size four is less than 1 microseconds, and a block size of eight is up to 1 microsecond using a single thread on an Intel Xeon CPU.

5 Conclusions, Limitations & Future Work

5.1 Conclusions

We present the first random access scheme (RANDM) that encodes depth maps using range-partition and computes a global dictionary. Our

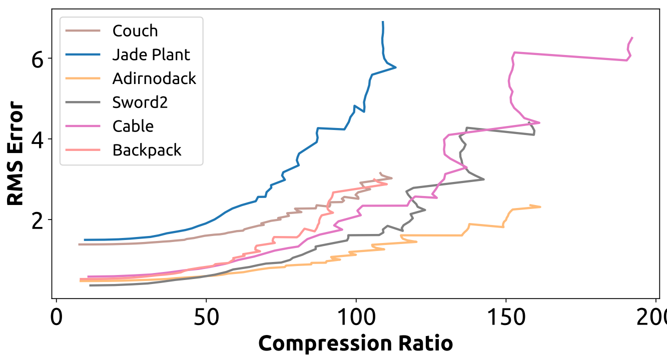
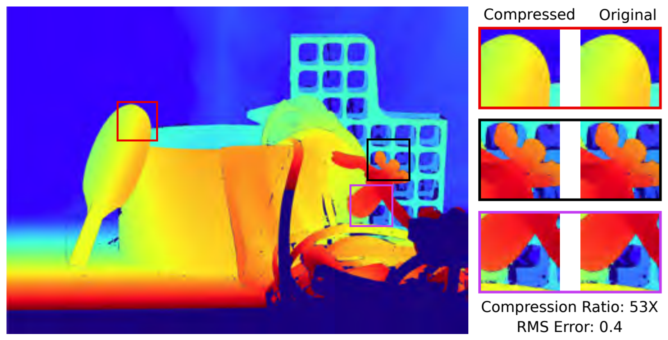
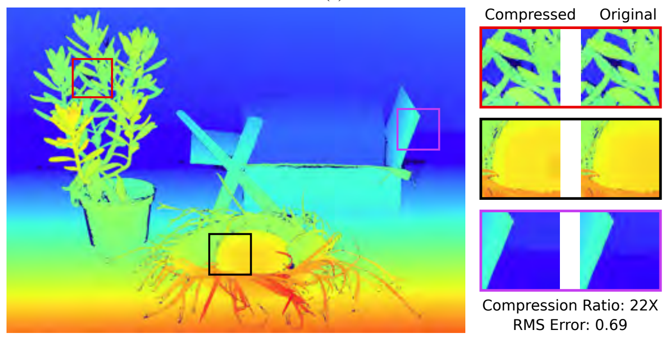


Fig. 7: The variation in RMS error with change in compression ratio is highlighted. As the compression ratio increases, the error in the disparities of the pixels increases, resulting in an overall increase in RMS error.



(a)



(b)

Fig. 8: Small regions of size 128×128 are selected and scaled to 2048×2048 around edges of a few interesting regions. We show a comparison of the scaled regions of the compressed depth map and uncompressed depth map. Edges are critical information in depth maps and we introduce no errors in the edges as highlighted.

method provides random access to blocks of depth pixels, supports fast parallel decoding, and is amenable for hardware decoding. We evaluated our approach on several depth maps of different dynamic ranges collected from several datasets. Our method achieves compression ratios similar to or better than the existing approaches. The average time to decode a block of pixels from the compressed stream is up to 1 microseconds and it can be used for interactive rendering.

5.2 Limitations

One of the primary limitations of our approach is that there is no clear or straightforward relationship between the encoding parameters and the compression quality and ratio. For a given set of encoding parameters, our method can result in low compression ratios and significant error at the same time. Other times, our method can result in a small RMS

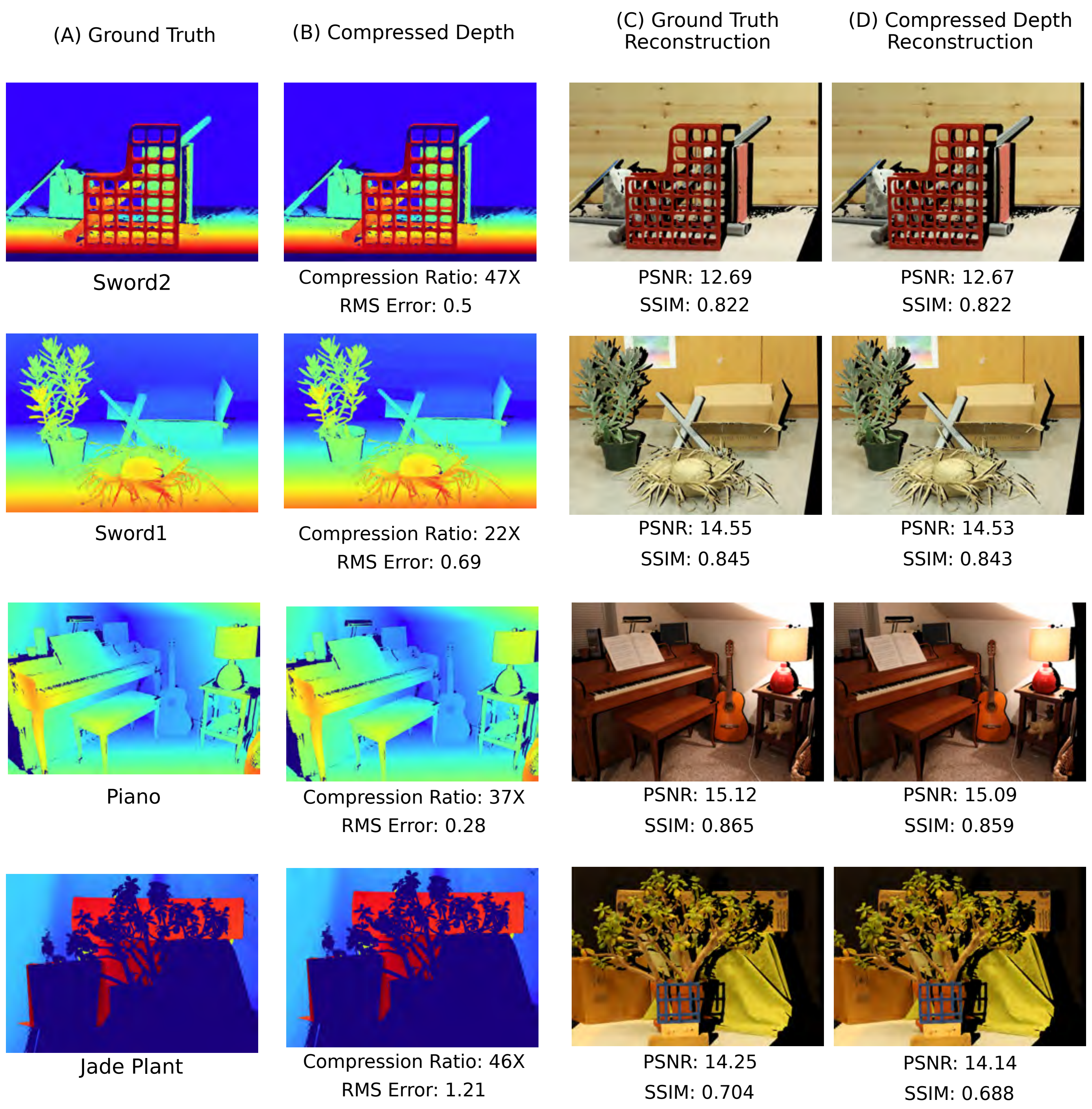


Fig. 9: We present a visual comparison between the ground truth depth map and a decompressed depth map compressed using our approach. We reconstruct the right-view from the depth map and the left-view for given stereo image pairs. A comparison between the right-view reconstructed from the uncompressed depth map and compressed depth map is highlighted. (A) Uncompressed ground truth depth maps. (B) The compressed depth maps using our approach. The corresponding *compression ratio* and the *RMS error* from our method are mentioned in the figure. (C) The right-view reconstructed from the uncompressed ground truth depth map. (D) The right-view reconstructed from the compressed depth map using our method. The PSNR and SSIM metrics between the reconstructed view and the ground truth right-view (stereo pairs) are mentioned in the figure.

error for a good compression ratio. However, the maximum error and percentage of bad pixels can be quite high, leading to noticeable errors in reconstruction or rendered images. Our method divides the depth range into intervals of equal size. If the input depth map has a big range but an uneven distribution of depth values in the depth map, the compression rate may not be high.

5.3 Future Work

Our method partitions the depth range into equal-sized intervals. Instead of equal sized intervals, we might consider intervals of uneven size depending on the exact distribution of depth values in the depth map. We speculate that a smaller number of uneven intervals covering all the depth pixels in the depth map might reduce the range of values in the three decomposed parts (**PI**, **DI**, *GDD*) leading to further compression. Currently, our approach is only for encoding static depth maps; in the future, we would like to extend the method for depth map videos. Finally, we would like to integrate our approach with a tele-presence system that performs realtime capture of depth maps, followed by compression, transmission and rendering.

References

[1] R. Anderson, D. Gallup, J. T. Barron, J. Kontkanen, N. Snavely, C. H. Esteban, S. Agarwal, and S. M. Seitz. Jump: Virtual reality video. *SIGGRAPH Asia*, 2016.

[2] A. C. Beers, M. Agrawala, and N. Chaddha. Rendering from compressed textures. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, SIGGRAPH '96, pp. 373–378. ACM, 1996. doi: 10.1145/237170.237276

[3] R. C. Bolles, H. H. Baker, and D. H. Marimont. Epipolar-plane image analysis: An approach to determining structure from motion. *International journal of computer vision*, 1(1):7–55, 1987.

[4] B.-B. Chai, S. Sethuraman, H. S. Sawhney, and P. Hatrack. Depth map compression for real-time view-based rendering. *Pattern Recognition Letters*, 25(7):755 – 766, 2004. Video Computing. doi: 10.1016/j.patrec.2004.01.002

[5] J.-X. Chai, X. Tong, S.-C. Chan, and H.-Y. Shum. Plenoptic sampling. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pp. 307–318. ACM Press/Addison-Wesley Publishing Co., 2000.

[6] A. Chang, A. Dai, T. Funkhouser, M. Halber, M. Niessner, M. Savva, S. Song, A. Zeng, and Y. Zhang. Matterport3d: Learning from rgb-d data in indoor environments. *International Conference on 3D Vision (3DV)*, 2017.

[7] J. Chen, J. Hou, and L.-P. Chau. Light field compression with disparity-guided sparse coding based on structural key views. *IEEE Transactions on Image Processing*, 27(1):314–324, 2018.

[8] E. Delp and O. Mitchell. Image compression using block truncation coding. *Communications, IEEE Transactions on*, 27(9):1335–1342, sep 1979. doi: 10.1109/TCOM.1979.1094560

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

[10] R. Geldreich. Advanced dxtc texture compression library. `https://github.com/richgel999/crunch`, 2012.

[11] B. Girod, C.-L. Chang, P. Ramanathan, and X. Zhu. Light field compression using disparity-compensated lifting. In *Acoustics, Speech, and Signal Processing, 2003. Proceedings.(ICASSP'03). 2003 IEEE International Conference on*, vol. 4, pp. IV–760. IEEE, 2003.

[12] S. B. Gokturk, H. Yalcin, and C. Bamji. A time-of-flight depth sensor - system description, issues and solutions. In *2004 Conference on Computer Vision and Pattern Recognition Workshop*, pp. 35–35, June 2004. doi: 10.1109/CVPR.2004.291

[13] M. M. Hannuksela, Ye-Kui Wang, and M. Gabbouj. Isolated regions in video coding. *IEEE Transactions on Multimedia*, 6(2):259–267, April 2004. doi: 10.1109/TMM.2003.822784

[14] J. Hasselgren and T. Akenine-Möller. Efficient depth buffer compression. In *Graphics Hardware*, pp. 103–110, 2006.

[15] P. Hedman, J. Philip, T. Price, J.-M. Frahm, G. Drettakis, and G. Brostow. Deep blending for free-viewpoint image-based rendering. *ACM Trans. Graph.*, 37(6):257:1–257:15, Dec. 2018. doi: 10.1145/3272127.3275084

[16] P. Hedman, T. Ritschel, G. Drettakis, and G. Brostow. Scalable inside-out image-based rendering. *ACM Trans. Graph.*, 35(6):231:1–231:11, Nov. 2016. doi: 10.1145/2980179.2982420

[17] Heung-Yeung Shum, Sing Bing Kang, and Shing-Chow Chan. Survey of image-based representations and compression techniques. *IEEE Transactions on Circuits and Systems for Video Technology*, 13(11):1020–1037, Nov 2003. doi: 10.1109/TCSVT.2003.817360

[18] H. Hirschmuller and D. Scharstein. Evaluation of cost functions for stereo matching. In *2007 IEEE Conference on Computer Vision and Pattern Recognition*, pp. 1–8, June 2007. doi: 10.1109/CVPR.2007.383248

[19] E. Horn and N. Kiryati. Toward optimal structured light patterns. *Image and Vision Computing*, 17(2):87–97, 1999.

[20] J. Huang, Z. Chen, D. Ceylan, and H. Jin. 6-dof vr videos with a single 360-camera. In *2017 IEEE Virtual Reality (VR)*, pp. 37–44, March 2017. doi: 10.1109/VR.2017.7892229

[21] K. I. Iourcha, K. S. Nayak, and Z. Hong. System and method for fixed-rate block-based image compression with inferred pixel values. U. S. Patent 5956431, 1999.

[22] A. Jagmohan, A. Sehgal, and N. Ahuja. Compression of lightfield rendered images using coset codes. In *Signals, Systems and Computers, 2004. Conference Record of the Thirty-Seventh Asilomar Conference on*, vol. 1, pp. 830–834. IEEE, 2003.

[23] Jian-Wen Chen, Cheng-Ru Chang, and Youn-Long Lin. A hardware accelerator for context-based adaptive binary arithmetic decoding in h.264/avc. In *2005 IEEE International Symposium on Circuits and Systems*, pp. 4525–4528 Vol. 5, May 2005. doi: 10.1109/ISCAS.2005.1465638

[24] L. Keselman, J. Iselin Woodfill, A. Grunnet-Jepsen, and A. Bhowmik. Intel realsense stereoscopic depth cameras. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*, July 2017.

[25] S. Kim and Y. Ho. Mesh-based depth coding for 3d video using hierarchical decomposition of depth maps. In *2007 IEEE International Conference on Image Processing*, vol. 5, pp. V – 117–V – 120, Sep. 2007. doi: 10.1109/ICIP.2007.4379779

[26] T. Koch, L. Liebel, F. Fraundorfer, and M. Korner. Evaluation of cnn-based single-image depth estimation methods. In *The European Conference on Computer Vision (ECCV) Workshops*, September 2018.

[27] P. Krajcevski, S. Pratapa, and D. Manocha. Gst: Gpu-based supercompressed textures. 2016.

[28] R. Krishnamurthy, Bing-Bing Chai, Hai Tao, and S. Sethuraman. Compression and transmission of depth maps for image-based rendering. In *Proceedings 2001 International Conference on Image Processing (Cat. No.01CH37205)*, vol. 3, pp. 828–831 vol.3, Oct 2001. doi: 10.1109/ICIP.2001.958248

[29] P. K. Lai, S. Xie, J. Lang, and R. Laqarure. Real-time panoramic depth maps from omni-directional stereo images for 6 dof videos in virtual reality. In *2019 IEEE Conference on Virtual Reality and 3D User Interfaces (VR)*, pp. 405–412, March 2019. doi: 10.1109/VR.2019.8798016

[30] M. Levoy and P. Hanrahan. Light field rendering. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '96, pp. 31–42. ACM, New York, NY, USA, 1996. doi: 10.1145/237170.237199

[31] M. Levoy, K. Pulli, B. Curless, S. Rusinkiewicz, D. Koller, L. Pereira, M. Ginzton, S. Anderson, J. Davis, J. Ginsberg, et al. The digital michelangelo project: 3d scanning of large statues. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pp. 131–144. ACM Press/Addison-Wesley Publishing Co., 2000.

[32] J. Li, R. Klein, and A. Yao. A two-streamed network for estimating fine-scaled depth maps from single rgb images. In *The IEEE International Conference on Computer Vision (ICCV)*, Oct 2017.

[33] Z. Lin and H.-Y. Shum. On the number of samples needed in light field rendering with constant-depth assumption. In *Computer Vision and Pattern Recognition, 2000. Proceedings. IEEE Conference on*, vol. 1, pp. 588–595. IEEE, 2000.

[34] D. Liu, L. Wang, L. Li, Z. Xiong, F. Wu, and W. Zeng. Pseudo-sequence-based light field image compression. In *Multimedia & Expo Workshops (ICMEW), 2016 IEEE International Conference on*, pp. 1–4. IEEE, 2016.

[35] F. Liu, C. Shen, and G. Lin. Deep convolutional neural fields for depth estimation from a single image. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2015.

[36] S. Liu, P. Lai, D. Tian, C. Gomila, and C. W. Chen. Joint trilateral filtering for depth map compression. In *Visual Communications and Image Processing 2010*, vol. 7744, p. 77440F. International Society for Optics and Photonics, 2010.

[37] W. Luo, A. G. Schwing, and R. Urtasun. Efficient deep learning for stereo matching. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016.

[38] M. Magnor and B. Girod. Data compression for light-field rendering. *IEEE Transactions on Circuits and Systems for Video Technology*, 10(3):338–343, 2000.

[39] A. Maimone and H. Fuchs. Encumbrance-free telepresence system with real-time 3d capture and display using commodity depth cameras. In *2011 10th IEEE International Symposium on Mixed and Augmented Reality*, pp. 137–146, Oct 2011. doi: 10.1109/ISMAR.2011.6092379

[40] H. S. Malvar. Adaptive run-length/golomb-rice encoding of quantized generalized gaussian sources with unknown statistics. In *Data Compression Conference (DCC'06)*, pp. 23–32, March 2006. doi: 10.1109/DCC.2006.5

[41] D. Marr. *Vision: A Computational Investigation into the Human Repre-*

sentation and Processing of Visual Information. Henry Holt and Co., Inc., New York, NY, USA, 1982.

[42] A. Mavlankar and B. Girod. Spatial-random-access-enabled video coding for interactive virtual pan/tilt/zoom functionality. *IEEE Transactions on Circuits and Systems for Video Technology*, 21(5):577–588, May 2011. doi: 10.1109/TCSVT.2011.2129170

[43] S. Mehrotra, Z. Zhang, Q. Cai, C. Zhang, and P. A. Chou. Low-complexity, near-lossless coding of depth maps from kinect-like depth cameras. In *2011 IEEE 13th International Workshop on Multimedia Signal Processing*, pp. 1–6, Oct 2011. doi: 10.1109/MMSP.2011.6093803

[44] J. L. Mitchell and W. B. Pennebaker. Optimal hardware and software arithmetic coding procedures for the q-coder. *IBM Journal of Research and Development*, 32(6):727–736, Nov 1988. doi: 10.1147/rd.326.0727

[45] P. K. Nathan Silberman, Derek Hoiem and R. Fergus. Indoor segmentation and support inference from rgbd images. In *ECCV*, 2012.

[46] J. Nystad, A. Lassen, A. Pomianowski, S. Ellis, and T. Olson. Adaptive scalable texture compression. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on High Performance Graphics*, HPG '12, pp. 105–114. Eurographics Association, 2012.

[47] J. O'Neal. Differential pulse-code modulation (pcm) with entropy coding. *IEEE Transactions on Information Theory*, 22(2):169–174, March 1976. doi: 10.1109/TIT.1976.1055534

[48] A. R. B. OpenGL. ARB_texture_compression_bptc. `http://www.opengl.org/registry/specs/ARB/texture_compression_bptc.txt`, 2010.

[49] R. S. Overbeck, D. Erickson, D. Evangelakos, and P. Debevec. Welcome to light fields. In *ACM SIGGRAPH 2018 Virtual, Augmented, and Mixed Reality*, SIGGRAPH '18, pp. 32:1–32:1. ACM, New York, NY, USA, 2018. doi: 10.1145/3226552.3226557

[50] C. Perra and P. Assuncao. High efficiency coding of light field images based on tiling and pseudo-temporal data arrangement. In *Multimedia & Expo Workshops (ICMEW), 2016 IEEE International Conference on*, pp. 1–4. IEEE, 2016.

[51] I. Peter and W. Straßer. The wavelet stream: Interactive multi resolution light field rendering. In *Rendering Techniques 2001*, pp. 127–138. Springer, 2001.

[52] S. Pratapa, P. Krajcevski, and D. Manocha. Mptc: Video rendering for virtual screens using compressed textures. In *Proceedings of the 21st ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, I3D '17, pp. 14:1–14:9. ACM, New York, NY, USA, 2017. doi: 10.1145/3023368.3023375

[53] S. Pratapa and D. Manocha. Rlfc: Random access light field compression using key views and bounded integer sequence encoding. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, I3D '19, pp. 9:1–9:10. ACM, New York, NY, USA, 2019. doi: 10.1145/3306131.3317018

[54] M. Sarkis and K. Diepold. Depth map compression via compressed sensing. In *2009 16th IEEE International Conference on Image Processing (ICIP)*, pp. 737–740, Nov 2009. doi: 10.1109/ICIP.2009.5414286

[55] E. Sayyad, P. Sen, and T. Höllerer. Panotrace: Interactive 3d modeling of surround-view panoramic images in virtual reality. In *Proceedings of the 23rd ACM Symposium on Virtual Reality Software and Technology*, VRST '17, pp. 32:1–32:10. ACM, New York, NY, USA, 2017. doi: 10.1145/3139131.3139158

[56] D. Scharstein, H. Hirschmüller, Y. Kitajima, G. Krathwohl, N. Nešić, X. Wang, and P. Westling. High-resolution stereo datasets with subpixel-accurate ground truth. In *German conference on pattern recognition*, pp. 31–42. Springer, 2014.

[57] D. Scharstein and C. Pal. Learning conditional random fields for stereo. In *2007 IEEE Conference on Computer Vision and Pattern Recognition*, pp. 1–8, June 2007. doi: 10.1109/CVPR.2007.383191

[58] N. D. Seif Allah Elmesloul Nasri, Khaled Khelil. Enhanced view random access ability for multiview video coding. *Journal of Electronic Imaging*, 25(2):1 – 13 – 13, 2016. doi: 10.1117/1.JEI.25.2.023027

[59] A. Serrano, I. Kim, Z. Chen, S. DiVerdi, D. Gutierrez, A. Hertzmann, and B. Masia. Motion parallax for 360 rgbd video. *IEEE Transactions on Visualization and Computer Graphics*, 25(5):1817–1827, May 2019. doi: 10.1109/TVCG.2019.2898757

[60] N. Silberman and R. Fergus. Indoor scene segmentation using a structured light sensor. In *Proceedings of the International Conference on Computer Vision - Workshop on 3D Representation and Recognition*, 2011.

[61] D. Sim and R. Park. Robust reweighted map motion estimation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 15(04):353–365, apr 1998. doi: 10.1109/34.677261

[62] J. Ström and T. Akenine-Möller. iPACKMAN: high-quality, low-complexity texture compression for mobile phones. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, HWWS '05, pp. 63–70. ACM, 2005. doi: 10.1145/1071866.1071877

[63] J. Ström and M. Pettersson. ETC2: texture compression using invalid combinations. In *Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, GH '07, pp. 49–54. Eurographics Association, 2007.

[64] J. Strom and P. Wennersten. Lossless compression of already compressed textures. In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*, pp. 177–182. ACM, 2011.

[65] J. Sturm, N. Engelhard, F. Endres, W. Burgard, and D. Cremers. A benchmark for the evaluation of rgb-d slam systems. In *Proc. of the International Conference on Intelligent Robot Systems (IROS)*, Oct. 2012.

[66] I. E. Sutherland, R. F. Sproull, and R. A. Schumacker. A characterization of ten hidden-surface algorithms. *ACM Computing Surveys (CSUR)*, 6(1):1–55, 1974.

[67] G. Tischler. Refinement of near random access video coding with weighted finite automata. In *International Conference on Implementation and Application of Automata*, pp. 46–57. Springer, 2006.

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

[69] A. D. Wilson. Fast lossless depth image compression. In *Proceedings of the 2017 ACM International Conference on Interactive Surfaces and Spaces*, ISS '17, pp. 100–105. ACM, New York, NY, USA, 2017. doi: 10.1145/3132272.3134144

[70] I. H. Witten, R. M. Neal, and J. G. Cleary. Arithmetic coding for data compression. *Communications of the ACM*, 30(6):520–540, 1987.

[71] J. Xiao, A. Owens, and A. Torralba. Sun3d: A database of big spaces reconstructed using sfm and object labels. In *The IEEE International Conference on Computer Vision (ICCV)*, December 2013.

[72] C. Zhang and J. Li. Compression of lumigraph with multiple reference frame (mrf) prediction and just-in-time rendering. In *Data Compression Conference, 2000. Proceedings. DCC 2000*, pp. 253–262. IEEE, 2000.