Fast Lossless Depth Image Compression

Andrew D. Wilson
Microsoft Research
Redmond, WA
awilson@microsoft.com

ABSTRACT
A lossless image compression technique for 16-bit single channel images typical of depth cameras such as Microsoft Kinect is presented. The proposed “RVL” algorithm achieves similar or better compression rates as existing lossless techniques, yet is much faster. Furthermore, the algorithm’s implementation can be very simple; a prototype implementation of less than one hundred lines of C is provided. The algorithm’s balance of speed and compression make it especially useful in interactive applications of multiple depth cameras on local area networks. RVL is compared to a variety of existing lossless techniques, and demonstrated in a network of eight Kinect v2 cameras.

Author Keywords
Depth image compression; interactive spaces

ACM Classification Keywords
H.5.m. Information interfaces and presentation (e.g., HCI): Miscellaneous.

INTRODUCTION
The Microsoft Kinect sensor is a popular device for building interactive spaces. Multiple sensors can be used to instrument larger interactive spaces, or to address sight line limitations of a single camera [6]. However, the use of multiple sensors is complicated by the need to calibrate the cameras to the same coordinate system. Another pragmatic concern is that a PC can host only one Kinect v2 device. This restriction is due to the fact that for every depth image, the camera sends multiple images to the PC which are combined on the GPU [13].

A workaround to the one camera per PC limit is to use multiple networked PCs, each hosting a camera and transmitting sensor data over the local area network. The RoomAlive Toolkit [12], for example, provides a server that sends depth images, color images, skeleton data, etc. Using multiple networked PCs has the added benefit that image processing tasks such as smoothing and background subtraction can be distributed to the hosting PCs.

Networking multiple cameras will be limited by network bandwidth. For example, transmitting the full HD color image from a single Kinect sensor at video rate will require more than 1.4Gbps network bandwidth. High quality JPEG compression (Q=50%) of each color image can reduce the bandwidth required to 30.7Mbps (30Hz) or 15.4Mbps (15Hz low light mode), allowing for multiple cameras on a typical 1Gbps local area network. More sophisticated video compression techniques such as H.264 or HEVC can reduce this bandwidth dramatically, and hardware encoders are commonly available on modern GPUs.

Kinect depth images are smaller and can be practically sent over a local area network at video rates (30Hz) without compression. Kinect depth images are 512×424, where each 16-bit pixel value directly encodes depth in the range of 80mm to 8000mm (13 bits). Each depth image is thus 424kB or 104Mbps. Transmitting both JPEG compressed color and uncompressed depth at 30Hz requires about 134Mbps. A 1Gbps network can support seven cameras in theory. Saturating a network in this way can add as much as 33ms latency to each image transmission.

Compressing the depth image may allow for the possibility of more than seven cameras, reduce latency, and leave network bandwidth available for other payloads (e.g., audio). Unfortunately, there appears to be no commonly available lossy image compression technique that does not adversely affect the geometric interpretation of depth images. For example, lossy compression can result in unacceptable artifacts at depth discontinuities, as near the edges of objects, and when valid depth pixels are adjacent to invalid depth values, which are typically zero.

Popular lossy compression techniques are optimized for color images and do not support 13bpp or 16bpp formats. Splitting 13-bit depth values across multiple color channels is a poor strategy, as errors due to lossy compression in the channel holding the most significant bits will cause large artifacts in the reconstructed depth image. Certain H.264 profiles support 14-bit color depth, and HEVC Version 2 supports 16-bit monochrome images, but neither handle depth discontinuities appropriately. Recent extensions to HEVC for 3D and multiview scenarios may be applicable.

Lossless techniques naturally avoid the problem of artifacts dramatically impacting depth images, yet there are few lossless implementations that are optimized for depth images. This paper presents a lossless compression technique that achieves similar compression rates as commonly available lossless techniques, yet runs significantly faster, making it suitable for real time, latency-sensitive interactive applications employing multiple Kinect cameras. It has the further benefit of being so simple that in the spirit of [16], its C implementation is included in the appendix.

RELATED WORK

Most previous work investigating Kinect depth image compression focuses on lossy techniques.

One strategy is to adapt existing video codecs. Pece et al. [11], propose a novel transform on depth image data that minimizes the impact of lossy compression when packing 16-bit depth image values into three channels for H.264 and VP8 codecs. Their technique suffers from noise at depth discontinuities. Liu et al. [7] evaluate using lossless compression on the most significant bits of the depth image, while using H.264 to encode the remaining bits. While existing lossy video compression techniques are not aware of contours, input depth image data may be pre-processed in certain ways to minimize artifacts around edges [3, 10].

Another approach is to address the geometrical interpretation of depth image data in the compression technique. Modi et al. approximate the scene as a series of planar surfaces [9], for example, while Jafarabad et al use "geometrical wavelets" to model surfaces in depth image [5]. Meanwhile, recent extensions to HEVC address 3D TV and multiview applications but seem to have not been applied to Kinect images. Such techniques typically are computationally expensive, particularly at the encoding stage.

A number of proposed techniques address the problem of edge artifacts by explicitly modeling contours in the depth image [17, 4, 15]. These techniques are also computationally expensive.

There are few published lossless or near-lossless depth image compression techniques. Perhaps the work that is most similar to the present work is that of Mehrotra et al., which similarly combines run length encoding and variable length encoding schemes to achieve lossless compression [8]. The approach proposed in this paper is simpler still, runs notably faster and achieve similar or better compression rates. Finally, there are informally reported results indicating that some degree of filtering (e.g., median filtering) and simple run length encoding or Lempel-Ziv encoding, in combination with frame-to-frame differencing, achieves good compression [2].

RVL COMPRESSION

The proposed "RVL" compression scheme is a mixture of <u>R</u>un length encoding and <u>V</u>ariable <u>L</u>ength encoding schemes.

Consider a depth image as alternating runs of zero and non-zero values in raster scan order. Without loss of generality, assume that the image begins with a run of zeros (possibly of zero length). Each successive pair of zero and non-zero runs is encoded in order as

- the number of zeros,
- the number of non-zeros,
- differences (deltas) of successive non-zero pixel values, where the difference for the first pixel in a run of non-zeros is based on the last pixel in the previous run of non-zeros

The number of zeros and non-zeros are positive integers written to the compressed bitstream with a base-8 variable length encoding scheme: starting with the least significant bits, the integer is broken into successive groups of three bits until all set bits are consumed. A fourth continuation bit is added to each group of three indicating whether it is the last group. For example, the 16 bit value 42 is encoded as 8 bits:

$$42 = 0000000000101010 \rightarrow \underline{0}010, \underline{1}101$$

where continuation bits are underlined.

Deltas are similarly variable-length encoded, but with the following elaboration to address both positive and negative values. Two's-complement representation of negative integers means that the highest bit of a negative delta value will be set, and the variable length encoding will always require the maximum number of groups of three bits, regardless of value (for example, the 32-bit representation of -1 will require 11 groups of three bits or 44 bits total). To avoid this, negative and positive deltas d are uniquely mapped to positive values u by "zigzag" encoding,

$$u = \begin{cases} 2d, & d \geq 0 \\ -2d - 1, & d < 0 \end{cases}$$

which maps small negative and small positive values to small positive values suitable for variable-length encoding. This encoding is efficiently implemented in C without branching as

```
int u = (d << 1) ^ (d >> 31)
```

with 32-bit signed integer `d` and the arithmetic right shift operator (note that right shift with negative numbers is not specified by the ISO C standard, and is compiler-specific).

The main feature of the variable length coding scheme is that it can encode any positive integer (and zero) with encoded length proportional to its value. Thus, in regions of the depth image with gradual changes, pixel delta values will be small and stored compactly. The smallest zigzag encoded delta will require four bits (4x compression), while the largest will require 20 bits. By considering runs of zeros independently from runs of non-zeros, large delta values near zeros in the depth image are avoided. In fact, the value chosen to indicate an invalid depth value is unimportant: it need only be unique.

Decompression is straightforward given the lengths of alternating runs of zeros and non-zeros. Zigzag encoding can be efficiently reversed in C with

```
int d = (u >> 1) ^ -(u & 1)
```

EVALUATION

To evaluate RVL, the following lossless compression techniques are compared, focusing on compression ratio and speed of encoding and decoding:

- RVL: our prototype implementation, compiled for x64 with speed optimizations enabled (/O2). See appendix for C code listing.
- JPEG-LS: CharLS implementation [1], compiled for x64, with speed optimizations enabled.
- JPEG-XR: Windows Imaging Component built-in codec, configured for lossless compression, 16bpp.
- RLGR: 64-bit library provided by authors of [8], lossless.
- PNG: Windows Imaging Component built-in codec, configured for 16bpp.

The Windows Imaging Component TIFF codec also supports 16bpp with lossless compression; however, its in-memory compression performance was not competitive, and was omitted as a comparison technique.

Additionally, two image filtering techniques were applied to the test images, resulting in three filtering conditions:

- Raw: no pre-processing of the image
- Filtered: single bilateral filtering [14] pass, with spatial weighting Gaussian with $\sigma = 3$ pixels, intensity weighting Gaussian with $\sigma = 40$mm, and a 7×7 window
- Quantized: every pixel value is divided by 4. Such quantization may be acceptable given the application and sensor noise, which is influenced by multiple factors including ambient light and distance. A simple experiment was conducted to estimate noise by setting the camera in front of a flat featureless wall about 1.5m away. Depth mean and standard deviation was computed at each pixel location over 1000 frames. The average standard deviation across all pixel locations was 2.5mm.

A series of 1000 consecutive test depth images was collected. The indoor scene includes clutter, areas of flat empty walls, and regions of invalid depth data (see Figure 1). To provide some variation in the test set, the camera was moved continuously through the capture session. Note that the presence of motion in the sequence has no bearing on compression, since all of the techniques tested work with single images only.

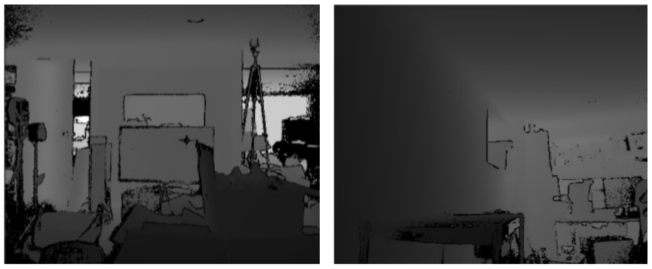

Figure 1. Two depth images from the test set.

Encoding and decoding times were measured on a Windows 10 PC with an Intel Core i7-3770K @ 3.50GHz. In all cases, compression was performed in memory. Timing did not include filter processing. Compression ratios were calculated against the 16-bit uncompressed version of the depth image. Mean and standard deviation of timing and compression ratio measures were logged to disk.

RESULTS

Compression ratio, compression time and decompression time means and standard deviations across compression technique and filtering condition are presented in Figure 2.

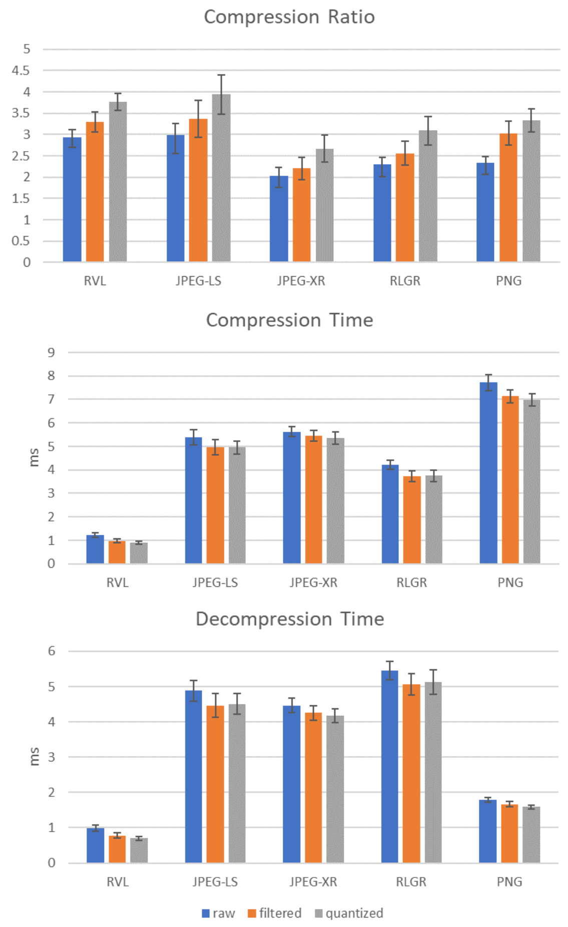

Figure 2. Compression ratio, compression time and decompression time across all comparison techniques and filter conditions. Error bars display ±1 standard deviation.

Our experiments show that the RVL algorithm achieves compression ratios comparable to existing commonly available lossless image compression techniques, as well as the experimental implementation of RLGR. It tends to perform better than JPEG-XR, RLGR and PNG, and sometimes JPEG-LS.

RVL has the lowest compression times (1.2ms without filtering), about four times faster than the next fastest technique (RLGR, 5.6ms), and the lowest decompression times (0.99ms), about twice as fast the next fastest technique

(PNG, 1.8ms). In our application setting every compressed image must be uncompressed, so it is meaningful to consider the sum of both. In this case RVL is about five times faster than the comparison techniques.

Filtering

Filtering improved compression ratios, with quantization showing the most improvement. The histogram over the number of 4-bit nibbles used in encoding non-zero pixel values for one sample image shows that many more were encoded with a single nibble when filtering is performed (see Figure 3). Filtering also had a slight impact on compression and decompression times.

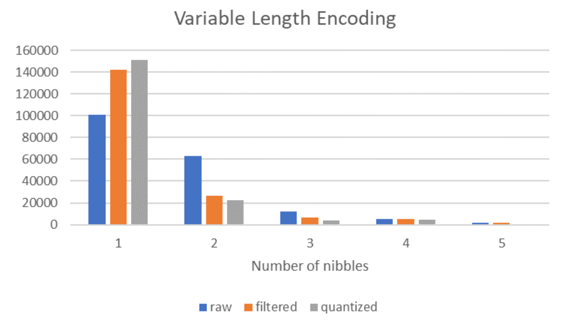

Figure 3. Histogram of number of 4-bit nibbles used to encode non-zero pixels, across filter conditions, for one sample image.

DISCUSSION

An important caveat in drawing conclusions from the evaluation is that developers of the comparison techniques may not have prioritized encoding and decoding speed, particularly when these codecs are commonly used with images on disk. Likewise, except for RLGR, the comparison techniques are not tuned for depth images in any way. Conversely, RVL has not been tested with a more general set of images, and may not perform well in comparison to other techniques designed for more general application.

Compression and Decompression Time

Without further detailed analysis, it is difficult to know why RVL runs much faster than the comparison techniques. However, in general it appears that the comparison techniques are more complex and therefore likely perform more instructions per pixel. JPEG-LS and RLGR use Golomb-Rice encoding, which involves bit-level manipulation, while RVL deals in four bit nibbles that easily pack into 32 bit words. The remaining techniques use entropy encoding methods such as Huffman or Lempel-Ziv encoding, which likely adds more instructions and memory bandwidth requirements.

Compression Ratio

Again, without further detailed analysis it is difficult to know why RVL achieves comparable compression ratios as the comparison techniques, despite the algorithm's simplicity.

RLGR, the algorithm most like RVL, features a more complex encoding scheme, yet appears to not handle transitions to and from zero values in any special way. Such transitions will be expensive to encode, as Golomb-Rice encoding uses unary encoding in part.

JPEG-LS uses more sophisticated context-sensitive prediction techniques. Preliminary experiments with a simple linear prediction technique in our prototype RVL implementation showed little difference in compression performance even in the filtered conditions.

Latency Complexity

Minimizing latency is critical in real time interactive applications. Compression time, network load and decompression time can increase depth image latency in different ways. For example, compression happens at each of the networked PCs and is thus fully parallelized. Network transport time will increase with more cameras or lower compression ratios. Decompression can be parallelized on multiprocessor architectures. More formally, the complexity of depth image latency is $O(C + \frac{n}{c}T + \frac{n}{m}D)$ where C denotes compression time, T network transmission time for an uncompressed image, D decompression time, n the number of cameras, c compression ratio and m the number of cores on the PC assembling the final scene. Further,

- If $n < m$, the image can be split into multiple subimages that can be parallelized to keep m cores busy.
- More complex compression techniques might achieve greater compression ratio c at the expense of compression and decompression time. This could be a good tradeoff if n or T is sufficiently large.
- RVL will give the lowest overall latency, but of the remaining techniques, PNG has rather high compression times, but second lowest decompression time and therefore is probably the best choice after RVL.

Temporal Coherence

In most interactive space applications the camera is statically mounted and the image does not change frame to frame aside from sensor noise. In these cases great improvements in compression might be gained by exploiting frame to frame coherence of the image. A preliminary RVL implementation encoding frame differences showed inferior compression to simply compressing each frame separately. This may be due to the fact that zeros in the depth image are typically in noisy regions of the image, and therefore will generate large frame to frame deltas; these might have appeared in a run of zeros if the image were coded singly.

Another strategy to exploit temporal coherence is to create two images; one modeling the static parts of the scene which is only transmitted occasionally, and another which has only moving objects. The latter image will likely be dominated by runs of zeros and will therefore highly compress. These two images could be composited on the client. This approach has the added benefit in that sensor noise on the static parts of the scene can be greatly reduced by simple frame to frame averaging. This is left for future work.

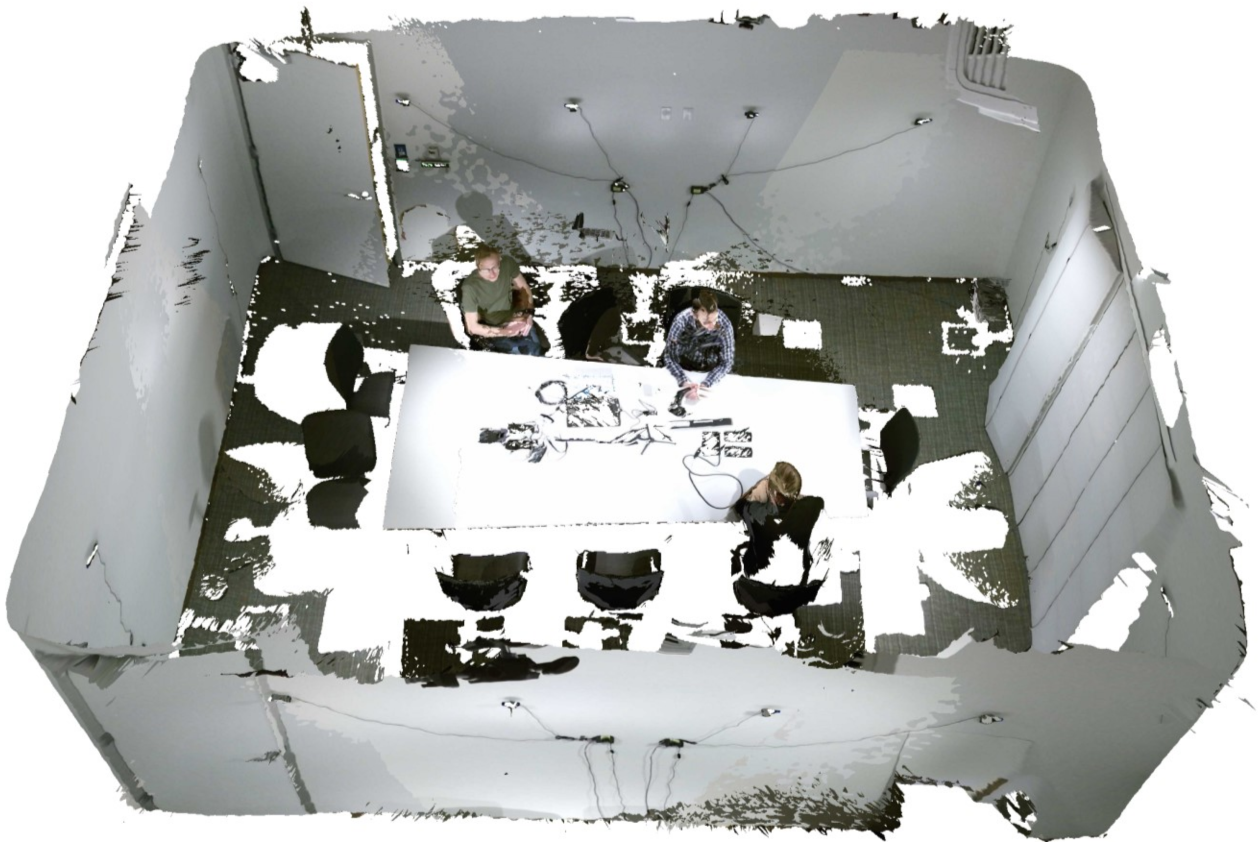
Figure 4. Rendering of live depth and color data from eight Kinect v2 cameras mounted in the ceiling to cover a conference room. Without depth image compression, the 1Gbps network is saturated, and many frames are dropped. With RVL compression, less than one half of the 1Gbps network bandwidth is used, and few frames are dropped (see accompanying video figure).

APPLICATION

The RVL codec was demonstrated in an interactive space equipped with eight Kinect v2 cameras. Each camera is hosted by an Intel NUC PC. The cameras were calibrated using the RoomAlive Toolkit [12]. Depth and JPEG compressed color images from each camera are transmitted over a 1Gbps network to a client PC which converts the depth images into textured meshes and renders them in real time (see Figure 4).

Without RVL compression, network bandwidth measured at the client PC exceeds 800Mbps, and many depth and color images are dropped. With RVL compression, few depth and color images are dropped, and network bandwidth is less than 500Mbps (see accompanying video figure).

CONCLUSION

RVL is a simple lossless image compression technique for 16-bit single channel images typical of depth cameras such as Microsoft Kinect. The technique's speed, competitive lossless compression rates, and simple implementation make it especially appropriate for practitioners building larger interactive spaces involving multiple networked depth cameras.

REFERENCES

1. CharLS, a C++ JPEG-LS library implementation. https://github.com/team-charls/charls.
2. Drossaers, M. Data Compression For the Kinect. https://thebytekitchen.com/2014/03/24/data-compression-for-the-kinect/.
3. Fu, J., Miao, D., Yu, W., Wang, S., Lu, Y., & Li, S. (2013). Kinect-like depth data compression. IEEE Transactions on Multimedia, 15(6), 1340-1352.
4. Gautier, J., Le Meur, O., & Guillemot, C. (2012, May). Efficient depth map compression based on lossless edge coding and diffusion. In Picture Coding Symposium (PCS), 2012 (pp. 81-84).
5. Jafarabad, M. Y., Kiani, V., Hamedani, T., & Harati, A. (2014, May). Depth image compression using geometrical wavelets. 6th Conference on Information and Knowledge Technology (IKT), 2014.
6. Jones, B., Rajinder Sodhi, R., Murdock, M., Mehra, R., Benko, H., Wilson, A., Ofek, E., MacIntyre, B., Raghuvanshi, N., and Shapira, L. 2014. RoomAlive: magical experiences enabled by scalable, adaptive projector-camera units. In Proc of the 27th annual ACM symposium on User interface software and technology (UIST '14). ACM, New York, NY, USA, 637-644.

7. Liu, Y., Beck, S., Wang, R., Li, J., Xu, H., Yao, S., ... & Froehlich, B. (2015, September). Hybrid lossless-lossy compression for real-time depth-sensor streams in 3D telepresence applications. In Pacific Rim Conference on Multimedia (pp. 442-452).

8. Mehrotra, S., Zhang, Z., Cai, Q., Zhang, C., & Chou, P. A. (2011, October). Low-complexity, near-lossless coding of depth maps from Kinect-like depth cameras. In Multimedia Signal Processing (MMSP), 2011 IEEE 13th International Workshop on (pp. 1-6).

9. Kanika Modi, Prem K. Kalra, and Subodh Kumar. 2014. Compression of Noisy Depth Image using Planes. In Proceedings of the 2014 Indian Conference on Computer Vision Graphics and Image Processing (ICVGIP '14). ACM, New York, NY, USA.

10. Lan, C., Xu, J. and Wu, F., 2012, July. Improving depth compression in HEVC by pre/post processing. In Multimedia and Expo Workshops (ICMEW), 2012 IEEE International Conference on (pp. 611-616).

11. Pece, F., Kautz, J., Weyrich, T.: Adapting standard video codecs for depth streaming. In: Proceedings of EGVE-JVRC 2011, pp. 59–66, Aire-la-Ville, Switzerland. Eurographics Association (2011)

12. RoomAlive Toolkit. https://github.com/Microsoft/RoomAliveToolkit.

13. Stuhmer, J., Nowozin, S., Fitzgibbon, A., Szeliski, R., Perry, T., Acharya, S., Cremers, D., and Shotton, J. 2015. Model-Based Tracking at 300Hz using Raw Time-of-Flight Observations. In ICCV, pp. 3577--3585.

14. Tomasi, C., and Manduchi, R. (1998). Bilateral filtering for gray and color images. In Sixth International Conference on Computer Vision (pp. 839-846).

15. Varadarajan, K. M., Zhou, K., & Vincze, M. (2012, November). RGB and depth intra-frame Cross-Compression for low bandwidth 3D video. In Pattern Recognition (ICPR), 2012 21st International Conference on (pp. 955-958).

16. Wobbrock, J. O., Wilson, A., and Li, Y. 2007. Gestures without libraries, toolkits or training: a $1 recognizer for user interface prototypes. In Proc. of the 20th annual ACM symposium on User interface software and technology (UIST '07). 159-168.

17. Yuan, Y., Cheung, G., Frossard, P., Le Callet, P., & Zhao, V. H. (2015, October). Contour approximation & depth image coding for virtual view synthesis. In Multimedia Signal Processing (MMSP), 2015 IEEE 17th International Workshop on (pp. 1-6).

APPENDIX

RVL source code is licensed under the MIT License:

```
int *buffer, *pBuffer, word, nibblesWritten;

void EncodeVLE(int value)
{
  do
  {
    int nibble = value & 0x7; // lower 3 bits
    if (value >>= 3) nibble |= 0x8; // more to come
    word <<= 4;
    word |= nibble;
    if (++nibblesWritten == 8) // output word
    {
      *pBuffer++ = word;
      nibblesWritten = 0;
      word = 0;
    }
  } while (value);
}

int DecodeVLE()
{
  unsigned int nibble;
  int value = 0, bits = 29;
  do
  {
    if (!nibblesWritten)
    {
      word = *pBuffer++; // load word
      nibblesWritten = 8;
    }
    nibble = word & 0xf0000000;
    value |= (nibble << 1) >> bits;
    word <<= 4;
    nibblesWritten--;
    bits -= 3;
  } while (nibble & 0x80000000);
  return value;
}

int CompressRVL(short* input, char* output, int numPixels)
{
  buffer = pBuffer = (int*)output;
  nibblesWritten = 0;
  short *end = input + numPixels;
  short previous = 0;
  while (input != end)
  {
    int zeros = 0, nonzeros = 0;
    for (; (input != end) && !*input; input++, zeros++);
    EncodeVLE(zeros); // number of zeros
    for (short* p = input; (p != end) && *p++; nonzeros++);
    EncodeVLE(nonzeros); // number of nonzeros
    for (int i = 0; i < nonzeros; i++)
    {
      short current = *input++;
      int delta = current - previous;
      int positive = (delta << 1) ^ (delta >> 31);
      EncodeVLE(positive); // nonzero value
      previous = current;
    }
  }
  if (nibblesWritten) // last few values
    *pBuffer++ = word << 4 * (8 - nibblesWritten);
  return int((char*)pBuffer - (char*)buffer); // num bytes
}

void DecompressRVL(char* input, short* output, int numPixels)
{
  buffer = pBuffer = (int*)input;
  nibblesWritten = 0;
  short current, previous = 0;
  int numPixelsToDecode = numPixels;
  while (numPixelsToDecode)
  {
    int zeros = DecodeVLE(); // number of zeros
    numPixelsToDecode -= zeros;
    for (; zeros; zeros--)
      *output++ = 0;
    int nonzeros = DecodeVLE(); // number of nonzeros
    numPixelsToDecode -= nonzeros;
    for (; nonzeros; nonzeros--)
    {
      int positive = DecodeVLE(); // nonzero value
      int delta = (positive >> 1) ^ -(positive & 1);
      current = previous + delta;
      *output++ = current;
      previous = current;
    }
  }
}
```