

RESEARCH

Open Access



FitDepth: fast and lite 16-bit depth image compression algorithm

Juan P. D'Amato^{1,2*}

*Correspondence:
juan.damato@gmail.com

¹ Pladema Institute, Campus
Universitario, UNICEN, Tandil,
Argentina

² National Scientific
and Technical Research
Council, CONICET, Buenos Aires,
Argentina

Abstract

This article presents a fast parallel lossless technique and a lossy image compression technique for 16-bit single-channel images. Nowadays, such techniques are “a must” in robotics and other areas where several depth cameras are used. Since many of these algorithms need to be run in low-profile hardware, as embedded systems, they should be very fast and customizable. The proposal is based on the consideration of depth images as surfaces, so the idea is to split the image into a set of polynomial functions that each describes a part of the surface. The developed algorithm herein proposed can achieve a similar—or better—compression rate and especially higher speed rates than the existing techniques. It also has the potential of being fully parallelizable and to run on several cores. This feature, compared to other approaches, makes it useful for handling and streaming multiple cameras simultaneously. The algorithm is assessed in different situations and hardware. Its implementation is rather simple and is carried out with LIDAR captured images. Therefore, this work is accompanied by an open implementation in C++.

Keywords: Depth image, Fast compression, Parallel implementation

1 Introduction

The massification of technologies—like 3D reconstruction, autonomous vehicles and robots—has increased the popularity of technologies like depth sensors. Many state-of-the-art smartphones already come with built-in depth sensors that make 3D data freely available. This depth data bring new ways of interacting as more accurate data are available in real-time. Also, it has an enormous impact on 3D design and industrial applications as it is mentioned in [3], as they provide high-quality data at a high speed and with a low cost, enabling real-time 3D reconstruction.

The Microsoft Kinect sensor is one of the first and most popular devices for obtaining 3D information. Multiple sensors can be used to instrument larger interactive spaces or to address sight line limitations of a single or multiple camera [23].

However, the newer Kinect v2 device is very CPU consuming, since, for every depth image, the camera sends multiple images to the PC, which are combined on the GPU. In order to deal with these limitations, many other devices have arisen in the market, such as IntelReal Sense [14] or Structure [15]. At the same time, a new generation of depth estimators, based on CNN inference models, like it is shown in [2] are arising, which

constitutes an interesting alternative to depth Cameras. Even that these methods have great quality, they are still not quite accurate and have high computer requirements for running.

All of these technologies provide their own SDK for development, but are focused on image capturing instead of storing. Therefore, efficient compression that exploits the characteristics of depth maps is still an open issue. As is well-known, there are two main types of compression algorithms: lossless and lossy. Lossless techniques naturally use original data without alteration. Although this avoids the problem of artifacts that dramatically impact on depth images, there are still a few lossless implementations that are optimized for depth images. On the other hand, popular lossy compression techniques, like “MP4”, are optimized for color images and do not naturally support 13bpp or 16bpp formats. Splitting 13-bit depth values across multiple color channels is a poor strategy. Errors due to lossy compression in the channel that holds the most significant bits will cause large artifacts to appear in the reconstructed depth image. Certain H.264 profiles support 14-bit color depth, and HEVC Version 2 supports 16-bit monochrome images. However, neither of them handles depth discontinuities appropriately, nor they require too much CPU effort.

This paper presents a fast technique that supports both lossless and lossy compression and which exploits the easiness of handling curved surfaces instead of pixels. The proposal is to split the image into parallel row scans and try to approximate it with splines or lines. The output of such method is a list of parametric functions that represent a part of an object. This method achieves similar compression rates as commonly available lossless techniques. Yet, it runs significantly faster, making it suitable for real-time or latency-sensitive interactive applications that employ multiple distributed depth cameras. It also supplements a residual encoding with a dictionary-based compression, such as the one in [4], from Facebook. Since it has the further benefit of being rather simple and open, its C++ implementation is included in a code repository. To validate the performance of this method, different depth-map compression scenarios are assessed and compared to standard 16-bit image formats. The comparison metrics used are the standard PSNR, compression rate and frames-per-second. This document is structured in four sections, as follows: Sect. 2 summarizes some of the existing works and proposals in the area; Sect. 3 gives a description of the proposed algorithm and its variants; Sect. 4 displays some proofs and examples, and Sect. 5 presents the final conclusion.

2 Related works

Our eyes estimate depth by comparing the images obtained by our left and right eye. The minor displacement between both viewpoints is enough to calculate an approximate depth map. The pair of images obtained by our eyes is referred to as a “stereo pair”. This, combined with our lens with variable focal length and our general experience of “seeing things”, allows us to have seamless 3D vision.

When engineers and researchers understood this concept, they tried to emulate the way our eyes work in order to extract depth information from the environment. There are numerous approaches that lead to the same outcome based on the following strategies:

- Dual camera technology: some devices have two cameras which are separated by a small distance, and compute depth using stereo-paired inference;
- Dual pixel technology: in this case, each pixel comprised two photodiodes, which are separated by a very small distance (less than a millimeter). Each photodiode receives the image signals separately, and then analyzes them as a stereo-image pair. Google Pixel 2 uses this technology;
- IR sensors: the first version of Kinect used an infra-red (IR) projector to compute depth. A pattern of IR dots is projected onto the environment, and a monochrome CMOS sensor (placed a few centimeters away) receives the reflected rays. To produce depth information, the difference between the expected and received IR dot position is calculated;
- Laser sensors: LIDAR systems fire laser pulses at the objects in the environment and measure either the flight time or the time the pulses take to get reflected back. These systems additionally measure the change in laser pulses' wavelength, providing accurate depth information.

IR captured images tend to have blurry edges, while Lidar images tend to retain the objects' contours. As many new devices are using laser-based sensors, our tests will focus on them. To sum up, nowadays several of these sensors are used synchronously. Some companies like [13] offer 3D reconstruction solutions based on several depth scanners connected to a local PC. When this process cannot be performed locally, there are various considerations to be taken. Streaming multiple cameras through the network will be limited by the 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. The high-quality JPEG compression ($Q = 50$) bandwidth required for 30.7Mbps (30 Hz) allows for multiple cameras on a typical 1 Gbps local area network.

More sophisticated video compression techniques—such as H.264 or HEVC—can reduce this bandwidth requirement dramatically, since hardware encoders are commonly available on modern GPUs. As could be inferred from Intel's Site [14], their proposal is for a small network of Raspberry PI to handle multiple cameras. Their idea is to use a Raspberry to control each camera and produce streaming, as can be seen in Fig. 1. In Intel's work, a stream of 1 frame-per-second is reached, which is relatively low. However, it is assured that leveraging better compression algorithms will allow for more USB3 camera modes to be reliably supported. Novel software and hardware 3D-compression schemes are being continuously published, what calls for closer evaluation. The above-mentioned work also highlights the importance of sharing 3D data with a wider audience for education and research purposes.

Kinect depth images are smaller and can be sent through a local area network at video rates (30 Hz) without compression. Having a 512×424 resolution and a 13-bit-pixel size would require 104Mbps. In theory, a 1Gbps network can support seven cameras, both for RGB and depth. In general, networks are WIFI-based, so they do not usually reach theoretical bandwidth. Saturating a network in this way could add latency to each image transmission.

Compressing the depth image may allow for the possibility of having more cameras, reducing latency, and leaving network bandwidth available for other payloads.

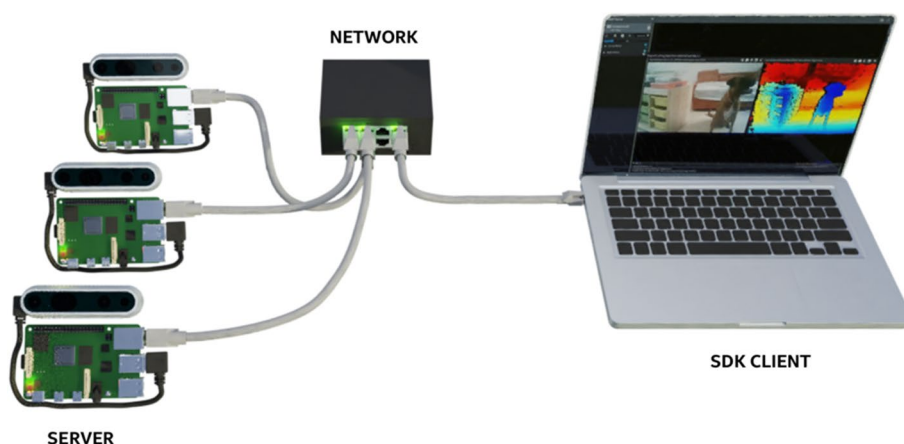


Fig. 1 Network configuration for handling several depth cameras

Unfortunately, all commonly available lossy image compression techniques seem to adversely affect the geometric interpretation of depth images. For example, a lossy compression can result in the appearance of unacceptable artifacts in depth discontinuities, such as near the edges of objects.

2.1 Existing methods

Most of the previous works that focused on Kinect depth-image compression use lossy techniques. One of the approaches is based on adapting existing video codecs. The authors of [23] propose a novel transformation of 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. This technique suffers from noise generation and depth discontinuities. In the work of [16], the use of lossless compression on the most significant bits of the depth image, and the use of H.264 to encode the remaining bits are evaluated.

While existing lossy video compression techniques are not aware of contours, input depth image data may be pre-processed in certain ways to minimize the appearance of artifacts around edges [21]. Another approach to depth-image compression is to address the geometrical interpretation of depth-image data in the compression technique. In [12], for example, its authors approximate the scene as a series of planar surfaces, whereas geometrical wavelets are used in [18] to model surfaces in depth images. Meanwhile, [6] proposes an extension to HEVC so it can support different formats. In general, such techniques are computationally expensive, specially in the encoding stage. Even though some of the proposed techniques address the problem of edge artifacts by explicitly modeling contours in the depth image, like in [9] these techniques also require too much computational effort. Besides, there are still several issues to overcome as depth compression remains an open challenge.

3 Methods

There are a few published lossless—or nearly lossless—depth-image compression techniques. The proposal of [12] is probably the one that mostly resembles this present work since its authors combine plane segmentation with run-length encoding schemes to achieve a lossy compression. Our proposal is focused on speed, but also

on keeping reasonable good-quality reconstructions after encoding. As images represent a scene, a row of such image is part of the object's surface, similar to an iso-surface. Our idea is to treat each depth image row separately and to describe it as a set of polynomial functions.

This schematic is presented in Fig. 2 where the RGB color picture is also presented. On the right, the depth is represented using pseudo-color (red is far, blue is close). Down, it is shown the real-time pipeline, with corresponding parameters. The dictionary for fast compression, as we will explain later, is generated off-line.

The proposed “lossy” algorithm evaluates each pixel in a sequential way, from left to right. If the gradient between consecutive pixels is low, it is enqueued into a vector of numbers called “spline”. In other cases, it is assumed that the pixel belongs to a different object, so a new vector is allocated. After visiting all pixels, each separated spline is evaluated in order to find the parameters of the function that best fits the elements.

The algorithm is supposed to be implemented using parallel technology. For this purpose, we have a memManager that handles the process memory, as it is explained in the next section. Following that same idea, the proposed “Polynomial lossy fit” algorithm is presented below.

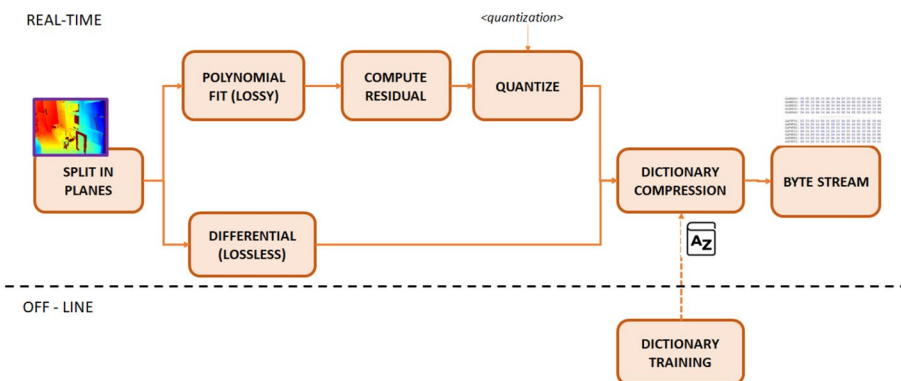
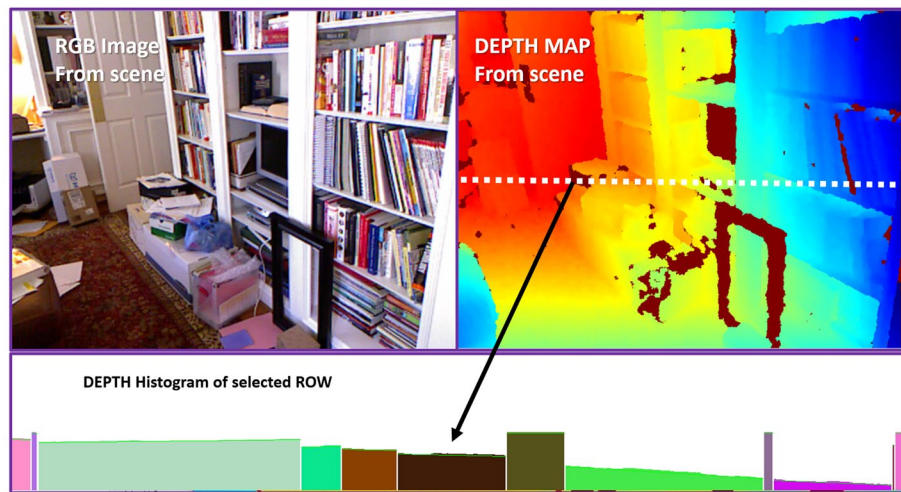


Fig. 2 Encoding pipeline (top). Schematic (bottom) step-by-step algorithm

Algorithm 1 Polynomial lossy fit

```

1: procedure LOSSYENCODING(image  $M$ )
2:   for  $row \in M$  do
3:      $spline \leftarrow memManager :: allocate()$ 
4:      $value \leftarrow M.at(row, x)$ 
5:     while  $(value - ant_{value}) \leq Threshold$  do
6:        $spline.push(value)$ 
7:        $ant_{value} \leftarrow value$ 
8:     end while
9:      $fit(spline)$ 
10:     $computeResidual(spline)$ 
11:  end for
12: end procedure

```

For each spline, a polynomial regression model in a general form is used as a fitting function. This could be expressed as $y_i = \beta_0 + \beta_1 x_i + \beta_2 x_i^2 + \dots + \beta_m x_i^m + \varepsilon_i$ ($i = 1, 2, \dots, k$). The vector of estimated polynomial regression coefficients is obtained using the “ordinary least squares estimation”. In our case, m is the amount of elements in each spline i , and n the exponential coefficient used. When n is small and fixed, this could be solved in almost constant time. Once all the necessary splines are evaluated, a final step—computing residual—is performed. In this step, the difference between estimation and real image is stored in a separated structure.

3.1 Compute residual and quantize

Unlike color information, depth is characterized by large smooth regions with very abrupt transitions. Preservation of these sharp edges during encoding process is crucial for applications that utilize depth information. In the literature, several approaches have been proposed for encoding depth images/videos. All methods aim at preserving edges while coding smooth regions with a minimum cost.

Our method tends to create more differences around the objects’ edges. There is a difference between the original sampled depth frame and the estimated one; this is called residual. In our case, residual is computed as $residual_{i,row} = frame_{i,row} - evaluate(spline, i, row)$, where i is the current pixel of the corresponding row in the image. The residual is stored in the same spline that describes the original pixel. This residual is finally stored using a linear fixed quantization (dividing each element by a constant). In future versions, an adaptative quantization could be used.

3.2 Differential encoding for lossless compression

In case a lossless compression is required, a similar procedure is carried out. Basically, instead of approximating the object’s surface by a curve, the idea is to store only the first element of a sequence, and then store the differential element. Its corresponding algorithm is shown below.

Algorithm 2 Differential lossless encoding

```

1: procedure LOSSLESSENCODING(image  $M$ )
2:   for row  $\in M$  do
3:     spline  $\leftarrow$  memManager :: allocate()
4:     value  $\leftarrow M.at(row, x)$ 
5:     while (value - antvalue)  $\leq$  Threshold do
6:       spline.push(value - antvalue)
7:       antvalue  $\leftarrow$  value
8:     end while
9:   end for
10: end procedure

```

A Threshold 128 is taken in order to encode the difference with an 8-bit value, what initially leads to lower memory requirements. As it was already mentioned about quantization, an adaptative solution could result in better encoding.

3.3 Parallel memory management

One of the major challenges of parallel implementation is the handling of simultaneous memory allocation that comes from multiple threads, as mentioned in [10] and in [7]. In general, every time a new variable needs to be allocated, the memory manager is locked until it organizes the heap.

In the proposed method, the amount of “splines” needed for encoding the image is unknown at first. Therefore, the splines need to be created dynamically, leading to lower CPU performance. In order to reduce the time required for structures to be created, we have proposed the use of a pre-allocated scheme, as shown in Fig. 3.

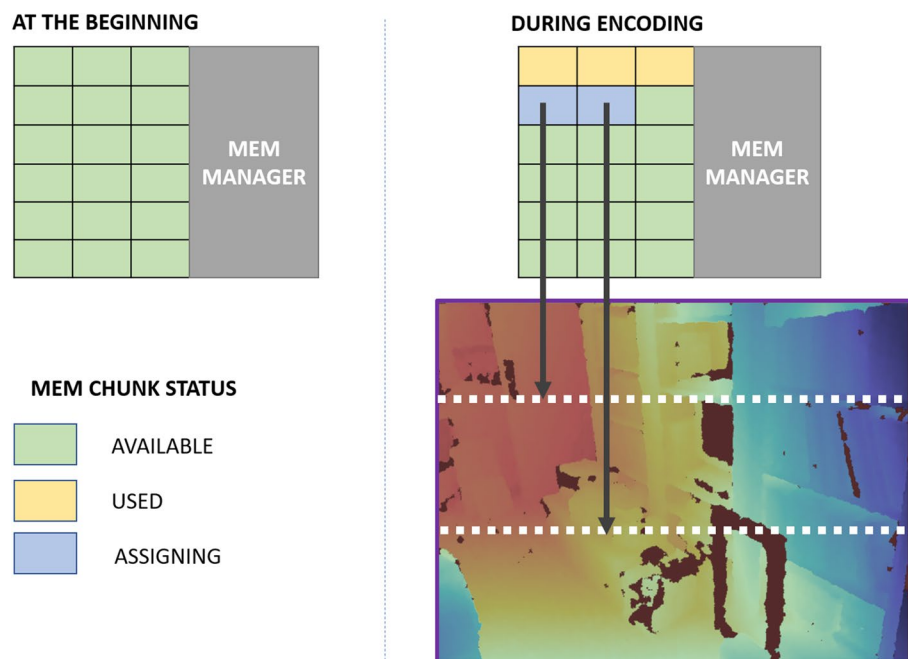


Fig. 3 Mem allocation scheme

At the beginning, the system estimates and allocates a fixed number of memory chunks that hold spline data. Such number is empirically computed from several previous algorithm runs with different depth images. In practice, this number is about 1/8 of the total amount of pixels.

Once the encoding begins, each new “spline” structure created is handled by the “mem-Manager” class. Within this class, an atomic semaphore is implemented, which controls the current index of available structures. The manager returns the pointer to the available memory chunk. Then, the structure is marked as used, as shown in Fig. 3. As the atomic operation entails the increasing an index—regardless of how many threads make the call at the same time—no bottleneck arises.

Once an image is completely encoded, the whole data vector is marked as “available”, and is prepared for another encoding.

3.4 Lossless compression configuration

In order to get the best compression rate, the final step is to apply a dictionary-based compression algorithm. (This step could also be applied in lossy compression.) In our implementation, the open source library of [4] is used. This library has an outstanding decompression rate, but it is not very fast for compression.

In order to reach a high framerate, our proposal is to train this algorithm with several compressed images using our algorithm to generate a default dictionary file with existing library tools. Later on, during encoding—in case this step is enabled—this computed dictionary is used. As can be seen in Fig. 4, some tests were run to validate this configuration. These figures show the time required for each frame and the resulting stream size in kbytes after compression (lower is better in both cases).

As it can be observed, using this pre-trained dictionary really improves the speed (it is faster) and the compression size (output is smaller) in both cases. For the upcoming sections, the best configuration described is used for any test that enables this encoding.

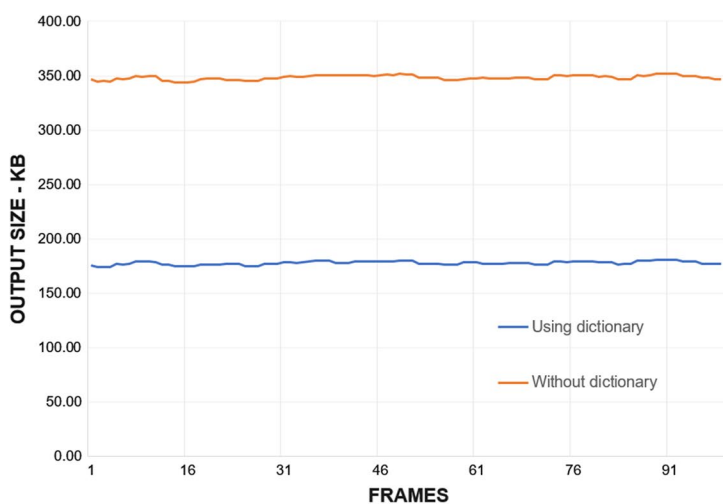
4 Results and discussion

In order to compare the proposed methods of compression, six different depth inner sequences, captured with a realSense L515 Lidar Camera, were generated. Each sequence was composed of 150 frames of 1024x768 resolution. At the same time, we used a dataset from [5] which contained a set of an office room captured with a RealSense D415 sensor (called scene 7) and a dataset from [11] (scene 8) that has 640x480 images captured with structured Kinect sensor V1.

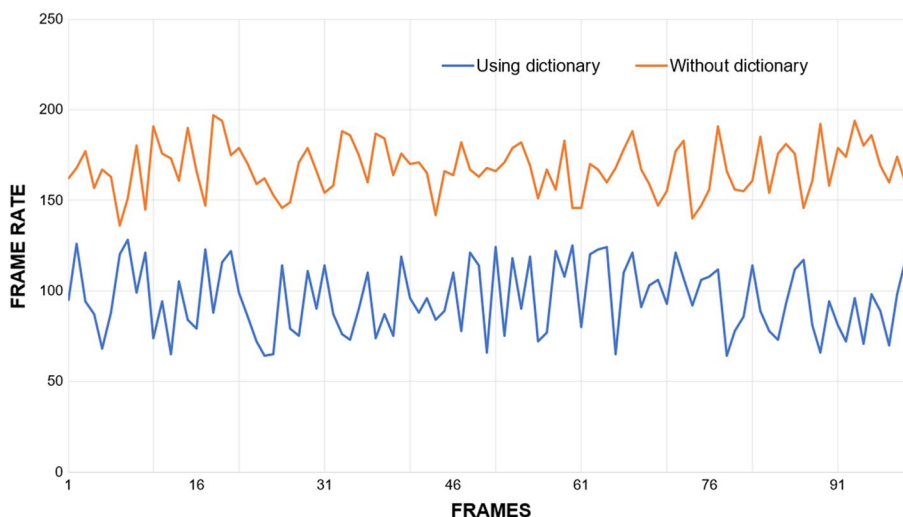
All of them were indoor scenes with different characteristics: from a person in front of the camera to part of a kitchen room. For measuring the quality of the encoding/decoding method, the peak signal-to-noise ratio (PSNR) was used. The PSNR, that has also been proposed in most of other related works, is defined as 1:

$$\text{PSNR} = 10 \log_{10} \frac{(2^d - 1)^2 WH}{\sum_{i=1}^W \sum_{j=1}^H (p[i, j] - p'[i, j])^2}, \quad (1)$$

where d is the bit depth of pixel, W is the image width, H is the image height, and $p[i, j]$, $p'[i, j]$ is the i th-row j th-column pixel in the original and compressed image, respectively.



(a) Obtained size of compressed image



(b) Estimated amount of frames processed per second

Fig. 4 Comparison with and without dictionary

Moreover, the compression rate—as the comparison between the original size and the one resulting after applying the method—was also used. The third metric used was the frames-per-second that can be reached while encoding.

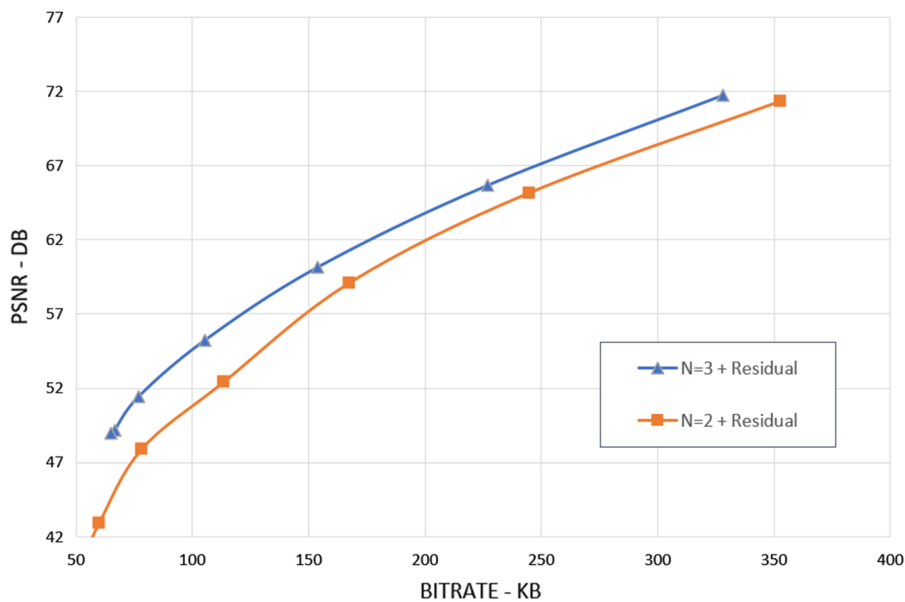
In addition to our implementation, the same sequences were compressed using an intra-coded mode of JPEG2000 coding standards, both in lossless and lossy modes. The PNG format, which is one of the fastest formats, was used. Finally, Jasper libraries were used for JPEG2000 [8]. All tests were run strictly on CPU.

4.1 Encoding residual

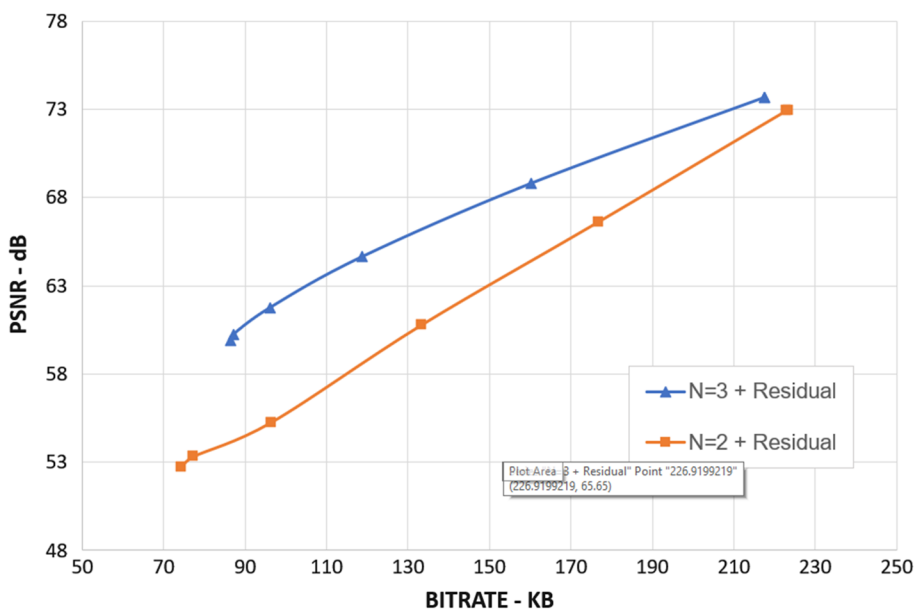
First of all, the resulting bitrate was evaluated in sample scenarios by adding residual encoding. In this case, the quantization parameter was varied and a lossless compression on the residual data was applied. In all cases, an evaluation was done on both,

the linear ($N=2$) and the cubic ($N=3$) fit version shown in Fig. 5. The idea was to try to choose the best encoding configuration.

As it was expected, the cubic fit ($N=3$) tended to work with higher quality and less residual size than the linear fit ($N=2$). In the following section, also a higher exponent ($N=5$) was also tested.



(a) Bitrate Scenario 0



(b) Bitrate Scenario 1

Fig. 5 Bitrate comparison with different fit parameters

4.2 Performance evaluation with different hardware

With regard to performance, the different algorithms were run in different hardware configurations. Two main configurations were used. The next section shows the impact of parallelism in each case:

- Intel i7 9300 with 16GB RAM laptop
- Raspberry PI 4 with 4GB RAM.

Both devices were tested with 100 frames, and the average times were obtained. Figure 6 shows the different cores each hardware has to exploit parallelism.

The Raspberry hardware has a lower clock rate of 1.1 GHZ, compared to the Intel 9300, which has a peak rate of 3.1 GHZ. As expected, this comparison allowed us to observe how the average time fell as the number of parallel threads increased. The Raspberry hardware has only 4 physical cores, which means it cannot accelerate more than it already does. On the CPU, an average of 60 FPS was obtained for encoding with tops of 100 FPS. In contrast, the Raspberry reached an average speed of 6 FPS, with a top of 7 fps.

4.3 Lossless compression comparison

The lossless version of the algorithm was applied to the proposed cases. In Table 1, the performance comparison of compression rates is presented with respect to the above-mentioned methods. Each metric used is the average of what was obtained from the entire sequence. A sample picture of each scene is also shown.

As it can be observed, the performance of the depth-coding algorithm showed better outcomes than the other methods. The key is to choose a good compression library. No deeper analysis was carried out.

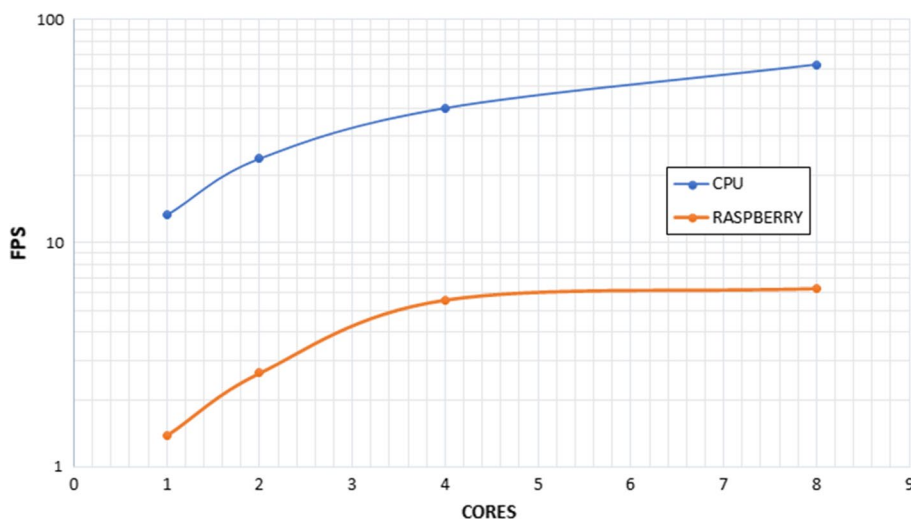


Fig. 6 Performance comparison with different cores

Table 1 Lossless compression rate



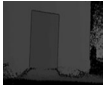





Name	Ours	PNG	J2K	Sample
Scene 0—kitchen	12.2%	29.8%	36.0%	
Scene 1—empty wall	21.4%	27.9%	23.6%	
Scene 2—box	22.1%	28.5%	25.0%	
Scene 3—object	23.2%	29.5%	26.0%	
Scene 4—animation	33.2%	44.5%	39.3%	
Scene 5—closer animation	24.4%	36.5%	27.8%	
Scene 6—market	6.6%	30.1%	31.1%	
Scene 7—office [5]	5.9%	23.5%	27.7%	

Table 1 (continued)


Name	Ours	PNG	J2K	Sample
Scene 8—assorted [11]	14.0%	44.0%	44.9%	

Table 2 Compression rate

		J2K		Basic			Residual		
		Q85	Q95	N=2	N=3	N=5	N=2	N=3	N=5
scene0	Rate	15.5%	20.4%	17.3%	18.1%	19.0%	13.6%	14.2%	15.1%
	PSNR	49.00	57.90	50.39	56.36	57.27	54.38	57.49	58.16
scene1	Rate	4.7%	6.5%	7.2%	7.5%	7.8%	5.6%	5.6%	6.1%
	PSNR	53.00	61.00	51.70	59.62	59.91	52.33	60.22	60.52
scene2	Rate	5.1%	7.1%	7.9%	8.5%	9.1%	6.4%	6.7%	7.4%
	PSNR	52.00	60.70	52.32	59.14	62.00	54.49	59.79	62.00
scene3	Rate	5.6%	7.8%	9.6%	10.3%	10.9%	8.1%	8.3%	8.9%
	PSNR	51.60	60.05	45.00	56.40	57.74	51.74	58.20	59.01
scene4	Rate	9.6%	12.8%	11.6%	13.2%	14.8%	11.5%	11.6%	13.0%
	PSNR	49.00	58.00	40.45	52.66	54.54	48.12	54.46	55.73
scene5	Rate	2.5%	3.9%	3.5%	4.5%	5.5%	5.5%	4.8%	5.3%
	PSNR	53.00	61.00	39.00	49.00	50.34	48.12	51.03	52.80
scene6	Rate	10.2%	13.7%	7.3%	8.4%	8.6%	8.2%	7.5%	8.3%
	PSNR	48.00	57.00	37.25	53.93	54.20	49.73	54.86	55.62
scene7	Rate	9.6%	13.0%	7.5%	8.4%	8.6%	7.68%	7.43%	8.03
	PSNR	51.00	56.00	40.09	52.68	53.20	50.78	54.50	56.13
scene8	Rate	9.2%	12.8%	20.0%	21.8%	23.6%	20.7%	19.3%	21.3%
	PSNR	49.00	58.00	39.99	42.29	42.84	47.01	48.23	51.84

4.4 Lossy compression comparison

At a later stage, the same scenes were processed using the lossy approach with several configurations. The algorithm was configured to fit different polynomial of degree N , $N=2$ (linear), $N=3$ (cubic) and $N=5$. The method variant was evaluated only by applying fitting (basic) and encoding residual with ZSTD compression. Residual was quantized using a 128 factor. All of these results are presented in Table 2.

For comparing the lossy version, the linear combination of SPEED, compression RATE and PSNR was used (best results are shown bolded), keeping the lower values at their best. It was observed that, in homogeneous scenes (like 1, 2, 3 and 8), the JPEG2000 format tends to work better than our algorithm. In all other cases (scenes 0, 4, 5, 6 and 7), our algorithm worked better in producing a better reconstruction quality (PSNR) and a minimal compression rate.

Table 3 Comparing encoding time measured in FPS

	OUR	PNG	J2K	Basic			Residual				
				LOSSLESS	LOSSY	N = 2	N = 3	N = 5	N = 2	N = 3	N = 5
scene0	E	7.33	26.40	4.40	5.94	62.74	56.48	48.00	7.19	7.41	7.44
	D	1211.38	56.44	4.56	6.16	1460.78	1318.58	961.29	1173.23	1128.79	925.47
scene1	E	12.35	24.83	6.41	8.66	106.50	79.94	64.31	13.67	13.52	12.68
	D	1342.34	59.48	6.49	8.75	1049.30	1006.76	973.86	1034.72	1020.55	943.04
scene2	E	11.37	24.71	6.23	8.41	95.27	68.66	54.94	12.05	11.96	11.39
	D	1505.05	58.04	6.28	8.48	1087.59	1020.55	925.47	1064.29	814.21	856.32
scene3	E	9.82	23.76	6.10	8.23	88.11	64.59	17.32	9.47	9.57	16.95
	D	1536.08	56.46	6.04	8.15	1111.94	1041.96	886.90	1079.71	973.86	16.95
scene4	E	8.40	20.43	4.83	6.52	76.69	57.64	45.71	7.78	8.25	7.91
	D	1027.59	44.25	4.88	6.58	986.75	955.13	772.02	949.04	973.86	818.68
scene5	E	3.67	19.22	5.27	7.12	56.59	37.22	29.08	9.90	12.18	11.40
	D	2525.42	43.45	5.28	7.13	1013.61	1040.50	997.32	931.25	955.13	772.02
scene6	E	9.80	21.74	4.83	6.81	90.91	47.62	64.47	8.33	9.90	5.91
	D	476.19	47.62	4.44	6.27	588.24	833.33	417.19	555.56	769.23	394.01
scene7	E	5.18	23.81	2.36	3.33	41.67	25.00	29.55	5.10	5.15	3.62
	D	862.07	25.64	2.44	3.44	497.51	52.67	352.85	322.58	304.88	228.78
scene8	E	11.40	35.47	7.31	10.31	107.12	83.43	28.01	8.70	9.52	8.58
	D	4450.00	102.83	7.80	11.00	4842.11	5545.45	4933.33	3597.12	3225.81	2551.15

4.5 Performance comparison with other methods

Table 3 shows the performance time obtained from the different algorithms, for both encoding (E) and decoding (D) tasks with the best hardware configuration.

In all cases, the linear fit reached top-frame rate. When residual was added, the performance fell dramatically. This happened because ZSTD has a one-thread implementation. When a higher polynomial function was evaluated, the gain in quality was not that notable with respect to the loss of speed. Our algorithm works in an asymmetrical way (since decoding is about 10 times faster than encoding). This is a good feature in algorithms of said kind, especially when decoding occurs in a low-profile hardware or when it is necessary to handle many cameras.

Finally, the results of scene 8 were compared to the work in [12]. Approximately, a 20% of compression and PSNR of 40 were reached. In the same scene, authors claimed to have about 10% of compression with a quality PSNR of 45. Our work could not be exactly compared with this one since there was no implementation of such technique available. On the other hand, a high speed of about 110 FPS was obtained with our algorithm while [12] does not mention the computational effort, a critical metric for validation that could be used for multi-camera streaming.

5 Conclusions

A novel method, based on multiple curve fittings for encoding noisy depth images, was presented in this paper. The proposed depth-image method was tested on several scenarios and hardware configurations. The method was then compared to JPEG2000 and PNG formats, where it achieved a better combination of compression and speed performance. Furthermore, the combination of the proposed depth segmentation and the residual coding scheme proved to outperform other similar depth segmentation algorithms when applied to depth compression.

Since depth cameras are very sensitive to light changes, they have several issues that cause a noisy image. One of the main problems is the extension of video encoding so that part of the information could be reused. Another one is the reduction of the number of curves, especially for planar objects. It is clear that there are still several issues to work on. The third problem is that the algorithm does not work well when the object contour is poorly limited. Particularly, this happens in depth images obtained by Structured Sensors, but not in Lidar ones. If different parameters are selected according to the source of image capture, the issue could be solved. Despite our best efforts—and contrary to earlier results—there is still more work to do in the evaluation of different cameras and in many different situations. Our next work will explore the implementation of such methods in GPU in order to get a better speed-up and defining a unique descriptor for automatically parametrizing the algorithms, balancing between quality and speed.

Abbreviations

CNN	Convolutional Neural Network
GPU	Graphics processing unit
PSNR	Peak signal-to-noise ratio

Acknowledgements

To the PLADEMA Institute's team for their infrastructure support for infrastructure supporting.

Author contributions

The idea, implementation and validation of this work was carried out by the main author. The author read and approved the final manuscript.

Authors' information

Mr. Juan Pablo D'Amato, PhD is a computer science researcher. He has been working for several years in topics related to Computer Vision and Computer Graphics and their applications. He is specially advocated to improving algorithms performance using new parallel strategies.

Funding

This work was partially funded by the National Agency of Scientific and Technological Promotion from Argentina (AGENCI—PICT Start Up 2020-0005 and the National Scientific and Technical Research Institute (CONICET).

Availability of data and materials

The source code in a GIT repository for free access is shared here <https://github.com/jpdamato/depthCompression>. Also, several open source databases available on the Internet were used for testing the algorithms.

Declarations

Competing interests

In accordance with Springer policy and my ethical obligation as a researcher, it is hereby informed that there are no conflict of interests with any company or organization.

Received: 31 March 2022 Accepted: 30 March 2023

Published online: 06 April 2023

References

1. C. Yan, B. Gong, Y. Wei, Y. Gao, Deep multi-view enhancement hashing for image retrieval. *IEEE Trans. Pattern Anal. Mach. Intell.* (2020)
2. A. Bozic, P. Palafox, J. Thies, A. Dai, M. Nießner, TransformerFusion: monocular RGB scene reconstruction using transformers. *NeurIPS* (2021)
3. J. Choe, S. Im, F. Rameau, M. Kang, I.S. Kweon, VolumeFusion: deep depth fusion for 3D scene reconstruction. In: *ICCV* (2021)
4. M. Kucherawy (Facebook), Zstandard Compression and the 'application/zstd' Media Type. <https://datatracker.ietf.org/doc/html/rfc8878> (accessed: 01.09.2021)
5. A. Dai, A. X. Chang, M. Savva, M. Halber, T. Funkhouser, M. Nießner, Scannet: richly-annotated 3d reconstructions of indoor scenes. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp 58285839 (2017)
6. C. Yan, Y. Zhang, Z. Xu, F. Dai, L. Li, Q. Dai, F. Wu, A highly parallel framework for HEVC Coding unit partitioning tree decision on many-core processors. *IEEE Signal Process. Lett.* **21**–5, 573–576 (2014). <https://doi.org/10.1109/LSP.2014.2310494>
7. B. Wang, D. Souza, M. Mesa, C.C. Chi, B. Juurlink, A. Ilic, N. Roma, L. Sousa, Highly parallel HEVC decoding for heterogeneous systems with CPU and GPU. *Signal Process. Image Commun.* (2017). <https://doi.org/10.1016/j.image.2017.12.009>
8. A. Skodras, C. Christopoulos, T. Ebrahimi, The JPEG 2000 still image compression standard. *IEEE Signal Process. Mag.* **18**(5), 36–58 (2000). [https://doi.org/10.1109/79.952804\(2001\)](https://doi.org/10.1109/79.952804(2001))
9. F. Jäer, Contour-based segmentation and coding for depth map compression. *Visual Communications and Image Processing (VCIP)*. <https://doi.org/10.1109/VCIP.2011.6115989> (2011)
10. Juan D'Amato, M. Vénere, A CPU-GPU framework for optimizing the quality of large meshes. *J. Parallel Distributed Computing*. **73**, 1127–1134 (2013). <https://doi.org/10.1016/j.jpdc.2013.03.007>
11. N. Silberman, D. Hoiem, P. Kohli, R. Fergus, Indoor segmentation and support inference from RGBD Images. *European Conference on Computer Vision* (2012)
12. S.H. Kumar, K.R. Ramakrishnan, Depth compression via planar segmentation. *Multimed. Tools Appl.* (2019). <https://doi.org/10.1007/s11042-018-6327-4>
13. T. Carter, 3D body scanner platform, Company site. <https://fit3d.com/> (accessed: 01.09.2021)
14. S. Dorodnicov, A. Grunnet-Jepsen, A. Puzhevich, D. Piro, Open-Source Ethernet Networking for Intel®RealSense™ Depth Cameras, Company site, <https://dev.intelrealsense.com/docs/open-source-ethernet-networking-for-intel-realsense-depth-cameras> (accessed: 01.09.2021)
15. Structure SDK (Cross-Platform), <https://structure.io/developers> (accessed: 01.09.2021)
16. W. Kim, A. Ortega, P. Lai, D. Tian, Depth map coding optimization using rendered view distortion for 3D video coding. *IEEE Trans. Image Process.* **24**(11), 3534–3545 (2015). <https://doi.org/10.1109/TIP.2015.2447737>
17. S. Shahriyar, M. Murshed, M. Ali, M. Paul, Efficient coding of depth map by exploiting temporal correlation, 2014 International Conference on Digital Image Computing: Techniques and Applications (DICTA), 1–8, (2014) <https://doi.org/10.1109/DICTA.2014.7008105>
18. M. Yaghouti Jafarabad, V. Kiani, T. Hamedani, A. Harati, Depth image compression using geometrical wavelets, 2014 6th Conference on Information and Knowledge Technology, 117–122, (2014), <https://doi.org/10.1109/IKT.2014.7030344>

19. J. Lei, S. Li, C. Zhu, M. Sun, C. Hou, Depth coding based on depth-texture motion and structure similarities. *IEEE Trans. Circ. Syst. Video Technol.* **25**(2), 275–286 (2015). <https://doi.org/10.1109/TCSVT.2014.2335471>
20. S. Schwarz, M. Preda, V. Baroncini, M. Budagavi, P. Cesar, P. Chou, R. Cohen, M. Krivokuća, S. Lasserre, L. Zhu, J. Llach, K. Mammou, R. Mekuria, O. Nakagami, E. Siahaan, A. Tabatabai, A. Tourapis, V. Zakharchenko, Emerging MPEG standards for point cloud compression, *IEEE J. Emerg. Sel. Top. Circ. Syst.* <https://doi.org/10.1109/JETCAS.2018.2885981> (2018)
21. M.M. Duch, J.R. Morros, J. Ruiz-Hidalgo, Depth map compression via 3D region-based representation. *IMultimed. Tools Appl.* **76**, 13761–13784 (2017). <https://doi.org/10.1007/s11042-016-3727-1>
22. S. Mehrotra, Z. Zhang, Q. Cai, C. Zhang, P. Chou, Low-complexity, near-lossless coding of depth maps from kinect-like depth cameras, *IEEE 13th International Workshop on Multimedia Signal Processing*, 1–6, <https://doi.org/10.1109/MMSP.2011.6093803> (2011)
23. Y. Morvan, Acquisition, compression and rendering of depth and texture for multi-view video. *Nano Letters - NANO LETT* (2009). <https://doi.org/10.6100/IR641964>
24. V. Delaitre, J. Sivic, I. Laptev, Learning person-object interactions for action recognition in still images, *Adv. Neural Inf. Process. Syst.* (2011)

Publisher's Note

Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Submit your manuscript to a SpringerOpen[®] journal and benefit from:

- ▶ Convenient online submission
- ▶ Rigorous peer review
- ▶ Open access: articles freely available online
- ▶ High visibility within the field
- ▶ Retaining the copyright to your article

Submit your next manuscript at ▶ [springeropen.com](https://www.springeropen.com)
