FitDepth: Fast and lite 16bits depth image compression algorithm

Juan Pablo DAmato (juan.damato@gmail.com)
Campus Universitario, UNICEN

Research

Keywords:

Posted Date: April 18th, 2022

DOI: https://doi.org/10.21203/rs.3.rs-1506832/v1

FitDepth : Fast and lite 16bits depth image compression algorithm

Juan P. D'Amato[1,2*]

[1]Pladema Institute, Campus Universitario, UNICEN, Tandil (Argentina) .
[2] National Scientific and Technical Research Council, CONICET, Buenos Aires (Argentina).

Corresponding author(s). E-mail(s): juan.damato@gmail.com;

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 integrated depth sensors that make 3D data freely available. This depth data also brings new ways of interacting, as more accurate data is available in real time. 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 [20].

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 [11] or Structure [12] .

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 [1], 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 16bits-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: section 2 summarizes some of the existing works and proposals in the area; section 3 gives a description of the proposed algorithm and its variants; section 4 displays some proofs and examples, and section 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.

Since 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 separated by a small distance, and compute depth using stereo-paired inference;
- Dual pixel technology: In this case, each pixel is comprised of 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;

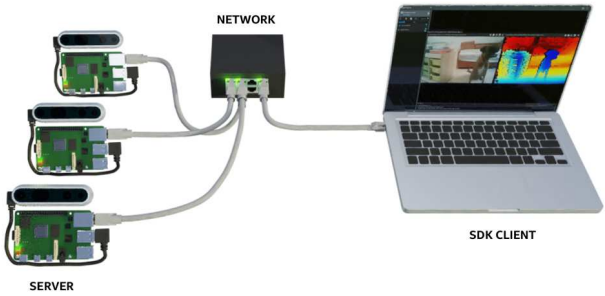


Fig. 1: Network configuration for handling several depth cameras

- 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 of the 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 [10] offer 3D reconstruction solutions based on several depth scanners connected to a local PC. When it is not possible to perform such process 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 (30Hz) allows 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, hardware encoders are commonly available on modern GPUs. As could be inferred from Intel's Site [11], 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 Figure 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 (30Hz) without compression. Having a 512×424 resolution and a 13-bits-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 theorical 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. 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 [20] 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 [13], 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 [18]. Another approach to depth-image compression is to address the geometrical interpretation of depth-image data in the compression technique. In [9], its authors approximate the scene as a series of planar surfaces, for example; whereas [15] uses "geometrical wavelets" to model surfaces in depth images. Meanwhile, [4] 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 [6]; however, 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 [9] 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. Our idea is to treat each image row separately and to describe it as a set of polynomial functions. Each function represents a part of the object´s surface, similar to an iso-surface. This schematic is presented in Figure 2 .

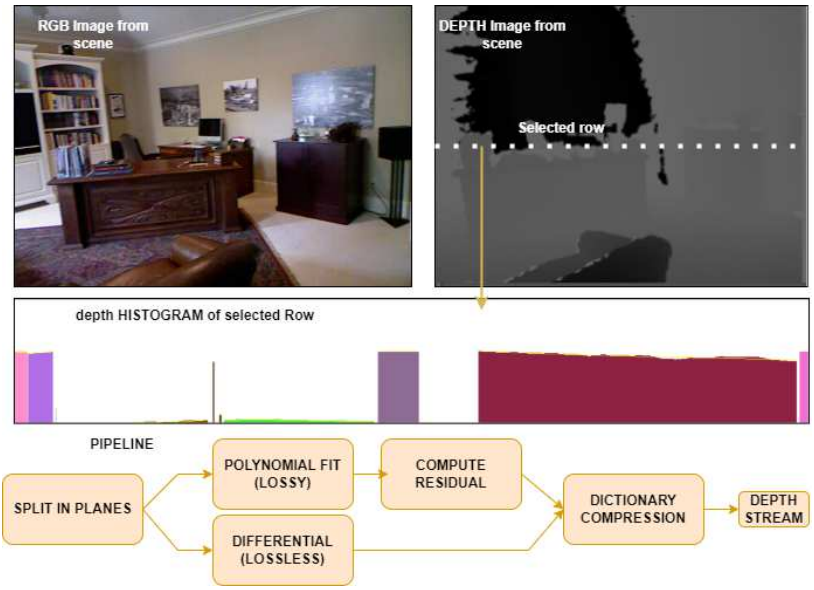


Fig. 2: Encoding pipeline

The proposed 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 algorithm "Polynomial lossy fit" is presented below:

For each spline, a polynomial regression model in a general form is used as a fitting function. This could be expressed as

$y_i = \beta_0 + \beta_1 x_i + \beta_2 x_i^2 + \cdots + \beta_m x_i^m + \varepsilon_i \ (i = 1, 2, \ldots, n)$

Using matrix notation, this could be expressed as $y = \mathbf{X}\vec{\beta} + \vec{\varepsilon}$. The vector of estimated polynomial regression coefficients (using "ordinary least squares estimation") is $\widehat{\vec{\beta}} = (\mathbf{X}^\mathsf{T}\mathbf{X})^{-1}\ \mathbf{X}^\mathsf{T}\vec{y}$, assuming $m < n$, which is required for the matrix to be invertible; then, since X is a "Vandermonde matrix", the invertibility condition is guaranteed to hold if all the x_i values are distinct.

Algorithm 1 Polynomial lossy fit

```
1: procedure LOSSYENCODING(image M)
2:     for row ∈ M do
3:         spline ← memManager :: allocate()
4:         value ← M.at(row, x)
5:         while (value − ant_value) ≤ Threshold do
6:             spline.push(value)
7:             ant_value ← value
8:         end while
9:         fit(spline)
10:        computeResidual(spline)
11:    end for
12: end procedure
```

This is the unique least squares solution. When n is small and fixed, this could be solved in 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 Residual quantization

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 stored using a linear fixed quantization. In future versions, an adaptative quantization could be used.

3.2 Differential encoding

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.

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

Algorithm 2 Differential lossless encoding

```
1: procedure LOSSLESSENCODING(image M)
2:     for row ∈ M do
3:         spline ← memManager :: allocate()
4:         value ← M.at(row, x)
5:         while (value − ant_value) ≤ Threshold do
6:             spline.push(value − ant_value)
7:             ant_value ← value
8:         end while
9:     end for
10: end procedure
```

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 [7] [3]. 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 a pre-allocated scheme, as shown in Figure 3.

At the beginning, the system estimates and allocates a fixed number of memory chunks that holds 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 that has to be created is handled by the class "memManager". Within this class, an atomic semaphore is implemented which controls the current index of available structures. The manager returns the pointer to this available memory chunk. Then, the structure is marked as used, as shown in Figure 3. As the atomic operation consists of 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 it 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 library of [1] 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

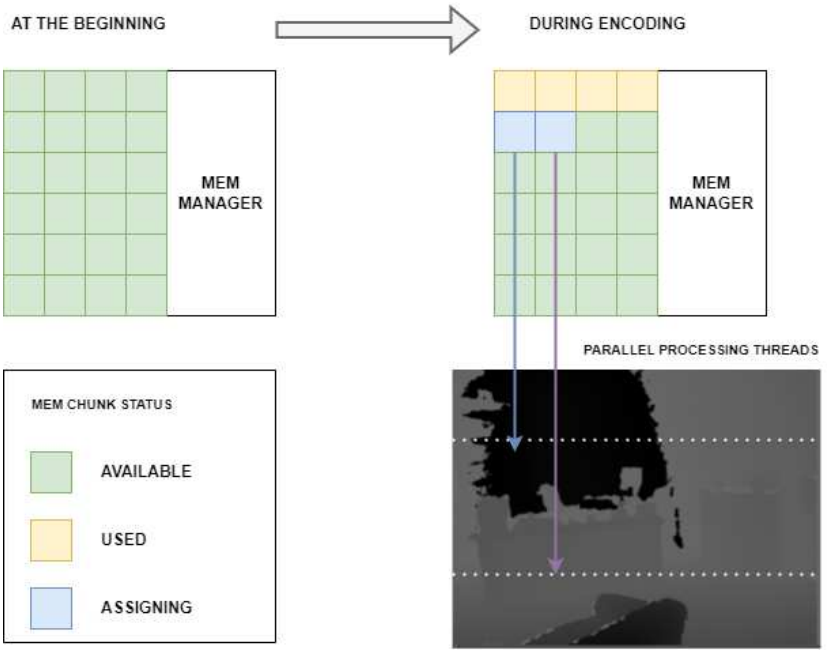
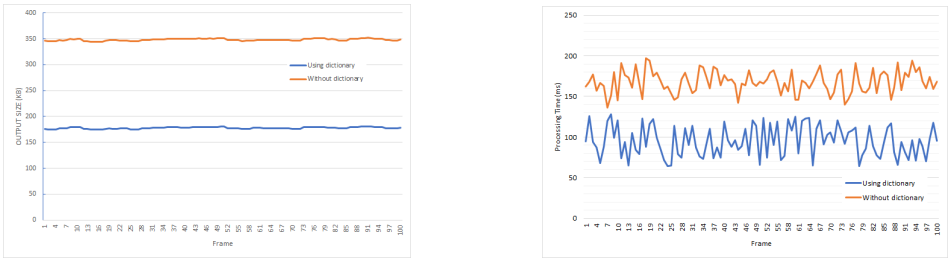


Fig. 3: Mem allocation scheme

dictionary file. Later on, during encoding — in case this step is enabled—, this computed dictionary is used. As can be seen in Figure 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).



(a) Time (b) Size

Fig. 4: Comparison with and without using dictionary

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

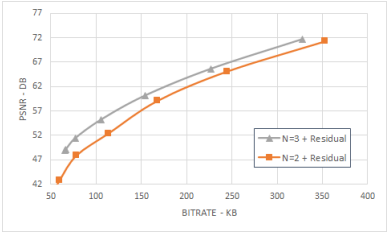
In order to compare the proposed methods of compression, six (6) different depth inner sequences, captured with a realSense L515 Lidar Camera, were generated. Each sequence had 150 frames of 1024x768 resolution. At the same time, we used a dataset from [2] which contained a set of office from RealSense D415 sensor (called scene 7) and a dataset from [8] (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, as has also been proposed in most of other related works. 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.

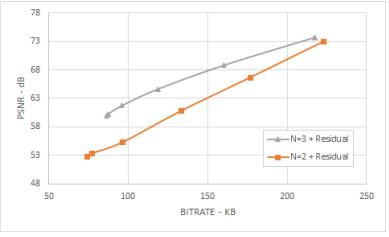
In addition to our implementation, the same sequences were compressed using an intra-coded mode of JPEG2000 coding standards, both in lossless and lossy modes. The PNG format, which is one of the fastest formats, was used. Finally, Jasper libraries were used for JPEG2000 [5]. All tests were run strictly 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 and the cubic fit version shown in Figure 5. The idea was to try to choose the best encoding configuration.



(a) Bitrate Scenario 0



(b) Bitrate Scenario 1

Fig. 5: Bitrate comparison with different fit parameters

As it was expected, the cubic fit tended to work with higher quality and less residual size than the linear fit.

4.2 Performance evaluation with different hardware

With regards 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.

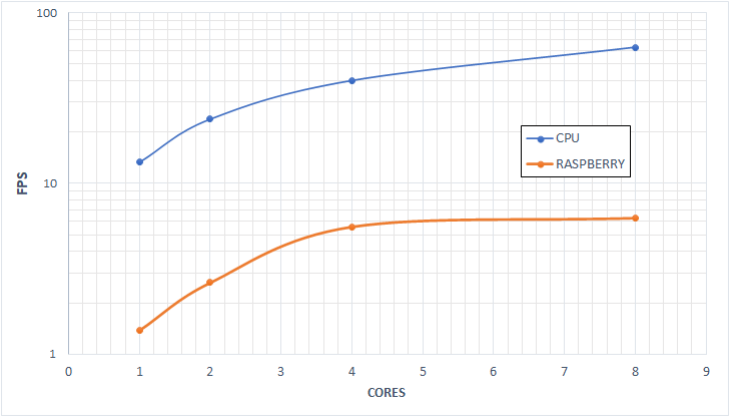


Fig. 6: Performance comparison with different cores

The Raspberry hardware has a lower clock rate of 1.1 GHZ, compared to the Intel 9300, which has a peek 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.

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 [2]	5.9%	23.5%	27.7%	
Scene 8 - Assorted [8]	14.0%	44.0%	44.9%	

Table 1: Lossless compression rate

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 degrees N, N=2 (lineal), N=3 (cubic) and N=5. The method variant was evaluated only by applying fitting (basic) and encoding residual with ZSTD compression. Residual is quantized using a 128 factor. All these results are presented in Table 2

For comparing the lossy version, the linear combination of SPEED, compression RATE and PSNR was used, keeping the lower values at their best. It could be 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 works better in producing a better reconstruction quality (PSNR) and a minimal 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

Table 2: Compression rate

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 reaches top-frame rate. When residual is added, the performance falls dramatically. This is because ZSTD has a one-thread implementation. When a higher polynomial function is evaluated, the gain in quality is 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 such algorithms, especially when decoding occurs in a low-profile hardware or when it is necessary to handle many cameras.

Finally, scene 8 results were compared to the work in [9] . 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 work since there is no implementation of such technique available. On the other hand, a high speed of about 110 fps was obtained (this work does not mention the computational effort) which is very useful for multi camera streaming, which is our main purpose.

		OUR	PNG	J2K		Basic			Residual		
		LOSSLESS		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

Table 3: Comparing encoding time measured in FPS

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 and 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 has issues when the object contour is poorly limited. This happens particularly in depth images obtained by Structured Sensors, but not on 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 more work to do in the evaluation of different cameras and in many different situations. Our next work will consist of implementing such methods in GPU in order to get a better speed-up and to define a unique descriptor for automatically parametrizing the algorithms, balancing between quality and speed.

6 Declarations

6.1 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 Internet were used for testing the algorithms.

6.2 Competing interests

In accordance with Springer policy and my ethical obligation as a researcher, I am reporting that I have not conflict of interests with any company or organization.

6.3 Funding

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

6.4 Authors' contributions

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

6.5 Acknowledgements

I thanks the PLADEMA's Institute team for infrastructure supporting.

6.6 Authors' information

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

References

[1] M. Kucherawy (Facebook),Zstandard Compression and the 'application/zstd' Media Type, url = https://datatracker.ietf.org/doc/html/rfc8878, (accessed: 01.09.2021).

[2] A. Dai, A. X. Chang, M. Savva, M. Halber, T. Funkhouser, and M. Nießner. Scannet: Richly-annotated 3d reconstructions of indoor scenes. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pages 5828–5839, 2017

[3] Wang, Biao Souza, Diego Mesa, Mauricio Chi, Chi Ching Juurlink, Ben Ilic, Aleksandar Roma, Nuno Sousa, Leonel. (2017). Highly parallel HEVC decoding for heterogeneous systems with CPU and GPU. Signal Processing: Image Communication. 62. 10.1016/j.image.2017.12.009.

[4] Yan, C. and Zhang, Y. and Xu, Z. and Dai, F. and Li, L. and Dai, Q. and Wu, F., A Highly Parallel Framework for HEVC Coding Unit Partitioning Tree Decision on Many-core Processors, IEEE Signal Processing Letters, 21-5,pp.573-576, doi=10.1109/LSP.2014.2310494 (2014).

[5] Skodras, A. and Christopoulos, C. and Ebrahimi, T., The JPEG 2000 still image compression standard , IEEE Signal Processing Magazine, 18(5), 36-58, doi=10.1109/79.952804 (2001)

[6] Jäger, F. , Contour-based segmentation and coding for depth map compression, Visual Communications and Image Processing (VCIP), doi=10.1109/VCIP.2011.6115989 (2011).

[7] D'Amato, Juan Vénere, M.. (2013). A CPU–GPU framework for optimizing the quality of large meshes. Journal of Parallel and Distributed Computing. 73. 1127–1134. 10.1016/j.jpdc.2013.03.007.

[8] Nathan Silberman, Derek Hoiem, Pushmeet Kohli and Rob Fergus, Indoor Segmentation and Support Inference from RGBD Images,European Conference on Computer Vision 2012.

[9] Kumar, S.H. and Ramakrishnan, K. R. , Depth compression via planar segmentation, Multimed Tools Appl,78, doi=https://doi.org/10.1007/s11042-018-6327-4 (2019).

[10] Carter, T , 3D body scanner platform, Company site, url = "https://fit3d.com/" (accessed: 01.09.2021).

[11] Dorodnicov,S. and Grunnet-Jepsen, A. and Puzhevich, a. and Piro d, Open-Source Ethernet Networking for Intel® RealSense™ Depth Cameras, Company site, url = ""https://dev.intelrealsense.com/docs/open-source-ethernet-networking-for-intel-realsense-depth-cameras" (accessed: 01.09.2021).

[12] Structure SDK (Cross-Platform), url = "https://structure.io/developers" (accessed: 01.09.2021).

[13] Kim, W. and Ortega, A. and Lai, P. and Tian, d. ,Depth Map Coding Optimization Using Rendered View Distortion for 3D Video Coding, IEEE Transactions on Image Processing,24 (11),3534-3545, doi=10.1109/TIP.2015.2447737 (2015).

[14] Shahriyar, S. and Murshed, M. and Ali, M. and Paul, M. ,Efficient Coding of Depth Map by Exploiting Temporal Correlation, 2014 International Conference on Digital Image Computing: Techniques and Applications (DICTA),1-8, doi=10.1109/DICTA.2014.7008105 (2014).

[15] Yaghouti Jafarabad, M. and Kiani, V. and Hamedani, T. and Harati, A. ,Depth image compression using geometrical wavelets, 2014 6th Conference on Information and Knowledge Technology,117-122, doi=10.1109/IKT.2014.7030344 (2014).

[16] Lei, J. and Li, S. and Zhu, C. and Sun, M. and Hou, C. ,Depth Coding Based on Depth-Texture Motion and Structure Similarities, IEEE Transactions on Circuits and Systems for Video Technology,25(2), 275-286, doi=10.1109/TCSVT.2014.2335471 (2015).

[17] Schwarz, S and Preda, M. and Baroncini, V. and Budagavi, M. and Cesar, P. and Chou, P. and Cohen, R. and Krivokuća, M. and Lasserre, S. and Li, Zhu and Llach, J. and Mammou, K. and Mekuria, R. and Nakagami, O. and Siahaan, E. and Tabatabai, A. and Tourapis, A. and Zakharchenko, V ,Emerging MPEG Standards for Point Cloud Compression, IEEE Journal on Emerging and Selected Topics in Circuits and Systems,12(1), doi=10.1109/JETCAS.2018.2885981 (2018).

[18] Duch, M.M., Morros, JR. and Ruiz-Hidalgo, J ,Depth map compression via 3D region-based representation, IMultimed Tools Appl, 76, 13761-13784, doi=https://doi.org/10.1007/s11042-016-3727-1 (2017).

[19] Mehrotra, S. and Zhang, Z. and Cai, Q. and Zhang, C. and Chou, P ,Low-complexity, near-lossless coding of depth maps from kinect-like depth cameras, IEEE 13th International Workshop on Multimedia Signal Processing, 1-6, doi=10.1109/MMSP.2011.6093803 (2011).

[20] Morvan, Y.,Acquisition, compression and rendering of depth and texture for multi-view video, Nano Letters - NANO LETT, doi=10.6100/IR641964 (2009).

[21] Delaitre, V. and Sivic, J. and Laptev, I.,Learning person-object interactions for action recognition in still images, Advances in Neural Information Processing Systems, (2011).