Efficient Compression for Server-Side G-Buffer Streaming in Web Applications

Sascha Räsch
Fraunhofer IGD, Germany

Maximilian Herz
TU Darmstadt

Johannes Behr
Fraunhofer IGD, Germany

Arjan Kuijper
Fraunhofer IGD, Germany

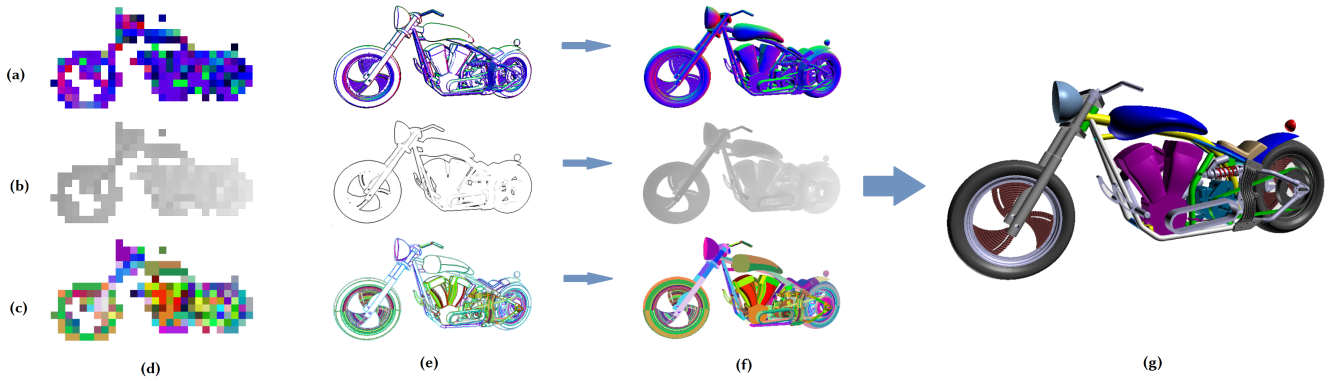


Figure 1: The multiple steps on the GPU for decoding a compressed G-Buffer. In (a), (b) and (c) the normal, depth and object id buffer is reconstructed, respectively. (d) are the buffers in low resolution, (e) are the sharp features and (f) are the decoded buffers. The final shaded image is shown in (g).

ABSTRACT

Remote rendering methods enable devices with low computing power like smart phones or tablets to visualize massive data. By transmitting G-Buffers , *Depth-Image-Based Rendering (DIBR)* methods can be used to compensate the artefacts caused by the latency. However, the drawback is that a G-Buffer has at least twice as much data as an image.

We present a method for compressing G-Buffers which provides an efficient decoding suitable for web applications. Depending on the computing power of the device, software methods, which run on the CPU, may not be fast enough for an interactive experience. Therefore, we developed a decoding which runs entirely on the GPU. As we use only standard WebGL for our implementation, our compression is suitable for every modern browser.

CCS CONCEPTS

• **Computing methodologies** → *Image-based rendering*; **Image compression**;

KEYWORDS

G-buffer, Streaming, Compression, Remote Rendering, Depth-Image-Based Rendering (DIBR), Mobile Devices, WebGL

ACM Reference format:
Sascha Räsch, Maximilian Herz, Johannes Behr, and Arjan Kuijper. 2017. Efficient Compression for Server-Side G-Buffer Streaming in Web Applications. In *Proceedings of Web3D '17, Brisbane, QLD, Australia, June 05-07, 2017,* 7 pages.
DOI: http://dx.doi.org/10.1145/3055624.3075952

Web3D '17, Brisbane, QLD, Australia

DOI: http://dx.doi.org/10.1145/3055624.3075952

1 INTRODUCTION

Modern Server-Side rendering concepts vary from streaming 3D model data to bitmap image streams according to Shi and Hsu (2015). Depending on the client's computing capabilities, the size of the 3D content, the bandwidth and the available resources on the server, a different approach is appropriate. Streaming 3D data to client devices is the costliest technique and is only practical for powerful clients. On the other hand, bitmap streams like NVIDIA (2017) are suitable for small devices such as smart phones or tablets. Behr et al. (2015) proposed a *Visual Computing as a Service infrastructure* which automatically determines which of both streaming methods fits best to the current situation.

However, the general drawback of bitmap streams is the latency between the user input and the update off the display. Moreover, the number of *frames per second (FPS)* is always limited by the

bandwidth. This is especially bad for virtual reality applications, as they require a high framerate according to Zielinski et al. (2015).

Streaming *Geometry Buffers (G-Buffers)* instead of just bitmaps solves both problems. G-Buffers contain additional information, e.g., the per-pixel depth, normal and the ID of the respective 3D object. By combining them with a depth buffer, the frame data is transformed into a 2.5 D representation. This enables for *Depth-Image-Based Rendering (DIBR)* methods, which are able to extrapolate new frames based on the already received data (see Xi et al. (2013)). Therefore, the framerate of the client device is decoupled from the actual received FPS and thus is not limited by the bandwidth anymore. Moreover, generating extrapolated frames can also be used to hide the latency artefacts.

In general, G-Buffers enable the client application to perform certain tasks without requesting new data from the server. For example, the normal and depth buffer can be used for shading and lighting. Thus, material and lighting settings can be changed without getting updates from the server. We use the object ID buffer to handle the selection and highlighting of objects.

However, the drawback of G-Buffers is their size. A single uncompressed G-Buffer of dimension 512×512 is already about 2.88 MB. With a framerate of just 8 FPS, we would require a bandwidth of 185 MBit/s. Therefore, an efficient compression is an indispensable requirement.

We propose a compression method for streaming G-Buffer data suitable for web client applications. Depending on the computing power of the device, common software decompression approaches, which run on the CPU, may not be fast enough. Therefore, we provide a decoding which runs entirely on the GPU of the device and thus no software decompression is necessary. As we use only standard *WebGL* technology, the decoding is viable on any standard web browser.

2 BACKGROUND AND RELATED WORK

2.1 G-Buffer

G-Buffers have originally been introduces by Saito and Takahashi (1990). They motivated the G-Buffer technique with decoupling the *geometric process* from the *physical* and *artificial process*. The physical and the artificial process are the steps performed in the screenspace shading.

Altenhofen et al. (2015) proposed a concept similar to G-Buffers called *Rich Pixels (Rixels)*. Each pixel has additional attributes like position data and physical quantities resulting from physical simulations. New Rixels are requested only if the viewport changes. Except for discarding the background pixels, no further steps have been applied to compress the Rixels data.

Doellner et al. (2012) presented an approach where G-Buffer cube maps are streamed to the client. The G-Buffer consists of an object id, a normal and depth buffer. In contrast to our definition, they add an additional buffer for storing color information. The buffers have been compressed using the PNG and JPEG image formats.

Kerzner and Salvi (2014) proposed a lossy compression to provide anti-aliasing and high visibility sampling rates to enable for fast deferred rendering. They exploit groups of primitives which define a surface with little to no curvature. This allows to locally reuse shaded samples and thus reduce the overall shading rate.

2.1.1 *Depth Buffer Compression.*

A depth buffer compression technique based on a combination of color and depth values has been developed by Förster et al. (2015). The correlation between the RGB and depth values is exploited to reduce the overall size of the encoded depth values. The decompression is based on an optimization problem which does not qualify for a fast web implementation.

G-Buffers and multipass rendering are standard techniques in game industry and have been optimized for many years. However, these optimization are not suitable for client server streaming.

A short survey over existing methods is presented by Hasselgren and Akenine-Möller (2006). They also introduced a new compression approach by extending a prediction method based on plane equations defined on small tiles.

Gautier et al. (2012) proposed a depth compression where the depth buffer is sampled in a regular grid pattern. Using an edge detection algorithm, additional samples are added to preserve sharp features. The decompression is performed by solving a PDE. Our compression (see Section 3) is based on this approach, but extended for arbitrary buffers and optimized for GPU decoding web applications.

2.1.2 *Normal Compression.*

An analysis by Meyer et al. (2010) of efficient normal encodings pointed out that the octahedron normal vectors are the best choice from a practical and theoretical point of view. An interesting alternative has been proposed by Keinert et al. (2015). They sample the unit sphere, the space of all normal vectors, using *Spherical Fibonacci* points. The decoding can be implemented in shader programs. However, one drawback is the loss of continuity. After mapping the normals into the Spherical Fibonacci point space, only discrete index values remain. Thus lossy compression methods cannot be applied to the encoded data.

2.2 Web Technologies

WebSocket is a modern protocol for two-way communication between a web client and a server. For a formal definition of the protocol see Fette (2011). It is generally available on all modern web browsers and can be used to stream arbitrary data. The *RFC 7692* extension (see Yoshino (2015)) has been proposed to compress the WebSocket data streams. Compressions like zlib deflate are applied on a per-message basis.

WebRTC is used for real-time audio and video streaming without depending on additional plugins. For a detailed description of WebRTC we refer the reader to Johnston and Burnett (2012). However, it is not available on all web browsers, e.g., Microsoft Internet Explorer. A single G-Buffer is composed of multiple buffers which cannot be encoded with the same compression. Thus, using WebRTC requires sending the buffers in asynchronous streams. Yet, previous experiments indicated that asynchronous streams can lead to heavy synchronization artifacts.

WebGL (see Marrin (2011)) is a JavaScript API for accessing the device's graphics hardware. The corresponding shading language GLSL enables implementing fast decoding algorithms by directly

using the shading units of the device. Version 1.0 of WebGL is generally available on all modern web browsers.

3 G-BUFFER COMPRESSION

In this section, we describe our new approach for compressing G-Buffer streams. Our G-Buffers are composed of three buffers, i.e., the object ID, the depth and the normal buffer as depicted in Figure 2.

(a) Object ID (b) Depth (c) Normal

Figure 2: Our definition of G-Buffers is composed of three different buffers, i.e., the id, depth and normal buffer.

A buffer can be distinguished according the continuity of the contained data. For example, the object ID buffer contains discrete information, whereas the depth and normal buffer are continuous for smooth surfaces. However, the intersection between different objects introduces discontinuities.

We decompose each buffer into patches of pixels which are either smooth if the buffer is continuous or constant if the buffer is discontinuous. For example, the ID buffer is discrete and thus discontinuous and has constant patches. On the other hand, the depth and normal buffer have patches which are not constant, but smooth. The question if the patches are smooth or constant is directly linked to the question if a buffer can be encoded lossy or lossless. An overview of the buffer properties is outlined in Table 1. A buffer is continuous if and only if it has smooth patches. We

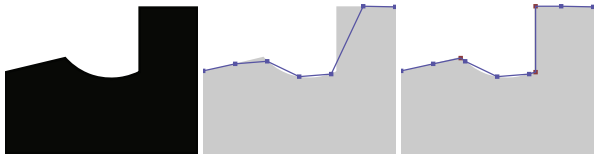
Buffer	Compression	Patches	Continuous
ID	Lossless	Constant	No
Depth	Lossy	Smooth	Yes
Normal	Lossy	Smooth	Yes

Table 1: The three sub-buffers which define our G-Buffer . A discontinuous buffer has constant patches and must be encoded lossless. Continuous buffers on the other hand, have smooth patches and thus can be encoded using lossy methods.

represent the patches by sampling them in a regular pattern. The values are reconstructed by interpolating between the samples. As shown in Figure 3, it is necessary to add additional samples to preserve sharp features and thus reducing artefacts.

3.1 Compression

We propose a compression method which is based on sub-sampling the original buffer. We have two different cases for creating a sample, i.e., a regular and an edge sample.:



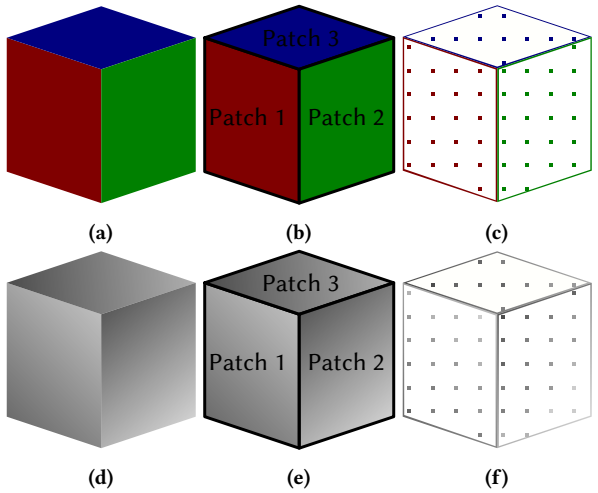
(a) (b) (c)

Figure 3: Figure (a) shows a continuous buffer illustrated as 1D surface. (b) shows a reconstructed surface by applying linear interpolation to the regular samples. However, sharp features are faded out. By adding additional sample points to cover them, the sharp features are preserved in (c).

Regular sample. A regular sample is created based on a regular grid-pattern, i.e., a pixel at position i, j is sampled if i and j are both divisible by some step size parameter $s \in \{2, 3, 4, \ldots\}$. Thus, for a buffer with size w, h, we have $\frac{w \cdot h}{s^2}$ samples, e.g., with a step size of $s := 16$, only 0.39% of the original pixels remain after the regular sampling.

Edge sample. Additional to the regular samples, we add further samples for covering the edges. That is, we add a sample for a pixel i, j if the function IsEdge returns **true** for some pixel i, j. IsEdge is a buffer specific edge filter function as defined in 3.1.1.

In Figure 4, we apply our sampling scheme to a discrete and a continuous buffer. First, we determine the edges using an edge filter to identify the patches 1, 2 and 3. Afterwards, regular and edge samples are created.



(a) (b) (c)

(d) (e) (f)

Figure 4: In (a) and (d), we have a discrete and a continuous buffer, respectively. We apply an edge filter to identify the different patches in (b) and (e). Finally, in (c) and (f), we sample each patch and add additional samples for covering the edges.

In order to efficiently encode the samples, we split them into two packages. We encode the regular samples as texture of size $\frac{w}{s} \times \frac{h}{s}$. The edge samples, which are not covered by the regular grid, are stored explicitly. That is, for each edge sample i, j with sample value p_{ij} we store (i, j, p_{ij}). We use 16 bit per coordinate and thus spend additional 32 bit for the position per edge sample.

3.1.1 Edge Filter. Let $w, h \in \{3, 4, \ldots\}$ be the width and height of the buffer and $p_{ij} \in V$ for $0 \leq i < w, 0 \leq j < h$ be the values of the buffer, where $V := \{0, 1, \ldots, N\}$ and $V := [-1, 1]$ for discrete and smooth buffers, respectively.

For discrete buffers, e.g., the object ID buffer, the edge filter at pixel i, j is defined as:

$$\text{IsEdge}(i,j) := \begin{cases} \textbf{true} & \text{if } p_{ij} \neq p_{i+1,j} \text{ or } p_{ij} \neq p_{i-1,j} \\ \textbf{true} & \text{if } p_{ij} \neq p_{i,j+1} \text{ or } p_{ij} \neq p_{i,j-1} \\ \textbf{false} & \text{else} \end{cases} \quad (1)$$

Thus, for discrete buffers, edges mark neighboring pixels with different values.

For smooth buffers, we check for the curvature, i.e., the second discrete derivative. If the curvature is 0, the neighboring pixels are linear to each other, which they are not if the curvature is non-zero. The more the curvature differs from 0, the more significant are the artifacts introduced by applying linear interpolation between the sample points.

We denote the curvature in direction X and Y for a value at pixel i, j as c_{ij}^X and c_{ij}^Y and define them formally as:

$$c_{ij}^X := (p_{i+1,j} - p_{ij}) - (p_{i,j} - p_{i-1,j}) = p_{i-1,j} - 2p_{ij} + p_{i+1,j} \quad (2)$$

$$c_{ij}^Y := (p_{i,j+1} - p_{ij}) - (p_{i,j} - p_{i,j-1}) = p_{i,j-1} - 2p_{ij} + p_{i,j+1} \quad (3)$$

An edge is detected if the curvature exceeds a certain threshold $\varepsilon \geq 0$. The smooth edge filter is then defined as:

$$\text{IsEdge}(i,j) := \begin{cases} \textbf{true} & \text{if } \max(|c_{ij}^X|, |c_{ij}^Y|) > \varepsilon \\ \textbf{false} & \text{else} \end{cases} \quad (4)$$

The larger we choose ε, the more aggressive is our lossy compression.

We have to slightly modify the edge filter for buffers with more than one component like the normal buffer. Let $\text{IsEdge}_k(i,j)$ be the edge filter as defined above for the k-th component of the buffer. Then we define the combined edge filter as:

$$\text{IsEdge}(i,j) := (\exists k)\ \text{IsEdge}_k(i,j) \quad (5)$$

Thus, we have an edge if the curvature of any of the components exceeds the threshold.

3.2 Streaming

We use WebSockets to stream the compressed G-Buffers as binary frames to the client device. The binary layout of a single frame is depicted in Figure 5. The frame header consists of the width and height of the G-Buffer and the body is divided into the data for the regular and irregular samples. The regular sample data is encoded as three distinct textures for the regular ID, depth and normal sample data, respectively. The irregular samples are packed into three arrays which are consecutively combined into a single binary chunk.

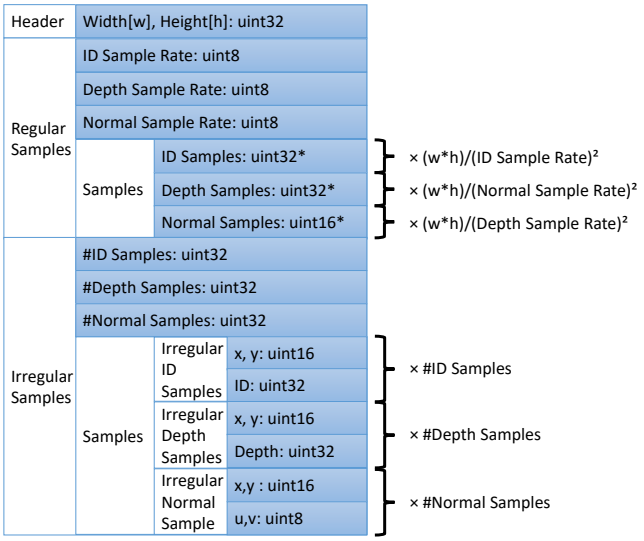


Figure 5: The binary structure of a single frame streamed to the client. The header consists of the width and height of the streamed G-Buffer. The body is divided into the regular and irregular samples. The regular samples are encoded as three textures of reduced size for the ID, depth and normal buffer. The irregular samples for all three buffers are packed together into a single binary chunk. Each irregular sample consists of a x, y position plus the actual value.

3.3 Decompression

We upload each chunk of regular samples as a single WebGL texture into the GPU memory and the irregular sample data is loaded as single WebGL buffer.

To decompress the G-Buffer , we need three draw calls as illustrated in Figure 6. In the first two drawcalls, we render the regular

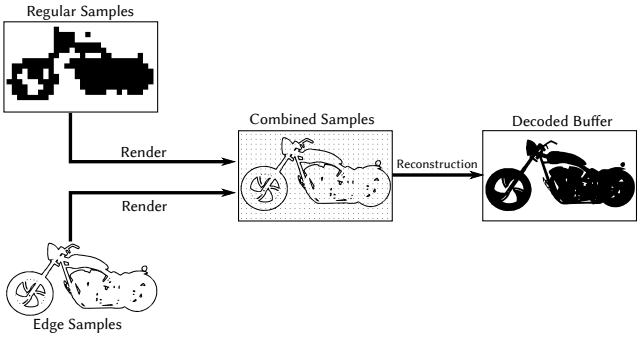
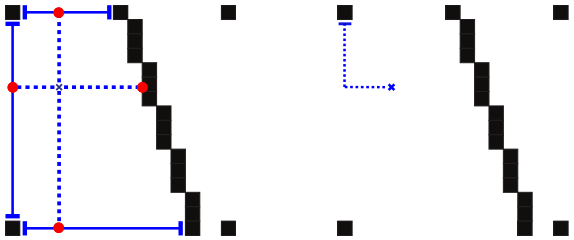


Figure 6: The flow diagram of decompressing a single buffer. First, the regular and edge samples are rendered into a combined buffer texture. The buffer texture is then used for the actual decoding in a second step. The decoding step can also be performed as part of the deferred shading.

and the irregular samples into a combined texture buffer. Rendering the regular samples is just a simple texture lookup and each set

of irregular samples is rendered as POINTS using vertex attribute buffers.



(a) (b)

Figure 7: (a) and (b) show the reconstruction process for continuous and discrete buffers, respectively. For the continuous case (a), the value is interpolated between the four red dots. The red dot values are either encountered sample points, or reconstructed values along the edges of the cell. In the discrete case (b), we just follow the iteration along the blue line until we encounter a sample.

In the last step, we use the combined sample texture buffer as input for our reconstruction step. As illustrated in Figure 7, we use linear interpolation between the samples for continuous buffers and a simple iteration rule for the discrete case.

For the discrete case, we simply iterate to the left until we either encounter another sample or reach the edge of the current grid cell. If we didn't hit any sample, we iterate up to the next regular grid sample. The result is the first encountered sample.

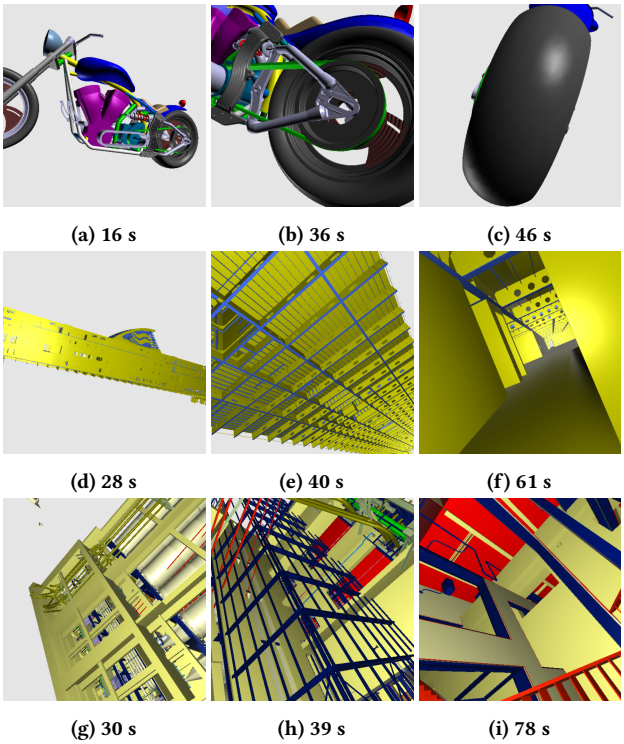
For continuous buffers, we iterate into the four directions $+X$, $-X$, $+Y$, $-Y$. Again, we stop if we either encounter a sample or reach the edge of the current grid cell. If we reached the edge, but didn't encounter any sample, we perform a linear interpolation along the edge. Finally, the result is a linear interpolation between the four reconstructed/encountered samples.

4 RESULTS

We created three different test cases for evaluating our compression method. Each test case is defined by one model and a series of camera operations like, rotate, move or zooming. In Table 2, we outline the properties of the model and the duration for the test cases. The *Harley* test case illustrates a bike with a rather low complexity of 285 thousand triangles. The *Ferry* test case shows a model of a complex ferry with cabins and corridors with a moderate size of 1.4 million triangles. The *Powerplant* model has the highest complexity and visualizes the UNC power plant model [1] with about 15.4 million triangles. Screenshots for each test case are depicted in Figure 8.

Model	Num. Vertices	Num. Triangles	Duration
Harley	341 K	285 K	67 s
Ferry	2.4 M	1.4 M	75 s
Powerplant	15.4 M	12.7 M	87.5 s

Table 2: The three different test cases used for our experiments with their respective properties.



(a) 16 s (b) 36 s (c) 46 s

(d) 28 s (e) 40 s (f) 61 s

(g) 30 s (h) 39 s (i) 78 s

Figure 8: Screenshots from the three different test cases.

In our test cases, the object IDs and the depth values are encoded with 32 bit precision. For the normal data, we use only 16 bit by applying an octahedron encoding.

4.1 Traffic

In this Subsection, we analyze the generated traffic by transmitting **8** frames per second with a G-Buffer resolution of **512 × 512**. As the raw uncompressed size of a single frame is about **2.88 MB**, we would require a bandwidth of **185 MBit/s**.

We compare our compression approach with the permessage-deflate compression extension for WebSockets, which is supported by modern web browsers like Firefox or Chrome. However, to perform the per frame decompression on the client, the device must have moderate computing power.

Therefore, we also analyze a combined method, where the size of our compressed frames is further reduced by applying the WebSocket deflate compression. In the following discussion, we call our compression technique *Sampling Compression*.

In Figure 9, we show the average compression of all three test cases with each of the three methods. The average compression for our Sampling Compression method ranges between 18 % and 23

[1] http://gamma.cs.unc.edu/POWERPLANT/index.html

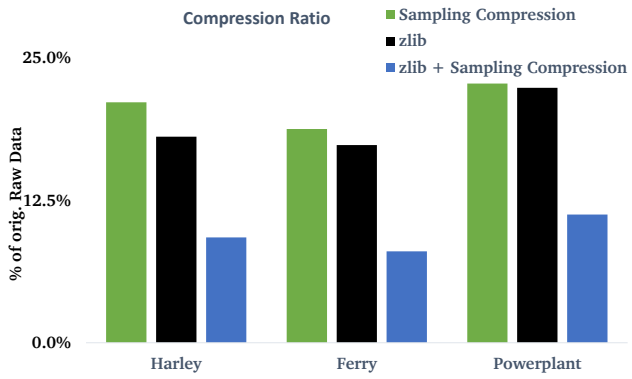


Figure 9: The average compression ratio in percentage for the different test cases and compression methods. The compression ratio indicates the size of the compressed data relative to the original uncompressed data size. Thus, less is better.

%. Thus, our method performs better than common hardware supported lossy texture compression formats like DXT2, DXT3, DXT4 and DXT5 which compress with 25 %. However, DXT compressions cannot be applied to the object ID buffer or the depth buffer. The WebSocket permessage-deflate compression extension slightly outperforms our approach and ranges between 18 % and 22 %. When combining our Sampling Compression with the WebSocket compression, we achieve the best compression ratios which are between 8 % and 11 %.

In Figure 10, we analyze the needed bandwidth in MBit/s overtime. The required bandwidth of our compression method has high peaks at second 36, 40 and 39 for the Harley, Ferry and Powerplant test case, respectively. The corresponding screenshots are depicted in Figure 8b, 8e and 8h. In all three screenshots, we have detailed small structures which causes a lot of edges in the compression. Thus, a lot of samples have to be stored which expand the size of the compressed frames. On the other hand, if we have large smooth surfaces as in Figure 8c, we only store a few edge samples and thus have small frames.

In second 61 and 78 in the Ferry and the Powerplant test case we outperform permessage-deflate. In both cases, the screen is covered by the 3D scene, but the image consists mainly of smooth surfaces. Again, just a few samples are required to encode the structure. However, due to slight numerical errors, the pixels are not perfectly linear. As deflate is lossless, it cannot eliminate these numerical artifacts and performs less effective.

5 CONCLUSION AND FUTURE WORK

In this paper, we have presented a compression for G-Buffers which can be applied for remote rendering using DIBR methods in web applications. Our compression method uses only standard web technology like WebGL and thus can be used on any client device. Our proposed compression provides a decompression which can be performed entirely on the hardware of the client. This is essential for devices like smartphones or tablets with low computing power.

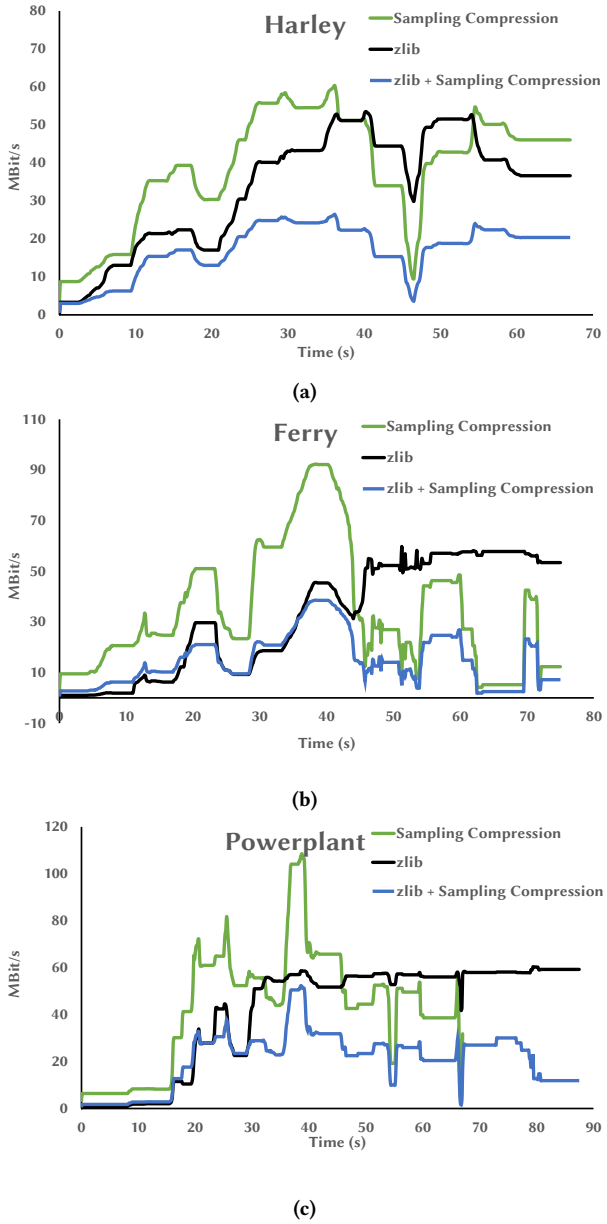


Figure 10: The MBit/s over time while moving and rotating the camera for a period of 67, 75 and 89 seconds for the models Harley, Ferry and Powerplant, respectively. We have a constant framerate of 8 fps and each G-Buffer has a resolution of 512×512. The traffic for the uncompressed data is constant and is about 185 MBit/s.

Based on our results, we have an average compression between 19% and 23% relative to the original uncompressed data. Moreover, for capable devices, we presented a combination of our method with the browser built in WebSocket permessage-deflate compression

extension, where we achieved even better average compression ratios between 9% and 11%.

A major concern for future work is to further improve the compression of the G-Buffers . As we stated before, the regular sampled data is only about 0.39% of the original data for frames of size 512×512. For larger resolutions, the percentage is even smaller. Thus, the majority of the transmitted data are the samples to cover the edges. We encoded these samples by storing their i, j buffer positions. Thus, half of the edge sample data are just the coordinates of the edge samples. As the edge data is often similar to a line strip, future research should analyze the potential advantages by using lines instead of points. This could reduce the overall amount of position data, but demands an efficient server side line detection algorithm which is executed every frame. Moreover, the overall number of edge samples can be reduced if linear interpolation is used for the detected line segments.

Another direction for future research is to focus more on the application in the context of DIBR. One could develop a heuristic to determine the required accuracy for the next frame, depending on the previously send frames and the current camera movements. By reducing the accuracy, less edge samples are generated and thus less data is transmitted.

REFERENCES

Christian Altenhofen, Andreas Dietrich, André Stork, and Dieter Fellner. 2015. Rixels: Towards Secure Interactive 3D Graphics in Engineering Clouds. *The IPSI BgD Transactions on Internet Research* (2015), 31.

Johannes Behr, Christophe Mouton, Samuel Parfouru, Julien Champeau, Clotilde Jeulin, Maik Thöner, Christian Stein, Michael Schmitt, Max Limper, Miguel de Sousa, Tobias Alexander Franke, and Gerrit Voss. 2015. webVis/Instant3DHub: Visual Computing As a Service Infrastructure to Deliver Adaptive, Secure and Scalable User Centric Data Visualisation. In *Proceedings of the 20th International Conference on 3D Web Technology (Web3D '15)*. ACM, New York, NY, USA, 39–47. DOI: http://dx.doi.org/10.1145/2775292.2775299

Juergen Doellner, Benjamin Hagedorn, and Jan Klimke. 2012. Server-based Rendering of Large 3D Scenes for Mobile Devices Using G-buffer Cube Maps. In *Proceedings of the 17th International Conference on 3D Web Technology (Web3D '12)*. ACM, New York, NY, USA, 97–100. DOI: http://dx.doi.org/10.1145/2338714.2338729

Ian Fette. 2011. The websocket protocol. (2011).

E. C. Förster, T. Löwe, S. Wenger, and M. Magnor. 2015. RGB-guided depth map compression via Compressed Sensing and Sparse Coding. In *2015 Picture Coding Symposium (PCS)*. 1–4. DOI: http://dx.doi.org/10.1109/PCS.2015.7170035

J. Gautier, O. Le Meur, and C. Guillemot. 2012. Efficient depth map compression based on lossless edge coding and diffusion. In *2012 Picture Coding Symposium*. 81–84. DOI: http://dx.doi.org/10.1109/PCS.2012.6213291

Jon Hasselgren and Tomas Akenine-Möller. 2006. Efficient depth buffer compression. In *Graphics Hardware*. 103–110.

Alan B. Johnston and Daniel C. Burnett. 2012. *WebRTC: APIs and RTCWEB Protocols of the HTML5 Real-Time Web*. Digital Codex LLC, USA.

Benjamin Keinert, Matthias Innmann, Michael Sänger, and Marc Stamminger. 2015. Spherical Fibonacci Mapping. *ACM Trans. Graph.* 34, 6, Article 193 (Oct. 2015), 7 pages. DOI: http://dx.doi.org/10.1145/2816795.2818131

Ethan Kerzner and Marco Salvi. 2014. Streaming g-buffer compression for multi-sample anti-aliasing. In *Proceedings of High Performance Graphics*. Eurographics Association, 1–7.

Chris Marrin. 2011. Webgl specification. *Khronos WebGL Working Group* (2011).

Quirin Meyer, Jochen Süßmuth, Gerd Sußner, Marc Stamminger, and Günther Greiner. 2010. On Floating-point Normal Vectors. In *Proceedings of the 21st Eurographics Conference on Rendering (EGSR'10)*. Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, 1405–1409. DOI: http://dx.doi.org/10.1111/j.1467-8659.2010.01737.x

NVIDIA. 2017. Nvidia grid: Stream applications and games on demand. http://www.nvidia.com/object/nvidia-grid.html. (2017).

Takafumi Saito and Tokiichiro Takahashi. 1990. Comprehensible Rendering of 3-D Shapes. *SIGGRAPH Comput. Graph.* 24, 4 (Sept. 1990), 197–206. DOI: http://dx.doi.org/10.1145/97880.97901

Shu Shi and Cheng-Hsin Hsu. 2015. A Survey of Interactive Remote Rendering Systems. *ACM Comput. Surv.* 47, 4, Article 57 (May 2015), 29 pages. DOI: http://dx.doi.org/10.1145/2719921

Ming Xi, Liang-Hao Wang, Qing-Qing Yang, Dong-Xiao Li, and Ming Zhang. 2013. Depth-image-based rendering with spatial and temporal texture synthesis for 3DTV. *EURASIP Journal on Image and Video Processing* 2013, 1 (2013), 9. DOI: http://dx.doi.org/10.1186/1687-5281-2013-9

Takeshi Yoshino. 2015. *Compression Extensions for WebSocket*. Technical Report.

D. J. Zielinski, H. M. Rao, M. A. Sommer, and R. Kopper. 2015. Exploring the effects of image persistence in low frame rate virtual environments. In *2015 IEEE Virtual Reality (VR)*. 19–26. DOI: http://dx.doi.org/10.1109/VR.2015.7223319