Vision, Modeling, and Visualization (2018)
F. Beck, C. Dachsbacher, and F. Sadlo (Eds.)

Web-based Volume Rendering using Progressive Importance-based Data Transfer

Finian Mwalongo[1], Michael Krone[2], Guido Reina[1], and Thomas Ertl[1]

[1]Visualization Research Center (VISUS), University of Stuttgart, Germany
[2]Big Data Visual Analytics in Life Sciences (BDVA), University of Tübingen, Germany

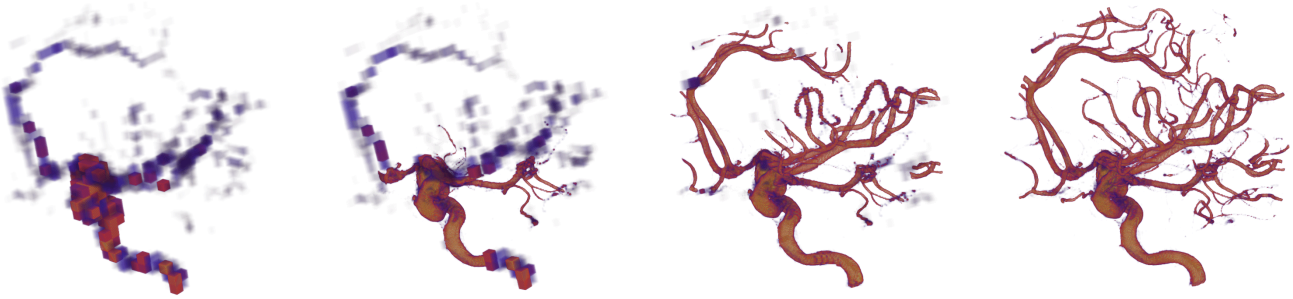

Figure 1: *Different snapshots of our bricked volume rendering method showing the aneurism data set. We use multiple levels of detail of the volumetric data, which are divided into bricks and progressively transferred to the client. The leftmost image shows the lowest resolution while the rightmost image shows the final image with the original resolution of the data set. Our volume ray marching was implemented in WebGL 2.0 and uses a simple 1D transfer function.*

Abstract

WebGL 2.0 makes it possible to implement efficient volume rendering that runs in browsers using 3D textures and complex fragment shaders. However, a typical bottleneck for web-based volume rendering is the size of the volumetric data sets. Transferring these data to the client for rendering can take a substantial amount of time, depending on the network speed. This can introduce latency that can in turn affect interactive rendering at the client. We address this challenge by introducing a multi-resolution bricked volume rendering, where data is transferred progressively. Similar to MIP-Mapping, the volume data is divided into multiple levels of detail. Each level of detail is broken down into bricks. The client requests the data brick by brick starting with the lowest resolution and renders each brick immediately as it is received. The 3D volume texture is updated as bricks with higher resolution are received asynchronously from the server. The advantages of this algorithm are that it reduces latency, the user can see at least a reduced-detail version of the data almost immediately, and the application always stays responsive while the data is updated. We also implemented a prioritization scheme for the bricks, where each brick can be assigned an importance value. Using this information, the client can request more important bricks first. Furthermore, we investigated the influence of data compression on the transfer and processing times.

CCS Concepts

• ***Computing methodologies*** → *Ray tracing; Scientific visualization; Volumetric models;*

1. Introduction

Advancements in volume data acquisition techniques and simulation technologies induce the generation of increasingly large volumetric data sets. Remote visualization of these data, especially in web browsers, poses a challenge in terms of bandwidth, latency, and computational power at the client side. While advances of the client hardware capabilities make it possible to use involved rendering methods like GPU-accelerated volume ray marching even on most mobile devices, network speed and especially latency are still limiting factors for achieving interactive browser-based visualization. Therefore, in a web-based environment, efficient low-latency data transfer from server to client is required. The bandwidth prob-

DOI: 10.2312/vmv.20181264

lem can be partially remedied by techniques that use compression to reduce the amount of data transferred to the client [MKRE16]. However, most of these techniques require that the whole data is transferred to the client at once. Moreover, decompression of large data sets on the client can introduce additional latency. Thus, compression has to be applied with caution [LWS*13].

Another popular option for remote visualization is image or video streaming, where the server renders the images and sends them to the client. While this is attractive for clients with limited rendering capabilities, it still requires a high network speed and low latency. Furthermore, the server has to carry out the rendering. Thus, this approach does not scale well for higher numbers of users.

For visualizing large data sets, multi-resolution rendering and bricking techniques have been introduced in desktop platforms in order to address the problems of CPU-GPU bandwidth and limited memory on GPUs [BHMF08,BHP15]. These techniques, combined with efficient progressive data transfer, can be exploited in the browser in order to minimize latency due to large data transfer over the network.

Contributions: We present a GPU-based volume ray marching technique that uses a progressive data transfer utilizing multiple levels of detail and bricking for interactive rendering of remote data sets. Our proposed algorithmic pipeline described in section 3 is optimized for bandwidth and minimizes latency for the user on the client. This is achieved by using a multi-resolution bricked volume representation with progressive importance-based data transfer, which is described in section 4. As demonstrated in section 5, this ensures interactive client-side rendering of large volume data.

As soon as the first brick with the lowest resolution is transferred, the client starts rendering. That is, the user almost immediately gets at least a first glimpse of the data at reduced detail and can start exploring the data while bricks with higher resolution are streamed, thus increasing the quality of the volume rendering progressively as shown in Figure 1. Optionally, the client can get importance information for the volume bricks from the server and request the bricks in this order, so that more important bricks are received first. In the results section, we show that our algorithmic pipeline is feasible to transfer and visualize large volumetric data sets at interactive frame rates for web-based volume rendering. We also investigate the effects of brick-level compression on data transfer and processing times on the client.

2. Related Work

Previous work on GPU-based volume rendering using WebGL mainly focuses on techniques that allow visualization of volumetric data without the use of 3D textures, since these works were implemented using WebGL 1.0 that has no support for 3D textures. Among the first works on web-based visualization using WebGL is that of Congote et al [CSK*11]. They implemented a GPU-based volume ray marching in WebGL to visualize medical volumes and weather radar volumetric data sets at interactive frame rates. Their technique uses a 2D texture atlas to store volumetric data due to lack of 3D texture support in WebGL 1.0. Movania and Feng [MF12] implement a single-pass GPU-based ray caster in WebGL for visualization of medical data sets. Noguera and Jimenéz [NJ12] extend the use of 2D texture mosaics not only to address the lack of 3D textures in WebGL 1.0 and in mobile devices supporting OpenGL ES 2.0, but also combine it with multi-texture support to use all available texture units on the GPU to store large volume data.

None of the aforementioned approaches uses multi-resolution volumes or bricking to address network latency and bandwidth issues. Such techniques have been employed in desktop applications for the interactive visualization of large volumetric data sets [GWGS02, GS04, EHK*06]. For example, multi-resolution techniques have been employed to reduce interaction latency by allowing the user to view a low-resolution model during interaction and render a high-resolution model when there is no user interaction [EHK*06]. Bricking techniques are mainly used for addressing the problem of visualizing large volume data by allowing individual bricks to be streamed to the GPU from memory or local disk [BHP15]. Frey et al. [FSE15] present a remote visualization approach for optimizing the rendering process through adaptive sampling and compression for efficient transfer of individual frames rendered by a server. The goal of their approach is to minimize and maintain constant response latency for user requests. Frey et al. [FSE13] developed *volumetric depth images* for remote volume rendering, which reduce the amount of data that has to be transferred to the client. In contrast to our algorithm, the rendering is done on the server, which requires the server to provide not only the storage but also the rendering infrastructure.

Progressive data transfer and compression on the web has so far mainly been used for rendering of mesh data, for example by Lavoue et al. [LCD13] or by Ponchio and Dellepiane [PD15]. Yang et al. [YSG15] presented a specialized compression technique for time-varying volumetric data that combines S3TC texture compression with *deflate* compression for efficient transmission of volumetric data to the browser. On the client side, the compressed data is inflated and uploaded to the GPU as video textures, which are directly supported by WebGL.

Obrul et.al [OLv12] present a method for progressive visualization of losslesly compressed DICOM files over the Internet to address bandwidth and storage issues. The compression is performed using a quad-tree based progressive lossless compression technique for volumetric datasets presented by Klajnsek et.al [KZNP08]. Decompression and rendering is done in the browser using a Java applet. A comprehensive overview of web-based volume visualization methods and compressed data transfer for remote rendering can be found in the recent state-of-the-art report by Mwalongo et al. [MKRE16].

Our work differs from these previous works in that it combines multi-resolution volumes and bricking to address the problem of interaction latency and efficient data transfer in a network environment using WebGL 2.0. First, a hierarchical multi-resolution volume is computed on the server, where each level has half the resolution of the previous level. Each level of detail is divided into uniformly sized bricks, leading to short per-brick transfer times even for low network bandwidth. By using a bricked multi-resolution volume representation, the user can start viewing the lowest-resolution volume while the volume is progressively refined

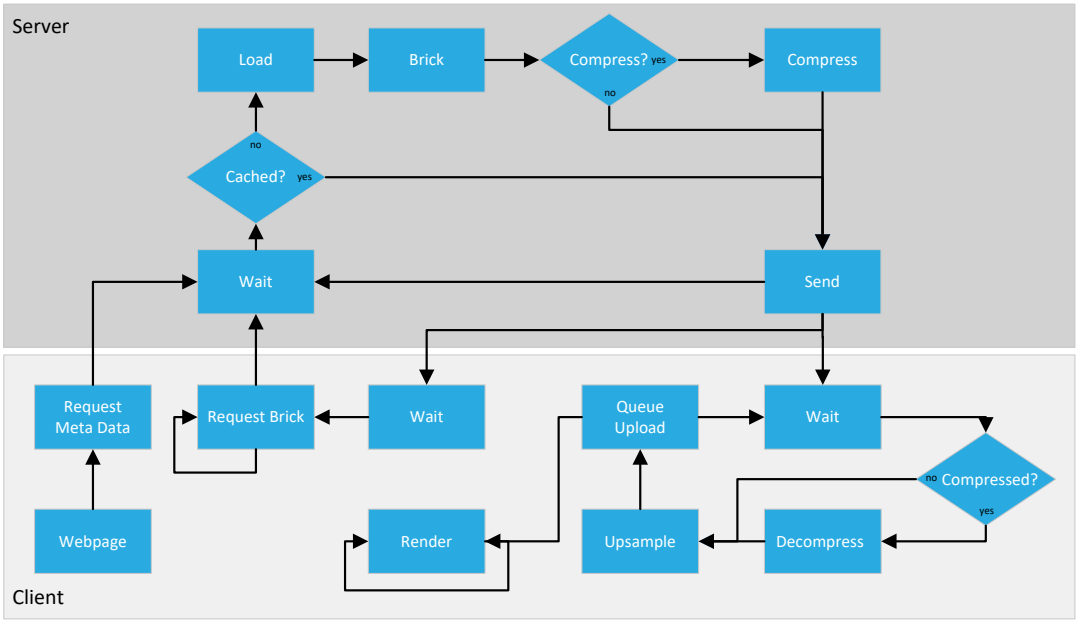

Figure 2: *Algorithmic pipeline of our client-server architecture for the bricked volume rendering.*

using higher-resolution bricks, which are asynchronously streamed from the server in a background thread.

3. Algorithmic Pipeline

Our algorithmic pipeline follows a client-server architecture. The server is responsible for storing the volumetric data, generating the bricks of the different levels of detail, and encoding the brick data before transferring them to the client. The client is responsible for decoding the brick data and rendering. An overview of the pipeline is shown in Figure 2. The application allows the user to visualize volumetric data stored on the server by providing a browser-based interface. The interaction begins with the client requesting meta data about the volume to be visualized via a web page. The server, upon receiving this request, firsts checks whether the data set is already cached. If not, the data set is loaded, a MIP-map-like level of detail pyramid is generated and each level is bricked into equally sized subvolumes. Optionally, the bricks are compressed to reduce the memory footprint. Afterwards—or if the data was already loaded—the volume meta data is sent to the client. These meta data include the original volume resolution, the brick resolution, and the number of levels of detail.

The client can now use this information to allocate texture memory for the whole volume. It also uses the meta data to generate requests for the bricks from the server. The initially allocated texture is updated with the brick data as it is received from the server. Receiving and decoding the brick data is done asynchronously in a separate thread to avoid stalling the main thread.

The volume texture is rendered continuously as soon as the first brick is received, processed, and uploaded. That is, the progressive refinement of the volume is visible as soon as it is available on the GPU. Since the volume texture always has the full resolution, the bricked structure of the data does not affect the rendering.

4. Implementation Details

This section describes our prototypical implementation of the algorithmic pipeline described in Section 3. We discuss implementation details of the server and the client side, and the data structures.

4.1. Server-side Brick Generation

We implemented our server using the popular *Node.js* runtime[†]. The brick generation is performed on the fly when any data of a particular volume data set is requested for the first time by a client. After loading the volume data set, the first step is to compute how many levels of detail have to be generated. Similar to a classical 2D texture MIP map, each level has half the resolution in each dimension as the previous level until the lowest level is reached, which consists of just one brick. The voxel values of the lower levels are computed by averaging the corresponding $2\times2\times2$ voxel values of the previous level. Each level is divided into fixed-size cubic bricks. We typically use a brick resolution of $32\times32\times32$ or $64\times64\times64$ voxels, which results in a reasonably small memory footprint per brick but also does not lead to too many HTTP requests by the client for the entirety of the bricks. Once the volume is loaded from disk and bricked, it is cached in memory and subsequent requests will directly get the data of a particular brick.

The additional memory for storing the lower-resolution bricks is very low. Since we divide the resolution by a factor of two in

[†] Node.js 9.5.0 https://nodejs.org/ (last accessed 02/21/2018).

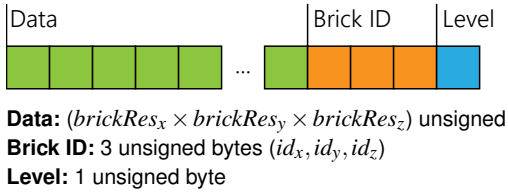

Figure 3: *Memory layout for brick data serialization between server and client.*

each dimension, the total memory can be computed as $\sum_{i=0}^{\infty}(1/8)^i = 1 + 1/7$. That is, the additional memory for all lower levels equals $1/7$ of the memory needed for the full volume.

The bricked volume data could also be precomputed and stored to disk to eliminate the processing time when loading a new volume data set. However, we did not include this in our prototype, since the bricking only has to be done once, as mentioned above.

4.2. Data Encoding and Transfer

The data format for sending the brick data from the server to the client is shown in Figure 3. The serialized message starts with the actual data of the brick, followed by the brick indices—i.e., the (x,y,z)-coordinates of the brick in the current level—and the level of detail to which the requested brick belongs. Since the bricks always have the same resolution, independent of the current level, the messages always have the same size. All values are stored as byte values (UINT8), that is, the current implementation can address 16k×16k×16k volumes if 64×64×64 voxel bricks are used. The information about the brick indices and level are needed since the receiving thread in the client cannot get this information from the main thread. That way, incoming brick data can be processed without stalling the rendering thread.

In order to keep the data that has to be transferred from the server to the client as small as possible, we optionally use compression. The above mentioned serialized brick data is compressed using the Snappy.JS library[‡]. If compression is used, the brick size is set to 64×64×64 voxels by default. Otherwise, the amount of data to be transferred for each brick request would be very low.

4.3. Client-side Data Processing

After the client has requested a new data set and it was loaded and bricked by the server, the client receives meta data about the data set (i.e., the voxel resolution of the data set, the voxel resolution of the bricks, the number of levels etc.). The client uses this information to allocate the required 3D texture storage, which has the same resolution as the original volume data. We use immutable textures, which are created using `texStorage3D()`, since these are more efficient than mutable textures as per WebGL 2.0 specification [Khr13]. After this, the volume bricks can be requested.

‡ Snappy.JS https://github.com/zhipeng-jia/snappyjs (last accessed 02/21/2018).

Each time a brick is received from the server, this texture memory has to be updated with the new brick data. If the data was compressed for transfer, it is of course first decompressed again. Since each brick has a different footprint in the full volume depending on which level of the volume it is in, the brick data has to be upsampled to the resolution of the original volume before being sent to the GPU. During the upsampling, the information about the level of the brick and the brick index is used to compute the corresponding global voxel offsets in the full-resolution 3D texture. The data is upsampled by duplicating the voxels by a factor that is computed as the ratio between the resolution of the original volume data and the resolution of the volume at the corresponding level of detail of the brick. Receiving and upsampling each brick are performed by a web worker [W3C12]. The transfer of data between this worker and the main thread is done using transferable objects to avoid copying of the data. This has already been previously demonstrated by Mwalongo et al. [MKB*15, MKB*16] for particle rendering. Finally, the upsampled brick data is used to update the volume texture using `texSubImage3D()`.

4.4. Prioritization of Volume Bricks

The initial bricking described in Section 4.1 allows the client to request the data level by level, starting with the lowest resolution (just one brick) to get an initial image very fast and adding higher-resolution data level by level until the full resolution is reached. However, volume data often has regions that are more important than other ones. That is, it can be worthwhile to transfer the bricks out of order, so that more important bricks are available earlier in the client than less important ones. Thus, important regions in the data set are available for rendering sooner in full resolution.

Since it is hard to find a measure of importance that applies to all kinds of data sets, we decided to use entropy as a measure of importance in our prototype. That is, we compute the entropy of each brick and assign it as the importance value of that brick. The reasoning behind this is that bricks with a higher disorder and more diverse voxel values contain more information. Our algorithmic pipeline would of course also work with other, arbitrary measures of importance (e.g., visual saliency metrics [KV06, JC10, MHD*18]).

After assigning an importance value to each brick, a list consisting of pairs of importance values and the corresponding brick indices—that is, the level and (x,y,z)-coordinate of each brick—is generated. This list is sorted by the importance values in ascending order. The client can request this sorted list and use it to prioritize the bricks. That is, the client can request the bricks in the order of their importance. The only exception from this rule is the lowest-resolution level consisting of just one brick. Since this brick should always be transferred first to the client to give a first impression of the whole data set, it is assigned the maximum importance value.

Note that the scheme described above can lead to bricks being requested out of level order, that is, a brick from a lower level (with higher resolution) can be received earlier than the corresponding brick with higher level (lower-resolution), which overlaps with the first brick. In this case, it has to be ensured that the lower-resolution brick does not overwrite the higher-resolution brick (see Figure 4).

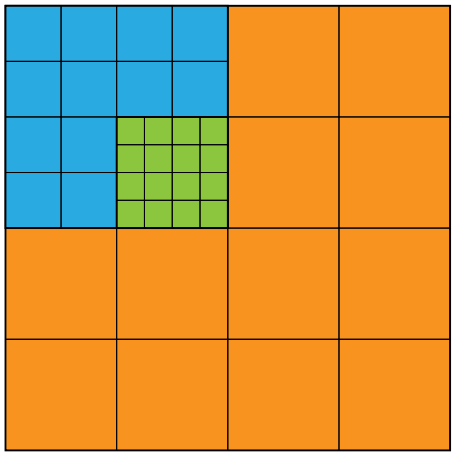

Figure 4: *Effects of importance-based prioritization of bricks. The lowest-resolution brick (orange) always has the highest importance value and is requested first. If the green brick has a higher importance than the blue one, it is requested earlier. Since the blue brick is at a higher level and overlaps the data of the already available green brick, the corresponding voxels in the 3D texture used for rendering must not be overwritten by the values of the blue brick. Note that the orange and blue voxels are upsampled to the original resolution of the volume (green brick).*

We handle these cases by storing all bricks in an octree-like data structure. If a brick is loaded and upsampled, we have to check recursively whether one or more of its eight child nodes already contain brick data. If this is the case, the respective areas are updated with the available high-resolution bricks prior to uploading the volume to the GPU. To improve the performance of this algorithm, we propagate the availability of high-resolution bricks to lower levels, so we do not have to traverse the whole tree. The importance-based prioritization of brick requests is of course only optional.

4.5. Client-side Rendering using WebGL

For the rendering, we implemented a basic 3D texture volume ray marching [EHK*06] in WebGL. As mentioned in Section 2, WebGL 2.0 [Khr13] supports 3D textures and complex fragment shaders that support, for example, dynamic loops. Only the front faces of the bounding box of the volume are rendered in order to initiate the volume rendering. The actual ray marching is performed in the fragment shader. Ray marching is performed by casting a ray into the volume from the eye position through the center of the current fragment and the volume is sampled along the ray. When using a 1D transfer function, the samples are then mapped to color through a texture lookup and composited using front to back compositing in order to get the final color value of the pixel. In case of maximum intesity projection, the maximum scalar value sampled from the volume along the ray is taken as the color for the pixel.

The user can switch between maximum intensity projection or a 1D transfer function. All transfer functions used for the screenshots in our paper were constructed using the Inviwo visualization framework [SSK*15] and stored locally on the client as *PNG* images. We use a `FileReader` object of the File API [W3C13] to read the transfer function data from the local disk and upload it to the GPU as a texture.

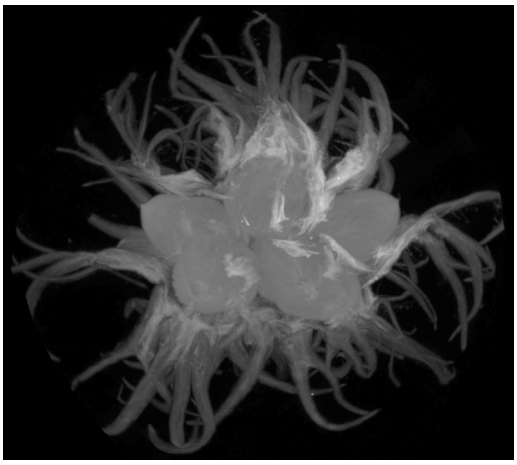

Figure 5: *Visualization of the hazelnuts volume data set using maximum intensity projection.*

As observable in Figure 1, the aggregation of densities over resolution levels cannot be properly performed via simple averaging. This has been investigated in detail by Sicat et al. [SKMH14]. They proposed to encode high-frequency data in the lower resolutions via probability distribution functions to obtain a scale-consistent rendering. Since this severely impacts the memory requirements, the volume is represented via a Gaussian mixture model (GMM) instead. The downside of this method is that the GMM fitting is extremely costly and the rendering requires a density histogram instead of discrete density values per voxel. Since the refinement of the volume converges rather quickly using our bricked data transfer, we decided to use simple averaging in our prototype. Actually, the artifacts arising from lower-level data make it easy for the user to spot locations where full-resolution data is still missing.

5. Results and Discussion

We measured transfer with and without decompression times on multiple machines and various data sets. To account for different network connections, we employed a workstation (custom built), a laptop (Microsoft Surface Book), and a typical smart phone (Samsung Galaxy S7). We used the hazelnuts data set ($512\times512\times512$ voxels, 128 MB; shown in Figure 5) for this test and varied the size of the transferred bricks from 512^3 (one brick) down to 64^3 (585 bricks over all levels of detail). The results and the hardware specifications of our test systems can be seen in Figure 6. Several interesting effects can be observed in the resulting graphs. One, for local wired network, compression does not improve the overall time. Although the Snappy compression results in roughly half the data set sizes, the time required for decompression roughly equals the time saved from shorter transmission. However, the improvements for compressed data are significant for slower network speeds (WiFi and LTE). Overall time for data transmission is reduced to less than 50% in all cases and decompression times are

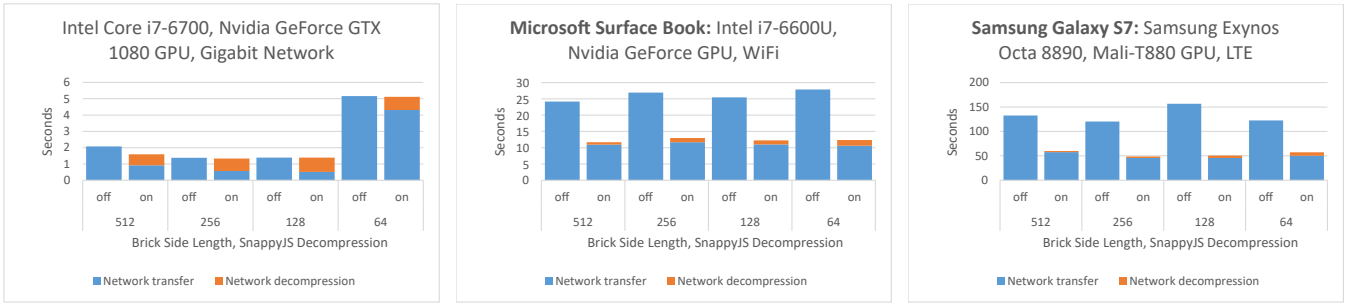

Figure 6: *Transfer and decompression times for the Hazelnut data set. The machines used for testing are the same as in Figure 7.*

negligible compared to transfer times. As expected, the additional data needed for the levels of detail is so low that it does not affect total transfer times significantly. That is, the total transfer time does not depend on the chosen bricking for the two wireless connections. The variance of the measurements in these cases can be attributed to constantly changing connection quality, which we also observed in the connection properties. This is due to taking the measurements in a real-world office environment. For example, in the WiFi case, we observed speeds between 39 Mbit/s and 117 Mbit/s (average was 72 Mbit/s). It is interesting to notice, however, that for the wired connection the single brick and the smallest bricks behave differently. We attribute the slightly higher transfer time for the single brick to the fact that only a single transfer is performed, while all other cases queue and execute several HTTP requests that will be served concurrently. This means that we could still improve the transfer of a whole volume even without levels of detail by bricking it all the same. We hypothesize that the sudden increase of transfer time for the smallest brick must stem from some inherent overhead in the HTTP requests that cannot be mitigated by the low latency of wired networking.

Since the main target of our method are networked connections with lower speed, not only the total transfer time, but also the per-brick transfer times are important. These directly influence the time a user has to wait until the visualization is available as well as the time it takes for each refinement of the available data. As mentioned above, the total transfer time does not vary much over brick sizes, so the trade-off for having a quick response with lower-resolution data is more than justified. Especially on a smartphone, where the absolute transfer times are very high, quick response times are essential. Based on our measurements, we therefore recommend the smallest brick size (64^3 voxels, compressed), which took about 20ms to transfer, in this case. For the WiFi connection, a brick size of 128^3 can be chosen to obtain comparable results (19ms).

Although our technique addresses a latency problem, still we needed to measure client-side rendering to check whether the volumes can be rendered at interactive frame rates. The rendering performance for the different client systems is shown in Figure 7. Note that we use a logarithmic scale for the ordinate. In addition to the hazelnuts, we also tested the engine ($256\times256\times128$ voxels, 8 MB; see Figure 8) and the aneurism (256^3 voxels, 16 MB; see Figure 1). For the frame rate measurements, we zoomed into the volume data sets until they reached maximum screen coverage while still being

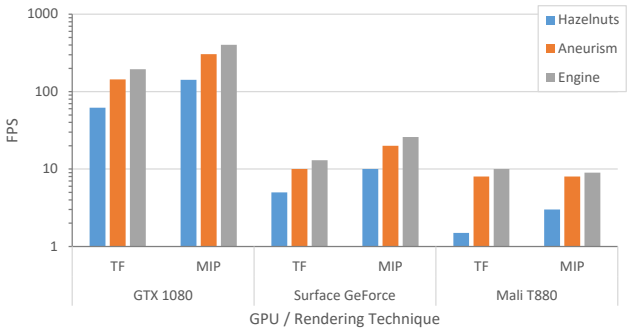

Figure 7: *Performance of our WebGL volume renderer for different data sets and rendering techniques: Maximum Intensity Projection (MIP) and volume ray marching using a Transfer Function (TF). A canvas size of 1024×768 was used. Frame rates were measured using the* stats.js *library (`https://github.com/mrdoob/stats.js/`).*

fully visible. Performance was measured after data set transmission was completed, although the background transfer and texture updates did not affect performance noticeably. The overall behavior of the different hardware is on par with expectations: while our simple volume ray marching reaches very high frame rates on the current desktop hardware, smaller (and lower-power) devices still struggle with this visualization technique. Especially the smartphone does not reach interactive frame rates for the largest data set. On both PC systems, MIP is twice as fast as using a transfer function. Interestingly, the performance of the smartphone does not change significantly if using a transfer function. This means that the dependent texture lookups have a much lower cost on the Mali architecture. The hazelnuts data set has such low performance that the performance difference is beyond measurement accuracy (1 vs. 2 FPS).

Our prioritization scheme described in Section 4.4 works well in conjunction with the transfer of levels of detail. Figure 1 shows the effect of the entropy-based prioritization of brick requests. In the intermediate snapshots, the central region of the data set is available in full resolution very early, since these bricks have the highest importance. As observable in the image, entropy is a very effective measure of importance for this data set, since the aneurism is available in high resolution first. Depending on the transfer function, this

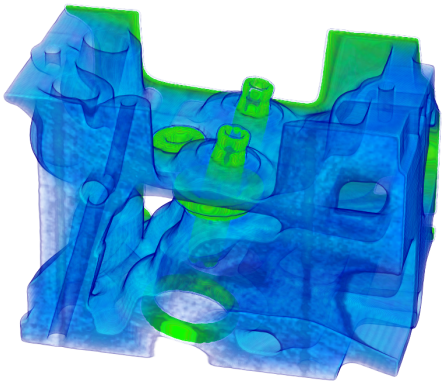

Figure 8: *Visualization of the engine volume data set using a transfer function.*

Figure 9: *Visualization of the flower data set with 1024^3 voxels using a transfer function. Our unoptimized WebGL 2.0 volume renderer reaches ∼2 fps on a Nvidia GeForce GTX 1080, which would be the expected frame rate for a similar desktop volume renderer.*

is not always the case. If it is known beforehand, the server could compute the entropy with respect to this classification. However, this is not applicable in an exploratory application case, where the user interactively designs the transfer function.

While the aneurism example shows that the entropy-based importance is an acceptable generic approach, more specific importance measures can be devised depending on the data sets and application domain. For known data sets, expert annotations could be used to derive importance. An alternative could be to use machine-learning-based classification, which is already routinely used in medical imaging [LKB*17], to identify important spots in the data. It would also be possible to use a weighted sum of different importance measures and allow an expert user to adjust the weighting in accordance with the task at hand.

6. Summary and Future Work

Our work has demonstrated the feasibility of visualizing volumetric data sets at interactive frame rates using multiple levels of detail and bricking techniques in the browser. This has been made possible by exploiting 3D texture support with immutable textures in WebGL 2.0 and modern web technologies like web workers that make it feasible to load the data asynchronously. Combining levels of detail, bricking, and importance-based streaming of compressed bricks to the client provides a base for visualization of large volumetric data that is only limited by texture memory of the GPU. By transferring individual bricks starting with the lowest level of resolution, this technique allows the user to start interacting with the volume without waiting for the entire volume to be transferred to the client. Moreover, this minimizes latency and network bandwidth requirements.

As future work, we plan to investigate multi-level volume rendering using multiple 3D textures to get rid of the CPU-based upsampling of bricks in the client. We would also like to investigate the use of *SharedArrayBuffer* between the web worker and the main thread to investigate if it can improve frame rates compared to the *Transferable Objects* that we have used in this implementation. Since the rendering performance of our current simple GPU-based volume ray marching is quite good for medium-sized data sets, there is room to improve the quality of the volume rendering. Possible directions are to add shading, use pre-integration and improved transfer functions (e.g., 2D textures), and add advanced lighting calculations like volumetric AO [JSYR14]. For large data sets like the flower shown in Figure 9, we need to investigate the applicability of common acceleration strategies like empty space skipping or early ray termination. Furthermore, we would like to extend our approach to support the visualization of time-varying or incrementally acquired volumetric data. We also want to experiment with different importance measures for the prioritization as discussed in section 5, for example by replacing the entropy calculation with application-specific methods.

Acknowledgments

This work was partially funded by German Academic Exchange Service (DAAD) and by German Research Foundation (DFG) as part of Collaborative Research Center SFB 716. We acknowledge the Computer-Assisted Paleoanthropology group and the Visualization and MultiMedia Lab at University of Zurich (UZH) for the acquisition of the µCT data sets (hazelnuts, flower).

References

[BHMF08] BEYER J., HADWIGER M., MÖLLER T., FRITZ L.: Smooth mixed-resolution gpu volume rendering. In *Proceedings of the EG/IEEE VGTC Conference on Point-Based Graphics* (2008), SPBG'08, pp. 163–170. doi:10.2312/VG/VG-PBG08/163-170. 2

[BHP15] BEYER J., HADWIGER M., PFISTER H.: State-of-the-art in gpu-based large-scale volume visualization. *Comput. Graph. Forum 34*, 8 (2015), 13–37. doi:10.1111/cgf.12605. 2

[CSK*11] CONGOTE J., SEGURA A., KABONGO L., MORENO A., POSADA J., RUIZ O.: Interactive visualization of volumetric data with webgl in real-time. In *Proceedings of the 16th International Conference*

on 3D Web Technology (New York, NY, USA, 2011), Web3D '11, ACM, pp. 137–146. doi:10.1145/2010425.2010449. 2

[EHK*06] ENGEL K., HADWIGER M., KNISS J. M., REZK-SALAMA C., WEISKOPF D.: *Real-time volume graphics*. A K Peters, 2006. 2, 5

[FSE13] FREY S., SADLO F., ERTL T.: Explorable Volumetric Depth Images from Raycasting. In *26th SIBGRAPI - Conference on Graphics, Patterns and Images (SIBGRAPI)* (2013), pp. 123–130. doi:10.1109/SIBGRAPI.2013.26. 2

[FSE15] FREY S., SADLO F., ERTL T.: Balanced Sampling and Compression for Remote Visualization. In *SIGGRAPH Asia 2015 Visualization in High Performance Computing* (New York, NY, USA, 2015), SA '15, ACM, pp. 1:1–1:4. doi:10.1145/2818517.2818529. 2

[GS04] GUTHE S., STRASSER W.: Advanced techniques for high-quality multi-resolution volume rendering. *Computers & Graphics 28*, 1 (2004), 51–58. 2

[GWGS02] GUTHE S., WAND M., GONSER J., STRASSER W.: Interactive rendering of large volume data sets. In *IEEE Visualization* (2002), pp. 53–60. doi:10.1109/VISUAL.2002.1183757. 2

[JC10] JÄNICKE H., CHEN M.: A Salience-based Quality Metric for Visualization. *Computer Graphics Forum* (2010). doi:10.1111/j.1467-8659.2009.01667.x. 4

[JSYR14] JÖNSSON D., SUNDÉN E., YNNERMAN A., ROPINSKI T.: A Survey of Volumetric Illumination Techniques for Interactive Volume Rendering. *Computer Graphics Forum 33*, 1 (2014), 27–51. doi:10.1111/cgf.12252. 7

[Khr13] KHRONOS: WebGL 2.0 Specification. http://www.khronos.org/registry/webgl/specs/latest/2.0/, 2013. [Online; accessed 2018/09/13]. 4, 5

[KV06] KIM Y., VARSHNEY A.: Saliency-guided enhancement for volume visualization. *IEEE Transactions on Visualization and Computer Graphics 12*, 5 (2006), 925–932. doi:10.1109/TVCG.2006.174. 4

[KZNP08] KLAJNSEK G., ZALIK B., NOVAK F., PAPA G.: A quadtree-based progressive lossless compression technique for volumetric data sets. *J. Inf. Sci. Eng. 24*, 4 (2008), 1187–1195. 2

[LCD13] LAVOUÉ G., CHEVALIER L., DUPONT F.: Streaming compressed 3d data on the web using javascript and webgl. In *Proceedings of the 18th International Conference on 3D Web Technology* (New York, NY, USA, 2013), Web3D '13, ACM, pp. 19–27. doi:10.1145/2466533.2466539. 2

[LKB*17] LITJENS G., KOOI T., BEJNORDI B. E., SETIO A. A. A., CIOMPI F., GHAFOORIAN M., VAN DER LAAK J. A., VAN GINNEKEN B., SÁNCHEZ C. I.: A survey on deep learning in medical image analysis. *Medical Image Analysis 42* (2017), 60 – 88. doi:https://doi.org/10.1016/j.media.2017.07.005. 7

[LWS*13] LIMPER M., WAGNER S., STEIN C., JUNG Y., STORK A.: Fast delivery of 3d web content: A case study. In *Proceedings of the 18th International Conference on 3D Web Technology* (2013), Web3D '13, ACM, pp. 11–17. doi:10.1145/2466533.2466536. 2

[MF12] MOVANIA M. M., FENG L.: High-performance volume rendering on the ubiquitous webgl platform. In *HPCC-ICESS* (2012), IEEE Computer Society, pp. 381–388. 2

[MHD*18] MATZEN L. E., HAASS M. J., DIVIS K. M., WANG Z., WILSON A. T.: Data visualization saliency model: A tool for evaluating abstract data visualizations. *IEEE Transactions on Visualization & Computer Graphics 24*, 1 (2018), 563–573. doi:10.1109/TVCG.2017.2743939. 4

[MKB*15] MWALONGO F., KRONE M., BECHER M., REINA G., ERTL T.: Remote visualization of dynamic molecular data using webgl. In *Proceedings of the 20th International Conference on 3D Web Technology* (2015), Web3D '15, ACM, pp. 115–122. doi:10.1145/2775292.2775307. 4

[MKB*16] MWALONGO F., KRONE M., BECHER M., REINA G., ERTL T.: Gpu-based remote visualization of dynamic molecular data on the web. *Graphical Models 88*, Supplement C (2016), 57 – 65. doi:https://doi.org/10.1016/j.gmod.2016.05.001. 4

[MKRE16] MWALONGO F., KRONE M., REINA G., ERTL T.: State-of-the-Art Report in Web-based Visualization. *Computer Graphics Forum 35*, 3 (2016), 553–575. doi:10.1111/cgf.12929. 2

[NJ12] NOGUERA J. M., JIMÉNEZ J.-R.: Visualization of very large 3d volumes on mobile devices and webgl. In *WSCG'2012 - 20-th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision'2012* (2012), pp. 105–112. 2

[OLv12] OBRUL D., LIU Y., ŽALIK B.: Progressive visualization of losslessly compressed dicom files over the internet. *J. Med. Syst. 36*, 3 (2012), 1927–1933. doi:10.1007/s10916-011-9652-y. 2

[PD15] PONCHIO F., DELLEPIANE M.: Fast decompression for web-based view-dependent 3d rendering. In *Proceedings of the 20th International Conference on 3D Web Technology* (2015), Web3D '15, ACM, pp. 199–207. doi:10.1145/2775292.2775308. 2

[SKMH14] SICAT R., KRÜGER J., MÖLLER T., HADWIGER M.: Sparse PDF Volumes for Consistent Multi-Resolution Volume Rendering. *IEEE Transactions on Visualization and Computer Graphics 20*, 12 (2014), 2417–2426. 5

[SSK*15] SUNDÉN E., STENETEG P., KOTTRAVEL S., JÖNSSON D., ENGLUND R., FALK M., ROPINSKI T.: Inviwo - An Extensible, Multi-Purpose Visualization Framework. Poster at IEEE Vis, 2015. 5

[W3C12] W3C: Web Workers: W3C Candidate Recommendation. http://www.w3.org/TR/workers/, 2012. [Online; accessed 2018/09/13]. 4

[W3C13] W3C: File API. http://www.w3.org/TR/FileAPI/, 2013. [Online; accessed 2018/09/13]. 5

[YSG15] YANG Y., SHARMA A., GIRIER A.: Volumetric texture data compression scheme for transmission. In *Proceedings of the 20th International Conference on 3D Web Technology* (2015), Web3D '15, ACM, pp. 65–68. doi:10.1145/2775292.2775323. 2