SLAMCast: Large-Scale, Real-Time 3D Reconstruction and Streaming for Immersive Multi-Client Live Telepresence – Supplemental Material

Patrick Stotko, Stefan Krumpen, Matthias B. Hullin, Michael Weinmann, and Reinhard Klein

In the scope of this supplemental material, we provide further implementation details of the components of our framework in order to facilitate its reproducibility. In this context, we also extend the discussion of the hash map serving as underlying data structure in our technique and provide an evaluation of our hashing approach with respect to approaches followed in recent literature.

1 Implementation Details

As mentioned in the accompanying paper, the main components of our framework are given by the reconstruction client, the server and the exploration client. In the following sections, we provide an in-depth discussion of implementation details for these components.

1.1 Reconstruction Client

To allow a transmission in the order of reconstruction, we attach each block added to the stream set with a unique counter. The transmission of the data is performed by a separate worker thread which extracts voxel block entries from the stream set up to the package size limit and collects the corresponding voxel data until the capturing process has ended and all data has been transmitted. Since InfiniTAM's streaming component allows blocks to be allocated in both volumes that results in a delayed internal streaming of the block, we first lookup and copy their data from the active visible volume and afterwards from the passive one. Already retrieved voxel data are overwritten as we prefer the passive version due to its higher robustness and lower susceptibility to noise. Note that this decision has no impact on the quality of the final model since at least one further update will be triggered once the active part is streamed out and merged with the passive one.

Although our system is developed to reconstruct static scenes, object interactions causing changes in the scene can still be handled to provide an updated virtual model. On demand, the user performing the acquisition can trigger a reset of the part of the scene that is currently visible for the sensor under its current pose. The list of visible blocks is then generated on the fly and all corresponding voxel data including the queued blocks in the stream set are erased. To maintain consistency across all users, the list is sent to the server which updates its model accordingly, and again forwards the message to all exploration clients which can then also reset the relevant parts of their models.

1.2 Server

The server component maintains both a copy of the transmitted TSDF voxel block model and a bandwidth-optimized representation based on Marching Cubes indices. Since only data from this optimized representation is transmitted to exploration clients, we store it in CPU memory whereas the received TSDF voxel block model resides in managed memory to allow fast processing with relaxed memory size restrictions. Recent NVIDIA GPUs and versions of the CUDA toolkit [6] support managed memory which not only allows direct accesses from both the CPU and GPU but also uses fast GPU-CPU paging to effectively relax the memory limit to the CPU rather than the GPU memory size.

When a new package from the reconstruction client arrives, the server first decompresses the data and integrates them into the TSDF voxel block model by allocating new blocks for not yet inserted parts and overriding already existing ones. In a second pass, we compute the MC voxel block data of the received blocks and their seven neighbors in negative direction, and add these blocks to each exploration client's stream hash set. In case a block is already inserted, the minimum of both unique counters is stored to make the update order-independent and to avoid holes in the client's model due to delayed streaming.

1.3 Exploration Client

For HMD-based visualization, the scene has to be rendered twice each frame, which results in a high memory bandwidth requirement for reading vertex data, and a high computational burden for the vertex shader. To cope with this, we store the position of each vertex relative to the bottom-left corner of the mesh block. Since we limit the number of voxel blocks inside a mesh block to 15^3, we can encode the position using one byte per dimension. This leads to 8 bytes per vertex, i.e. 3 bytes per position, 4 bytes per color which is stored in RGBA format, and 1 byte to align the structure. The client also stores all received voxel blocks, together with the indices of the triangles and points generated for them, in the CPU memory. This is needed for the case that a voxel block is received twice, and the structure inside the block changes or is deleted entirely, e.g. in case of partial resets of the model. If primitives need to be removed from a mesh block, we set the alpha value of the color to zero, which makes the primitives invisible and marks them as removed. However, as these invisible primitives also need to pass the vertex shader, we prune them by rebuilding the whole mesh block if the amount of them passes a certain threshold.

After a reconstruction thread has processed the update or rebuild for a mesh block, it is queued with all mesh blocks for which the geometry data needs to be uploaded to the GPU. When rendering a frame, we upload mesh data based on a predefined time window of 0.5 milliseconds. This time limit has proven to be sufficient to ensure proper uploading to the GPU with low latency while providing a smooth visual experience when rendering at 90Hz on a HMD.

To allow the visualization of fine texture details, e.g. required for reading texts or measurement instruments, we transmit the current RGB image from the reconstruction client upon request together with the estimated camera pose and the known intrinsic camera parameters. Using these data, the texture can be visualized in terms of a virtual 2D display or by a direct projection into the scene model. After receiving the data, the image is loaded into a texture which is passed to the fragment shader together with the camera parameters. The fragment shader first uses the camera parameters to project each fragment into the RGB image to obtain pixel coordinates and then checks if the coordinates lie within the texture boundaries. In this case, the texture is sampled and the resulting color is used as a fragment color. Otherwise, if the projected fragment lies outside the texture bounds, the fragment is assigned the voxel color of the 3D model and its brightness is reduced. This darkening of areas outside the projected image helps the user to

• *Patrick Stotko, Stefan Krumpen, Matthias B. Hullin, Michael Weinmann, and Reinhard Klein are with University of Bonn. E-mail: {stotko, krumpen, hullin, mw, rk}@cs.uni-bonn.de.*

focus on the relevant parts.

2 Hash Map and Set Data Structures

In this section, we want to elaborate on the requirements and design choices of our hash map data structure to realize a real-time remote collaboration system. Key to an interactive user experience is a fast and reliable data management that scales across multiple clients. Due to its constant amortized runtime complexity for insertion, retrieval, and removal and especially the successful integration into real-time 3D reconstruction GPU frameworks [1–3, 5], we have chosen spatial hashing techniques to manage our data workflow.

2.1 General Design

Since the hash map and set data structure is heavily used in the whole telepresence system, it must be highly reliable regarding insertion, retrieval, and removal to avoid artifacts such as holes in the exploration client's virtual model during transmission. In order to maintain key uniqueness to avoid duplicate and inconsistent voxel data, the retrieval operation must be performed – at least internally – inside insertion and removal such that those duplicates can be detected and correctly handled. Therefore, both concurrent insertion and retrieval, and concurrent removal and retrieval must be supported by the underlying data structure. Although only these two concurrency requirements are strictly required by our telepresence system, all major obstacles regarding thread synchronization to enable concurrent insertion and removal have also been implicitly resolved. Therefore, we also propose a minor extension to the internal stack data structure which enables full concurrency of insertion, removal and retrieval for the stack and directly propagates this property to the hash map and set data structure. We believe that our fully concurrent hash data structure is beneficial for various applications beyond to spatial hashing (see Sect. 2.3).

Considering the design choices to implement such a hash map and set data structure on the GPU, the actual implementation can be performed either on a thread level or a kernel level. We designed it on a thread level as this provides many important advantages compared to a kernel-leveled version. Each operation is hidden behind a function and enables very simple usage and a high re-usability across our system. Furthermore, data management can be easily done in any scenario and does not rely on additional synchronization steps. The most important aspect, however, is that function or thread-leveled operations are immune to synchronization errors from outside as the whole management is performed inside the function. While kernel-leveled versions can instead be hand-tuned to the particular scenario and may be faster, they are less generic and more susceptible to subtle errors. In Algorithm 1 (retrieval), Algorithm 2 (insertion), and Algorithm 3 (removal), we provide more implementation details regarding the proposed thread-safe hash operations.

Since a kernel-leveled implementation that also enforces the required guarantees is provided in the open-source implementation of the original voxel block hashing framework, which we consider an extension to the originally proposed technique by Nießner et al. [5], we will evaluate the implications in terms of runtime regarding this design choice.

2.2 Stack

One core element of our hash data structures is a stack structure that manages the available linked list positions. This stack structure is capable of adding and removing elements to and from its end in parallel. Although a simple implementation based on an atomic counter is sufficient to make the insertion and removal operations thread-safe, we extended it to support concurrent insertion and removal as this property directly propagates to the hash map and set and requires no further modification there. More details regarding these two operations are provided in Algorithm 4 (insertion), and Algorithm 5 (removal). Thus, we need to store the elements, indicators for each element determining whether the entry is occupied, locks for synchronization, the current size and the capacity of the container. Since the underlying arrays are all preallocated, the maximum size is limited.

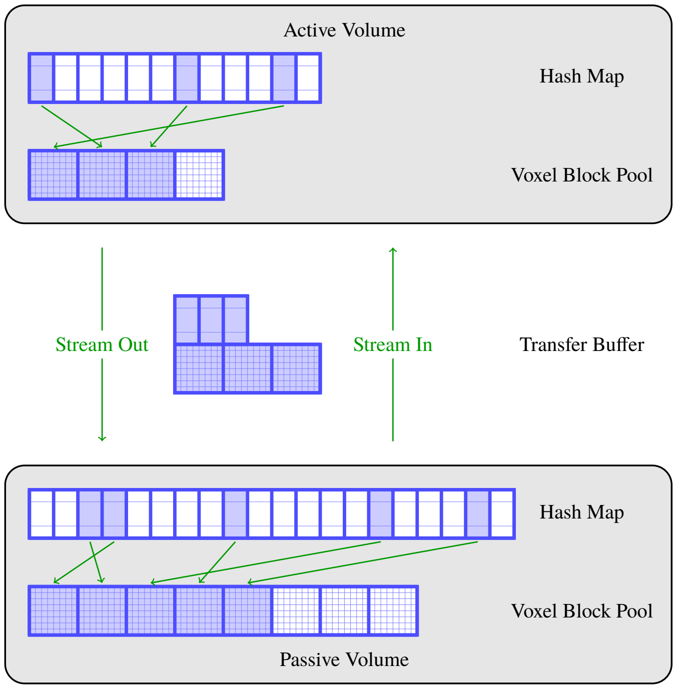


Fig. 1. Our novel two-hash-map streaming approach does require any knowledge about the hash map's implementation details and allows to adjust the size of each component independent of the others.

Insertion Similar to the simple counter-based approach, the insertion position is obtained by atomically incrementing the size. In particular, this operation reads the current value, increments and writes it back, and finally returns the old read value. This minimizes synchronization overhead since determining the position is decoupled from the actual insertion. Next, we try to lock the entry at the acquired position and check whether it is not yet occupied. Since concurrent insertion and removal is supported, the entry might be occupied indicating that another thread will remove it. In this case, we stop this attempt by unlocking and repeat this step until success. This guarantees that the operation order is correctly resolved. If the non-blocking locking operation succeeds and the entry is free, we write the given value into the entry, mark it as occupied and finally release the lock.

Removal Returning and erasing the last element of the stack is performed similar to insertion. First, we obtain the removal position by atomically decrementing the size and, in addition to insertion, decrementing the returned value again to obtain the correct index. Then we try to lock the corresponding entry, check whether it is occupied and get its stored value. In case it is not occupied, we retry this step until success to ensure a correct order of operations.

2.3 Applications

Besides the heavy usage within our telepresence system, our novel hash map data structure is beneficial for several applications such as the ones mentioned below.

Voxel Block Streaming Classical swapping techniques, that are part of several out-of-core scenarios, are used to relax memory size restrictions and enable applications that scale much better such as the voxel block hashing pipeline with its CPU-GPU streaming component. Whereas the original approach by Nießner et al. [5] uses a simple list on the CPU-site to manage the streamed data, the InfiniTAM system [3] reuses the GPU hash map to implicitly manage both the GPU and the CPU volume using an index-based mapping and special flags to indicate the streaming state. This introduced coupling between the hash map size and the CPU volume size has been relaxed by Mossel and Kröter [4], who added an auxiliary index array between these two to reduce the memory footprint. However, all these approaches either shift at least

Table 1. Time measurements of our system for various numbers of connected exploration clients. We compared the time required to stream the whole model with 512 blocks/request and 100Hz request rate to all exploration clients. Missing values refer to configurations where the GPU ran out of memory. The reconstruction speed, given by the time until the reconstruction client (RC) transmitted the model with 512 blocks/request and unlimited rate, serves as a lower bound. Across all scenes, the server can handle up to 2 and 6 clients at 5mm and 10mm voxel size respectively at reconstruction speed. When the work load exceeds this speed, the performances of the server scales linearly with the number of clients.

(a) Time measurements when using the NVIDIA GTX 780 with 3GB VRAM in the server component.

Dataset	Voxel Size [mm]	Time [min]											Model Size [# Voxel Blocks]
		1	2	3	4	5	6	7	8	9	10	RC	
heating_room	5	-	-	-	-	-	-	-	-	-	-	2:31	897×10^3
pool	5	-	-	-	-	-	-	-	-	-	-	1:08	637×10^3
fr1/desk2	5	0:27	0:28	0:33	0:39	0:44	0:49	0:54	0:59	1:04	1:09	0:22	134×10^3
fr1/room	5	-	-	-	-	-	-	-	-	-	-	0:56	467×10^3
heating_room	10	1:44	1:44	1:44	1:44	1:44	1:51	1:58	2:07	2:15	2:23	1:44	147×10^3
pool	10	0:50	0:50	0:50	0:50	0:50	0:50	0:53	0:56	1:00	1:04	0:50	104×10^3
fr1/desk2	10	0:19	0:19	0:19	0:19	0:19	0:19	0:20	0:21	0:23	0:24	0:18	23×10^3
fr1/room	10	0:41	0:41	0:41	0:41	0:42	0:43	0:46	0:48	0:49	0:51	0:41	86×10^3

(b) Time measurements when using the NVIDIA GTX 980 with 4GB VRAM in the server component.

Dataset	Voxel Size [mm]	Time [min]											Model Size [# Voxel Blocks]
		1	2	3	4	5	6	7	8	9	10	RC	
heating_room	5	-	-	-	-	-	-	-	-	-	-	2:31	897×10^3
pool	5	-	-	-	-	-	-	-	-	-	-	1:08	637×10^3
fr1/desk2	5	0:27	0:28	0:31	0:37	0:43	0:48	0:52	0:58	1:02	-	0:22	134×10^3
fr1/room	5	1:01	1:04	1:13	1:22	1:34	1:46	1:57	2:08	2:15	-	0:56	467×10^3
heating_room	10	1:44	1:44	1:44	1:44	1:44	1:50	1:58	2:06	2:13	-	1:44	147×10^3
pool	10	0:50	0:50	0:50	0:50	0:50	0:50	0:52	0:55	0:59	-	0:50	104×10^3
fr1/desk2	10	0:19	0:19	0:19	0:19	0:19	0:19	0:20	0:21	0:23	-	0:18	23×10^3
fr1/room	10	0:41	0:41	0:41	0:41	0:42	0:43	0:45	0:47	0:49	-	0:41	86×10^3

(c) Time measurements when using the NVIDIA GTX 1080 with 8GB VRAM in the server component.

Dataset	Voxel Size [mm]	Time [min]											Model Size [# Voxel Blocks]
		1	2	3	4	5	6	7	8	9	10	RC	
heating_room	5	2:40	2:40	2:58	3:19	3:41	4:00	4:26	4:44	5:14	5:33	2:31	897×10^3
pool	5	1:12	1:12	1:14	1:23	1:32	1:40	1:49	1:59	2:08	2:09	1:08	637×10^3
fr1/desk2	5	0:27	0:27	0:29	0:35	0:41	0:44	0:50	0:54	0:59	1:04	0:22	134×10^3
fr1/room	5	1:01	1:01	1:09	1:17	1:28	1:38	1:48	2:00	2:11	2:22	0:56	467×10^3
heating_room	10	1:44	1:44	1:44	1:44	1:44	1:47	1:56	2:04	2:12	2:20	1:44	147×10^3
pool	10	0:50	0:50	0:50	0:50	0:50	0:50	0:51	0:54	0:58	1:02	0:50	104×10^3
fr1/desk2	10	0:19	0:19	0:19	0:19	0:19	0:19	0:19	0:20	0:22	0:24	0:18	23×10^3
fr1/room	10	0:41	0:41	0:41	0:41	0:41	0:42	0:43	0:44	0:46	0:48	0:41	86×10^3

one part to the CPU-cite or effectively use only a single GPU hash map. In contrast, we use two hash maps: One for the active GPU and one for the passive CPU volume (see Fig. 1). Both volumes are implemented by pool data structures and both hash maps are allocated on the GPU so that all data management can be performed efficiently in parallel. Only the voxel block data buffer of the passive pool is stored in CPU memory whereas the remaining parts reside in GPU memory which has several advantages. First, there is no need for managing streaming-related logic in the raycasting or fusion step to differentiate between active and passive voxel blocks which completely decouples the streaming component from the rest of the pipeline. Furthermore, we can drop the requirement that the passive voxel block pool must have the same size as the hash map since no index hacks or streaming state management are needed anymore. This also avoids the auxiliary index buffer approach of Mossel and Kröter [4] and moves and unifies all voxel block data management to the hash map data structure. A direct consequence of our approach is that during streaming, a voxel block might be allocated in both the active and the passive volume when limiting the size of the transfer buffer [3]. Therefore, we simplify the two-step copy-and-merge streaming technique of InfiniTAM and consider merging as the only needed operation. In particular, our transfer buffers do not store indices to hash map entries but the actual inserted pointers to voxel blocks together with the voxel block data. This also decouples the streaming from the actual hash map implementation.

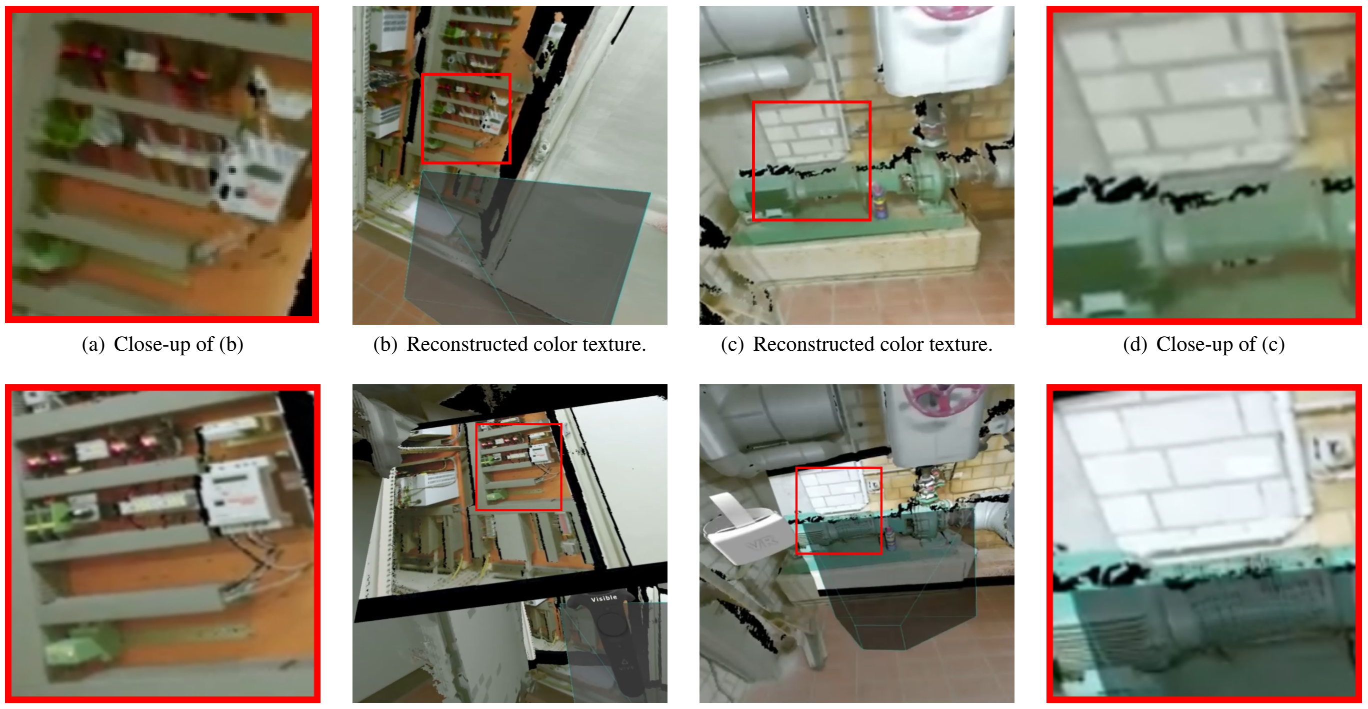
Beyond Spatial Hashing While we follow the specific hash function definition for 3D spatial hashing used by Nießner et al. [5] and Kähler et al. [3], our data structure has no limitations regarding key-value pair size or exchangeability of the actual hash function. Thus, it can be applied to various problems beyond computer graphics that need proper and reliable on-the-fly data management of millions of entries on the GPU with enforced key uniqueness preservation. This includes file indexing in data centers or large databases with non-standard data in economics and other fields.

3 Evaluation

Beyond the evaluation in the main paper, we provide a further assessment of our novel remote collaboration system regarding the involved projective texture mapping approach, the scalability of our server component, as well as the runtime performance of our novel hash map and set data structure. If not explicitly stated, we used the same parameter set as described in the main paper for each experiment.

3.1 System Scalability

In order to evaluate the scalability of our server component, we measured the time until all connected benchmark clients received the whole



(a) Close-up of (b) (b) Reconstructed color texture. (c) Reconstructed color texture. (d) Close-up of (c)

(e) Close-up of (f) (f) Projection on virtual 2D display. (g) Projection on virtual 3D model. (h) Close-up of (g)

Fig. 2. Visual comparison of our projective texture mapping approach for the *heating_room* dataset: In comparison to the reconstructed color texture (top row), projecting the current input image on a virtual display device (f) as well as the reconstructed virtual model (g) improves remote collaboration by allowing users to inspect fine details in the scene, which are crucial for decision making. Note that the brightness of the reconstructed texture is decreased during texture mapping to emphasize the projection region. Furthermore, the brightness of the projected texture might vary according to the camera's automatically chosen exposure time.

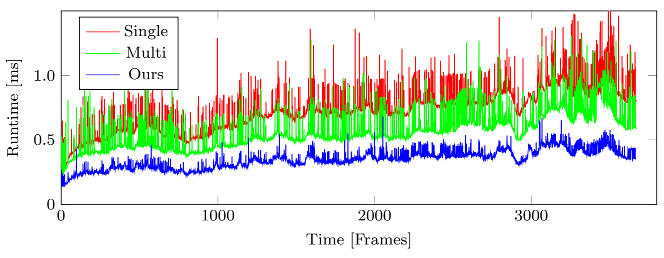
reconstructed model from the server where the latter is equipped with different GPUs. Throughout the experiment, the model is reconstructed on the computer equipped with the NVIDIA GTX TITAN X, then streamed to the server (second computer) and further to up to 10 benchmark clients (all on the third computer). The benchmark client is started at the same time as to the reconstruction client, requests voxel blocks with a fixed predefined frame rate of 100Hz, and directly discards the received data to avoid overheads. Since all computers are within a local network and the benchmark clients do not perform any processing, we observed no measurement overhead compared to a setup with up to 12 computers where at most one benchmark client is executed on each of them. For a comparison with weaker hardware, we equipped the server with either a NVIDIA GTX 780 with 3GB VRAM, a NVIDIA GTX 980 with 4GB VRAM, or a NVIDIA GTX 1080 with 8GB VRAM for our experiments. While a voxel block pool size of 2^{20} blocks is used for the NVIDIA GTX 1080 like in all other experiments, we lowered this size to 2^{19} blocks (NVIDIA GTX 980) and 2^{18} blocks (NVIDIA GTX 780) to make the pool fit into the VRAM of the respective GPUs. Note that this also affected the excess list size of the corresponding hash map. Furthermore, we reduced the number of buckets to 2^{21} buckets for both the NVIDIA GTX 980 and NVIDIA GTX 780 to account for the limited VRAM. The results of our experiment are shown in Table 1.

Across all scenes and GPUs, we observed that our server implementation is capable of handling up to 2 clients at 5mm and up to 6 clients at 10mm voxel size at the speed of the reconstruction client. Further increasing the number of clients in the benchmark affects the server performance in a linear manner. For a voxel size of 5mm, the time until 10 benchmark clients received the whole model is more than twice as high as the reconstruction times. Since parts of the model are rescanned during reconstruction, the number of streamed voxel blocks is larger than the model size which results in an enormous amount of data that need to be processed and transmitted for each connected benchmark client. Thus, the overall performance is mostly determined by the bandwidth of both the CPU and GPU memory which becomes apparent when the server is equipped with weaker hardware. Although the NVIDIA GTX 1080 has a significantly higher computational power, the time difference to a NVIDIA GTX 780 is only within a few seconds. However, the major limitation of the weaker GPUs is the significantly smaller memory size that prohibits the reconstruction of large models with a fine resolution, i.e. 5mm voxel size (see Table 1(a) and Table 1(b)). Note that this also limits the number of exploration/benchmark clients since each of them is assigned a stream hash set. For the NVIDIA GTX 980 and the corresponding parameters as described above, only up to 9 benchmark clients can be handled by the server until the memory limit is reached. Since the size of the voxel block pool is further reduced by a factor of two for the NVIDIA GTX 780, more than 9 clients can connect to the server, however, at the cost of limiting the reconstructed models to even smaller sizes.

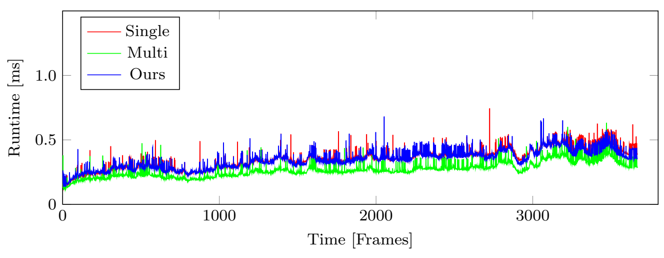
In addition to the evaluation regarding the scalability of the server component, we also analyzed the GPU requirements at the exploration client's side. Since rendering the reconstructed scene does not involve complex shading operations, the computational burden due to the number of triangles within the scene outweighs the computational costs due to the resolution of the HMD. To meet the native frame rate of 90Hz for both eyes of the used HMD, we need to render 180 images per second which results in a huge load on the vertex shader even for a relatively low triangle count in the scene. Therefore, the rasterization of the geometry for VR applications imposes a much higher computational burden than in the usual rendering scenario where only 60Hz on a normal 2D screen are required. We observed that increasing the distance in which the next level of detail (LoD) is rendered, or disabling the LoD completely, which increases the number of processed triangles per frame, has a greater impact on the frame rate than increasing the rendering resolution beyond the display resolution of the HMD. Due to this observations, we used the NVIDIA GTX 1080 GPU for the exploration client and did not consider using less powerful GPUs.

Table 2. Runtime measurements of our hash map data structure for various scenes during reconstruction. We compared the mean (and standard deviation) runtime in milliseconds to similar techniques that either operate on a thread level and allow failures, or on a kernel level with guarantees. While our data structure guarantees successful insertion, it is much faster than kernel-based approaches and only slightly slower than unsafe thread-leveled techniques.

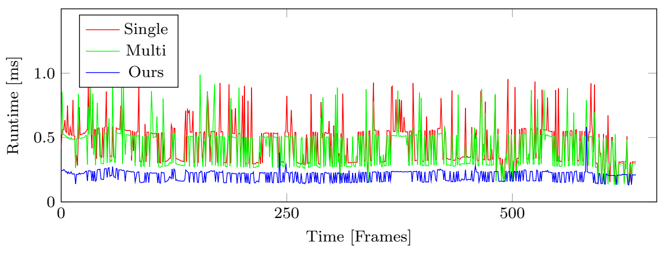
Dataset	Voxel Size [mm]	Thread Level		Kernel Level			Model Size [# Voxel Blocks]
		Multi Entry	Single Entry	Multi Entry	Single Entry	Ours	
heating_room	5	0.27 (0.06)	0.36 (0.08)	0.58 (0.15)	0.74 (0.18)	0.35 (0.07)	897×10^3
pool	5	0.28 (0.06)	0.37 (0.08)	0.63 (0.16)	0.80 (0.20)	0.37 (0.08)	637×10^3
fr1/desk2	5	0.25 (0.04)	0.29 (0.04)	0.57 (0.16)	0.65 (0.17)	0.29 (0.05)	134×10^3
fr1/room	5	0.29 (0.06)	0.37 (0.08)	0.67 (0.18)	0.82 (0.22)	0.37 (0.08)	467×10^3
heating_room	10	0.14 (0.04)	0.16 (0.04)	0.32 (0.14)	0.37 (0.14)	0.16 (0.04)	147×10^3
pool	10	0.15 (0.04)	0.18 (0.05)	0.37 (0.14)	0.43 (0.15)	0.18 (0.05)	104×10^3
fr1/desk2	10	0.18 (0.04)	0.19 (0.04)	0.42 (0.16)	0.45 (0.16)	0.20 (0.04)	23×10^3
fr1/room	10	0.19 (0.04)	0.22 (0.04)	0.46 (0.15)	0.51 (0.15)	0.22 (0.04)	86×10^3



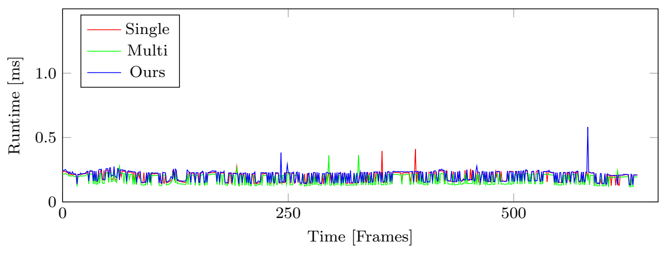
(a) Kernel-leveled Data Structures.



(b) Thread-leveled Data Structures.



(c) Kernel-leveled Data Structures.



(d) Thread-leveled Data Structures.

Fig. 3. Runtime comparison between hash data structures for the *heating_room* scene at 5mm voxel resolution (see (a) and (b)) and for the *fr1/desk2* scene at 10mm voxel resolution (see (c) and (d)).

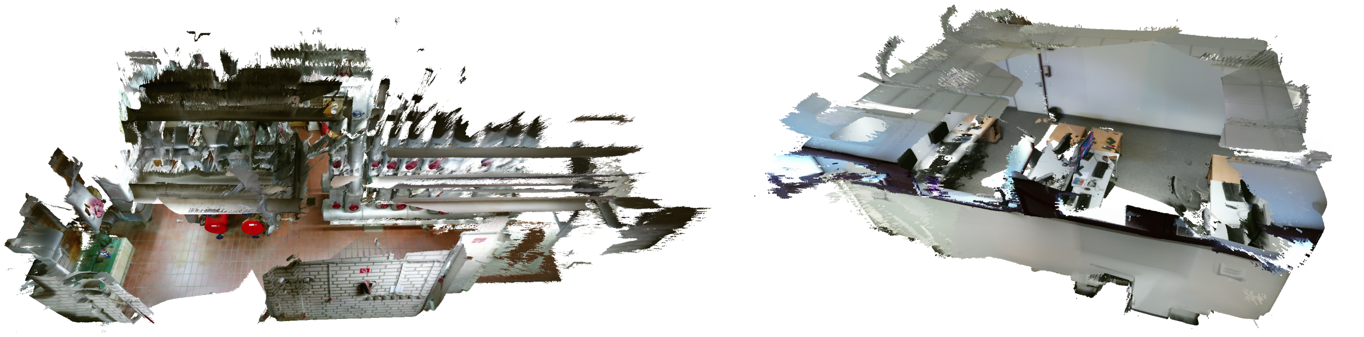
3.2 Subjective User Experience

To evaluate the user experience, we conducted a study where 15 participants with ages between 25 and 57 years were immersed into the scene as remote users. The participants were assigned the roles of maintenance and safety experts and asked to fulfill related tasks. Depending on the assigned role, they had to navigate through the scene to the respective objects of interest and had to interact with the scene in terms of performing measurements such as the size of objects, the heights of objects above the ground as well as door heights and widths or reading measurement devices. In general, most of the users got familiar with the scene exploration via the teleport-operation to reach more distant scene parts and physical movements to inspect the nearby environment within about 30 seconds. Navigating to individual objects upon request was easily performed by all of the participants. We observed that the participants took great care to avoid collisions with scene contents and, therefore, navigated around obstacles such as chairs and tables in office environments or machines in the heating room scenario. In the heating room scenario, all participants even crouched below tubes mounted at the ceiling to access respectively interesting areas. Scene interaction was measured in terms of user feedback regarding the easiness of the measurement as well as the accuracy of the measured lengths. Most of the users enjoyed the task of taking measurements and after taking a few (about 5 - 10) measurements, they reached an accuracy of about 1cm to 2cm when measuring lengths between 10cm and 250cm. This accuracy is influenced by both subjective factors of the user interaction and the quality of the underlying model. The latter is affected by the quality of the reconstruction that is determined by the voxel resolution and sensor noise, the accuracy of the camera tracking as well as the compression in terms of Marching Cubes voxels.

Regarding the model quality, the participants mostly rated the texture resolution to be acceptable and appreciated the possibility of high-resolution texture projection of the live-image of the camera onto a virtual 2D display device or the surface directly. Thereby, the participants were able to report e.g. the states of measurement devices and were able to even recognize smaller objects (such as the WD-40 spray as shown in the supplemental video). In addition, all users rated for a high relevance of perceiving other remote users as well as the view frustum of the RGB-D camera used for scanning within the same scene. Thereby, the communication between the different users via headsets was improved regarding the collaboration of the remote experts, e.g. when analyzing the same scene parts, and regarding the collaboration of the remote experts with the local user to guide the capturing process.

3.3 Projective Texture Mapping

In the following, we provide a qualitative evaluation of our projective texture mapping approach. For this purpose, we demonstrate the quality of the texture improvement over the typical accuracy of voxel block hashing approaches in Fig. 2. We also measured the bandwidth implications for image requests and observed only a minimal impact since the image data are efficiently compressed before transmission and such



(a) *heating_room*. (b) *pool*.

Fig. 4. Reconstructed models of our datasets.

requests only occur rarely on the remote user's demand.

Due to the limited voxel resolution, fine structures and object details are blurred and, therefore, hardly visible in the reconstructed and transmitted color texture (see top row of Fig. 2). Small alignment errors during 3D reconstruction also affect the texture quality. In contrast, the requested current RGB image projected on a virtual 2D display (see Fig. 2(f)) has a much higher resolution and contains fine details. Projecting the image onto the transmitted surface geometry leads to similar results where object textures are much sharper (see Fig. 2(d) and Fig. 2(h)). Since the projected texture represents the perspective of the camera held by the local user, color values around object boundaries might be incorrectly projected onto adjacent objects due to the imperfectly estimated camera pose. However, the users reported that all the important details are clearly visible and such small errors as well as the varying exposure time of the requested RGB image only slightly affected the visual experience.

3.4 Hashing Performance

To verify the efficiency of our hash data structure, we provide a detailed performance evaluation. Similar to the bandwidth analysis provided in the paper, we used the scenes *heating_room* and *pool* (see Fig. 4) recorded with the Kinect v2 and two further datasets captured with the Kinect v1 [7]. For the purpose of a fair comparison, we reimplemented and evaluated thread-leveled versions of the data structures following the description by Nießner et al. [5], i.e. a multi-entry hash map which resolves collisions through a neighborhood search, and by Kähler et al. [3], i.e. a single-entry hash map with a stack data structure implemented via a simple atomic counter. While both of these approaches do not provide strong guarantees beneficial in the context of a remote collaboration system, they are still suitable for high-frame-rate 3D reconstruction. Since it is also possible to ensure successful insertion and removal by looping over the kernel and testing whether the size has changed, we further compare our thread-leveled data structure with kernel-leveled multi-entry and single-entry versions as e.g. included in the extended voxel block hashing framework accompanying the work by Nießner et al. [5]. We used a bucket size of 2^{20} buckets for all hash maps as well as two entries per bucket for the multi-entry version and excess list sizes of 2^{19} elements for the single-entry and our version. In order to evaluate the insertion performance of each approach and minimize side effects by other parts of the voxel block hashing pipeline, we measured the runtime of the voxel block allocation step. The results of this comparison are shown in Table 2 and Fig. 3.

Across all scenes, we observed that the kernel-leveled approaches are significantly slower, i.e. exhibit runtimes of more than a factor of 2, in comparison to their thread-leveled counterparts. This is a result of the need for at least two calls of the kernel where in the second run all insertion failures are corrected. Furthermore, obtaining the hash map size involves additional costly and inefficient memory copies from GPU to CPU memory. Note that kernel performance optimizations could reduce the gap but this requires careful manual hand-tuning. When comparing single and multi-entry approaches, we observe that the multi-entry technique is approximately 20% faster since first-order collisions are directly handled and no additional stack data structure is required. However, this comes at the cost of an increased memory footprint where most secondary entries remain empty and unused. Our data structure's performance is approximately on par with the other thread-leveled approaches but provides the reliability of their kernel-leveled counterparts without further hand-tuning.

We also observed that the runtime scales almost linearly with the voxel size since the allocation step traverses over all visible blocks in the view frustum. For datasets captured with the Kinect v1, the difference between 5mm and 10mm resolution is less than for the datasets recorded with the Kinect v2. This is mainly caused by the lower field of view and the higher image resolution of the Kinect v1 sensor. Thus, the impact of the volume traversal is higher at 10mm since more pixels try to the insert a single block. The hash map efficiently handles this by immediately returning if the block has been already inserted, thus keeping the cost in such cases as low as possible. Over the course of time, the runtimes of all hash map data structures remain constant (see Fig. 3(c) and Fig. 3(d)). At higher load factors where more collisions are observed and also over the course of time (see Fig. 3(a) and Fig. 3(b)), we observed slightly increasing values which are caused by traversing colliding entries in the linked lists. However, this impact is rather small compared to the total runtime and underlines the constant amortized asymptotic complexity of hash data structures.

References

[1] A. Dai, M. Nießner, M. Zollhöfer, S. Izadi, and C. Theobalt. BundleFusion: Real-time Globally Consistent 3D Reconstruction using On-the-fly Surface Reintegration. *ACM Trans. Graph.*, 36(3):24, 2017.

[2] O. Kähler, V. A. Prisacariu, and D. W. Murray. Real-Time Large-Scale Dense 3D Reconstruction with Loop Closure. In *European Conference on Computer Vision*, pp. 500–516, 2016.

[3] O. Kähler, V. A. Prisacariu, C. Y. Ren, X. Sun, P. Torr, and D. Murray. Very High Frame Rate Volumetric Integration of Depth Images on Mobile Devices. *IEEE Trans. on Visualization and Computer Graphics*, 21(11):1241–1250, 2015.

[4] A. Mossel and M. Kröter. Streaming and exploration of dynamically changing dense 3d reconstructions in immersive virtual reality. In *Proc. of IEEE Int. Symp. on Mixed and Augmented Reality*, pp. 43–48, 2016.

[5] M. Nießner, M. Zollhöfer, S. Izadi, and M. Stamminger. Real-time 3D Reconstruction at Scale Using Voxel Hashing. *ACM Trans. Graph.*, 32(6):169:1–169:11, 2013.

[6] NVIDIA Corporation. CUDA Toolkit Documentation. `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`, 2016. Accessed: 2019-01-29.

[7] J. Sturm, N. Engelhard, F. Endres, W. Burgard, and D. Cremers. A Benchmark for the Evaluation of RGB-D SLAM Systems. In *Proc. of the Int. Conf. on Intelligent Robot Systems*, 2012.

Algorithm 1 Our Hash Map and Set Retrieval Function

Input: Key of the requested hash entry
Output: Pointer to corresponding hash entry if found, nullptr otherwise

```
function FIND(key)
    index $\leftarrow$ bucket(key)                                                          $\triangleright$ Check bucket
    if occupancyFlags[index] $=$ true and getKey(values[index]) $=$ key then
        return values $+$ index
    end if
    while offsets[index] $\neq 0$ do
        index $\leftarrow$ index $+$ offsets[index]                                          $\triangleright$ Check linked list
        if occupancyFlags[index] $=$ true and getKey(values[index]) $=$ key then
            return values $+$ index
        end if
    end while
    return nullptr
end function
```

Algorithm 2 Our Hash Map and Set Insertion Function

Input: Key or key-value-pair to insert
Output: Pointer to inserted hash entry if inserted *by this thread*, nullptr otherwise

```
function INSERT(value)
    result $\leftarrow$ nullptr
    while find(getKey(value)) $=$ nullptr do
        result $\leftarrow$ tryInsert(value)
    end while
    return result
end function

function TRYINSERT(value)
    result $\leftarrow$ nullptr
    if find(getKey(value)) $\neq$ nullptr then                              $\triangleright$ Already inserted, so return
        return result
    end if
    index $\leftarrow$ bucket(getKey(value))
    if occupancyFlags[index] $=$ false then                                 $\triangleright$ Bucket is free: Insert there
        if locks.tryLock(index) $=$ true then
            if find(getKey(value)) $=$ nullptr and occupancyFlags[index] $=$ false then   $\triangleright$ Double check insertion and occupancy status
                values[index] $\leftarrow$ value
                atomicAdd(count, 1)
                occupancyFlags[index] $\leftarrow$ true
                result $\leftarrow$ values $+$ index
            end if
            locks.unlock(index)
        end if
    else                                                                     $\triangleright$ Bucket is not free: Insert into linked list
        linkedListEnd $\leftarrow$ findLinkedListEnd(index)
        if locks.tryLock(linkedListEnd) $=$ true then
            if find(getKey(value)) $=$ nullptr and offsets[linkedListEnd] $= 0$ then   $\triangleright$ Double check insertion and offset status
                newLinkedListEnd $\leftarrow$ excessList.pop()
                values[newLinkedListEnd] $\leftarrow$ value
                offsets[newLinkedListEnd] $\leftarrow 0$                    $\triangleright$ *Must be reset to zero:* Skipped in tryErase()
                offsets[linkedListEnd] $\leftarrow$ newLinkedListEnd $-$ linkedListEnd
                atomicAdd(count, 1)
                occupancyFlags[newLinkedListEnd] $\leftarrow$ true
                result $\leftarrow$ values $+$ newLinkedListEnd
            end if
            locks.unlock(linkedListEnd)
        end if
    end if
    return result
end function
```

Algorithm 3 Our Hash Map and Set Removal Function

Input: Key of the to-be-erased hash entry
Output: True if the entry has been erased *by this thread*, false otherwise

```
function ERASE(key)
    result ← false
    while find(key) ≠ nullptr do
        result ← tryErase(key)
    end while
    return result
end function

function TRYERASE(key)
    result ← false
    pointer ← find(key)
    index ← pointer − values
    if pointer = nullptr then                                         ▷ Already erased, so return
        return result
    end if
    if index = bucket(key) then                                       ▷ Entry inside bucket
        if locks.tryLock(index) = true then
            if find(key) = pointer then                               ▷ Double check removal status
                occupancyFlags[index] ← false
                atomicSub(count, 1)
                values[index] ← default or hole value
                result ← true
            end if
            locks.unlock(index)
        end if
    else                                                              ▷ Entry inside linked list
        previousIndex ← findPreviousEntry(index)
        if locks.tryLock(index, previousIndex) = true then
            if find(key) = pointer and findPreviousEntry(index) = previousIndex then   ▷ Double check removal and linked list status
                if offsets[index] ≠ 0 then
                    offsets[previousIndex] ← offsets[previousIndex] + offsets[index]
                else
                    offsets[previousIndex] ← 0
                end if
                occupancyFlags[index] ← false
                atomicSub(count, 1)
                values[index] ← default or hole value         ▷ Do not reset offset[index]: Avoids synchronization in find()
                excessList.push(index)
                result ← true
            end if
            locks.unlock(index)
            locks.unlock(previousIndex)
        end if
    end if
    return result
end function
```

Algorithm 4 Our Stack Push Function

Input: Value to push at the end of the stack

```
function PUSH(value)
    pushed ← false
    index ← atomicAdd(size, 1)
    while not pushed do
        if locks.tryLock(index) = true then
            if occupancyFlags[index] = false then     ▷ Entry is free: Insert now, otherwise resolve push/pop order by reattempting
                values[index] ← value
                occupancyFlags[index] ← true
                pushed ← true
            end if
            locks.unlock(index)
        end if
    end while
end function
```

Algorithm 5 Our Stack Pop Function

Output: Extracted value from the end of the stack

```
1:  function POP
2:      result ← default or hole value
3:      popped ← false
4:      index ← atomicSub(size, 1) - 1
5:      while not popped do
6:          if locks.tryLock(index) = true then
7:              if occupancyFlags[index] = true then    ▷ Entry is not free: Extract now, otherwise resolve push/pop order by reattempting
8:                  occupancyFlags[index] ← false
9:                  result ← values[index]
10:                 values[index] ← default or hole value
11:                 popped ← true
12:             end if
13:             locks.unlock(index)
14:         end if
15:     end while
16:     return result
17: end function
```