AN END-TO-END SYSTEM FOR REAL-TIME DYNAMIC POINT CLOUD VISUALIZATION

Hansjörg Hofer, Florian Seitner

emotion3D GmbH
Gartengasse 21/3
A-1050 Vienna, Austria

Margrit Gelautz

Interactive Media Systems Group
Vienna University of Technology
Favoritenstr. 9-11/193-06
A-1040 Vienna, Austria

ABSTRACT

The growing availability of RGB-D data, as delivered by current depth sensing devices, forms the basis for a variety of mixed reality (MR) applications, in which real and synthetic scene content is combined for interaction in real-time. The processing of dynamic point clouds with possible fast and unconstrained movement poses special challenges to the surface reconstruction and rendering algorithms. We propose an end-to-end system for dynamic point cloud visualization from RGB-D input data that takes advantage of the *Unity3D* game engine for efficient state-of-the-art rendering and platform-independence. We discuss specific requirements and key components of the overall system along with selected aspects of its implementation. Our experimental evaluation demonstrates that high-quality and versatile visualization results can be obtained for datasets of up to 5 million points in real-time.

***Index Terms*—** point cloud, rendering, real-time, reconstruction, unity3d, visualization, rgb-d

1. INTRODUCTION

The increasing popularity of virtual reality (VR), augmented reality (AR) and mixed reality (MR) applications in commercial and industrial environments, due to the emergence of powerful mobile devices and novel hardware, encourages further advancements in computer vision and computer graphic technologies. Such technologies allow to overcome the boundaries between reality and virtuality. We can observe that the popularity of semi-automated 3D reconstruction of real-world objects and scenes is steadily increasing [6] and finding its way into commercial products. Modern approaches try to minimize the manual labor of digital content creation, while increasing the visual quality of the 3D assets. Real 3D content - for example, reconstructed by means of photogrammetry [20] - is often more authentic than hand-made replications and forms the basis for creating photo-realistic environments rendered in real-time [6].

As opposed to static environments, dynamic scenes are more challenging to process in an efficient and automated manner, due to potential unconstrained topological changes and fast frame-to-frame movements. A seamless integration of concurrent reconstruction and visualization of live captured 3D data would be highly beneficial for a variety of MR applications such as remote virtual collaboration or broadcasting of live events captured in 3D. This becomes even more relevant with the recent advent of depth sensors for mobile devices [11], which open a completely new area of possibilities. For example, mobile telepresence systems with a live 3D projection of the dialogue partner, making it possible to talk to a person as if he or she was physically present.

In order to support such promising MR applications, we propose an efficient visualization system that takes dynamic RGB-D data, as delivered by modern depth sensing devices, as input to generate novel and augmented views of the captured original scenes in real-time. The main contributions of this end-to-end visualization pipeline are (1) platform-independent system architecture taking advantage of common GPU features, (2) optimized reconstruction and visualization of arbitrary dynamic scenes with no frame-to-frame dependencies and (3) the incorporation of robust algorithms allowing to handle fast movements and potentially large topological changes. After a review of related work in Section 2, we discuss the principal components of the processing pipeline and implemented algorithms in Section 3. The visual quality of the rendering results and a runtime analysis demonstrating our implementation's real-time capability are presented afterwards in Section 4.

2. RELATED WORK

Real-time 3D surface reconstruction and visualization from RGB and RGB-D camera setups is a steadily growing field of research. Related techniques can be categorized in two main branches. The first branch relies on point cloud meshing, where the incoming point based data is used to construct and continuously refine a polygonal geometry mesh. Such approaches only became feasible, in real-time, with the advent of GPGPU capabilities of current hardware. The second major branch is point based surface visualization. These algorithms enrich the dynamic positional data with surface

978-1-5386-7590-8/18/$31.00

properties, creating *surfel* datasets [18]. Point clouds with additional surface information are then visualized using special rendering techniques, beyond the classical rendering pipeline [3, 4, 5, 7, 24].

Real-time meshing algorithms for dynamic point clouds used in [17, 21] are based on the construction and triangulation of implicit surfaces. This process can be optimized for parallel processing on GPGPU kernels. Nevertheless, the processing of higher volumetric resolutions often already exceeds the real-time frame-budget. To overcome these performance limitations, researchers have introduced various assumptions, pre-trained models or movement constraints [8, 12, 16, 23], to reduce the required processing time without limiting the reconstruction quality. Despite the astounding results and the performance of these algorithms, they often introduce a new level of complexity to the initial challenge, which also generally limits the possible area of applications, while still having high hardware requirements. Therefore, we pursued a different approach for our point cloud visualization.

Contrary to the reconstruction via meshing and the resultant limitations, Preiner et al. [19] present a surface reconstruction algorithm by estimating per-point normals in screen space. Their approach determines the surface normal of each visible point by calculating the supporting plane of its k nearest neighbors (kNN). The algorithm exploits the screen space representation and takes advantage of the highly parallelized fragment shader stage on the GPU. Their processing pipeline works on a per-frame basis and is therefore not affected by rapid movements or temporal tracking errors. While the approach of [19] does not require any temproal coherence, it has the additional advantage of exploiting standard GPU shader stages which are consistently supported by current graphics hardware.

After successful reconstruction of the surface normals, the oriented points can be rendered using surface splatting [24], similar to [14]. Surface splatting was first introduced by Zwicker et al. [24] in 2001. Later adaptations for modern lighting models and GPUs were proposed by Botsch et al. [3, 4] a few years later. But even today this technique is still an efficient solution which can be integrated into modern state-of-the-art deferred shading pipelines and provides stunning high-quality results. Point cloud splatting is also used in the recent work of Bonatto et al. [2] to visualize very large environmental scans in VR, where high frame rates are crucial for a satisfying experience. Contrary to our system, [2] focus on static content and can therefore precalculate the surface normals in an offline processing step.

3. IMPLEMENTATION

We present an end-to-end 3D visualization system for RGB-D camera setups. The system is fully integrated in the well-known *Unity3D* game engine [22] and benefits from its state-of-the-art rendering engine and post-processing effects. Furthermore, the engine simplifies the utilization of the point cloud visualization and enables its usage in a variety of applications and platforms, such as different desktop and mobile operating systems (Section 3.2).

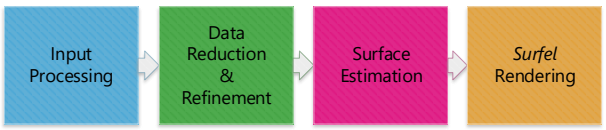


Fig. 1. Flowchart of the visualization pipeline stages and their execution order.

3.1. System Overview

Our processing pipeline consists of four different stages, as shown in the flow chart in Figure 1. First, the RGB-D camera streams (color and depth) are decoded and mapped into a common reference coordinate system, if necessary. Aside from sensor devices, a multitude of other point cloud inputs (e.g. static/animated geometry files, RGB-D frame sequences, remote network streams) are supported and handled accordingly (Section 3.3). Then, the provided color and depth data is enhanced for further processing. This process includes 2D image filtering for noise reduction, re-projection into 3D space and view-frustum and occlusion culling (Section 3.4). The culling step effectively reduces the number of points for further processing, allowing high performances even for large datasets. In the third step, the surface information of the reduced dataset is reconstructed (Section 3.5). Finally, a visually closed surface - that is, a surface that contains no holes - is rendered from the *surfel* data and injected into the deferred rendering pipeline of the *Unity3D* engine (Section 3.6).

Each of the described stages is implemented with modularity in mind and offers scalability in quality and performance. The focus lies on the robustness, easy expandability and adequate hardware requirement of the end-to-end system. Similar to [21], we allow to distribute various processing steps among different physical machines. The communication between the interfaces of two subsequent stages can be piped through a network connection. In an experimental implementation we outsourced the input processing stage to a remote workstation, streaming the captured color and depth values to the client application. The client processes the data further and displays the result. The uncompressed data stream from an RGB-D camera with a resolution of 640×480 pixels at 30 frames per second results in a bandwidth usage of approximately 40 Mbps. By considering compression algorithms for depth maps and color frames, a much lower bandwidth usage can be achieved [21]. In case the server application controls the view, also the surface reconstruction can be outsourced to a remote workstation. This further reduces the processing load on the client device. Depending on the current view, the transferred color, normal and positional data requires a band-

with of approximately 50 Mbps for a scene with 200K visible points (which was the average for our captured scenes).

3.2. Engine Integration

Unity3D is a widespread game engine, enabling the simultaneous development for desktop, video game consoles and mobile operating systems. Furthermore, it simplifies development for common VR, AR and MR devices, like the HTC Vive, Oculus Rift or Microsoft Hololens [22], due to its platform-independence and the out-of-the-box support for many input and output devices.

The proposed system allows to place, configure and render dynamic point clouds in *Unity3D*, taking full advantage of the rich integrated development environment (IDE). The integrated 3D scene editor simplifies the utilization and customization of our system and facilitates the combination of dynamic point cloud data with classical geometry meshes. Each of the four processing modules shown in Figure 1 is represented by a component in *Unity3D*. Various implementations of our processing stages allow to exchange components based on the applications needs (e.g. higher performance versus superior visual quality), independently for each point cloud. Figure 2 shows the intermediate results of each component in the high-quality visualization pipeline for RGB-D input streams.

3.3. Input Processing

Generally, dynamic 3D point cloud data can have different representations. Most commonly such datasets are recorded from depth ranging sensors like time-of-flight sensors, structured light sensors or stereo setups. In our system we focus on RGB-D devices, which combine RGB data with distance measurements. Usually such devices incorporate two separate sensors to capture color and distance values simultaneously. In order to combine the recorded data, a mapping from one, or both, sensors into a reference sensor space needs to be defined in the first step.

We integrated the Intel RealSense SR300 RGB-D camera [10], which relies on a structured light principle, to capture and reconstruct live 3D data. The device is suited for close range usage and provides a full HD (1920×1080px) RGB color stream and VGA (640×480px) depth stream at 30 FPS. The RGB data stream has a color depth of 8 bits per channel, resulting in 24 bits per pixel. The depth values are streamed as 16 bit integer values, representing the distance of the imaged scene points in millimeters. The captured frames are accessed with the official API [13] maintained by Intel. The camera interface was directly exposed through a custom native plugin for *Unity3D*, allowing to efficiently upload and update the frames on the GPU using the platform dependent graphics API.

In addition to the RealSense SR300, also other input sources like image file sequences, network streams, as well as static point cloud geometry files are supported by our implementation. Each distinct input source is realized as a separate interchangeable component in *Unity3D*.

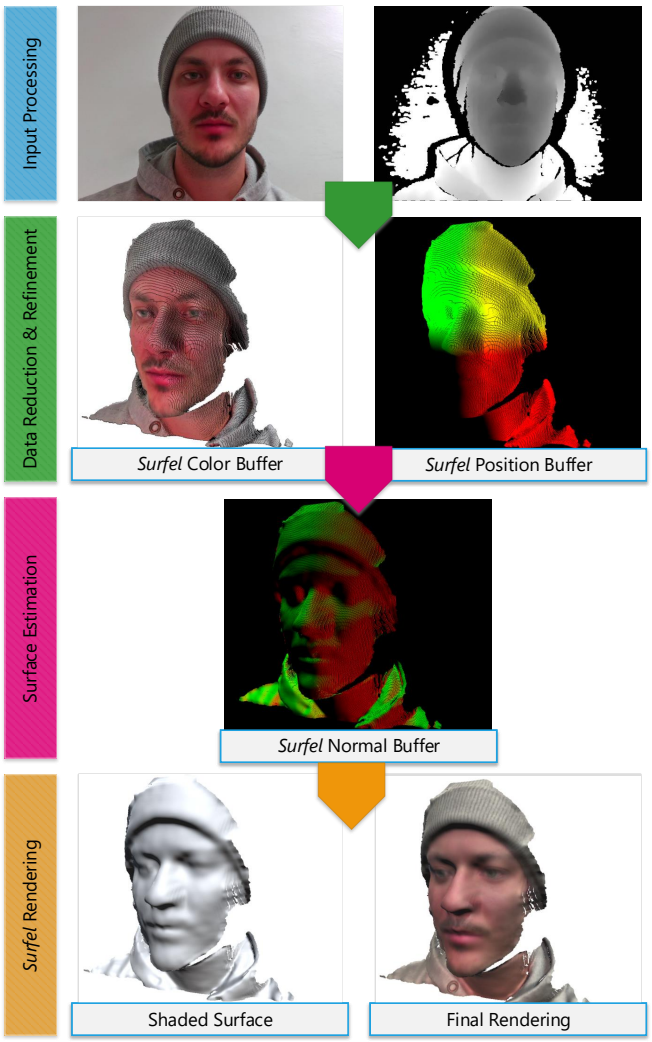


Fig. 2. Intermediate results of the processing pipeline components. The raw RGB-D data is reprojected and stored in screen space buffers containing color, position and normal values, which are then used to render the dataset.

3.4. Data Reduction and Refinement

After retrieving the raw data from the various input sources, it is necessary to reduce the data as far as possible before further processing. The goal is to suppress erroneous points and to minimize the additional overhead of processing superfluous information such as data which is not visible in the final rendered image. This process is generally called culling and can be achieved by various approaches.

Bounding volume hierarchies for fast geometry removal, known from classic culling approaches, have proven efficient

for spatially coherent datasets, but often lack flexibility when it comes to dynamic, fast changing and unconnected data [1]. More flexible approaches needed to be considered for live sensor stream processing. The implementation of our depth processing module exploits the 2D representation of the raw input data. Image filters provide an efficient way to reduce sensor noise and enhance the overall sensor output.

Such filters are able to manipulate the depth values in order to remove outliers or reduce sensor noise [9]. Our point cloud culling module incorporates two image filters: a median filter, reducing depth outliers originating from sensor errors, and a modified bilateral filter, to smoothen neighboring point positions while preserving characteristic edges.

Our median filter is based on an efficient GPU implementation introduced by McGuire [15], exploiting the parallelization capabilities of the fragment shader stage using a 5×5 kernel. The modified bilateral filter was implemented in a similar fashion and is defined as follows:

$$f(p,q) = g_\sigma(\|D(q) - D(p)\|)g_\varsigma(\|q - p\|) \tag{1}$$

$$W(p) = \sum_{q \in N(p)} f(p,q)v(q) \tag{2}$$

$$D(p)' = \frac{v(p)}{W(p)} \sum_{q \in N(p)} f(p,q)I(q)v(q) \tag{3}$$

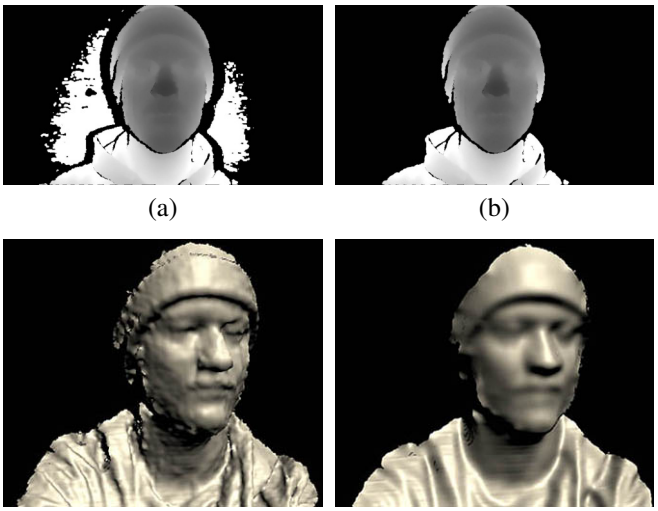
where $D(p)$ is the depth value at position p and $D(p)'$ is the resulting filtered depth. q represents a neighboring pixel in the (in our case, 9×9) window $N(p)$ centered at p. g_σ and g_ς are Gaussian functions smoothing the differences in depth and the 2D spatial differences, respectively. Additional to the standard definitions of a bilateral filter, we added the $v(p)$ term which restricts the smoothing to valid depth pixels:

$$v(p) = \begin{cases} 1 & \text{if } p \text{ is a valid depth value} \\ 0 & \text{else} \end{cases} \tag{4}$$

Figure 3 shows our reconstructed surface before (c) and after (d) the filtering passes.

Additionally, sensor depth images can be clipped in our implementation, using manually adjustable near and far planes in the sensor camera space. This process, which is denoted as distance culling, allows discarding points that are located too close or far away from the sensor's optimal working range and hence tend to be unreliable. The results can be most notably seen in Figure 3, original RGB-D output depth (a) and after distance culling (b).

As the last culling step, we project the 3D point positions into the screen space of the novel virtual view. This effectively removes occluded points and allows subsequent processing steps to focus only on the visible portion of points. The output of the data reduction and refinement step is an enhanced subset of the original dataset, containing only data which is relevant for the current view to be rendered (see Figure 3).



(a) (b) (c) (d)

Fig. 3. Unprocessed raw depth map as provided by the RGB-D camera (a) and the same depth map after distance culling and filtering using a median filter and modified bilateral filter (b), as described in Section 3.4. Reconstructed surface without (c) and with (d) filtering operations applied, corresponding to the 2D depth maps from (a) and (b), respectively.

3.5. Surface Estimation

Point positions and colors are usually not sufficient to render a point cloud with a smooth, visually appealing, hole-free surface. In addition, the surface orientation is crucial for lighting and reflection calculations. If the required surface normal is not available in the first place, an offline preprocessing step is commonly performed to recalculate the orientations for the complete dataset. However, offline preprocessing is not always applicable, especially when the point cloud needs to be captured and rendered in real-time. This section describes the pipeline stage which provides the functionality to estimate the surface of the remaining point cloud subset, after the data culling stage.

Our surface reconstruction is based on the AutoSplats algorithm, proposed in the previous work of Preiner et al. [19]. AutoSplats estimates the surface normal in screen space by finding the kNN for every point and fitting a plane into its local neighborhood. The algorithm achieves real-time performance, even for large point-clouds, through resourceful usages of fragment shader programs. We use the techniques presented in AutoSplats in our reconstruction module to estimate per-point normals for all visible points in each frame independently. Our adaptation approximates the kNN computation with a fixed radius, contrary to the iterative search in the original publication. This simplification allows our normal estimation to run in only three fragment shader passes, while still resulting in believable surface normals. Similar to [19] we use an initial *gather* pass to retrieve the *feedback radii* for

each individual point, then the components of the covariance matrix are computed with a second *gather* pass. Finally, the singular value decomposition (SVD) is solved and the eigenvector to the smallest eigenvalue is stored as the point normal. As opposed to the point sprite option utilized in [19], we adapted the geometry shader stage to draw the splats.

A big advantage of this approach is that it does not rely on spatial or temporal coherence. This frame-by-frame independence improves the robustness when dealing with rapidly changing point clouds with large changes in the topology.

3.6. Rendering

As the final step in our processing pipeline we render a hole-free, smooth surface from the oriented points resulting from the previous surface reconstruction step. Our implemented technique for point rendering relies on *surface splatting* [3]. The concept of *surface splatting* is similar to painting differently colored dots onto an object in the real world. The larger the painted dots the more they start to blend into each other and the gaps between the dots vanish. These painted dots or *splats* are created using a geometry shader program, emitting oriented quads at each point position. The size of these quads has to be chosen carefully, in order to create a sufficiently large overlap to close holes between neighboring points, while at the same time avoiding blurring artifacts. Generally, *surface splatting* encompasses three rendering passes: A visibility pass, an attribute pass and a shading pass. We implemented the first two passes as described in [3], the final pass normalizes and injects the blended surface attributes in *Unity3D*'s deferred G-Buffer. This enables state-of-the-art physically based illumination and environmental reflections on our dynamically reconstructed point cloud surface. Furthermore, *Unity3D*'s post-processing stack is fully applicable to the final output and allows further visual enhancements.

4. RESULTS

4.1. Performance Evaluation

In this section we evaluate the performance of our point cloud processing workflow on a high-end consumer system, using a desktop PC running Windows 10 with an Intel i7 CPU and an NVIDIA GTX1060 GPU at a rendering resolution of 1920×1080px.

Table 1 shows the processing time of each individual step in our pipeline. The performance measurements are narrowed down to the actual GPU processing time of the custom processing stages in milliseconds, which are computed every frame. These stages encompass view-dependent point culling, normal estimation and smooth surface rendering. Loading times for the specific file formats or stream upload times were neglected. This leaves us with the performance evaluation of the algorithms in the culling, surface reconstruction and rendering stage.

Points	**Visible**	**Cull**	**Recon.**	**Render**	**Overall**
#	#	ms	ms	ms	ms
36K	34K	0.12	1.80	2.00	**3.92**
307K	145K	1.07	4.89	1.60	**7.56**
3.6M	364K	2.62	13.30	4.00	**19.92**
5M	628K	4.10	17.10	7.70	**28.90**

Table 1. Performance evaluation of the GPU time of the culling, reconstruction and rendering stage on an NVIDIA GTX1060 at 1920×1080px. The second row corresponds with the RealSense depth resolution of 640×480px.

The measurements in Table 1 show that the proposed visualization pipeline performs in real-time, even on very large point clouds with 5 million points. However, the performance of the normal estimation stage and the rendering stage is largely dependent on the chosen fixed neighborhood and splat radius, respectively. Furthermore, as we reconstruct and render the point cloud in screen space, the projected size of the dataset impacts the visible points which have to be processed in the pipeline. Higher rendering resolutions lead to more points to be processed because fewer point-to-point occlusions take place. This is especially a challenge for mobile devices, where the processing power is very limited compared to the high display resolutions. Nevertheless, we could run our system on a mobile device, more specifically, on a Google Pixel XL Android smartphone with a quad-core processor and an Adreno 530 GPU. This shows that our end-to-end system has the means to bring real-time 3D surface reconstruction to mobile devices.

Furthermore, a direct comparison to the results of AutoSplats [19] shows that our implementation leads to faster computation times (Figure 4). Since both algorithms perform the reconstruction only for points projected to screen space, only the visible points were taken into account for this comparison. Figure 5 shows the angular error of the reconstructed normals of both algorithms. Blue areas have no error in the reconstruction, green areas have an error below 22.5 degrees and red values have an error above that. Figure 5 confirms that our simplification is still able to reconstruct accurate surfaces. The larger errors on sharp edges (e.g. the Stanford Armadillo's ears) occur due to our fixed neighborhood radius, as too few neighboring points are enclosed on silhouette points or edges.

4.2. Visual Quality

Synthetic datasets were used to evaluate the reconstruction quality on a set of typical dynamic scenes. Multiple 3D models of human faces with different emotions were rendered from a virtual camera view point, generating ground truth depth and surface normals. Figure 6 shows the color coded difference between the reconstructed surface normals and the ground truth. The main parts of the scanned faces are recon-

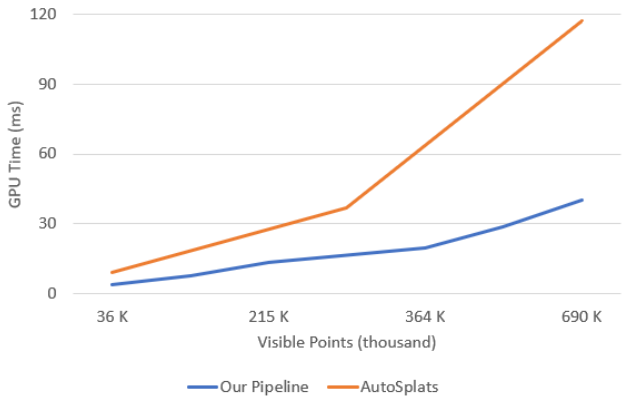


Fig. 4. Runtime comparison of our visualization pipeline with AutoSplats [19]. We use the actually visible points to compare both approaches in order to avoid view dependent differences.

structed without errors, small details and sharp edges around the eyes, mouth and hair still pose challenges to our approach and can result in artifacts.

The full integration of our visualization processing chain in the *Unity3D* rendering pipeline results in photo-realistic visual 3D reconstructions of the captured point clouds, as can be seen in Figure 7. The output is fully compatible with the engine's environmental reflection system and can be further polished with its predefined post-processing effects, like anti-aliasing or ambient occlusion [1]. Figure 7 displays four exemplary renderings of our RGB-D camera capturings. (a) displays a virtually shaded surface reconstruction of a person's head. (b) shows a mixed scene with a dynamic point cloud as well as geometry mesh elements. The green sphere and the two cubes in purple and red are rendered from classic triangle meshes. The interaction between these fundamentally different object representations can also be observed: the sphere casts a shadow onto the right hand, while the purple cube intersects with the head and the red box occludes the left hand. In Figure 7 (c) and (d) a sample of different material settings is shown. Materials are customizable combinations of shading properties, colors and textures. A glossy material setting is chosen to give the impression of a glazed coating in (c), whilst in (d) a gold-like metallic setting is used to display the environmental reflections and multiple colored light sources.

In Figure 8 some examples of distinct movements are visualized. The large topological changes resulting from touching the face with one or both hands are correctly handled by our processing pipeline and visualized accordingly. Such cases are challenging for meshing algorithms which rely on temporally coherent frames.

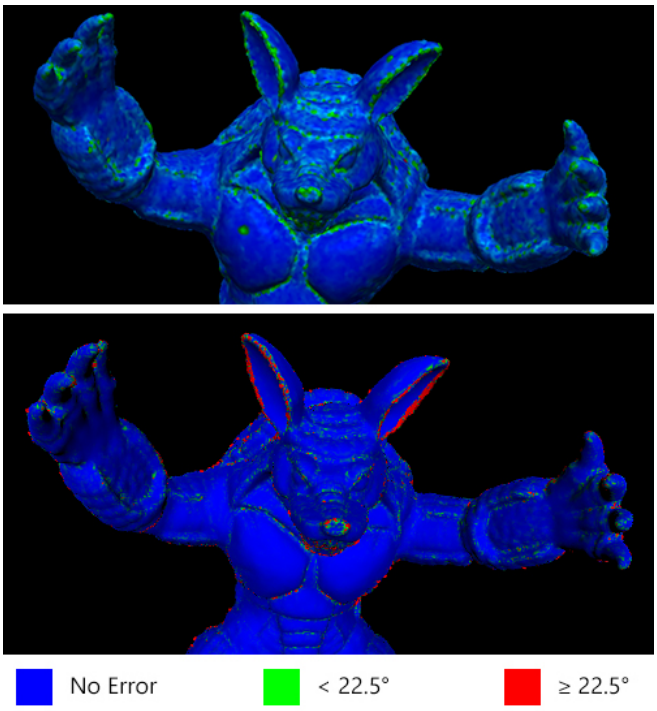


Fig. 5. Quality comparison of computed surface normals from AutoSplats [19] (top) and our approach (bottom). Blue areas have no angular error, green values have an error below 22.5 degrees and larger errors are displayed in red.

4.3. Limitations

A limitation of our approach to dynamic point cloud visualization lies in the higher computational effort for larger resolutions. This refers to the reconstruction as well as the rendering stage and can particularly affect mobile devices, where the display resolutions are high and the GPUs less powerful. Another possible drawback is the view dependency of our visualization. Client-server models as described in [21] can be implemented by sending the view changes from the client back to the reconstruction server.

5. CONCLUSION AND FUTURE WORK

We have proposed an end-to-end framework for the visualization of dynamic RGB-D data, which comprises the major processing stages of data culling, surface reconstruction and point rendering. The system's real-time performance was demonstrated for point cloud sizes of up to 5 million on an NVIDIA GTX1060 architecture. Compared to the state-of-the-art in point based reconstruction techniques, a significant increase in runtime performance was achieved while providing results of similar quality. We discussed selected aspects of our algorithms' implementation using the *Unity3D* game engine, which allows us to exploit the advantages of a state-

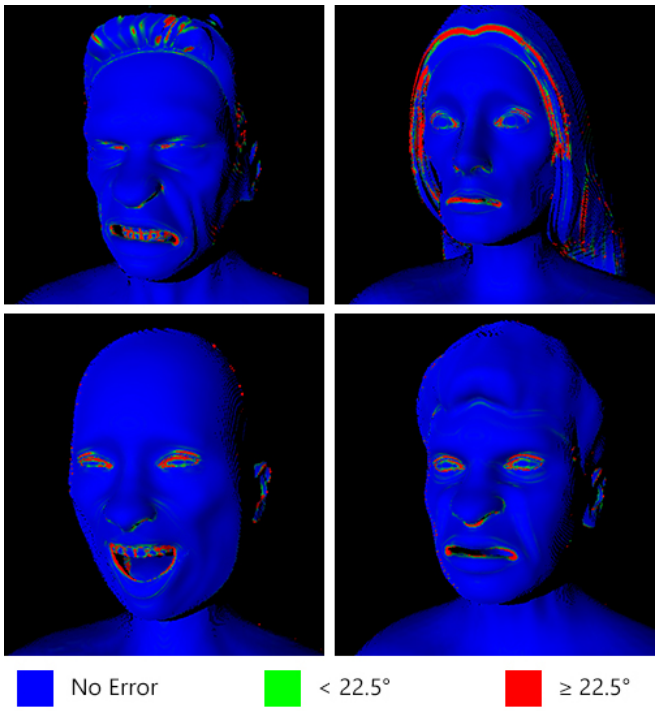


Fig. 6. Quality evaluation of surface normals with synthetically scanned data. The angular error between the ground truth and the reconstructed normals is color coded. Blue areas have no angular error, green values have an error below 22.5 degrees and larger errors are displayed in red.

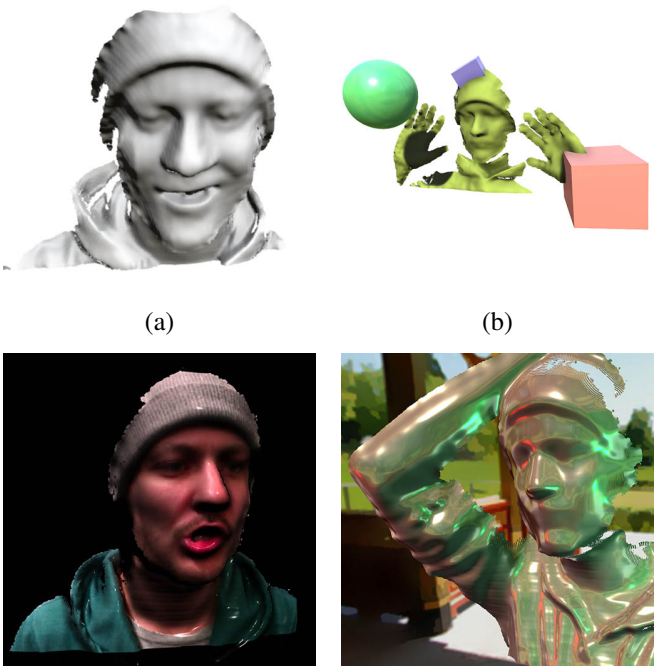


Fig. 7. Uncolored shaded surface of reconstructed face (a), hybrid scene containing a geometry mesh and a dynamic point cloud with similar material properties (b). Different rendering properties: glazed coating effect (c), metallic material with environmental reflections (d).

Fig. 8. Robust surface normal reconstruction even with large topological changes.

of-the-art rendering engine and platform interoperability. Our visualization results show the system's capability to achieve high-quality novel views with flexible adjustment of material properties and illumination effects, and seamless blending of real with synthetic content in mixed reality scenes. In order to overcome the system's current limitations on mobile devices, a possible topic for future work would be to optimize the system for rendering on higher resolution displays. This encompasses more efficient occlusion culling techniques as well as accelerations in the surface estimation and rendering stage.

6. ACKNOWLEDGEMENTS

This work has been funded by the Austrian Research Promotion Agency (FFG) and the Austrian Ministry BMVIT under the program ICT of the Future (project "Precise3D", grant no. 6905496). The color coded error image of the Stanford Armadillo using AutoSplats is provided by courtesy of Dominik Schörkhuber.

7. REFERENCES

[1] Tomas Akenine-Möller, Eric Haines, and Naty Hoffman. *Real-time rendering*. CRC Press, 2008.

[2] Daniele Bonatto, Ségolène Rogge, Arnaud Schenkel, Rudy Ercek, and Gauthier Lafruit. Explorations for real-time point cloud rendering of natural scenes in virtual reality. In *Proceedings of International Conference on 3D Imaging (IC3D)*, pages 1–7. IEEE, 2016.

[3] Mario Botsch, Alexander Hornung, Matthias Zwicker, and Leif Kobbelt. High-quality surface splatting on today's GPUs. In *Proceedings of VGTC Symposium on Point-Based Graphics*, pages 17–141. Eurographics, 2005.

[4] Mario Botsch, Michael Spernat, and Leif Kobbelt. Phong splatting. In *Proceedings of 1st Conference on Point-Based Graphics*, pages 25–32. Eurographics, 2004.

[5] Christian Boucheny. *Visualisation Scientifique de Grands Volumes de Données: Pour une Approche Perceptive*. PhD thesis, Université Joseph-Fourier-Grenoble I, 2009.

[6] Paul Bourke. Automatic 3D reconstruction: An exploration of the state of the art. *GSTF Journal on Computing (JoC)*, 2(3), 2018.

[7] Petar Dobrev, Paul Rosenthal, and Lars Linsen. An image-space approach to interactive point cloud rendering including shadows and transparency. *Computer Graphics and Geometry*, 12(3):2–25, 2010.

[8] Mingsong Dou, Sameh Khamis, Yury Degtyarev, Philip Davidson, Sean Ryan Fanello, Adarsh Kowdle, Sergio Orts Escolano, Christoph Rhemann, David Kim, and Jonathan Taylor. Fusion4d: Real-time performance capture of challenging scenes. *ACM Transactions on Graphics (TOG)*, 35(4):114, 2016.

[9] Marcin Grzegorzek, Christian Theobalt, Reinhard Koch, and Andreas Kolb. *Time-of-Flight and Depth Imaging. Sensors, Algorithms and Applications: Dagstuhl Seminar 2012 and GCPR Workshop on Imaging New Modalities*, volume 8200. Springer, 2013.

[10] Intel®. *RealSense™ Camera SR300 Product Datasheet*, 6 2016. Revision 1.

[11] Apple® iPhone Xs TrueDepth Camera. `www.apple.com/iphone-xs/cameras`. Accessed: 2018-09-14.

[12] Shahram Izadi, David Kim, Otmar Hilliges, David Molyneaux, Richard Newcombe, Pushmeet Kohli, Jamie Shotton, Steve Hodges, Dustin Freeman, and Andrew Davison. Kinectfusion: real-time 3D reconstruction and interaction using a moving depth camera. In *Proceedings of the 24th annual ACM symposium on User interface software and technology*, pages 559–568. ACM, 2011.

[13] Intel® RealSense™ SDK on GitHub. `www.github.com/IntelRealSense/librealsense`. Accessed: 2018-09-14.

[14] Charles Loop, Cha Zhang, and Zhengyou Zhang. Real-time high-resolution sparse voxelization with application to image-based modeling. In *Proceedings of the 5th High-Performance Graphics Conference*, pages 73–79. ACM, 2013.

[15] Morgan McGuire. A fast, small-radius GPU median filter. *Shader X*, February 2008.

[16] Richard A Newcombe, Dieter Fox, and Steven M Seitz. Dynamicfusion: Reconstruction and tracking of non-rigid scenes in real-time. In *Proceedings of Conference on Computer Vision and Pattern Recognition*, pages 343–352. IEEE, 2015.

[17] Sergio Orts-Escolano, Christoph Rhemann, Sean Fanello, Wayne Chang, Adarsh Kowdle, Yury Degtyarev, David Kim, Philip L Davidson, Sameh Khamis, Mingsong Dou, et al. Holoportation: Virtual 3D teleportation in real-time. In *Proceedings of the 29th Annual Symposium on User Interface Software and Technology*, pages 741–754. ACM, 2016.

[18] Hanspeter Pfister, Matthias Zwicker, Jeroen Van Baar, and Markus Gross. Surfels: Surface elements as rendering primitives. In *Proceedings of 27th Annual Conference on Computer Graphics and Interactive Techniques*, pages 335–342. ACM Press/Addison-Wesley Publishing Co., 2000.

[19] Reinhold Preiner, Stefan Jeschke, and Michael Wimmer. Auto splats: Dynamic point cloud visualization on the GPU. In *Proceedings of EGPGV Symposium on Parallel Graphics and Visualization*, pages 139–148. Eurographics, 2012.

[20] Fabio Remondino and Sabry El-Hakim. Image-based 3D modelling: a review. *The Photogrammetric Record*, 21(115):269–291, 2006.

[21] Donny Tytgat, Maarten Aerts, Jeroen De Busser, Sammy Lievens, Patrice Rondao Alface, and Jean-Francois Macq. A real-time 3D end-to-end augmented reality system (and its representation transformations). In *Proceedings of Applications of Digital Image Processing XXXIX*, volume 9971. International Society for Optics and Photonics, 2016.

[22] Unity 3D. `www.unity3d.com`. Accessed: 2018-09-14.

[23] Tao Yu, Kaiwen Guo, Feng Xu, Yuan Dong, Zhaoqi Su, Jianhui Zhao, Jianguo Li, Qionghai Dai, and Yebin Liu. Bodyfusion: Real-time capture of human motion and surface geometry using a single depth camera. In *Proceedings of International Conference on Computer Vision (ICCV)*, pages 910–919. IEEE, 2017.

[24] Matthias Zwicker, Hanspeter Pfister, Jeroen Van Baar, and Markus Gross. Surface splatting. In *Proceedings of 28th Annual Conference on Computer Graphics and Interactive Techniques*, pages 371–378. ACM, 2001.