International Conference on Artificial Reality and Telexistence
Eurographics Symposium on Virtual Environments (2024)
S. Hasegawa, N. Sakata and V. Sundstedt (Editors)

BlendPCR: Seamless and Efficient Rendering of Dynamic Point Clouds captured by Multiple RGB-D Cameras

A. Mühlenbrock[1] R. Weller[1] and G. Zachmann[1]

[1]Computer Graphics and Virtual Reality Research Lab, University of Bremen, Germany

(a) *Uniform Splats* **(b)** *Separate Meshes* **(c)** *TSDF192* **(d)** *TSDF512* **(e)** *BlendPCR* **(f)** *BlendPCR (HR)*

Figure 1: *Comparison of rendering techniques on the noisy CWIPC-SXR dataset in the S3 Flight Attendant scene. Besides comparing the six techniques depicted here, these figures facilitate comparison with additional state-of-the-art methods such as Pointersect [CCR*23] and P2ENet [HGSW24], which are shown in the supplementary material of [HGSW24] and illustrate the same scene segment. Both the baseline techniques (a-d) presented herein and the state-of-the-art methods depicted in the supplementary material of [HGSW24] exhibit seam flickering artifacts. In contrast, our technique (e-f) effectively eliminates these artifacts while simultaneously preserving details.*

Abstract

Traditional techniques for rendering continuous surfaces from dynamic, noisy point clouds using multi-camera setups often suffer from disruptive artifacts in overlapping areas, similar to z-fighting. We introduce BlendPCR, an advanced rendering technique that effectively addresses these artifacts through a dual approach of point cloud processing and screen space blending. Additionally, we present a UV coordinate encoding scheme to enable high-resolution texture mapping via standard camera SDKs. We demonstrate that our approach offers superior visual rendering quality over traditional splat and mesh-based methods and exhibits no artifacts in those overlapping areas, which still occur in leading-edge NeRF and Gaussian Splat based approaches like Pointersect and P2ENet. In practical tests with seven Microsoft Azure Kinects, processing, including uploading the point clouds to GPU, requires only 13.8 ms (when using one color per point) or 29.2 ms (using high-resolution color textures), and rendering at a resolution of 3580 x 2066 takes just 3.2 ms, proving its suitability for real-time VR applications.

CCS Concepts

• ***Computing methodologies*** → ***Rendering; Virtual reality;*** *Point-based models; Mesh geometry models;*

1. Introduction

Rendering dynamic 3D point cloud from RGB-D streams of multiple cameras is crucial for various applications, including the use of point cloud avatars in VR for enhanced social presence [GCC*20, YGE*21], performance capture systems [DKD*16], and dynamic geometry reconstruction, such as VR tele-assistance in surgical settings [RYP*21, GJS*21, FMK*22].

To address this, diverse approaches have been developed to optimally render point clouds, including volumetric techniques based on TSDF [DKD*16, DDF*17, YZG*21]. While learning-based methods like NeRFs have been limited to static scenes due to extensive training times [ASK*20, DZL*20, XXP*22, HXLJ23], recent advancements have enabled the rendering of dynamic point clouds using neural networks without prior scene-specific train-

DOI: 10.2312/egve.20241366

ing [CCR*23, HGSW24]. However, these techniques often struggle with performance issues; volumetric methods, which usually reconstruct the surface via Marching Cubes are computationally expensive, and pixel-wise rendering in NeRF-related or ray-tracing methods is costly at high resolutions, rendering them unsuitable for real-time VR applications. A recent technique that transforms dynamic point clouds into Gaussian splats using neural networks has been introduced for real-time rendering [HGSW24].

However, the use of multiple RGB-D cameras leads to visible artifacts not yet accounted for by current rendering techniques: When multiple cameras are used for real-time capture of a scene, identical surfaces are partially captured simultaneously by several cameras. Since different cameras have different positions, specular reflections differ quite noticeably, especially in the overlapping surface regions. This could be exacerbated by white balancing occurring in the cameras, or slightly different color gamuts. Thus, some parts of the surfaces in the scene are rendered multiple times, which leads to artifacts such as noticeable seams or effects with similar appearance to z-fighting. In the following, we will refer to these artifacts together as *seam flickering* (see Figures 1, 5, and 6, as well as our supplementary material).

We introduce a method for rendering continuous surfaces from point cloud streams of multiple RGB-D cameras, which specifically addresses these undesirable *seam flickering* artifacts through blending, offering significant advantages:

- Produces seamless transitions at overlaps between point clouds from multiple cameras through weight-based blending.
- Supports real-time performance using multiple cameras with screen resolutions and framerates suitable for VR.
- Allows for the use of different resolutions between depth images and RGB images, enabling higher color resolution than the actual point cloud size.
- Our implementation,[†], which relies solely on C++/OpenGL 3.3, is thus compatible with a wide range of platforms.

Additionally, we present a universal encoding scheme that efficiently and efficiently encodes 2D coordinates into 4D coordinates, preserving integrity during channel-wise linear interpolation. By applying this scheme to encode pixel coordinates (e.g. 16 Bit per channel) into RGBA color values (e.g. 8 Bit per channel), we leveraging the APIs of standard RGB-D sensors to efficiently generate UV coordinates for mapping the raw high-resolution color image onto the lower-resolution point cloud, see Section 3.4.

2. Related Work

Point cloud rendering, particularly of dynamic clouds, has long been a key challenge in computer graphics. We can roughly divide the approaches into three different categories: splat-based rendering techniques (Section 2.1), volumetric approaches (Section 2.2), and learning-based rendering methods (Section 2.3).

[†] GitHub: https://github.com/muehlenb/BlendPCR

2.1. Splat-based Rendering

While contemporary debug views frequently employ uniform rendering of 2D points without reconstructing a continuous surface, as exemplified by the Point Cloud Library [RC11] and Open3D [ZPK18], surface reconstruction techniques have evolved significantly over the decades. A pivotal advancement in this field is Surface Splatting [ZPvBG01], which utilizes ellipsoidal splats combined with EWA filtering for surface reconstruction. Subsequent enhancements to splatting include GPU implementations [BK03], rendering without preprocessing [WS06], and automatic surface fitting on the GPU using Screen Space KNN search [PJW12]. A related field of research involves the high-performance rendering of large static point cloud datasets. In this domain, real-time rendering of datasets containing over a billion points has been achieved, designed explicitly for rendering static artifact-free point clouds [SKW21, SKW22]. In point cloud rendering, addressing the challenge of splats initially having only a single color has led to the development of various methods to apply textures using pre-processed texture atlases. These methods aim to enhance rendering performance by reducing the number of points without significantly compromising quality or improving the quality itself [SSLK13, APS*14, CTCG19]. These techniques are primarily applied to static, pre-processed point clouds.

A significant contribution to current research is 3D Gaussian Splatting [KKLD23], initially developed for the photorealistic reconstruction of static scenes using an array of RGB images and requiring non-real-time preprocessing. Recent research has explored the conversion of points from RGB-D images into Gaussian splats using the neural network P2ENet [HGSW24], aiming to adapt the differentiable Gaussian Splatting renderer for real-time rendering of dynamic point clouds by RGB-D sensor streams.

2.2. Volumetric Fusion

Besides methods that render splats of separate cameras, volumetric approaches integrate data from one or more depth sensors into a unified 3D voxel grid. One of the earliest developments in this field was [CL96], which utilized a Signed Distance Function (SDF) for reconstruction. KinectFusion expanded on this by introducing a Truncated Signed Distance Function (TSDF) for real-time applications and also developed an ICP-based tracking algorithm for reconstructing static scenes using a moving camera [NIH*11]. Techniques like DynamicFusion and VolumeDeform adapted this approach for deformable objects by reconstructing a canonical model over time and applying detected deformations to this model. However, these methods are limited to mild movements, deformations without topological changes, and single camera usage [NFS15, IZN*16]. More recent methods such as Fusion4D, Function4D, and Motion2Fusion, which are also known as multi-view performance capture systems, have advanced the integration of multi-camera scenes into a common voxel grid [DKD*16, DDF*17, YZG*21]. They employ sophisticated techniques like adapted ED graphs [SSP07], texture atlases, and neural networks for deep implicit surface reconstruction. Even though methods for processing individually captured persons can be executed in real-time on powerful hardware, the required performance

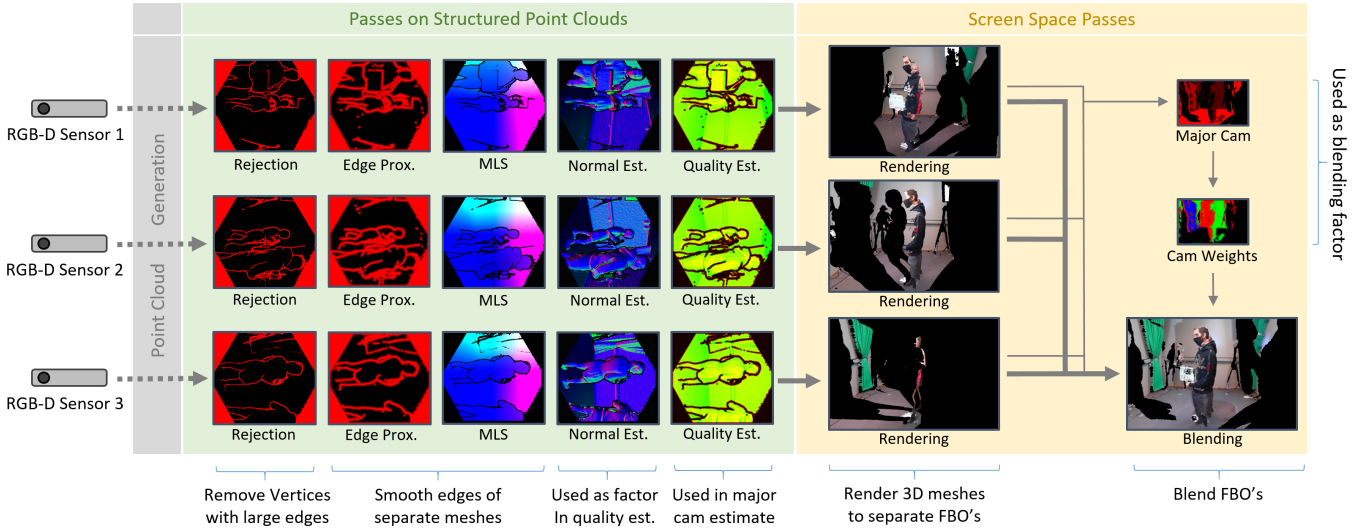


Figure 2: *Our pipeline for seamlessly rendering point clouds from multiple depth cameras involves various depth image-sized passes on the structured point clouds to smooth edges, determine normals, and estimate reconstruction quality. Subsequently, the structured point clouds are rendered as meshes on separate Frame Buffer Objects (FBOs) and blended, as explained in Section 3.*

is typically high due to voxelization, which limits its suitability for XR applications – especially in environments with multiple avatars.

2.3. Learning-based Rendering

Another approach involves learning-based methods that use a point cloud as input and learn to render it realistically using ground truth photos, for example, [ASK*20, DZL*20]. Additionally, some methods adapt Neural Radiance Fields (NeRF) [MST*20] to use point clouds as inputs in the rendering pipeline, as seen in [XXP*22, HXLJ23]. While these techniques achieve impressive results on static point clouds akin to NeRFs and Gaussian Splatting, their applicability to dynamic point cloud streams is constrained due to the requirement for the static scene to be learned over minutes to hours of training. Yet, recent advancements, such as Pointersect [CCR*23] using a NeRF-related approach or the Gaussian Splatting adaptation using P2ENet by [HGSW24], aim to mitigate this limitation for live point cloud streams. However, processing or rendering times at moderate resolutions are typically high, exacerbating the challenge for resolutions needed in XR applications. Furthermore, they usually do not take into account artifacts caused by the overlapping of separate point clouds from multiple consumer RGB-D cameras.

3. Our Method

In this section, we will first present an overview of our method, and then present each step in more detail.

3.1. Overview

Our main goals are to meet the performance requirements for VR applications while still visually merge continuous smooth surfaces from multiple RGB-D cameras seamlessly. To achieve this goal, we propose a new rendering method for dynamic point clouds that reconstructs meshes and blends them without visibe artifacts. Basically, for each separate structured point cloud, we use the inherent structure to reconstruct a canonical mesh (see Figure 3). We start by removing triangles that connect the foreground with the background, smoothing vertices at edges, generate normals, and estimate a reconstruction quality factor of how well each camera can reconstruct a specific area of the object, see Section 3.2. This is performed directly on the separate structured point clouds that are arranged in the 2D configuration of the original depth image. In the second phase, we explicitly render the meshes to separate framebuffers. Based on the estimated quality factor, we blend these framebuffers together. This second phase is performed in screen space and detailed in Section 3.3. This pipeline is depicted in Figure 2.

3.2. Point Cloud Processing

We manipulate a structured point cloud aligned with the depth image from which it originates during the point cloud processing stages. We apply standard techniques for edge smoothing, normal estimation, and estimation of reconstruction quality. All of these operations are performed at the depth image's original resolution. To achieve high compatibility and high performance through GPU acceleration, we have implemented these passes in separate GLSL fragment shaders, which are executed in order for each and every RGB-D camera, and use textures to upload structured point clouds and color textures.

3.2.1. Rejection Texture

Initially, the canonical meshes are created over the entire depth image per camera (see Figure 3), causing background and foreground objects to melt together. To sever these connections, we define a binary rejection texture which defines whether a point is valid: for

each point i, we determine the distance ed_i to the nearest point in the order of the 2D depth image that either exceeds a specific depth threshold from its neighbors or is invalid. Assuming $\mathbf{c_i}$ to be the vector pointing from a point i to the RGB-D camera position, the acceptable point-wise threshold t_i is defined based on the camera distance $|\mathbf{c_i}|$ and a constant k (we chose $k = 2.5\,\frac{\text{cm}}{\text{m}}$):

$$t_i = k \cdot |\mathbf{c_i}|$$

When we render the mesh later in the render pass (see Section 3.3.1), we remove all triangles that were marked as invalid by this rejection texture.

3.2.2. Edge Smoothing

To eliminate uneven silhouettes due to the flying pixel effect and sharp binary vertex deletion, we first identify vertices near object edges and smooth them based on their proximity to these boundaries. To streamline this process, we calculate a *truncated edge-proximity texture* ('Edge Prox.' in Figure 2): For each pixel of that texture, we calculate the pixel-wise edge proximity ep_i by scaling ed_i between 0 and 1 and capping distances greater than 10 pixels to enable its use as a straightforward influence factor.

$$\text{ep}_i = clamp(1 - \frac{\text{ed}_i}{10}, 0, 1)$$

To smooth the edges of the meshes, we calculate the Weighted Moving Least Squares (MLS) for all points x_i where $\text{ep}_i > 0$:

$$a_i = \frac{\sum_{j=1}^{k} \theta(||x_i - x_j||) p_j}{\sum_{j=1}^{k} \theta(||x_i - x_j||)}$$

with k = 121 using only 11 x 11 points around x_i in the structured point cloud for efficiency reasons, and with

$$\theta(d) = e^{-d^2/h^2},\ h = 0.02$$

With these smoothed vertices a_i, we can now update the position x_i of point i based on its proximity to an edge:

$$x_i \leftarrow \text{ep}_i \cdot a_i + (1 - \text{ep}_i) \cdot x_i$$

3.2.3. Normal Estimation

In order to render a smooth transition between meshes in overlapping areas, we propose to estimate the reconstruction quality of each part of each mesh. This allows us, then, to weight the parts of meshes during the blending according to the quality they will contribute to the final image. To effectively determine which mesh exhibits higher reconstruction quality in overlapping areas, it is essential to estimate the normal (refer to Section 3.2.4). To do so, we begin by calculating the weighted covariance matrix $B = (b_{ij}) \in \mathbb{R}^{3\times3}$:

$$b_{jh} = \sum_{l=1}^{k} \theta(||x_i - x_l||)(x_{l,j} - x_{i,j})(x_{l,h} - x_{i,h})$$

Using the characteristic polynomial and trigonometric methods to solve this polynomial, we calculate the eigenvalues of the matrix and, subsequently, the third eigenvector using Cholesky decomposition. For efficiency, we have implemented this in GLSL, too, using $k = 121$, i.e. 11×11 surrounding points for each point x_i.

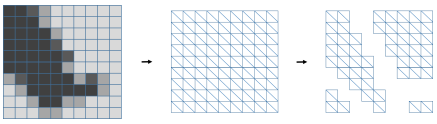


Figure 3: *Conceptual view of generating separate meshes. From the depth image, the generated 3D points are used as vertices for a contiguous grid-like 3D mesh. Triangles with edges too large in relation to camera distance are removed.*

3.2.4. Estimation of Reconstruction Quality

The estimated quality of a mesh at a point i depends on several factors:

1. **Distance to Camera** $||\mathbf{c_i}||$: The closer a mesh is to the camera, the higher the point density of the captured surface.
2. **Surface Normal** $\mathbf{n_i}$: The more parallel the surface normal $\mathbf{n_i}$ is to the vector $\mathbf{c_i}$ pointing from the surface towards the camera, the higher the point density.
3. **Proximity to Edge** ep_i: Pixels further from an object's edge, as viewed from the camera, experience fewer disturbances, such as those from the flying pixel effect.

Assuming an ideal camera model, a factor r that estimates the quality of reconstruction for a point can be approximated as follows, where c_{Max} defines the maximal point-to-camera distance that can occur, and where $\hat{\mathbf{n}}_\mathbf{i}$ and $\hat{\mathbf{c}}_\mathbf{i}$ are normalized (see $\mathbf{n_i}$ and $\mathbf{c_i}$):

$$r_i = ((c_{\text{Max}})^2 - ||\mathbf{c_i}||^2) \cdot (\hat{\mathbf{n}}_\mathbf{i} \cdot \hat{\mathbf{c}}_\mathbf{i})$$

For each point i in the point cloud, we estimate the reconstruction quality r_i. Additionally, the proximity to edge ep_i is calculated and stored for each point in a secondary texture channel. To ensure that smoothing operations in screen space do not compromise object boundaries, we do not yet combine factors ep_i and r_i.

3.3. Screen Passes

Screen passes are rendered at the resolution of the viewport and from the perspective of the virtual camera. In this passes, the separate meshes are ultimately rendered and subsequently blended.

3.3.1. Rendering Pass

In previous point cloud processing, we operated on structured point clouds, implicitly treating points as vertices of a mesh without explicitly creating it. Now, we actually render each separate point cloud in its own Frame Buffer Object (FBO) as mesh in a grid-like structure, as depicted in Figure 3. During this process, we use the Geometry Shader to remove all triangles with a vertex marked for deletion. This ensures that foreground objects do not merge with the background and that the triangle geometries remain distinct. The edge smoothing and geometric approximation have already been achieved through the modifications of x_i in the point cloud passes.

For further processing, we also output for each screen pixel the value of the Edge Proximity ep_i, the normal vector $\mathbf{n}$, and the estimated reconstruction quality r_i and store them in their respective frame buffer objects (FBO).

3.3.2. Camera Weight Pass

Various approaches can be employed to blend multiple mesh rendering, each stored in a different FBO. One method involves weighting all pixels based on their estimated reconstruction quality r_i to blend across large regions. However, blending over extensive areas using multiple cameras can lead to blurring of color textures due to camera-specific noise misalignment or minor registration errors. This issue can be mitigated by predominantly rendering pixels from the camera with the highest estimated reconstruction quality, hereafter referred to as *Major Cam*. Blending should only occur in regions where the dominant camera changes within the screen space.

To implement this efficiently, our approach is twofold. Initially, in the first screen pass, we determine the Major Cam for each pixel, identified as the camera providing the highest estimated reconstruction quality r_i for that pixel, and encode this information as an integer value in a texture (see Major Cam Est. in Figure 2). In a subsequent pass, we apply a simple smoothing operation using a 21×21 kernel on the Major Cam Est. texture and store in a separate channel of the output texture(s) a factor ranging from 0 to 1, indicating the influence of each camera on the corresponding pixel. This creates camera weight textures in screen space, enabling blending only in areas where a change in the optimal camera occurs. To conserve resources, both the Major-Cam and the Cam-Weights textures have a resolution of $1/16$th of the screen resolution, equivalent to $1/4$th along each dimension of height and width. This allows us to utilize the inherent smoothing of the `GL_Linear` texture option for further optimization.

3.3.3. Blending Pass

In the final blending pass, the separate meshes rendered in the Frame Buffer Objects (FBOs) at overlapping regions are blended according to the *Cam Weights* cw_i^h and *Edge Proximity* ep_i^h values for each FBO h and pixel i, to ensure a fading out at the object edges. The rendered color of a pixel i is determined by the formula, assuming col_i^h to be the color of FBO h at pixel i, and only including the k FBO's colors those fragments do not exceed a very small distance threshold to the fragment nearest to the virtual camera:

$$\mathrm{col}_i = \frac{\sum_{h=1}^{k} \mathrm{cw}_i^h \cdot (1 - \mathrm{ep}_i^h) \cdot \mathrm{col}_i^h}{\sum_{h=1}^{k} \mathrm{cw}_i^h \cdot (1 - \mathrm{ep}_i^h)}$$

3.4. BlendPCR (HR): Using High Resolution Textures

By employing separate continuous meshes for each camera, we allow for the rendering of color data using textures. This approach facilitates the use of disparate resolutions for the depth and color images—for instance, 640x576 for depth and 2048x1536 for color images. Such a configuration allows for superimposing high-resolution color information onto a lower-resolution mesh, as is often done in computer graphics.

However, devices such as the Azure Kinect, which utilize distinct sensors for capturing color and depth information, necessitate a per-pixel mapping from the coordinates of the depth image to those of the color image. This mapping, essentially a UV map for the mesh, can be generated by creating an image that stores the 2D color pixel coordinates. This coordinate image is then transformed from the color image coordinate into the depth image coordinate using the camera's API, an API that is usually meant to transform the real color image into the depth image directly. It is important to note that limitations exist within the camera API; for example, the Azure Kinect SDK provides only a function for transforming images from color camera coordinates to depth camera coordinates that operates with BGRA32 format (8 bits per channel) and always interpolates channels separately, since it was originally intended for the transformation of the color image directly, and not an image that stores coordinates in its pixels. However, when pixel coordinates are naïvely encoded into BGRA32 values through bit shifting, the coordinates in non-continuous regions are corrupted by the interpolation, as shown in Figure 4.

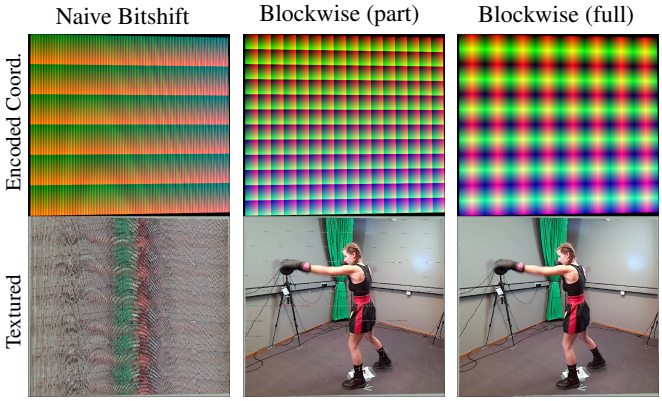


Figure 4: *Effect of linear interpolation to different encoding schemes when encoding 2D image coordinates into RGB values. On the left, naïve bit shifting is shown, where the pixel ID* $(2048 \cdot y + x)$ *is split bitwise across r, g, b. In the center, our block-based approach is displayed without oscillation and vertical repair. On the right, our full block-based method is applied as described in Section 3.4. At the top, the coordinates encoded in RGB are displayed, while at the bottom, the reconstructed textures using the decoded coordinates are shown. The activation of linear interpolation leads to the destruction of the encoded coordinates in the left and middle case.*

To overcome this issue, we devised an encoding scheme for 2D pixel coordinates into BGRA32-values that remain unaffected by channel-wise linear interpolation. This scheme even leverages interpolation to achieve subpixel precision in the resulting values.

To encode the pixel coordinates c_x and c_y, within the ranges [0, 2047] and [0, 1535], respectively, into BGRA32 format, our scheme segments the coordinate space into 64 x 64 blocks.

The blue channel (i_b) encodes the id of an 64 x 64 block:

$$i_b = \mathrm{int}(\frac{x}{64}) + \mathrm{int}(\frac{y}{64}) \cdot 32$$

The red channel (i_r) oscillates along the x axis within each block. The mode of oscillation—ascending or descending—is determined by the even or odd nature of i_b, ensuring a continuous and, therefore

interpolation-invariant encoding of the coordinates c_x and c_y:

$$i_r = \begin{cases} (c_x \cdot 4) \mod 256 & \text{if } i_b \equiv 0 \mod 2 \\ (255 - c_x \cdot 4) \mod 256 & \text{if } i_b \equiv 1 \mod 2 \end{cases}$$

Multiplying by 4 allows for a 2.5 mm sub-pixel precision of the UV coordinates during interpolation and can be adapted for higher precision. Similarly, the green channel (i_g) modulates based on c_y, with the direction of modulation controlled by the parity of i_b divided by 32, reflecting its y-axis block positioning:

$$i_g = \begin{cases} (c_y \cdot 4) \mod 256 & \text{if } \frac{i_b}{32} \equiv 0 \mod 2 \\ (255 - c_y \cdot 4) \mod 256 & \text{if } \frac{i_b}{32} \equiv 1 \mod 2 \end{cases}$$

Due to the structured nature of this encoding, every 64 units along the c_y-axis correspond to a jump of 32 in the block ID (i_b), the only vulnerable point for errors due to interpolation in this encoding scheme. However, this kind of error in the interpolated block ID can be rectified quite easily because the constructed block id i_b remains constant over a larger area of the image: For each pixel, the block ID is corrected based on the most frequent block IDs observed in the few pixels directly above and below, effectively eliminating aberrant interpolations.

When the block ID exceeds 255, we simply store its upper bits in the alpha channel i_a. By first decoding the blue and alpha channels together and then correcting the errors along the y-axis, aberrant interpolations are also eliminated here.

4. Evaluation

In the subsequent sections, we evaluate the visual rendering quality and compare it both with traditional techniques suitable for VR applications, as well as further state-of-the-art techniques, as detailed in Section 4.1. Furthermore, we assess performance metrics in Section 4.2.

4.1. Visual Comparison

In the subsequent sections, we evaluate our method's visual quality, specifically its ability to smooth overlapping regions between multiple cameras. To benchmark the visual quality against other studies, we utilize the *CWIPC-SXR* dataset for evaluation [RAJ*21].[‡] This dataset consists of 45 unique dynamic sequences of people in typical Social XR use cases, recorded simultaneously with seven co-registered Microsoft Azure Kinects. Depth images were captured at 640×576, while simultaneous color recordings were made at a resolution of 2048×1536, enabling us to test our high-resolution texture mapping.

We compare traditional rendering techniques currently employed or potentially suitable for real-time VR applications. Each method is discussed in the context of its application and expected performance:

- *Uniform Splats*: Points from point clouds are rendered as uniformly sized splats to form a continuous surface. Variations of these techniques have been successfully applied in various VR applications, e.g. [GCC*20], [GJS*21], and [FMK*22].
- *Separate Meshes*: Points are considered as vertices of a grid-like mesh, from which overly large triangles are removed. This serves as a foundation for our approach and has also been built upon in previous work, e.g., [YGE*21], [RYP*21], and [FMK*22].
- *TSDF192*: Uses a 3D TSDF with a voxel grid of size $192 \times 192 \times 192 \approx 7\,\text{M}$ voxels with a density of 1 voxel per cubic cm, combined with a Marching Cubes algorithm implemented in CUDA, upon which volumetric performance capture systems are often built.
- *TSDF512*: Represents the same volume as *TSDF192*, but with a voxel grid of size $512 \times 512 \times 512 \approx 134\,\text{M}$ voxels, resulting in a density of ≈ 19 voxels per cubic cm.
- *BlendPCR*: Our method without the high-resolution textures described in Section 3.4. Each vertex contains a color value based on the resolution of the original depth image.
- *BlendPCR (HR)*: Our advanced method with high-resolution textures (see Section 3.4), generated from depth images of size 640×576 and color textures of size 2048×1536 per camera.

For the visual comparison with additional state-of-the-art techniques, we utilized the *CWIPC-SXR* dataset [RAJ*21] that was also used by [HGSW24] and selected segments comparable to those shown there. By doing so, the comparative images in Figures 1 and 5 are also comparable to the state-of-the-art techniques showcased in the paper and supplementary material of [HGSW24]. Please refer to these materials for further comparison. Note that due to the noisy point cloud data, we employ a very simple spatial hole-filler and a basic erosion filter for object edges. These have also been applied to our other displayed techniques such as *Uniform Splats*, *Separate Mesh*, and *TSDF* to achieve a fair comparison.

Our results presented in Figures 1 and 5 demonstrate that our method effectively generates seamless transitions between the point clouds of multiple RGB-D sensors and achieves the highest quality in terms of preserving details, compared to uniform splatting, Separate Meshes, and the techniques described in [HGSW24], which suffer from *seam flickering* artifacts. Additionally, Figure 6 highlights the effectiveness of artifact removal by our rendering method in fully rendered scenes. Particularly, artifacts at the edges of objects in the Separate Meshes approach were almost completely eliminated using our simple processing strategy with *Weighted Moving Least Squares (MLS)*, weighted by the *Truncated Edge Proximity Texture*. Compared to the TSDF-based method, it is evident that our approach preserves details significantly better. Compared to uniform splats, our method does not compromise detail by using overly large points, nor does it result in a porous surface due to excessively small points. Furthermore, as illustrated in Figures 1 and 5, as well as in our supplementary material, our *BlendPCR (HR)* method achieves a higher level of detail through the use of higher-resolution color textures. Our encoding scheme facilitates the straightforward application of these raw textures by the color camera.

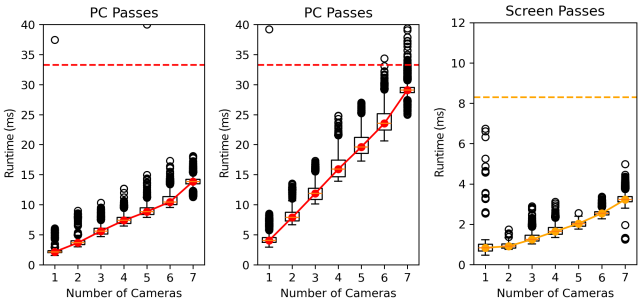
[‡] Note that other datasets often used for comparison, like THuman 2.0 [YZG*21], which contains static captured meshes of people in various poses, and BlendedMVS [YLL*20], which provides large static scenes as ground truth for Multi-View Stereo networks (e.g., NeRFs), are only partially suitable as they do not exhibit the typical artifacts that RGB-D cameras generate in multi-camera scenarios.

(a) *Uniform Splats* **(b)** *Separate Meshes* **(c)** *TSDF192* **(d)** *TSDF512* **(e)** *BlendPCR* **(f)** *BlendPCR (HR)*

Figure 5: *Comparison of rendering techniques applied to the noisy CWIPC-SXR dataset in the S13 Card Trick scene. This figure demonstrates that our techniques (e-f) achieves superior detail preservation and effectively eliminates* seam flickering *artifacts. Additionally, this figure enables comparison with state-of-the-art methods such as Pointersect [CCR*23] and P2ENet [HGSW24], which are also depicted in [HGSW24] and display visible* seam flickering *artifacts.*

(a) *Separate Meshes* **(b)** *BlendPCR (HR)*

Figure 6: *Removal of disruptive* seam flickering *artifacts in complete scenes. The image highlights the significant visibility of* seam flickering *when rendering full scenes. Compared to the Separate Mesh rendering (a) and other techniques, our BlendPCR technique effectively eliminates such artifacts.*



(a) *Integration* **(b)** *Integration (HR)* **(c)** *Rendering*

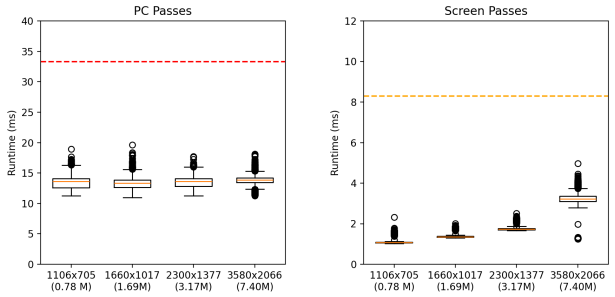
Figure 7: *Performance analysis of point cloud processing with varying numbers of RGB-D cameras, demonstrating the feasibility of real-time rendering. The analysis includes scenarios with high-resolution color textures (a), without high-resolution color textures (b), and screen passes that are executed in all cases (c). Dashed lines indicate the maximum runtime allowed for real-time rendering. Since point cloud passes are executed for each new point cloud, this corresponds to a refresh rate of 30 Hz on an Azure Kinect, as shown by the red dashed line. For rendering, we have set 120 FPS as the minimum required frame rate considered real-time, as it is deemed an optimal frame rate for VR applications, indicated by the yellow dashed line. Note that the increased overhead in scenario (b) arises solely from uploading up to seven color textures, each 2048x1536, via* `glTexSubImage2D`*. Screen rendering was performed at a resolution of 3580x2066.*

4.2. Runtime

Runtime performance of point cloud rendering is crucial for VR applications due to high demands on frame rate and resolution. In addition, there are numerous other tasks, such as decoding point cloud streams for telepresence applications, rendering the rest of the scene, and managing application mechanics, which must be executed alongside rendering. Therefore, we comprehensively test the performance of our approach in terms of the number of cameras, rendered screen resolution, and the contribution of individual passes to total execution time. All benchmarks were conducted on a workstation with an AMD Ryzen 9 3900X, an NVIDIA GeForce RTX 4090, and 32 GB of RAM, using the complete point clouds without clipping.

Runtime considerations in our evaluation are distinctly segmented between (a) the integration of point clouds, mainly through the Point Cloud Passes, and (b) the rendering of point clouds, particularly through the Screen Passes, as these processes have different frame rate requirements and are typically parallelizable. It is important to note that within our *BlendPCR* method, the integration process (a) encompasses not only the Point Cloud Passes but also the uploading of point clouds from RAM to VRAM via `glTexSubImage2D`. This step is time-intensive, especially when dealing with high-resolution color images from the cameras.

We evaluate runtime performance relative to the number of cameras used, as illustrated in Figure 7. The results indicate that even with seven cameras, real-time capability is maintained, with the processing of all seven point clouds from a single frame taking only 13.8 ms without high-resolution (HR) textures, and only 29.2 ms with HR textures on average. Rendering for seven RGB-D cameras



(a) *Processing* **(b)** *Rendering*

Figure 8: *Performance analysis across various resolutions reveals that screen resolution does not impact point cloud passes but, as expected, affects screen passes. Notably, even at high resolutions, such as 3580x2066, less than 4 ms is required for screen passes.*

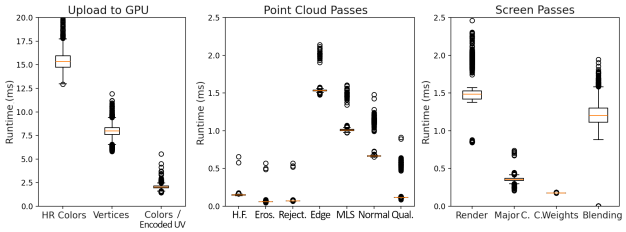


Figure 9: *Comparison of runtime for individual tasks or passes with seven RGB-D cameras and a screen resolution of 3580 x 2066 is depicted. We observed that uploading point cloud data to GPU consumes the most performance. During the processing of the point clouds, the Edge Proximity, MLS, and Normal Passes require substantially more performance than the Rejection and Quality Estimation Passes. For the sake of completeness, we also show the performance of the Hole Filling (H.F.) and Erosion (Eros.) filters, which were used upstream of all rendering techniques in the visual comparison and were not a dedicated part of BlendPCR.*

at a resolution of 3580 x 2066 is consistently achievable in 3.2 ms on average, rendering our method suitable for VR applications. Figure 8 illustrates that, as expected, the performance of the Point Cloud Passes is independently of screen resolution. Additionally, the runtime of the Screen Passes (rendering) per frame decreases significantly with a reduction in resolution. Furthermore, Figure 9 details the runtime of individual passes, as depicted in Figure 2.

4.3. Discussion and Limitations

Our BlendPCR method for rendering point clouds, as demonstrated in the previous section, is highly efficient. Conceptually, it should be feasible to distribute the processing of structured point clouds and the individual rendering passes across multiple GPUs or distributed systems to support more than seven cameras in real-time.

In terms of GPU memory requirements, our BlendPCR method can be executed on modern consumer hardware, even when handling multiple cameras. For processing structured point clouds, including pre-processing with hole-filling and erosion filters, we generate 8 textures requiring a total of 70 bytes per pixel in uncompressed memory. Each separate rendering pass per camera uses a framebuffer with 32 bytes per pixel at screen space resolution. For a depth resolution of 640 x 576, we allocate 24.6 MiB of GPU memory per camera, and for the separate rendering passes at a screen resolution of 3580 x 2066, we allocate an additional 225.7 MiB of GPU memory per camera. Additionally, there is a constant memory requirement for components like the default framebuffer, the Major Camera, and Camera Weights textures. However, the latter textures, due to their lower resolution and minimal memory need (9 bytes per pixel), consume approximately 4 MiB. Note that our current implementation has significant potential for further optimization through clever reuse of buffers.

Typical noise and minor errors in calibrating and registering depth sensors can cause slight color blurring at the boundaries where point clouds blend. However, our method minimizes this issue by reducing the blend areas and selecting a major camera for each fragment, as detailed in Section 3.3.2. While methods that reconstruct separate meshes are prone to producing small visible holes due to invalid pixels in depth images—as all triangles containing an invalid vertex are discarded—we mitigate this issue with an initial hole-filling filter. Note that when point cloud passes (e.g., at 30 Hz) and screen passes (e.g., at 120 Hz) are computed concurrently on a single GPU, distributing the point cloud passes over multiple frames and scheduling them in between screen passes might be crucial to avoid micro-stutters.

5. Conclusion

We have developed an efficient method for rendering continuous surfaces of dynamic point clouds in multi-camera scenarios. Our method effectively solves the issue of seam flickering artifacts similar to z-fighting, while preserving very high levels of detail of the original point cloud. Our method is particularly well-suited for VR applications that demand low latency rendering and high frame rates. On average, it requires only 3.2 ms for the screen passes at a resolution of 3580 x 2066, while simultaneously blending point clouds from seven RGB-D cameras. In addition, our method is fairly easy to implement in shaders, which can be seen in the source code we provide at www.github.com/muehlenb/BlendPCR.

Future research could investigate the selective uploading of only essential parts of high-resolution textures, which significantly impact performance during processing. Furthermore, integrating our blending approach with other rendering techniques, such as Neural Radiance Fields (NeRFs) or Gaussian Splatting, could be explored to similarly prevent *seam flickering*. Moreover, developing algorithms for completing point clouds suitable for use with BlendPCR to address gaps caused by self-shadowing could advance the field of multi-camera point cloud setups significantly. Finally, a subjective comparative study could be conducted to compare BlendPCR with other state-of-the-art rendering techniques, particularly in VR, and the development of standardized benchmarks using objective metrics should also be pursued.

6. Acknowledgements

This work was partially supported by BMBF grant 16SV9239.

References

[APS*14] ARIKAN M., PREINER R., SCHEIBLAUER C., JESCHKE S., WIMMER M.: Large-scale point-cloud visualization through localized textured surface reconstruction. *IEEE Transactions on Visualization and Computer Graphics 20* (2014), 1280–1292. 2

[ASK*20] ALIEV K.-A., SEVASTOPOLSKY A., KOLOS M., ULYANOV D., LEMPITSKY V.: Neural point-based graphics. In *Computer Vision – ECCV 2020: 16th European Conference, Glasgow, UK, August 23–28, 2020, Proceedings, Part XXII* (Berlin, Heidelberg, 2020), Springer-Verlag, p. 696–712. doi:10.1007/978-3-030-58542-6_42. 1, 3

[BK03] BOTSCH M., KOBBELT L.: High-quality point-based rendering on modern gpus. In *11th Pacific Conference onComputer Graphics and Applications, 2003. Proceedings.* (2003), pp. 335–343. doi:10.1109/PCCGA.2003.1238275. 2

[CCR*23] CHANG J.-H. R., CHEN W.-Y., RANJAN A., YI K. M., TUZEL O.: Pointersect: Neural rendering with cloud-ray intersection. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (2023). 1, 2, 3, 7

[CL96] CURLESS B., LEVOY M.: A volumetric method for building complex models from range images. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques* (New York, NY, USA, 1996), SIGGRAPH '96, Association for Computing Machinery, p. 303–312. doi:10.1145/237170.237269. 2

[CTCG19] COMINO TRINIDAD M., CALAF A. C., GRAN C. A.: View-dependent Hierarchical Rendering of Massive Point Clouds through Textured Splats. In *Spanish Computer Graphics Conference (CEIG)* (2019), Casas D., Jarabo A., (Eds.), The Eurographics Association. doi:10.2312/ceig.20191203. 2

[DDF*17] DOU M., DAVIDSON P., FANELLO S. R., KHAMIS S., KOWDLE A., RHEMANN C., TANKOVICH V., IZADI S.: Motion2fusion: real-time volumetric performance capture. *ACM Trans. Graph. 36*, 6 (nov 2017). doi:10.1145/3130800.3130801. 1, 2

[DKD*16] DOU M., KHAMIS S., DEGTYAREV Y., DAVIDSON P., FANELLO S. R., KOWDLE A., ESCOLANO S. O., RHEMANN C., KIM D., TAYLOR J., KOHLI P., TANKOVICH V., IZADI S.: Fusion4d: real-time performance capture of challenging scenes. *ACM Trans. Graph. 35*, 4 (jul 2016). doi:10.1145/2897824.2925969. 1, 2

[DZL*20] DAI P., ZHANG Y., LI Z., LIU S., ZENG B.: Neural point cloud rendering via multi-plane projection. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition* (2020), pp. 7830–7839. 1, 3

[FMK*22] FISCHER R., MÜHLENBROCK A., KULAPICHITR F., USLAR V. N., WEYHE D., ZACHMANN G.: Evaluation of point cloud streaming and rendering for vr-based telepresence in the or. In *Virtual Reality and Mixed Reality: 19th EuroXR International Conference, EuroXR 2022, Stuttgart, Germany, September 14–16, 2022, Proceedings* (Berlin, Heidelberg, 2022), Springer-Verlag, p. 89–110. doi:10.1007/978-3-031-16234-3_6. 1, 6

[GCC*20] GAMELIN G., CHELLALI A., CHEIKH S., RICCA A., DUMAS C., OTMANE S.: Point-cloud avatars to improve spatial communication in immersive collaborative virtual environments. *Personal Ubiquitous Comput. 25*, 3 (jul 2020), 467–484. URL: https://doi.org/10.1007/s00779-020-01431-1, doi:10.1007/s00779-020-01431-1. 1, 6

[GJS*21] GASQUES D., JOHNSON J. G., SHARKEY T., FENG Y., WANG R., XU Z. R., ZAVALA E., ZHANG Y., XIE W., ZHANG X., DAVIS K., YIP M., WEIBEL N.: Artemis: A collaborative mixed-reality system for immersive surgical telementoring. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems* (New York, NY, USA, 2021), CHI '21, Association for Computing Machinery. doi:10.1145/3411764.3445576. 1, 6

[HGSW24] HU Y., GONG R., SUN Q., WANG Y.: Low latency point cloud rendering with learned splatting. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR) Workshops* (June 2024), pp. 5752–5761. 1, 2, 3, 6, 7

[HXLJ23] HU T., XU X., LIU S., JIA J.: Point2pix: Photo-realistic point cloud rendering via neural radiance fields. In *2023 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)* (Los Alamitos, CA, USA, jun 2023), IEEE Computer Society, pp. 8349–8358. doi:10.1109/CVPR52729.2023.00807. 1, 3

[IZN*16] INNMANN M., ZOLLHÖFER M., NIESSNER M., THEOBALT C., STAMMINGER M.: Volumedeform: Real-time volumetric non-rigid reconstruction. In *Computer Vision – ECCV 2016* (Cham, 2016), Leibe B., Matas J., Sebe N., Welling M., (Eds.), Springer International Publishing, pp. 362–379. 2

[KKLD23] KERBL B., KOPANAS G., LEIMKUEHLER T., DRETTAKIS G.: 3d gaussian splatting for real-time radiance field rendering. *ACM Trans. Graph. 42*, 4 (jul 2023). doi:10.1145/3592433. 2

[MST*20] MILDENHALL B., SRINIVASAN P. P., TANCIK M., BARRON J. T., RAMAMOORTHI R., NG R.: Nerf: Representing scenes as neural radiance fields for view synthesis. In *ECCV* (2020). 3

[NFS15] NEWCOMBE R. A., FOX D., SEITZ S. M.: Dynamicfusion: Reconstruction and tracking of non-rigid scenes in real-time. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (2015), pp. 343–352. doi:10.1109/CVPR.2015.7298631. 2

[NIH*11] NEWCOMBE R. A., IZADI S., HILLIGES O., MOLYNEAUX D., KIM D., DAVISON A. J., KOHI P., SHOTTON J., HODGES S., FITZGIBBON A.: Kinectfusion: Real-time dense surface mapping and tracking. In *2011 10th IEEE International Symposium on Mixed and Augmented Reality* (2011), pp. 127–136. doi:10.1109/ISMAR.2011.6092378. 2

[PJW12] PREINER R., JESCHKE S., WIMMER M.: Auto splats: Dynamic point cloud visualization on the gpu. In *Proceedings of Eurographics Symposium on Parallel Graphics and Visualization* (May 2012), Childs H., Kuhlen T., (Eds.), Eurographics Association 2012, pp. 139–148. 2

[RAJ*21] REIMAT I., ALEXIOU E., JANSEN J., VIOLA I., SUBRAMANYAM S., CESAR P.: Cwipc-sxr: Point cloud dynamic human dataset for social xr. In *Proceedings of the 12th ACM Multimedia Systems Conference* (New York, NY, USA, 2021), MMSys '21, Association for Computing Machinery, p. 300–306. URL: https://doi.org/10.1145/3458305.3478452, doi:10.1145/3458305.3478452. 6

[RC11] RUSU R. B., COUSINS S.: 3D is here: Point Cloud Library (PCL). In *IEEE International Conference on Robotics and Automation (ICRA)* (Shanghai, China, May 9-13 2011), IEEE. 2

[RYP*21] ROTH D., YU K., PANKRATZ F., GORBACHEV G., KELLER A., LAZAROVICI M., WILHELM D., WEIDERT S., NAVAB N., ECK U.: Real-time mixed reality teleconsultation for intensive care units in pandemic situations. In *2021 IEEE Conference on Virtual Reality and 3D User Interfaces Abstracts and Workshops (VRW)* (Los Alamitos, CA, USA, apr 2021), IEEE Computer Society, pp. 693–694. doi:10.1109/VRW52623.2021.00229. 1, 6

[SKW21] SCHÜTZ M., KERBL B., WIMMER M.: Rendering point clouds with compute shaders and vertex order optimization. *Computer Graphics Forum 40*, 4 (2021), 115–126. doi:10.1111/cgf.14345. 2

[SKW22] SCHÜTZ M., KERBL B., WIMMER M.: Software rasterization of 2 billion points in real time. *Proc. ACM Comput. Graph. Interact. Tech. 5*, 3 (jul 2022). doi:10.1145/3543863. 2

[SSLK13] SIBBING D., SATTLER T., LEIBE B., KOBBELT L.: Sift-realistic rendering. In *2013 International Conference on 3D Vision - 3DV 2013* (2013), pp. 56–63. doi:10.1109/3DV.2013.16. 2

[SSP07] SUMNER R. W., SCHMID J., PAULY M.: Embedded deformation for shape manipulation. *ACM Trans. Graph. 26*, 3 (jul 2007), 80–es. doi:10.1145/1276377.1276478. 2

[WS06] WIMMER M., SCHEIBLAUER C.: Instant Points: Fast Rendering

of Unprocessed Point Clouds. In *Symposium on Point-Based Graphics* (2006), Botsch M., Chen B., Pauly M., Zwicker M., (Eds.), The Eurographics Association. doi:/10.2312/SPBG/SPBG06/129-136. 2

[XXP*22] XU Q., XU Z., PHILIP J., BI S., SHU Z., SUNKAVALLI K., NEUMANN U.: Point-nerf: Point-based neural radiance fields. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition* (2022), pp. 5438–5448. 1, 3

[YGE*21] YU K., GORBACHEV G., ECK U., PANKRATZ F., NAVAB N., ROTH D.: Avatars for teleconsultation: Effects of avatar embodiment techniques on user perception in 3d asymmetric telepresence. *IEEE Transactions on Visualization and Computer Graphics 27*, 11 (2021), 4129–4139. doi:10.1109/TVCG.2021.3106480. 1, 6

[YLL*20] YAO Y., LUO Z., LI S., ZHANG J., REN Y., ZHOU L., FANG T., QUAN L.: Blendedmvs: A large-scale dataset for generalized multi-view stereo networks. *Computer Vision and Pattern Recognition (CVPR)* (2020). 6

[YZG*21] YU T., ZHENG Z., GUO K., LIU P., DAI Q., LIU Y.: Function4d: Real-time human volumetric capture from very sparse consumer rgbd sensors. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR2021)* (June 2021). 1, 2, 6

[ZPK18] ZHOU Q.-Y., PARK J., KOLTUN V.: Open3D: A modern library for 3D data processing. *arXiv:1801.09847* (2018). 2

[ZPvBG01] ZWICKER M., PFISTER H., VAN BAAR J., GROSS M.: Surface splatting. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques* (New York, NY, USA, 2001), SIGGRAPH '01, Association for Computing Machinery, p. 371–378. doi:10.1145/383259.383300. 2