Article

Real-Time Interpolated Rendering of Terrain Point Cloud Data

Jaka Kordež [1], Matija Marolt [1], and Ciril Bohak [1,2,*]

[1] University of Ljubljana, Faculty of Computer and Information Science
[2] King Abdullah University of Science and Technology
* Correspondence: ciril.bohak@kaust.edu.sa

Abstract: Most real-time terrain point cloud rendering techniques do not address the empty space between the points but rather try to minimize it by changing the way how the points are rendered by either rendering them bigger or with more appropriate shapes such as paraboloids. In this work, we propose an alternative approach to point cloud rendering, which addresses the empty space between the points and tries to fill it with appropriate values to achieve the best possible output. The proposed approach runs in real time and outperforms several existing point cloud rendering techniques in terms of speed and render quality.

Keywords: point clouds; interpolation methods; terrain rendering

Three-dimensional acquisition sensors allow us to easily capture diverse shapes into a digital model resulting in a set of points, called a point cloud, that lie on the surfaces of objects. Such scanners are used in many different scenarios *e.g.*, object acquisition [1,2], acquisition of environment for self-driving car navigation to improve environmental awareness [3], autonomous planetary exploration [4] or environment acquisition for geographic and geodetic purposes with airborne sensors [5]. The best way to efficiently show this kind of data is by rendering it on screen. This can be done using different rendering techniques, but one should aim for an ideal rendering output, which would produce an image indistinguishable from a photograph taken from the same point as the point cloud data was acquired. On the other hand, rendering such data in real-time is often desired, even if this means sacrificing quality.

This is even more true when the data is visualized using limited resources such as on mobile devices or in the web browser. For such purposes, developing dedicated techniques that balance the rendering speed and quality is meaningful. Besides the image quality, rendering time is also crucial. 3D sensors on self-driving cars can produce a new point cloud every 50 milliseconds. Because of other moving objects (nearby traffic, pedestrians, *etc.*) and the fact that the car itself is moving through space, the new point cloud can be very different from the previous one. If we want to render the data in real time, an efficient algorithm is needed that doesn't require any time-consuming preprocessing steps.

In our case, we are interested in rendering terrain point cloud data acquired using LiDAR [2] sensors mounted on airplanes while flying over the desired part of the terrain. Such datasets consist of billions of points that store different parameters, such as the intensity of the returned signal, return number (one signal can be reflected from multiple targets), time of acquisition, and other data. For visualization purposes, LiDAR data is usually coupled with color information obtained from the ortho-photo data, which makes direct point cloud rendering of terrain data much more meaningful. Due to the amount of data available in LiDAR datasets, the need for fast real-time point cloud rendering techniques is in high demand.

The main contribution of this work is a novel adaptive real-time point cloud rendering technique that identifies which points are on the frontmost surfaces using a depth map and filters out the distant points. We evaluate our method by comparing it with several existing point cloud rendering techniques regarding rendering speed and image quality.

 doi:10.20944/preprints202211.0508.v1

1. Related Work

1.1. Point cloud rendering

The most straightforward way to render point cloud data is to draw points on the canvas using fixed-sized objects. A comparison of such techniques was made by Sainz and Pajarola [6]. These objects can be squares, circles, or any other shape. The only thing we need to consider when doing that is their order, which means starting with the most distant ones and drawing closer ones over them. This approach is straightforward to implement as most graphical libraries already support rendering with fixed-sized objects, and the ordering is already an integral step in the graphical pipeline. Rendering an extremely large point cloud in real time using different dedicated acceleration structures and exploitation of rendering hardware architecture was researched by Schütz *et al. et al.* [7–11]. While this is an appropriate way to render wast amount of points, the approaches do not address how to fill the empty space between them.

Fixed-sized objects approach can be improved by bending the edges of fixed size away from the camera. The result is a parabolic shape that ensures correct occlusions when points are very close to each other, as was presented by Schütz and Wimmer [12] on 3D scanned point cloud data. If the objects are large enough to fill the empty regions, the result will be the same as with the nearest neighbor algorithm. Furthermore, the researchers have used transparent splats [13] in combination with deferred rendering [14]. One of the ways to fill the gaps between the primitives is to use elliptically weighted averaging [15,16], which also prevents noise in areas with higher point densities.

In its basic form, the nearest neighbor algorithm searches each pixel's surroundings to find the nearest projected point. The images produced by this technique are called Voronoi diagrams. They never have any holes between the points, but the edges are sharp, and the method doesn't consider the point's depth. Implementation can be done by following the basic definition, but that turns out to be very inefficient due to intense texture sampling, which is slow on graphics cards. A faster way is using the JumpFlooding algorithm [17] that computes the image in a logarithmic number of steps with respect to the maximal image dimension. A smaller number of steps may even be used if we know that the points are close enough to each other.

To smooth the sharp edges in the Voronoi diagram, the inverse distance weighting interpolation combines the colors of several nearby points. The color of a particular pixel is a weighted sum of all colors in the point cloud. The weights are usually calculated as inverses distances to the points. The distance may also be raised to the power of the parameter p. If p is a small number, distant points will have similar weight to the near ones, and the picture will be blurry. On the other hand, a very big value of p reduces the influence of distant points, and the image gets similar to the Voronoi diagram.

Unlike inverse distance weighting, natural neighbor interpolation presented by Tsidaev [18] uses only the values of the nearest points around the pixel. To be precise, it takes only those points from which the pixel would remove some area if added to the Voronoi diagram as a new point. The colors are weighted with one of the two defined equations. The Sibson weights are equal to the area removed from each point, and the Laplace weights consider the length of the border in the removed region in combination with the distance to the point.

The above-mentioned space-filling methods give adequate results and are the ones our approach builds upon and aims to improve.

Another way to approach the sparseness of points is to upsample the point clouds. Zhang *et al.* [19] present a progressive method for point cloud upsampling via differentiable rendering, which addresses the non-uniform point distribution within the point cloud and is capable of learning local and global point features to cope with non-uniform point distribution and outlier removal. Yu *et al.* [20] present PU-Net, a data-driven point cloud upsampling approach on point patches capable of learning multi-level point features and expanding a set of points using a multi-branch convolution unit implicitly in feature space. These features are split into a new set of features used for upsampling the point set.

 doi:10.20944/preprints202211.0508.v1

Li *et al.* [21] present PU-GAN, a point cloud upsampling approach that uses a generative adversarial network to learn a variety of point distributions in the latent space and uses this information to upsample points point patches of surfaces. Qian *et al.* [22] present PUGeo-Net, a geometry-centric network for 3D point cloud upsampling, which uses discrete differential geometry and incorporates it into deep learning by learning the first and second fundamental forms for full representation of the local geometry unique up to rigid motion. Li *et al.* [23] present a method for point cloud upsampling via disentangled refinement, which uses two cascaded sub-networks, a dense generator for coarse but dense surface output and a spatial refiner for further fine-tuning the individual point location using local and global refinement units.

While rendering denser point clouds gives us better results, it also affects the rendering performance. In many cases of rendering aerial point cloud data, we are already tackling hundreds of millions or even billions of points. Since the mentioned methods do not offer real-time point cloud upsampling, this requires extensive preprocessing and affects the rendering performance significantly.

Point cloud reconstruction aims to reconstruct an implicit or explicit surface representation of the data for rendering purposes. Moreover, the point cloud acquisition methods usually return noisy point cloud representations of the acquired data. Researchers have used different approaches to tackle these problems. Mandikal *et al.* [24] present DensePCR, a deep hierarchical model for predicting point clouds of increasing resolution with architecture that predicts a low-resolution point cloud first and afterward hierarchically increases its resolution by aggregating local and global point features for grid deformation, yielding uniform, dense and accurate point clouds. Tachella *et al.* citeTachella2019 presented a real-time 3D reconstruction from a single-photon LiDAR data using point cloud denoisers by combining statistical models with highly scalable computational tools from the computer graphics community and demonstrating 3D reconstruction of complex outdoor scenes. Luo and Hu [25] present an approach for learning the underlying manifold of a noisy point cloud from differentiably subsampled points with trivial noise perturbation and their embedded neighborhood feature, aiming to capture intrinsic structures in point clouds. An autoencoder-like network encoding stage learns both local and global feature representations of points. It samples them with low noise using adaptive differentiable pooling operation, and in decoding, infers the underlying manifold by transforming each sampled point along with the embedded features of its neighborhood to a local surface centered around the point. Choe *et al.* [26] presented a deep point cloud reconstruction network that simultaneously solves point cloud sparseness, irregularity, and noisiness by first using a 3D spars stacked-hourglass network for the initial densification and denoising. Next, refines the result using transformers to convert discrete voxels into 3D points.

The above methods give good results but have limitations. Most of them are not capable of real-time denoising and reconstruction, and the one which is, was only tested on low field-of-view scenes. Moreover, most of the above methods were designed for specific domains and do not guarantee good results for general point cloud data.

An alternative approach is to use indirect techniques for rendering point clouds, which first transform point clouds into an alternative representation, such as mesh geometry of signed distance fields, and then render the newly obtained representation. Berger *et al.* [27] present an overview of surface reconstruction methods up to 2014, which was recently revised by Camuffo *et al.* [28]. Researchers have developed numerous approaches for point cloud reconstruction and completion by empowering deep learning. Here we briefly present a few selected works which tackle the point cloud reconstruction and completion problem. Han *et al.* [29] present a high-resolution shape completion approach that uses deep neural networks for global structure and local geometry inference. Yuan *et al.* [30] present a point completion network that directly operates on raw point clouds without structural assumption or annotation about the underlying shape and features a decoder design that enables the generation of fine-grained completion using a small number of parameters. Wen *et al.* [31] present a point cloud completion approach that uses a skip-attention network

 doi:10.20944/preprints202211.0508.v1

with hierarchical folding. Williams *et al.* [32] present a neural kernel fields approach for reconstructing implicit 3D shapes based on a learned kernel ridge regression. The method works on a variety of point clouds representing single objects or big scenes

The above methods achieve great results but are mostly not real-time methods which again requires extensive preprocessing before real-time rendering is possible.

1.2. Web-based point cloud rendering

Several approaches have tackled the possibility of web-based point cloud rendering. Kuder and Žalik [33] have presented a web-based rendering approach for point-based rendering with client-server design, where original LAS data is converted into quadtree representation and a RESTful web server for serving portions of data on different level-of-detail (LoD), mimicking the Web Map Server design. A similar approach was presented by Evans *et al.* [34], which supports more point cloud formats and utilizes different acceleration structures. Schütz [35] presented an optimized and modular web-based rendering framework that introduced multiple rendering approaches, including paraboloid rendering of massive point cloud data, which does not need a specialized server but includes a data preparation tool for converting the data into an appropriate format for streamed loading of data with LoD support. Discher *et al.* [36] presented a scalable WebGL-based approach for visualizing massive 3D point clouds using semantics-dependent rendering techniques for the joint rendering of 2D and 3D geodetic data as well as client-side or server-side rendering.

The presented approaches cover diverse use cases but require either additional server-side functionality or extensive preprocessing of data. Our implementation was done using RenderCore[1] - a WebGL 2.0 web-based rendering framework with extensions for processing the point cloud data.

2. Background

This section briefly introduces the data point cloud data used for testing different rendering methods and commonly used real-time point cloud rendering techniques. We compared the results of our method with the results of these methods to show the advantages and disadvantages of each technique.

2.1. Data

The data used in our experiments is publicly available from the Ministry of the Environment and Spatial Planning of Slovenia and consists of multiple datasets. In the presented work, we use LiDAR point cloud data of the Slovenian landscape, orthophoto images of Slovenia, and the digital terrain model.

The LiDAR point cloud dataset of Slovenian landscape[2] was acquired using the RIEGL LMS-Q780 laser scanner, the IGI Aerocontrol Mark II.E 256Hz IMU system and the Novatel OEMV-3 GNSS positioning system at altitudes of 1200 to 1400 m above ground. The postprocessing of the acquired data is presented in-depth in the acquisition report [37]. The data was acquired in 2014 and 2015. On average, the dataset contains 5 points (first return of LiDAR sensor) per m^2. The data is georeferenced in both the Gauß-Krüger coordinate system D48/GK and in the geodetic datum 96 coordinate system D96/TM.

Orthophoto images were obtained from *Portal eProstor*[37] - a repository of publicly available Slovenian geodetic data. The images are similarly georeferenced as the LiDAR data. They are available through a WMTS/REST service as small image tiles of 256 $\times$ 256 pixels on 17 scale levels and with a resolution of 0.5$\times$0.5 m per pixel on the largest scale.

[1] https://github.com/UL-FRI-LGM/RenderCore
[2] http://gis.arso.gov.si/evode/

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 28 November 2022 doi:10.20944/preprints202211.0508.v1

The digital terrain model was also obtained from the database of Slovenian public data[3]. The data is stored as a regular grid with a resolution of 1 m, with the corresponding heights with an accuracy of 0.1 m.

2.2. Fixed point size rendering

The simplest point cloud rendering method renders points as fixed-sized primitives such as squares, circles, or other planar primitives. Implementation is simple and straightforward since most graphics APIs already include such rendering. This is also true in our case where we used WebGL 2.0, which supports rendering of primitives `GL_POINTS`, with the ability to set its size through parameter `GL_PointSize` in the vertex shader. The API draws such points as rectangles, and their overlapping is handled using a depth test built into the graphics pipeline. When we want to render circles, we can discard the fragments outside the desired radius in the fragment shader. Their distance defines the size of the objects we see, and we can implement a similar transformation also in the rendering process. This is why the primitives closer to the viewer are usually rendered larger than the primitives further away. Despite setting the primitive size in accordance with their distance, the problem of non-uniform point cloud density still results in empty regions – holes in the sparsest parts of the point cloud. The following methods try to address these problems in different ways.

2.3. Nearest neighbor rendering

Pixels in the resulting image in the color of the background are those for which no point from point cloud data is mapped into. One of the most straightforward approaches to approximate its color is by using the color of its nearest neighbor – the point which is close to the pixel in the resulting image. The resulting image of such an approach is called the Voronoi diagram. We can obtain such a result by first rendering the points we have and then iteratively coloring the remaining points in the resulting image with the color of their closest neighbor. Due to the specific hardware architecture of graphical processing units (GPUs), this is not an optimal approach. Such a method requires many texture sampling steps, which makes it inefficient. In the case of rendering an image with resolution 256×256 pixels, this would result in $256^2 = 4,294,967,296$ texture samplings. Several approaches tried to address this problem; one of the most commonly used is Jump Flooding [17,38].

The jump Flooding algorithm builds the Voronoi diagram for an image in $\lceil log_2(d) \rceil$ steps, where d is the maximal image dimension. In the case of the previously mentioned image of resolution 256×256 pixels, this results in 8 steps or $8 \times 256^2 * 8 = 524,288$ texture samplings, where the first 8 corresponds to the number of steps (jumps) and the second 8 to the neighboring pixels. Jump length is determined as 2^{k-i}, where k is the number of all steps and i to *i-th* step. In the case above, this results in jump lengths of 128, 64, ..., 4, 2, and 1. We show how the Jump Flooding algorithm works on a square image of resolution 16×16 pixels in Figure 1.

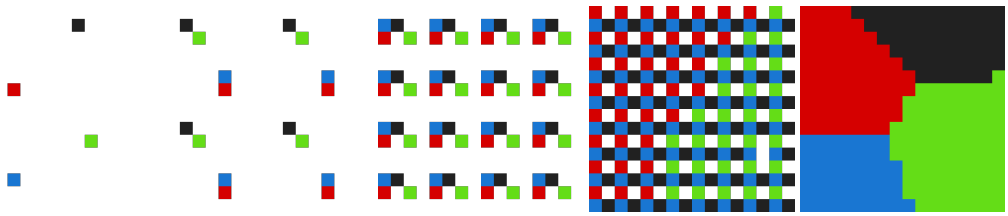


Figure 1. From left to right: consecutive steps of Jump Flooding algorithm for image of resolution 16×16 pixels.

[3] https://ipi.eprostor.gov.si/jgp/menu

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 28 November 2022 doi:10.20944/preprints202211.0508.v1

In WebGL, Jump Flooding can be implemented using multiple rendering passes, where first, we render the point cloud with one pixel-sized point and set the background as transparent. In the following render passed, we render one step of the Jump Flooding algorithm at a time. We store the coordinates of the nearest neighbors in additional texture after each step. This texture, together with the original point cloud rendering, is passed into the next rendering pass. In the last pass, we generate the Voronoi diagram by rendering the nearest neighbor for every uncolored pixel in the image.

2.4. Inverse distant weighted rendering

While nearest neighbor rendering fills all the empty spaces between points, the resulting image is not very precise. The edges between regions are sharp, which makes the results less convincing. Alternatively, the Inverse distance weighting [39] tries to address this by taking into account several neighbors applied to spatial risk distribution in contaminated site assessment. In the case of point cloud rendering, this scenario is mapped to the color of points and their distribution in the empty space between the points. The smaller the distance, the greater the influence of the neighboring point color is. The color of an arbitrary point in the image is defined as:

$$f(p) = \sum_{i=1}^{n} \frac{f(x_i)}{d(p, x_i)^e}, \tag{1}$$

where n is number of points, $f(x)$ is point color, $d(x, y)$ is distance between x and y, and e is weighting parameter. The greater the e, the higher the influence of nearby points is. By increasing the e, the output image gets closer to the nearest neighbor rendering. The smaller the e, the higher the influence of all neighboring points, resulting in a more blurred output image.

2.5. Natural neighbor rendering

The downside of the Inverse distant weighted rendering is that you need to consider all the points for calculating arbitrary pixel color. This means that rendering denser point cloud regions is substantially more computationally demanding than rendering sparser regions. The Natural neighbors method [40] only uses a few nearest neighbors to calculate the color of a selected pixel. These are the points by which the Voronoi diagram cell surface area gets affected. The color of a point is calculated as:

$$f(p) = \sum_{i=1}^{n} w(x_i, p) * f(x_i), \tag{2}$$

where n is number of points in the image, $f(x)$ is point color, and $w_i(x)$ is its weight. An efficient implementation can build on top of Jump Flooding by approximating the portions of surface area the selected pixel would take from each neighboring point in the Voronoi diagram. The approximation can be achieved by sparsely sampling the changed portion of the Voronoi diagram as is shown in Figure 2

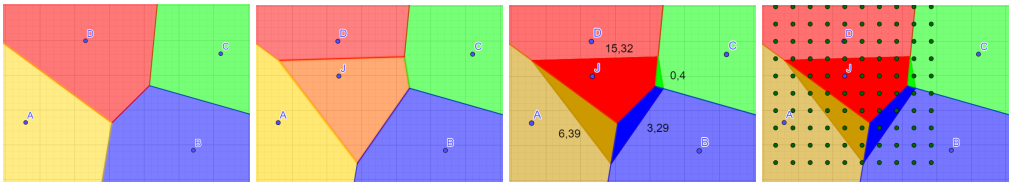


Figure 2. From left to right: Original Voronoi diagram, updated Voronoi diagram with new point, surface area portions taken by the new point, and sparse sampling of affected part of Voronoi diagram for estimating the surface area portions.

 doi:10.20944/preprints202211.0508.v1

2.6. Paraboloid rendering

None of the above-presented interpolation rendering methods take into account the depth of points and their overlapping. Even when rendering the points with different-sized primitives, both distant and near points are treated similarly in the rendering process, apart from their size, resulting in rendering the points which should be occluded. We can address this problem by rendering with fixed-sized primitives, curving them away from the viewer and thus turning them into curved surfaces. The final result ideally resembles the Voronoi diagram but additionally implicitly takes into account the occlusion of points. One can use different functions to curve back the edges of the primitives, but most commonly, the quadratic function is used, resulting in a parabolic shape – paraboloid. This is one of the most widely used approaches and is also used in the Potree web-based point cloud visualization framework [35].

In the development of our own approach, we identified two main problems with the existing methods:

1. efficient implementation on the graphical processing unit, and
2. unwanted rendering of distant points which should have been occluded.

The first problem is best addressed with Inverse distance weighted rendering (see subsection 2.4), which can be efficiently implemented. On the other hand, we can almost entirely suppress the calculation errors, which appear as color noise, by selecting appropriate parameters. The Paraboloid rendering method best addresses the second problem (see subsection 2.6).

3. Methods

In the previous section, we made a general review of the point cloud rendering techniques. We identified the two main challenges when using interpolation for real-time rendering: (1) the removal of occluded points from the scene and (2) the efficient blending of remaining points into a clear image. Based on the experience from observing the implemented methods, we constructed a new algorithm that combines some ideas from the presented methods to achieve better performance. We present the idea and the algorithms in the following sections.

Our proposed method for rendering point clouds comprises three steps:

1. a depth map of occlusion points is prepared,
2. filtering occluded points,
3. rendering the points as splats which are combined into the final image,

which are presented in more detail in the following subsections.

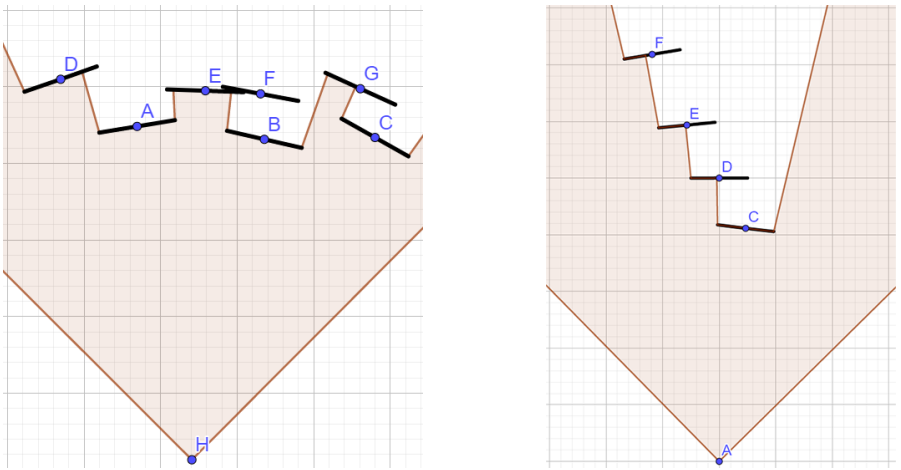
3.1. Depth map calculation

A depth map is an image that contains depth information about the rendered scene. The color of each pixel indicates the distance to the first object in that direction. WebGL automatically produces it with the main result (usually a colored image). We can use it to filter occluded points from the scene by comparing the distance to each point with the depth map value in the same direction. A smaller depth map values mean that something else is in the way and the point should be removed.

The main challenge when producing a good depth map is using the best shape and size of primitives. Since there is a limited number of points in the point cloud, the surfaces aren't completely covered. Points need to be rendered as larger objects to fill the holes between them and produce a depth map with solid surfaces, as shown in Figure 3a. Any surface gap could cause occluded points leakage as the filtering algorithm wouldn't remove them. On the other hand, the primitives also shouldn't be too large as they could cover smaller holes and remove essential details from the image or occlude a whole chain of points as shown in Figure 3b.

Our method uses default square-shaped primitives to generate the depth map. Their size only depends on their distance from the camera, so points that are further away from

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 28 November 2022 doi:10.20944/preprints202211.0508.v1



(a) Points F and G are filtered out, since they are occluded by points B in C.

(b) A side view on the surface defined with points C – F. Point C occludes point D, which in turn occludes point E, which in turn occludes point F.

Figure 3. Point filtering with use of depth map.

the viewer are smaller. To avoid covering smaller holes, fragment depth is increased towards the edges. Because the increase is based on a quadratic function, the primitives are shaped as paraboloids.

3.2. Filtering

The depth map obtained in the previous step is used to filter out the occluded points in the point cloud. The filtering is done on points within the rendering frustum in the vertex processing stage, where the occluded points are simply discarded by moving them out of the viewing frustum, thus ignoring them in the fragment processing stage of the rendering.

3.3. Rendering with splats

The third part of the method focuses on incorporating the remaining points into a final image. This can be achieved in many ways using different approaches. Most use interpolation techniques to calculate values of empty pixels based on nearby points. These approaches often require a lot of texture sampling to find those nearby points, which is very inefficient on most GPU architectures. A different approach must be used to achieve our real-time rendering goal.

Our algorithm tries to approximate the inverse distance weighting interpolation and can be described with the following equations:

$$f(p) = \frac{\sum_{i=1}^{n} w(p,x_i) * f(x_i)}{\sum_{i=1}^{n} w(p,x_i)}, \tag{3}$$

$$w(p,x_i) = \frac{1}{d(p,x_i)^e}, \tag{4}$$

where Equation 3 is an adaptation of the Inverse distance weighted rendering presented in subsection 2.4 with Equation 1, weighted using Equation 4.

Instead of searching every pixel's neighborhood for points, it adds points to the canvas as splats and calculates their weighted averages in the final step. Splats are modeled as square primitives, and their weight is decreasing relative to the distance from the point.

To achieve optimal results, Gaussian probability density is defined as:

$$g(x) = \frac{1}{\sigma\sqrt{2 * \pi}} * \exp\left(-\frac{1}{2} * \frac{x^2}{\sigma^2}\right). \tag{5}$$

 doi:10.20944/preprints202211.0508.v1

It was used instead of an exponential function in the Equation 4. The high values around the center produce a sharp image, while values for $x \gg \sigma$ and $x \ll -\sigma$ fill distant holes between points. The variance (σ^2) can be used as a parameter to determine how sharp the result will be. If this parameter is low, the curve will descend quickly, and the final result will be similar to the nearest neighbor interpolation. On the other hand, high values will cause more overlaps of colors and effectively blur the image.

Our method uses a normalized Gaussian curve, where $\frac{1}{\sigma\sqrt{2\pi}}$ is removed from the Equation 5. The weight in the center is consequently always equal to 1 and is not dependent on the variance.

The canvas is represented as a 4-dimensional floating-point texture in WebGL. The first three dimensions keep the color information, while the fourth dimension keeps the sum of weights. To ensure correct calculation, the colors are multiplied with the weight before adding them to the canvas. Adding is done on the GPU with blending (using the *GL_FUNC_ADD* function). Parameters $S = 1$ and $D = 1$ ensure that all splats are added to the texture unmodified. This part calculates the numerator part of Equation 1.

The last rendering step calculates the final color of each pixel by simply dividing each color component with the sum of the weights (denominator part of Equation 1).

The resulting images from the individual steps of the proposed method are displayed in Figure 4.

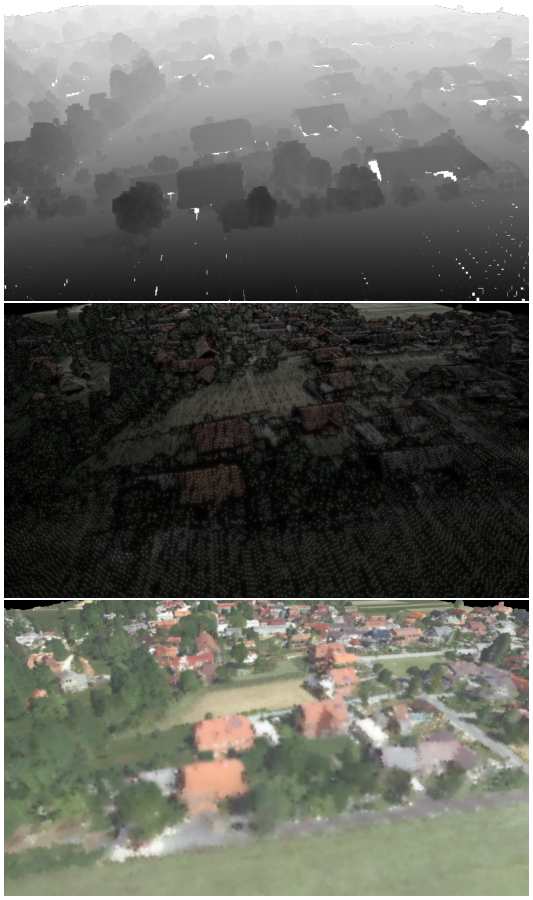


Figure 4. From left to right: step 1 – depth map of occlusion points, step 2 – filtering occluded points, step 3 – final rendering using splatting.

4. Results

To evaluate our rendering algorithm, we measured its performance in terms of output quality and rendering time. Results were compared to other popular interpolation-based rendering techniques. All tests were conducted on six scenes, always rendered from the

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 28 November 2022 doi:10.20944/preprints202211.0508.v1

same angle. The scenes were taken from the results of the Slovenian national surface LiDAR dataset [37]. Several locations were picked from the dataset, cropped, and filtered to reduce their size. Color information was added using LiDAR - Orthophoto Merging software[4].

The test cases were selected so that all the challenges related to point cloud rendering would be covered equally. The scenes contain flat surfaces with complex textures to test texture reconstruction, as well as overlapping objects (e.g., buildings and mountains) to test the rendering of occluded points. The tests were conducted on a consumer-grade computer with an AMD Ryzen 1700 processor, 16 GB of memory, and an NVidia GeForce RTX 2070 graphics card with 8 GB of memory.

4.1. Performance evaluation

As the algorithm was implemented using WebGL 2.0. We used Google Chrome and its developer tools to measure rendering time. We used the performance profiler to measure the execution of the rendering function as well as the GPU utilization. The measured values were averaged over multiple consecutive frames for all six test scenes. The average results are available for each tested algorithm in Table 1.

4.2. Qualitative evaluation

Our test models were obtained by scanning the terrain using LiDAR technology, so we didn't have any mesh models or photos for reference. We used a 2D Delaunay triangulation to reconstruct mesh models, render them and use those images as our targets. A simple C# program was used to project the points on a horizontal plane by removing their vertical coordinate. The algorithm from the *DelaunatorSharp*[5] library connected those points with triangles. In the end, the removed coordinate was returned to the points, and the model was saved as an OBJ file. An example scene rendered with all of the rendering methods is shown together with the close-up in Figure 5. Some additional comparisons are shown in Appendix, Figure A1.

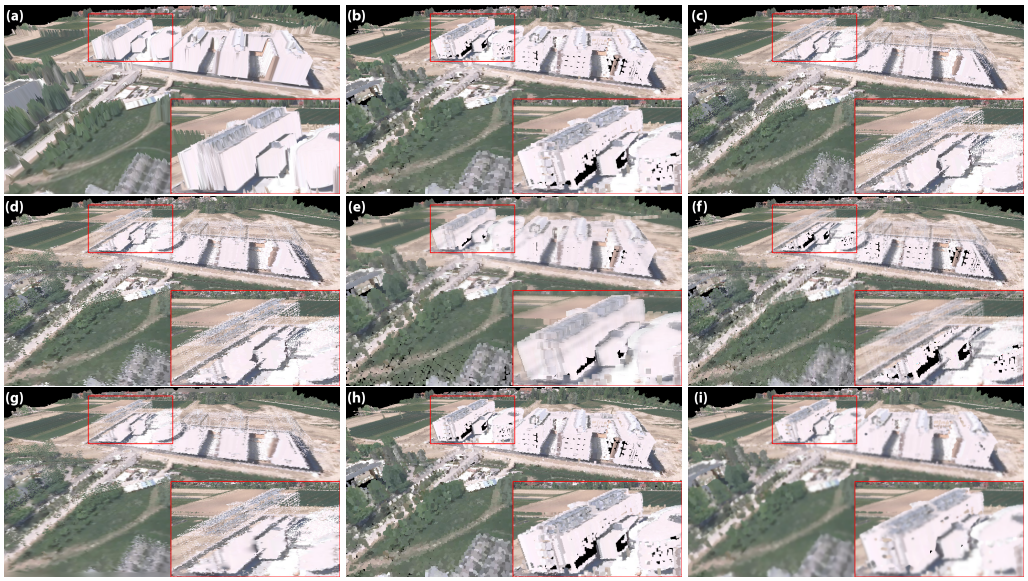


Figure 5. Rendering results of the presented rendering methods: (a) mesh, (b) fixed points, (c) nearest neighbor, (d) jump flooding, (e) inverse distance weighted, (f) inverse distance weighted with splats, (g) natural neighbor, (h) paraboloids, and (i) our approach.

The similarity between the algorithm output and the reference image was calculated using three metrics:

[4] https://github.com/jakakordez/lidar-orto-photo
[5] https://github.com/nol1fe/delaunator-sharp

 doi:10.20944/preprints202211.0508.v1

Learned Perceptual Image Patch Similarity (LPIPS) metric presented by Zhang et al. [41] is used to estimate the perceptual similarity between two images and essentially computes the similarity between the activations of two image patches for some predefined network, and it matches human perception well. A low LPIPS score means that images are more perceptually similar.

Peak Signal-to-Noise Ratio (PSNR) metrics expresses a ratio between the maximum possible value of a signal and the power of distorting noise that affects the representation quality. A high PSNR score means that the two images are more similar.

Structural Similarity Index Measure (SSIM) [42] is a perceptual metric for quantifying image quality degradation caused by processing, data compression, or transmission losses. A high SSIM score means that images are more perceptually similar.

For all six test scenes, we calculated all of the above metrics between the rendering output from a particular method and the target image produced by the mesh rendering method. The average score values for each metric are presented in Table 1. The detailed results for each technique and scene are available in Appendix, subsection A.2.

Table 1. Results of all implemented algorithms with values averaged across all test cases.

Approach	Performance		Quality		
	Time ↓	GPU ↓	LPIPS ↓	PSNR ↑	SSIM ↑
Rendering with fixed points	**0.2 ms**	**0.2 ms**	0.135	18.452	0.491
Nearest neighbors interpolation	11 ms	9.1 ms	0.165	18.290	0.526
Jump Flooding algorithm	132 ms	108 ms	0.164	18.256	0.524
Inverse distance weighting	233 ms	232 ms	0.145	19.033	0.495
Inverse distance weighting with splats	5.2 ms	2.7 ms	0.233	17.344	0.438
Natural neighbors interpolation	188 ms	161 ms	0.137	18.935	0.577
Rendering with paraboloids	0.3 ms	**0.2 ms**	0.139	18.541	0.497
Our approach	5.3 ms	3.0 ms	**0.091**	**21.475**	**0.655**

5. Discussion

The performance evaluation shows that our approach is slower than the two fastest methods (rendering with fixed points and paraboloids) and has similar performance as inverse distance weighted rendering with splats. Still, the method is a real-time rendering method with more than 180 frames-per-second performance on the test scenes. On the other hand, our approach has the best similarity metrics scores on average. While the comparison to the mesh representation of the point cloud data does not perfectly reflect how good the reconstruction is for non-top-down views, it is the best approximation we could obtain using the available data. The images show that our method fills most of the holes in the point clouds, resulting in a better-reconstructed scene.

Despite promising results, our method could use further improvements. The most noticeable flaw is observed when the camera moves around the scene. The depth image that the filtering part of the algorithm uses, can still have smaller holes. Consequently, some points from the background may become visible for a short time as the camera moves. That causes a flickering effect on the screen, which should be addressed in the future. Similar flickering is also present in some of the other presented methods.

The presented method is especially meaningful when the point cloud data is less flat (e.g., in mountains and residential areas). In these scenes, our method outperforms others even more than in flat scenes.

One downside of our method is that some regions in the resulting images are blurrier than with other methods; the effect is more apparent in scenes with smaller objects and lower densities of point clouds. This could be solved to some degree with the use of a post-processing sharpening filter.

While newer deep-learning-based methods, which were not assessed in this paper, undoubtedly produce better results, the inference step is still far from real-time. And until

 doi:10.20944/preprints202211.0508.v1

new hardware and deep model optimization enable us to run inference of such model in real time, the presented method is still very relevant for point cloud rendering.

6. Conclusions

During this study, we managed to develop our own algorithm for real-time point cloud rendering. The algorithm is able to filter occluded points from the scene and fill empty spaces on the canvas with colors from nearby points. Other interpolation-based algorithms were also implemented for performance and qualitative comparison. All algorithms were evaluated on a set of six test scenes. Reference renderings were also made from mesh reconstructions of those models. The results show that the presented point cloud rendering method is capable of real-time rendering of point cloud data and outperforms other popular real-time or interactive point cloud rendering methods.

The method could be further improved with a better filtering step, which would use some kind of density map. A density map is an image of the scene with higher pixel values where points are projected closer one to another. Such a map can be efficiently generated using the same technique as the color blending part of our method. If the render pass that is generating the depth image would use this density map to increase the size of paraboloids where density is low, it could potentially fill all empty spaces and therefore give better results. Additionally, when the computational power of new hardware will enable real-time inference of deep models, the method could be adapted in such a way, that the filtered points would be sent to a deep model, which would then infer the output image. An additional problem that might arise is how to address the temporal coherence of such models throughout the sequential frames.

Author Contributions: Individual authors contributed to the research presented in this paper as follows: "Conceptualization, C.B. and J.K.; methodology, C.B.; software, C.B. and J.K.; validation, C.B., J.K. and M.M.; formal analysis, C.B. and M.M.; investigation, C.B. and J.K.; resources, C.B. and J.K.; data curation, C.B. and J.K; writing—original draft preparation, C.B., J.K and M.M.; writing—review and editing, C.B.; visualization, C.B. and J.K.; supervision, C. B. and M.M.; project administration, C.B.

Informed Consent Statement: Not applicable.

Data Availability Statement: Not applicable.

Acknowledgments: Authors would like to thank Slovenian Environmental Agency for providing the data which was used in the experiments.

Conflicts of Interest: Declare conflicts of interest or state "The authors declare no conflict of interest."

Abbreviations

The following abbreviations are used in this manuscript:

LiDAR	Laser Imaging, Detection, and Ranging
LoD	Level-of-Detail

Appendix A

Appendix A.1 Additional rendering results

In this section, we show additional rendering results using the presented methods from two scenes in Figure A1.

Appendix A.2 Detailed qualitative evaluation results

In this section, we present detailed results of our qualitative evaluation. We used three metrics for evaluating and comparing our approach to other popular point cloud rendering

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 28 November 2022 doi:10.20944/preprints202211.0508.v1

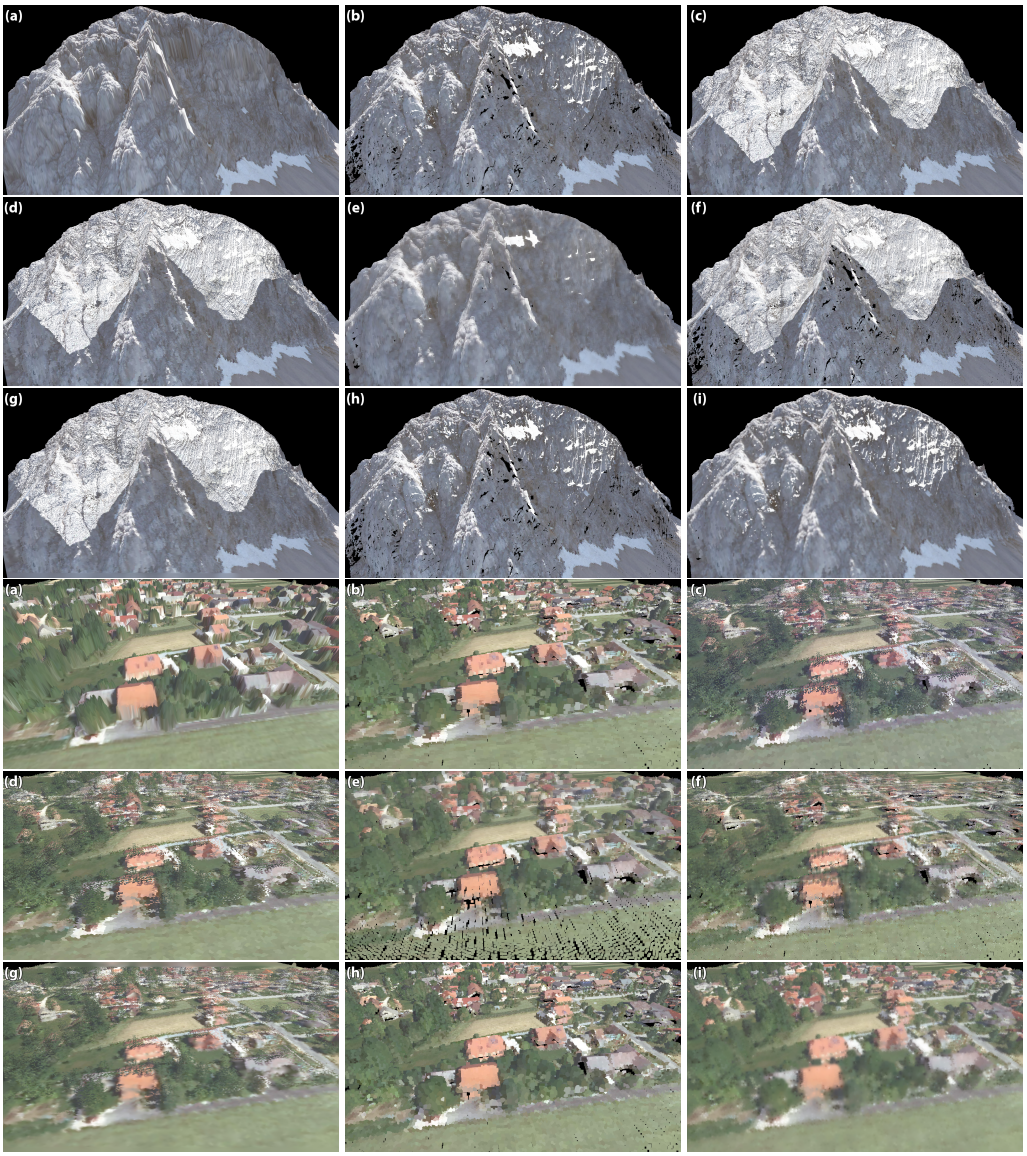


Figure A1. Rendering results of the presented rendering methods: (a) mesh, (b) fixed points, (c) nearest neighbor, (d) jump flooding, (e) inverse distance weighted, (f) inverse distance weighted with splats, (g) natural neighbor, (h) paraboloids, and (i) our approach.

techniques. We use LPIPS [41], PSNR [42], and SSIM [42] scores in our evaluation. In Table A1, Table A2, and Table A3 we present detailed results for all the test scenes.

The tested approaches with their acronyms are:

1. **FIXED** – Rendering with fixed points
2. **NEAR** – Nearest neighbors interpolation
3. **JF** – Jump Flooding algorithm
4. **IDW** – Inverse distance weighting
5. **IDW-S** – Inverse distance weighting with splats
6. **NAT** – Natural neighbors interpolation
7. **PARA** – Rendering with paraboloids
8. **OUR** – Our approach

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 28 November 2022 doi:10.20944/preprints202211.0508.v1

Table A1. LPIPS scores of the presented approaches compared with our own approach. Lower score means higher similarity.

Scene	FIXED	NEAR	JF	IDW	IDW-S	NAT	PARA	OUR
Scene #1	0.143	0.162	0.161	0.218	0.271	0.122	0.148	**0.092**
Scene #2	0.168	0.300	0.301	0.097	0.271	0.271	0.169	**0.095**
Scene #3	0.071	0.097	0.098	**0.064**	0.140	0.083	0.078	0.075
Scene #4	0.116	0.058	0.060	0.056	0.237	0.043	0.120	**0.036**
Scene #5	0.177	0.170	0.170	0.179	0.226	0.151	0.184	**0.149**
Scene #6	0.136	0.205	0.196	0.259	0.252	0.152	0.134	**0.096**
Mean	0.135	0.165	0.164	0.145	0.233	0.137	0.139	**0.091**

Table A2. PSNR scores of the presented approaches compared with our own approach. Higher score means higher similarity.

Scene	FIXED	NEAR	JF	IDW	IDW-S	NAT	PARA	OUR
Scene #1	18.548	18.902	18.934	18.644	17.996	19.819	18.657	**22.463**
Scene #2	17.900	12.763	12.773	**21.189**	13.600	12.921	17.881	19.317
Scene #3	18.617	19.397	19.360	18.917	18.679	20.383	18.197	**22.480**
Scene #4	22.930	26.565	26.305	24.669	21.614	27.332	23.680	**28.737**
Scene #5	14.611	14.587	14.587	14.708	14.480	14.748	14.572	**14.828**
Scene #6	18.107	17.527	17.579	16.070	17.694	18.407	18.263	**21.028**
Mean	18.452	18.290	18.256	19.033	17.344	18.935	18.541	**21.475**

Table A3. SSIM scores of the presented approaches compared with our own approach. Higher score means higher similarity.

Scene	FIXED	NEAR	JF	IDW	IDW-S	NAT	PARA	OUR
Scene #1	0.547	0.569	0.571	0.509	0.494	0.639	0.563	**0.715**
Scene #2	0.534	0.499	0.498	0.618	0.421	0.525	0.546	**0.696**
Scene #3	0.357	0.464	0.459	0.326	0.397	0.527	0.315	**0.562**
Scene #4	0.630	0.780	0.770	0.689	0.540	0.808	0.678	**0.844**
Scene #5	0.411	0.397	0.396	0.437	0.354	0.436	0.400	**0.467**
Scene #6	0.468	0.447	0.449	0.389	0.423	0.527	0.483	**0.644**
Mean	0.491	0.526	0.524	0.495	0.438	0.577	0.497	**0.655**

References

1. Rocchini, C.; Cignoni, P.; Montani, C.; Pingi, P.; Scopigno, R. A low cost 3D scanner based on structured light. Computer Graphics Forum. Wiley Online Library, 2001, Vol. 20, pp. 299–308.
2. Reutebuch, S.E.; Andersen, H.E.; McGaughey, R.J. Light detection and ranging (LIDAR): an emerging tool for multiple resource inventory. *Journal of forestry* **2005**, *103*, 286–292.
3. Hecht, J. Lidar for self-driving cars. *Optics and Photonics News* **2018**, *29*, 26–33.
4. Rekleitis, I.; Bedwani, J.L.; Dupuis, E. Autonomous planetary exploration using LIDAR data. 2009 IEEE International Conference on Robotics and Automation. IEEE, 2009, pp. 3025–3030.
5. Zhou, Q.Y.; Neumann, U. Complete residential urban area reconstruction from dense aerial LiDAR point clouds. *Graphical Models* **2013**, *75*, 118–125. doi:https://doi.org/10.1016/j.gmod.2012.09.001.
6. Sainz, M.; Pajarola, R. Point-based rendering techniques. *Computers & Graphics* **2004**, *28*, 869–879. doi:https://doi.org/10.1016/j.cag.2004.08.
7. Schütz, M.; Krösl, K.; Wimmer, M. Real-Time Continuous Level of Detail Rendering of Point Clouds. 2019 IEEE Conference on Virtual Reality and 3D User Interfaces (VR), 2019, pp. 103–110. doi:10.1109/VR.2019.8798284.
8. Schütz, M.; Mandlburger, G.; Otepka, J.; Wimmer, M. Progressive Real-Time Rendering of One Billion Points Without Hierarchical Acceleration Structures. *Computer Graphics Forum* **2020**, *39*, 51–64. doi:https://doi.org/10.1111/cgf.13911.
9. Otepka, J.; Mandlburger, G.; Schütz, M.; Pfeifer, N.; Wimmer, M. Efficient Loading and Visualization of Massive Feature-Rich Point Clouds Without Hierarchical Acceleration Structures. *The International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences* **2020**, *XLIII-B2-2020*, 293–300. doi:10.5194/isprs-archives-XLIII-B2-2020-293-2020.

10. Schütz, M.; Kerbl, B.; Wimmer, M. Rendering Point Clouds with Compute Shaders and Vertex Order Optimization. *Computer Graphics Forum* **2021**, *40*, 115–126. doi:10.1111/cgf.14345.
11. Schütz, M.; Kerbl, B.; Wimmer, M. Software Rasterization of 2 Billion Points in Real Time. *Proc. ACM Comput. Graph. Interact. Tech.* **2022**, *5*, 1–16.
12. Schütz, M.; Wimmer, M. High-quality point-based rendering using fast single-pass interpolation. 2015 Digital Heritage, 2015, Vol. 1, pp. 369–372. doi:10.1109/DigitalHeritage.2015.7413904.
13. Pfister, H.; Zwicker, M.; Van Baar, J.; Gross, M. Surfels: Surface elements as rendering primitives. Proceedings of the 27th annual conference on Computer graphics and interactive techniques, 2000, pp. 335–342.
14. Zhang, Y.; Pajarola, R. Deferred blending: Image composition for single-pass point rendering. *Computers & Graphics* **2007**, *31*, 175–189.
15. Zwicker, M.; Pfister, H.; Van Baar, J.; Gross, M. Surface splatting. Proceedings of the 28th annual conference on Computer graphics and interactive techniques, 2001, pp. 371–378.
16. Botsch, M.; Hornung, A.; Zwicker, M.; Kobbelt, L. High-quality surface splatting on today's GPUs. Proceedings Eurographics/IEEE VGTC Symposium Point-Based Graphics, 2005. IEEE, 2005, pp. 17–141.
17. Rong, G.; Tan, T.S. Jump Flooding in GPU with Applications to Voronoi Diagram and Distance Transform. Proceedings of the 2006 Symposium on Interactive 3D Graphics and Games; Association for Computing Machinery: New York, NY, USA, 2006; I3D '06, p. 109–116. doi:10.1145/1111411.1111431.
18. Tsidaev, A. Parallel algorithm for natural neighbor interpolation. Proceedings of the 2nd Ural Workshop on Parallel, Distributed, and Cloud Computing for Young Scientists, Yekaterinburg, Russia, 2016, Vol. 6.
19. Zhang, P.; Wang, X.; Ma, L.; Wang, S.; Kwong, S.; Jiang, J. Progressive Point Cloud Upsampling via Differentiable Rendering. *IEEE Transactions on Circuits and Systems for Video Technology* **2021**, *31*, 4673–4685. doi:10.1109/TCSVT.2021.3100134.
20. Yu, L.; Li, X.; Fu, C.W.; Cohen-Or, D.; Heng, P.A. PU-Net: Point Cloud Upsampling Network. Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2018.
21. Li, R.; Li, X.; Fu, C.W.; Cohen-Or, D.; Heng, P.A. PU-GAN: A Point Cloud Upsampling Adversarial Network. Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV), 2019.
22. Qian, Y.; Hou, J.; Kwong, S.; He, Y. PUGeo-Net: A Geometry-Centric Network for 3D Point Cloud Upsampling. Computer Vision – ECCV 2020; Vedaldi, A.; Bischof, H.; Brox, T.; Frahm, J.M., Eds.; Springer International Publishing: Cham, 2020; pp. 752–769.
23. Li, R.; Li, X.; Heng, P.A.; Fu, C.W. Point Cloud Upsampling via Disentangled Refinement. Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), 2021, pp. 344–353.
24. Mandikal, P.; Radhakrishnan, V.B. Dense 3D Point Cloud Reconstruction Using a Deep Pyramid Network. 2019 IEEE Winter Conference on Applications of Computer Vision (WACV), 2019, pp. 1052–1060. doi:10.1109/WACV.2019.00117.
25. Luo, S.; Hu, W. Differentiable Manifold Reconstruction for Point Cloud Denoising. Proceedings of the 28th ACM International Conference on Multimedia. Association for Computing Machinery, 2020, p. 1330–1338. doi:10.1145/3394171.3413727.
26. Choe, J.; Joung, B.; Rameau, F.; Park, J.; Kweon, I.S. Deep Point Cloud Reconstruction. Proceedings of International Conference on Learning Representations 2022, 2022.
27. Berger, M.; Tagliasacchi, A.; Seversky, L.; Alliez, P.; Levine, J.; Sharf, A.; Silva, C. State of the Art in Surface Reconstruction from Point Clouds. Eurographics 2014 - State of the Art Reports; , 2014; Vol. 1, *EUROGRAPHICS star report*, pp. 161–185. doi:10.2312/egst.20141040.
28. Camuffo, E.; Mari, D.; Milani, S. Recent Advancements in Learning Algorithms for Point Clouds: An Updated Overview. *Sensors* **2022**, *22*. doi:10.3390/s22041357.
29. Han, X.; Li, Z.; Huang, H.; Kalogerakis, E.; Yu, Y. High-Resolution Shape Completion Using Deep Neural Networks for Global Structure and Local Geometry Inference. Proceedings of the IEEE International Conference on Computer Vision (ICCV), 2017.
30. Yuan, W.; Khot, T.; Held, D.; Mertz, C.; Hebert, M. PCN: Point Completion Network. 2018 International Conference on 3D Vision (3DV), 2018, pp. 728–737. doi:10.1109/3DV.2018.00088.
31. Wen, X.; Li, T.; Han, Z.; Liu, Y.S. Point Cloud Completion by Skip-Attention Network With Hierarchical Folding. Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), 2020.
32. Williams, F.; Gojcic, Z.; Khamis, S.; Zorin, D.; Bruna, J.; Fidler, S.; Litany, O. Neural Fields As Learnable Kernels for 3D Reconstruction. Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), 2022, pp. 18500–18510.
33. Kuder, M.; Žalik, B. Web-Based LiDAR Visualization with Point-Based Rendering. 2011 Seventh International Conference on Signal Image Technology & Internet-Based Systems, 2011, pp. 38–45. doi:10.1109/SITIS.2011.36.
34. Evans, A.; Agenjo, J.; Blat, J. Web-Based Visualisation of on-Set Point Cloud Data. Proceedings of the 11th European Conference on Visual Media Production. Association for Computing Machinery, 2014. doi:10.1145/2668904.2668937.
35. Schütz, M. Potree: Rendering Large Point Clouds in Web Browsers. Master's thesis, Institute of Computer Graphics and Algorithms, Vienna University of Technology, Favoritenstrasse 9-11/E193-02, A-1040 Vienna, Austria, 2016.
36. Discher, S.; Richter, R.; Döllner, J. A Scalable WebGL-Based Approach for Visualizing Massive 3D Point Clouds Using Semantics-Dependent Rendering Techniques. Proceedings of the 23rd International ACM Conference on 3D Web Technology. Association for Computing Machinery, 2018. doi:10.1145/3208806.3208816.

 doi:10.20944/preprints202211.0508.v1

37. institute of Slovenia, G. izvedba laserskega skeniranja Slovenije. Blok 35 – Tehnično poročilo o izdelavi izdelkov. Technical report, Geodetic institute of Slovenia, 2015.
38. Rong, G.; Tan, T.S. Variants of Jump Flooding Algorithm for Computing Discrete Voronoi Diagrams. 4th International Symposium on Voronoi Diagrams in Science and Engineering (ISVD 2007). IEEE, 2007, pp. 176–181.
39. Yang, W.; Wang, R.S.; Huang, J.L.; Chen, Z.; Li, F. Application of inverse distance weighted interpolation method in contaminated site assessment. *The journal of applied ecology* **2007**, *18*, 2013–2018.
40. Boissonnat, J.D.; Cazals, F. Smooth surface reconstruction via natural neighbour interpolation of distance functions. *Computational Geometry* **2002**, *22*, 185–203.
41. Zhang, R.; Isola, P.; Efros, A.A.; Shechtman, E.; Wang, O. The Unreasonable Effectiveness of Deep Features as a Perceptual Metric. CVPR, 2018.
42. Wang, Z.; Bovik, A.; Sheikh, H.; Simoncelli, E. Image Quality Assessment: From Error Visibility to Structural Similarity. *IEEE Transactions on Image Processing* **2004**, *13*, 600–612. doi:10.1109/TIP.2003.819861.