POINT CLOUD RENDERING USING JUMP FLOODING

Renato Farias

Dissertação de Mestrado apresentada ao Programa de Pós-graduação em Engenharia de Sistemas e Computação, COPPE, da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Mestre em Engenharia de Sistemas e Computação.

Orientador: Ricardo Marroquim

Rio de Janeiro
Julho de 2014

POINT CLOUD RENDERING USING JUMP FLOODING

Renato Farias

RIO DE JANEIRO, RJ – BRASIL
JULHO DE 2014

Farias, Renato

Point Cloud Rendering Using Jump Flooding/Renato Farias. – Rio de Janeiro: UFRJ/COPPE, 2014.

XI, 43 p.: il.; $29,7$cm.

Orientador: Ricardo Marroquim

Dissertação (mestrado) – UFRJ/COPPE/Programa de Engenharia de Sistemas e Computação, 2014.

Referências Bibliográficas: p. 41 – 43.

1. point rendering. 2. raw point cloud. 3. jump flooding. I. Marroquim, Ricardo. II. Universidade Federal do Rio de Janeiro, COPPE, Programa de Engenharia de Sistemas e Computação. III. Título.

A minha mãe Vânia, que não pôde presenciar o fim desta jornada.

Agradecimentos

Primeiramente agradeço à minha família, que sempre me apoiou, nos momentos bons e nos ruins, me dando a base para minha formação pessoal.
Agradeço também ao meu orientador, por sua paciência e toda sua ajuda.
Por fim, agradeço ao Filippo Costanzo e à HyperReality pela parceria, e à Academia de España en Roma pela permissão de usar seus modelos para fins científicos.

Resumo da Dissertação apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc.)

POINT CLOUD RENDERING USING JUMP FLOODING

Renato Farias

Julho/2014

Orientador: Ricardo Marroquim

Programa: Engenharia de Sistemas e Computação

Neste trabalho, apresentamos um método de renderização de nuvem pontos em GPU sem ter informação de densidade e sem pré-processamento. Fazemos buscas em espaço de imagem usando o algorítmo de Jump Flooding para encontrar camadas e vizinhos mais próximos e usamos estas informações para renderizar uma aproximação da superfície. Nosso método trabalha com nuvens de baixa e alta densidade de pontos. Trabalha também com os casos onde temos os dois ao mesmo tempo, por exemplo quando se aproxima de um primeiro plano esparso com um fundo muito denso. Apresentamos os tempos de execução e renderizações que conseguimos em modelos de diversos números de vértices.

Abstract of Dissertation presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

POINT CLOUD RENDERING USING JUMP FLOODING

Renato Farias

July/2014

Advisor: Ricardo Marroquim

Department: Systems Engineering and Computer Science

In this work, we present a method of rendering a raw point cloud on the GPU with no density information or pre-processing. We perform image-space searches using the Jump Flooding algorithm to find layers and closest neighbors and use this information to render an approximation of the surface. Our method can work with both low and high density clouds. It can also work with cases where we have both at the same time, such as when zoomed in to a sparse foreground with a dense background. We present the execution times and renderings we achieved on models of varying number of vertices.

Sumário

Lista de Figuras **ix**

Lista de Tabelas **xi**

1 Introduction **1**

2 Related Works **3**

2.1 Surface Reconstruction 3

2.2 Point-Based Rendering 4

3 Jump Flooding Algorithm **8**

4 Algorithm **11**

4.1 Projection 13

4.2 Layer Search 13

4.3 Closest Neighbor Search 17

4.4 Rendering 18

5 Results **20**

5.1 Implementation Details 37

6 Conclusions **38**

7 Future Works **40**

Bibliography **41**

Lista de Figuras

3.1 An illustration of the jumps that pixel (3,3) makes when the JFA is executed on an 8x8 image. The initial jump step in this case is 4. Note that in the first iteration, only three of the jumps are actually inside the bounds of the image. At each iteration, every other pixel is also performing similar jumps. 9
3.2 Iteration-by-iteration generation of a Voronoi diagram using the JFA. Each seed was assigned a different color. The color of each pixel matches the color of its current closest seed (black if it has yet to find a seed). As can be seen, a pixel's current closest seed often changes from one iteration to the next. 10

4.1 Illustration of the steps of our algorithm. 12
4.2 When zoomed in, the background (green) points become much more numerous than the foreground (blue) points. 14
4.3 Every pixel should store the closest green layer and blue layer. Four pixels are highlighted here as examples: two pixels with green points, one with a blue point, and a pixel with no point. 14
4.4 Layers that a smooth cube could theoretically have. Points in layers 2, 3 and 5 could be paired up with points from adjacent faces of the cube depending on how strict we want to be, but we clearly want layers 1, 4, and 6 to remain separate. 15
4.5 Possible situations in the Layer Search, with the numbers representing the distance from the pixel making the jump. a) Candidate does not belong to any existing layers, and there is space for it. b) Candidate belongs to an existing layer, but is not closer, so it is discarded. c) Candidate belongs to an existing layer and is closer, thus replacing the existing point. d) When all spots are filled, candidate can replace existing layer if it is closer. 16

4.6 Illustration of the second condition. The black dotted lines are the surfaces implicitly defined by the points. In both scenarios, the red points have similar normals and are close to each other. However, by projecting the vector from one point to the other (blue lines) onto either point's normal (red dotted lines), we find that in a) the projection is too long, meaning it is too far forward, while in b) it is close enough. 17
4.7 Splat radius with 1, 2, 3, and 4 closest neighbors. 18

5.1 The DaFeltre model is an extreme example of the difference the viewpoint can make in the execution time of the method. Rendering the model's wide axis (a) takes around 280ms, twice more than the 140ms it takes to render the thin axis (b). 20
5.2 Cube 23
5.3 Stay Puft 23
5.4 Hand 24
5.5 DaFeltre 25
5.6 Arco (1) 26
5.7 Arco (2) 27
5.8 Armadillo 28
5.9 Blade (1). This dataset contains layers that are very close together. Our rendering (b) exposes interior details of the model, as can be seen when compared to the Meshlab rendering with triangles (c). These layers are not separated by the candidate test because they have similar normals, nor are they eliminated by the max distance threshold because of their proximity. 29
5.10 Blade (2) 30
5.11 Asian Dragon 31
5.12 Buddha 32
5.13 Schimele 33
5.14 Strada Ruffini 34
5.15 Tempietto 35
5.16 Esportazione (1) 35
5.17 Esportazione (2) 36
5.18 Sparse foreground and dense background for the Esportazione model. 36

Lista de Tabelas

5.1 Number of vertices of each model used for testing and average FPS achieved with them. 22
5.2 Execution times of each of the steps (in milliseconds). 22

Chapter 1

Introduction

3D scanning is a technology that produces a point cloud with geometric information that describes the scanned object. Other information can be extracted as well, such as colors and normals. Advances in this technology have made possible the acquisition of large-scale objects like buildings and environments. It is widely used in the entertainment industry and in applications like industrial design, reverse engineering, and the documentation of cultural heritage sites.

Because the data acquired from scanning comes in the form of point clouds, no connectivity information is present. A process of surface reconstruction can be applied to generate such connectivity information, producing a 3D model, for example a polygonal mesh or NURBS surface model. This model can then be visualized using traditional rendering methods.

The point cloud can also be rendered directly. There are many reasons why we might want to work with the point cloud directly. Point clouds are simple; they can have as little information as just a set of coordinates for each vertex. This is good because datasets acquired from 3D scanning are often already quite large. Performing surface reconstruction on these datasets can be a time-consuming process, and producing a good reconstruction is challenging. It may not even be possible to perform a reconstruction, such as in the case that an immediate visualization of the data is required. An example of such a situation is when the data, such as from a dynamic scene, is being streamed to a different machine that will be doing the rendering. If we can render a good approximation of the surface defined by the vertices of the point cloud, we eschew the need for an explicit reconstruction.

Point clouds obtained from scanning can have noise from the scanning process, inaccuracies inherent to the scanning equipment, and their sheer size can be a problem, too. Because of these challenges, and because of their simplicity and usefulness, point cloud rendering is a research topic that has received a good deal of attention. Many methods, while not explicitly reconstructing the surface, still require pre-processing steps for calculating density-per-point, inserting or removing points from the cloud, or determining a function that implicitly defines the surface.

In this work, we propose a method for visualizing a point cloud directly on the GPU. We project the point cloud, perform searches using the Jump Flooding algorithm to find layers and closest neighbors, and then use this information to render an approximation of the surface. The searches are performed in image space, meaning that it scales well with the size of the dataset and the rendering resolution. Our method does not require any density information, like radius-per-point, to work; in fact, it needs no pre-processing at all, nor extra information besides positions and normals. There are other methods that can also operate on this little information; we discuss some of them in Section 2.2. However, those methods tend to run into problems when it comes to varying and complex densities, whereas our method can work with both low and high density clouds, and cases where we have both at the same time.

This dissertation is structured as follows. Chapter 2 gives an overview of works related to ours or to the domain of point cloud rendering. Chapter 3 takes a close look at the technique we use to propagate information on the GPU. Chapter 4 begins by giving a high-level look at the method, then goes into the details of each step. Chapter 5 presents and discusses the results we achieved and implementation details. Chapter 6 summarizes the method and its advantages and disadvantages while Chapter 7 wraps up with possible directions for future works.

Chapter 2

Related Works

In this chapter we will go over related works.

Rong and Tan discuss using the Jump Flooding algorithm (JFA) as a communication pattern for the GPU. They apply the JFA to Voronoi diagrams and distance transform [1] [2], and image-based soft shadows [3], showing that it propagates information in constant time to the number of seeds. This technique is useful for us because of its efficiency in finding closest neighbors, and is explained more detail in Chapter 3.

2.1 Surface Reconstruction

There are a variety of works related to point cloud rendering that focus on reconstructing the surface represented by the cloud.

Carr et al. [4] use polyharmonic Radial Basis Functions (RBFs) to perform their reconstruction. They propose that the implicit representation of surfaces with RBFs simplifies the problems of smoothing and remeshing a noisy surface, as interpolation and extrapolation are inherent in the functional representation of the RBF. They fit an RBF to the cloud's data; the RBF's zero set is then the implicit surface.

Ohtake et al. [5] introduce a new surface representation called Sparse Low-degree IMplicits (SLIM) consisting of a sparse multi-scale set of nonconforming surface primitives. The SLIM representation consists of a sparse and hierarchical set of surfels, where each surfel is a triplet containing the center of a ball, the radius of the ball, and a function that delivers a local surface approximation inside the ball. Quadratic and cubic polynomials are used for the local approximations in the interest of speed. The local approximations are blended along view rays for rendering.

Kazhdan et al. [6] propose casting the surface reconstruction problem as a spatial Poisson problem. The main insight they point out is the relationship between oriented points from the surface of the model and the indicator function of the model, where the oriented points can be viewed as gradients of the model's indicator function.

The method of Mullen et al. [7] performs a reconstruction from a raw pointset. The first step of their pipeline calculates an unsigned distance function. In the second step, they compute a global, stochastic sign estimation of the distance, from which they get sign guesses for the unsigned distances. The last step smooths the second step's estimate to compute the signed distance through a linear solve, reconstructing a smooth, closed surface.

Hoppe et al. [8] present an algorithm that produces a representation of a surface given a raw dataset of points. The first stage of their algorithm defines a function that gives the signed geometric distance between a point in the region near the data to the surface being reconstructed. The zero set of this function is their estimate for the surface, and in the second stage it is approximated by a simplicial surface using a contouring algorithm. To create the signed distance function, they estimate tangent planes for each data point using a local point neighborhood.

2.2 Point-Based Rendering

There are many works that use splats for rendering point clouds. In this section we will go over the ones most closely related to our own work.

Kawata and Kanai [9] present a GPU-based approach that takes as input a point cloud with positional information. They dynamically determine the resolution of the image buffer that they will project the points into using the resolution of the point set. For this, they require a pre-processing step where they build a k-neighbor graph to determine the distance from each point to its closest neighbor. This results in the image buffer either being equal in size or smaller than the frame buffer. In the first pass, they project the points and store the point closest to the view position in each pixel. In the second pass, they project the points again, selecting for each pixel one point that is the farthest from the point selected in the first pass but still within a certain distance γ. A third pass looks at the two selected points for each pixel and saves their average in another buffer, for use as the splat position in the rendering. They calculate normals in the fourth pass by solving linear equations that use the two selected points of each pixel's 3x3 neighborhood. They do not store the normal, but instead immediately use it to apply shading to the color and save that instead. In the fifth pass, they use this color and the average point from the third pass to render splats, magnifying the image buffers because they are smaller than the frame buffer. However, their work requires a pre-processing step for the point selection process where a graph must be constructed, which is a non-negligible cost for large datasets. Further, their use of 3x3 neighborhoods makes their method dependent on high density; in the case that a point cloud is sparse, either by nature or because the user has zoomed in, it will not operate as well.

Diankov and Bajcsy [10] tackle the problem of rendering noisy point clouds from stereo algorithms with a multi-pass method they call Adapative Point Splatting (APS), which is based on Elliptical Weighted Average Splatting. The point cloud is first rendered to a small render target. Outliers are removed and a mask is created. To preserve edges and not incorrectly fill gaps, pixels are only set in the mask if there are enough pixels around it. Pixels not set will not be changed in later iterations. This operation is performed several times to fill larger holes. They then perform visibility, accumulation, and normalization passes 3-5 times on the pixels set in the mask. The visibility pass creates a depth map, the accumulation pass accumulates color and probability, and the normalization pass normalizes all pixels that pass the probability threshold, resetting the corresponding pixel in the mask so that it gets ignored from then on, and increases variance for each point's distribution.

Preiner et al. [11] devise a GPU approach to render dynamic scenes in real-time by performing a per-frame computation of surface aligned splats. They first project the points, saving their 2D pixel and 3D world positions in two separate buffers. To communicate information between neighboring points in a parallel manner, they use distribution and gathering passes with splats: they render splats with a certain radius r depending on that point's neighborhood, and each fragment created either writes to or reads from the point if they are contained within the radius. To find a k-neighborhood for each point, they begin with an initial radius estimation, r_0, and perform a search where they iteratively increase the radius until it contains k neighbors. Outliers are removed at this point depending on certain conditions, such as not finding a neighbor after 4 iterations. Finally, they fit a plane to the local neighborhood around each point with the plane's coordinates being the mean of the neighborhood's coordinates, and the normal being the eigenvector to the lowest eigenvalue of the scaled covariance matrix. As mentioned in their own work, this method can become inefficient when dealing with complex density distributions. If the foreground is sparse compared to the background, it will spend a lot of time constructing local neighborhoods for background points. They try to address this issue with grid culling.

In another image-space approach, Dobrev et al. [12] use depth peeling to render point cloud surfaces with transparency and shadows. Depth peeling is a multi-pass technique for extracting surface layers with respect to a given viewpoint. While a standard depth test finds the closest layer, depth peeling with n passes would find n layers. There must be a minimum depth distance between points for them to be considered different layers. For each pass, the entire scene must be rendered again. Transparency is achieved by blending the surfaces in order of extraction. For shadows, they construct a shadow texture, which is essentially a Boolean array that stores which points are lit and which are unlit. To construct the shadow texture they render the scene from the light source's position with depth test enabled and mark the visible points in the texture. The lit points are rendered with ambient, diffuse, and specular components as per the Phong model, while unlit points receive only the ambient component. Depth peeling has the problem of possibly confusing layers when there are gaps in the point cloud; this is a serious problem if the background is denser than the foreground. They attempt to address this using 3x3 masks, but this will fill only very small holes.

Pintus et al. [13] use a multipass GPU-based technique for rendering point clouds. The points are first projected into a texture. A visibility pass marks each point as visible if a large enough solid angle viewed by the point can be built that contains a line to the camera but no other points inside. The anisotropic filling pass considers 3x3 pixel neighborhoods and updates the central pixel's depth and color with a weighted sum of its neighbors. This provides 2.5D geometry and color textures which are used in the final stage of shape depiction, where they apply a custom shading term which is a mix of directional light, ambient occlusion and line drawing. They use a kd-tree to partition the dataset. This is another method which uses a 3x3 mask (or 7x7 mask for the visibility pass) in its execution. As discussed before, this makes the process dependent on the density of the point cloud; a very sparse cloud by nature, or a sparse view of the cloud because of zoom, will not produce the results of the same quality.

The method proposed by Yang et al. [14] receives as input a point cloud and a set of texture images with their corresponding projection matrices. It generates a splat primitive for each point and source view, orienting them by applying scale, rotation and translation transformations. The scale is determined by the largest edge distance from each point to its set of neighbors. The rotation aligns the primitive to the point's normal, or to the viewing direction if no normal is available. The translation moves the primitive to the Euclidean position of the point. The input textures are blended, with a weight determined by the angle between the desired viewpoint and the source views. They liken the visibility problem to the shadow texturing, and use the same idea: render once from the desired viewpoint, and one for each other view, to see which points are visible and should contribute to the final color. This method is interesting but relies on the availability of several high-quality texture images of the model being rendered.

To try to assuage the problem of magnification in splat-based rendering, Guennebaud et al. [15] propose an upscaling algorithm that iteratively refines the geometry of the point cloud. It is based on the idea of adding a new point for every pair of neighbors in the cloud. The first step is the selection operator, where they compute local neighborhoods. This helps them select good pairs to insert new points evenly throughout the cloud. They interpolate the position and normal of the new point based its selected local neighborhood. Unlike this method, our work aims to work on the point cloud directly, without generating any new geometry.

Marroquim et al. [16] present a method that does not use splats directly, instead projecting the points into single pixels then using pull-push interpolation render the surface. The pull phase of the interpolation creates the coarser levels of an image pyramid by consolidating groups of 4 pixels into 1, reducing the dimensions by a factor of two each level. The attributes of the new coarse pixel are generally the average of the valid, unoccluded pixels from the 4 that create it. The push phase works in the opposite direction, from the coarse levels down to the finer levels, and only the pixels that are invalid (empty) or occluded are changed. The attributes of 4 pixels from the coarser level are weighted according to a biquadratic B-spline subdivision, and this interpolation is the new value of the finer pixel. After the process is finished at the finest level, it can be rendered using deferred shading. However, this method requires local density information in order to preserve the background and not blend together different surfaces of the image. The dataset must be pre-processed to calculate a radius for each pixel so that this information can be used to limit the propagation of information during pull-push interpolation.

In developing this work, we focused on three main objectives. First, our method should require no pre-processing and no information besides what most raw point clouds will be able to provide (positions and normals). Second, as much of the method as possible should operate on image-space, in the interest of scalability. And third, the method must be able to deal with any kind of density, and must be able to deal with different kinds of densities together. While many of the works we cite above are able to achieve one or two of these objectives, none of them are able to unite all three.

Chapter 3

Jump Flooding Algorithm

The Jump Flooding Algorithm (JFA) is useful for propagating information, with applications such as Voronoi diagram, distance transform, and image-based soft shadows [1][3][2].

The JFA consists of successive iterations where each pixel in the image "jumps" to pixels in 8 directions. These 8 directions are the four cardinal and four ordinal directions (see figure 3.1). How far the pixel jumps in a given iteration is called the jump step, or simply step. The step is initially the largest power of 2 contained in the image's dimensions, and is divided by 2 after each iteration until it is equal to 1 in the last iteration. At each of the 8 jumps the origin pixel samples the information stored in the jump pixel, keeping that information if it is more desirable according to some criterion.

To give a concrete example, we will use the generation of a Voronoi diagram, as illustrated in figure 3.2. Consider a texture of dimensions 1280x720. In this texture 13 points are placed as in figure 3.2. These are the seeds of the Voronoi diagram. Every pixel in this texture contains a pair of screen-space coordinates: the coordinates of its closest neighbor. The seeds start with their own coordinates, and every other pixel besides them start with empty, non-valid coordinates. The initial step is 1024 (largest power of 2 smaller than 1280). In the first iteration, every pixel in the texture jumps 1024 positions in the 8 directions, and fetches from the texture the coordinates stored in that jump pixel. If the pixel currently has no valid coordinates, and if it just fetched valid coordinates, then it stores those coordinates. If the pixel has valid coordinates and also fetches valid coordinates, it keeps whichever pair of coordinates is closer to its screen-space position. At the start of the next iteration, the step is halved to 512, and all of the pixels jump again. In the third iteration the step is halved to 256, then to 128 in the fourth, and so on, until in the tenth and last iteration the step is 1.

Note that each pixel fetches information from several pixels but only writes to itself. This means that the jumps of an iteration can be performed independently,

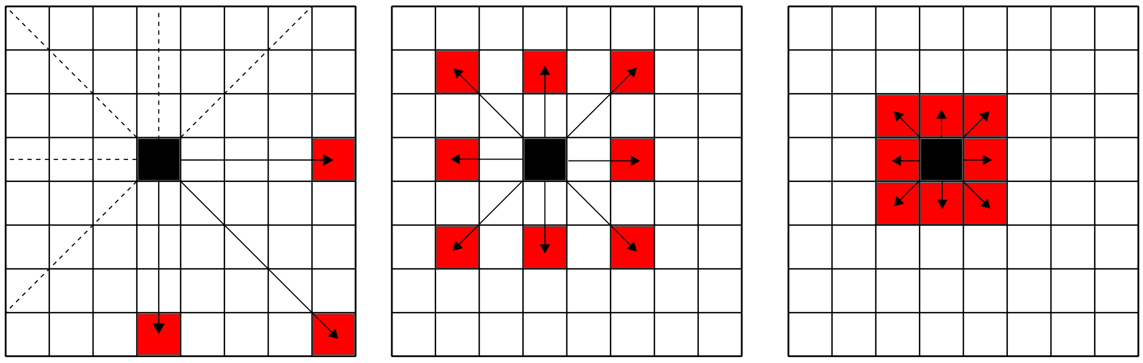

(a) First Iteration (b) Second Iteration (c) Third Iteration

Figure 3.1: An illustration of the jumps that pixel (3,3) makes when the JFA is executed on an 8x8 image. The initial jump step in this case is 4. Note that in the first iteration, only three of the jumps are actually inside the bounds of the image. At each iteration, every other pixel is also performing similar jumps.

which is well-suited to the parallel nature of the GPU. However, because an iteration fetches information from the previous iteration, two textures of similar size are required to perform the process described in the previous paragraph. One texture is the "read" texture containing the previous iteration's results while the other is the "write" texture for the current iteration's jumps. The two textures swap roles in the following iteration. This technique where two sets of textures swap the "read" and "write" roles between iterations is known as "ping-pong" and is a common technique in GPU programming.

A key advantage of the JFA is its speed and scalability. The jumping pattern adapts naturally to discrete grid structures like image textures; calculating the jump pixel coordinates is as simple as adding the step in each of the 8 directions. More importantly, the number of iterations increases logarithmically with the image's largest dimension, so it scales well with the rendering resolution. A Voronoi diagram with double the dimensions of our example, 2560x1440, would need 11 iterations to be generated, only 1 more than our example of 1280x720, which needed 10 iterations. Aside from the time it takes to put the seeds into the texture in the first place, the actual number of seeds does not influence the JFA time.

In Sections 4.2 and 4.3 we explain the differences between how we use the JFA in our method and the Voronoi example given above.

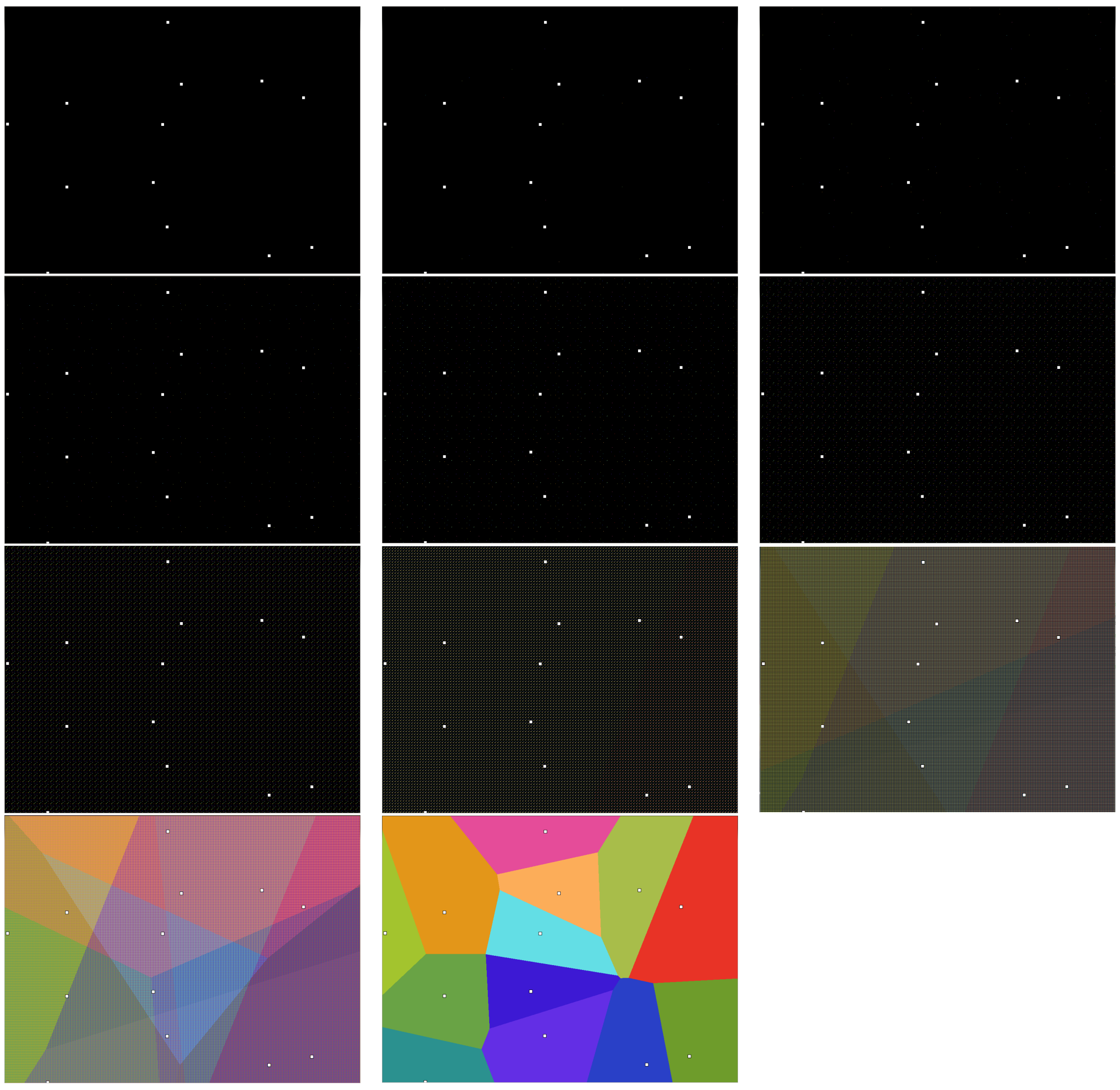


Figure 3.2: Iteration-by-iteration generation of a Voronoi diagram using the JFA. Each seed was assigned a different color. The color of each pixel matches the color of its current closest seed (black if it has yet to find a seed). As can be seen, a pixel's current closest seed often changes from one iteration to the next.

Chapter 4

Algorithm

The goal of our method is to render a smooth surface from a raw set of points. The method uses image-space searches to construct a neighborhood of closest points in world-space for each point in the dataset. A key component of the process is the differentiation of the "layers" of the dataset for use in the closest neighbor search. Our definition of layer here is the intuitive idea of points belonging to the same surface. As an example, consider the points of a wall: they have spatial continuity and similar normals. Being able to separate layers helps to construct more appropriate neighborhoods of points.

Our method receives as input a point cloud with coordinates and normals, and optionally colors. As illustrated in figure 4.1, the method is comprised of four steps: Projection (Section 4.1), a Layer Search (Section 4.2), a Closest Neighbor Search (Section 4.3), and Rendering (Section 4.4).

We start by projecting the points of the cloud, saving their information in buffers on the GPU. We execute the Jump Flooding algorithm to find surface layers, then execute it again to find each projected point's closest world-space neighbors located on its layer. With this information, we perform a first rendering pass to create a depth buffer. Then, after eliminating points that fail a depth test against this buffer, we render the splats again, blending them to calculate the final normal and color for each pixel. These steps are described in more detail in the following sections.

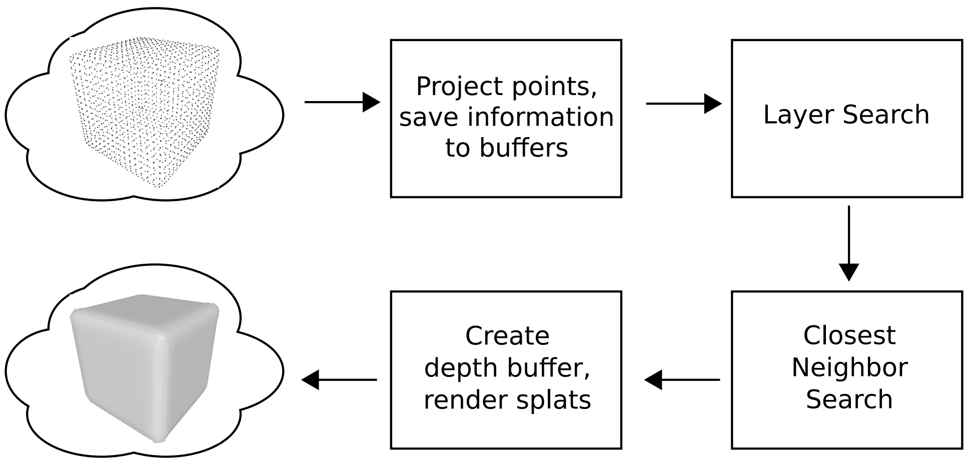

Project points, save information to buffers
Layer Search
Closest Neighbor Search
Create depth buffer, render splats

Figure 4.1: Illustration of the steps of our algorithm.

4.1 Projection

The first step of the algorithm is the projection of the point cloud. We allocate 4 textures on the GPU to store the cloud's information. We save the projected screen-space coordinates, the original world-space coordinates, the normals, and (if present) the colors to these buffers.

The resolution of these buffers is equal to half the rendering resolution. This speeds up the projection and subsequent steps. Depth culling is enabled for this step, so the reduced resolution also increases the chance of background points being culled as they are projected into the same pixels as foreground points. Because the final rendering is done in the normal resolution, this initial reduction does not noticeably affect the visual quality of the results. Note that this size reduction is simply a way for us to speed up our implementation, and is not required by the algorithm.

4.2 Layer Search

It is intuitive to think of the points of a given surface, such as a wall, as belonging to the same layer; they have spatial continuity and similar normals. Naturally we want points belonging to the same surface to be considered as neighbors when reconstructing the surface for rendering. However, if we simply search for each pixel's closest neighbor in image-space, we have no guarantee that this will happen. This is because for all but the simplest models, more than one surface can be projected into the same area of the image, mixing points from visible and occluded surfaces. Further, in the case that we are zoomed in close to a surface, the background points will be much denser than the foreground points, possibly drowning out the foreground points.

Figure 4.2 illustrates this situation. The background points greatly outnumber the foreground points. A simple closest neighbor search would propagate almost nothing but background points. Instead of using image-space distance, we could use world-space distance, and for some situations this can work. However, this is not robust enough if we are working with complex densities, because the main problem of the background points overwhelming the foreground points and preventing them from propagating persists.

To overcome this, many methods require an involved pre-processing step, either constructing spatial partitioning structures like kd-trees or using techniques like depth peeling. The approach we adopt in our method is to use the concept of layers, where each layer represents projected surfaces of the model. Figure 4.4 shows an example. To eventually render splats, each point will need a local neighborhood.

However, a point's neighborhood should only contain other points from the same layer.

Consider figure 4.2 again: we want the blue points to construct a local neighborhood from other blue points, and not green points. Likewise, we want the green points to consider other green points for their neighborhood. Of course, what we really want is for the background points to eventually be occluded, but this should come automatically with the rendering of splats. In essence, we want to "divide" the image into layers, so that the next step, the Closest Neighbor Search, can more easily pair up points that are in the same layer. In practice, this means that each pixel should store information from numerous layers, as exemplified in figure 4.3.

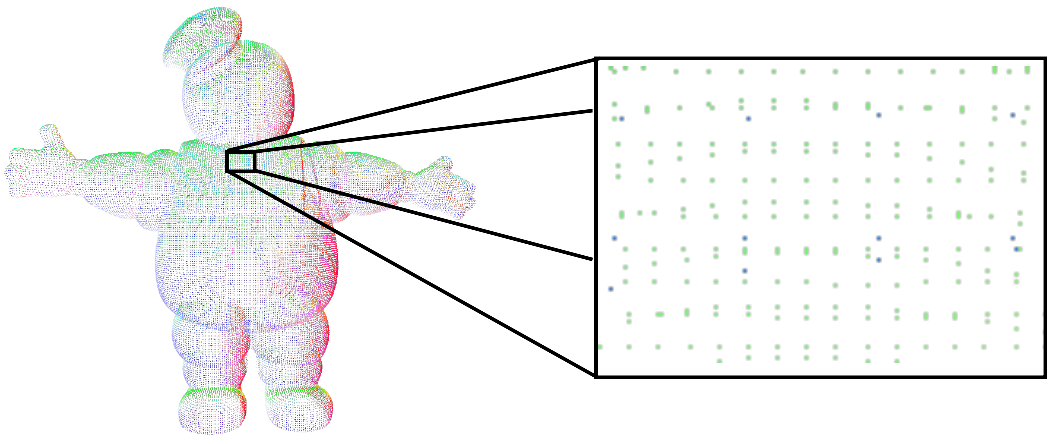


Figure 4.2: When zoomed in, the background (green) points become much more numerous than the foreground (blue) points.

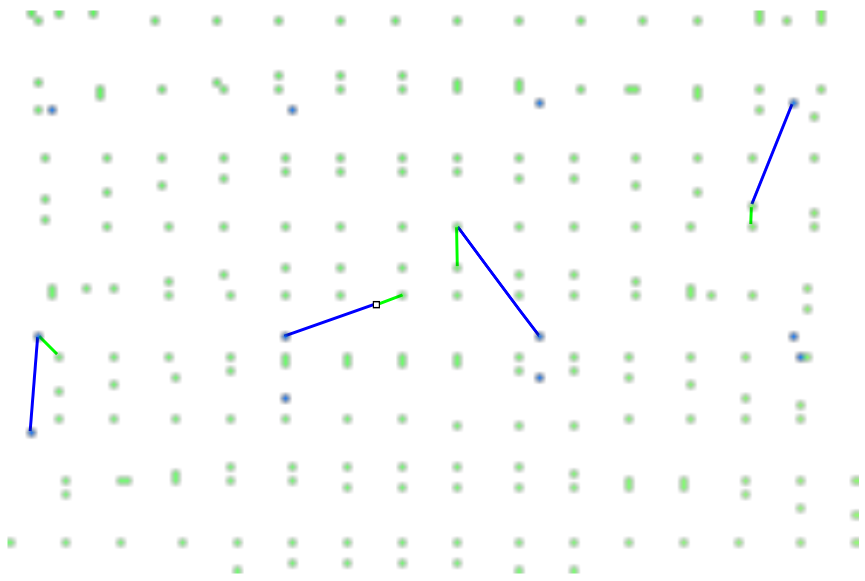


Figure 4.3: Every pixel should store the closest green layer and blue layer. Four pixels are highlighted here as examples: two pixels with green points, one with a blue point, and a pixel with no point.

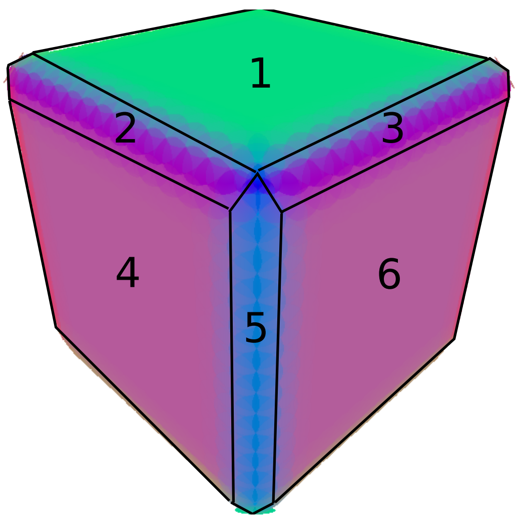


Figure 4.4: Layers that a smooth cube could theoretically have. Points in layers 2, 3 and 5 could be paired up with points from adjacent faces of the cube depending on how strict we want to be, but we clearly want layers 1, 4, and 6 to remain separate.

After we project the original model's points and store their information in buffers, the first thing we do is attempt to divide the image into layers. That is, for each pixel, we will look for the n closest points in image-space that belong to different layers. We perform the JFA as described in Section 3, but with some differences. First, each pixel will store not one pair of coordinates, but n pairs. Pixels making jumps will consider all n pairs of the jump pixel, possibly replacing and storing more than one pair of coordinates per jump. We use $n = 4$ layers in this search.

The way information is compared and stored is different here compared to the Voronoi diagram example of Section 3. After the pixel fetches a candidate, it first checks to see if it belongs to the same layer as one of its n stored coordinates. The way we determine if two points belong to the same layer is using the candidate test described below. If it does belong to an existing layer, then it either replaces the stored coordinates if it is closer in image-space, or is discarded if it is not closer (even if there are still open spots where it could have been stored). If the candidate belongs to none of the current layers and the n spots have not been filled yet, it is stored in one of the empty spots. After all n spots are filled, new candidates are still compared to current neighbors to see if they belong to the layers of any of them, as described in the previous paragraph. However, if a candidate is then found to belong to none of them, it can replace an existing layer if it is closer in image-space. This is to prevent the first n layers found to persist even if other, closer layers are found later on. At the end of the process, each pixel has its closest n layers in image-space. Taking the zoomed in region of figure 4.2 as an example, each pixel would have the coordinates of the closest green point, and also the coordinates of the closest blue point.

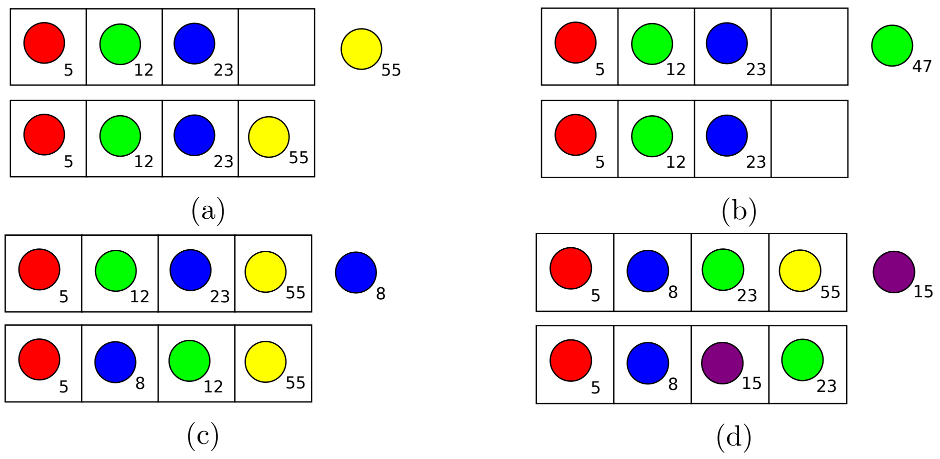


Figure 4.5: Possible situations in the Layer Search, with the numbers representing the distance from the pixel making the jump. a) Candidate does not belong to any existing layers, and there is space for it. b) Candidate belongs to an existing layer, but is not closer, so it is discarded. c) Candidate belongs to an existing layer and is closer, thus replacing the existing point. d) When all spots are filled, candidate can replace existing layer if it is closer.

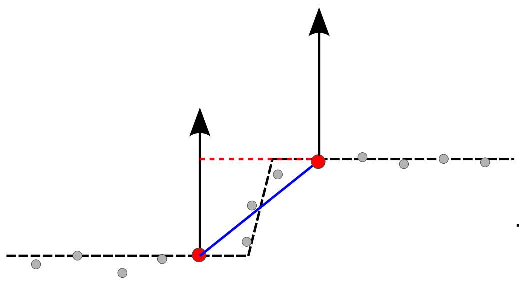
The candidate test that determines whether two points belong to the same layer takes the two world-space coordinates p_i and p_j and their normals n_i and n_j and checks the following conditions:

1. $n_i \cdot n_j \geq \textit{angle threshold}$
2. $|(p_j - p_i) \cdot n_i| < \textit{continuity threshold}$

Both conditions must pass for the points to be considered to be on the same layer. Condition 1 ensures that the two points' normals are sufficiently similar; points that are located on the same surface will tend to have similar normals, even if the surface is not perfectly flat. Satisfying condition 2 means that the perpendicular axis of the vector between the points is longer than the parallel axis, which is usually true of points on the same surface, as exemplified in figure 4.6.

Note that this candidate test is the only step in our method that requires normal information to be present. Later on we use normals to orient the splats and for lighting calculations during rendering, but at that point we have each point's local neighborhood and could compute the normal ourselves. However, the candidate test adds robustness to the layer separation and to the Closest Neighbor Search in Section 4.3.

We limit the max distance possible between the two points being considered by the candidate test; if the distance between them is larger than a certain threshold, the test fails. The idea of this limitation is similar to the outlier removal applied by Preiner et al. [11], where they discard points that have no neighbors after a certain


(a) Does not pass condition

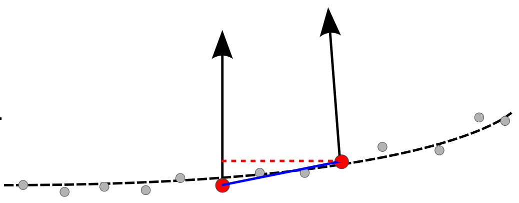

(b) Passes condition

Figure 4.6: Illustration of the second condition. The black dotted lines are the surfaces implicitly defined by the points. In both scenarios, the red points have similar normals and are close to each other. However, by projecting the vector from one point to the other (blue lines) onto either point's normal (red dotted lines), we find that in a) the projection is too long, meaning it is too far forward, while in b) it is close enough.

number of iterations (in a way, limiting how far a point can reach for neighbors). This limitation prevents the process from pairing up points that are too far apart even if they would otherwise belong to the same layer and there are no other closer layer points between them, because the resulting splat would be too big, adversely affecting the result.

4.3 Closest Neighbor Search

At the end of the Layer Search, each pixel has up to n coordinates of points that belong to different layers. We now execute the JFA again to find a local neighborhood consisting of the k closest neighbors for each projected point in the image. Each point builds a local neighborhood comprised only of other points that belong to the same layer. This k-neighborhood is important in determining how large each splat should be, since our method operates on raw point clouds with no density information. We use $k = 4$ closest neighbors in this search.

The importance of the layers now becomes apparent. In a situation where we have varying densities, like in figure 4.2, the densest surface would tend to drown out the rest. The JFA would be dominated by the denser surfaces, and points would incorrectly construct neighborhoods containing points in different surfaces. This would lead to splats that are too big, or oriented incorrectly if the normal were calculated from the neighborhood. Unless a model is very complex, we can expect that no more than n layers will be projected on the same area of the image. Therefore, the layer list ensures that every surface gets propagated fairly throughout this execution of the JFA and can be found by points on it.

Previously, every pixel in the image participated in the JFA; for the Closest Neighbor Search, only pixels that contain projected points from the model participate. At each jump, the points check the layers found in the previous step. There is no candidate test this time around. Candidates are stored if they are closer to the point than any of the current neighbors. Additionally, if a point jumps to a pixel that contains a projected point, they also look at the projected point's current local neighborhood in search of more candidates. The distance used here is in world-space rather than image-space.

4.4 Rendering

With each point's k closest neighbors in hand, we proceed to the rendering. The rendering follows a standard splat rendering pipeline. As noted in Section 4.1, the rendering buffer has normal rendering resolution, while the previous steps operated on buffers of half size. This simply means that when accessing the information from projection or the local neighborhood, we must adjust the texture coordinates accordingly, dividing them by two.

The original cloud's points are reprojected in this step. We make use of geometry shaders to generate the splats. In the geometry shader, for each point, we retrieve its world-space coordinates, its normal, and its local neighborhood. We perform backface culling, discarding the points whose normals point away from the viewing direction. The splat radius r is determined by the center point's closest neighbors, being equal to either the median, or the average of medians if there is more than one median, as illustrated in figure 4.7.

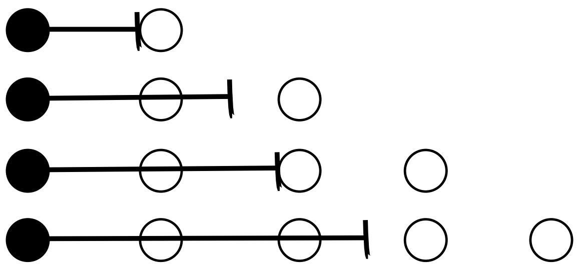


Figure 4.7: Splat radius with 1, 2, 3, and 4 closest neighbors.

Given d_c, the distance between the point and its closest neighbor, max, a max radius threshold, and max_r, a max relative radius threshold, we also discard points if:

1. $r \leq 0$,
2. $r > max$, or
3. $r > \mathrm{d}_c \times \, max_r$

Instead of explicitly rendering several triangles to draw the splats, we circumscribe a square about the circle that represents the splat. To achieve the splat effect, we first output from the geometry shader two triangles made up of 4 vertices rotated around the central splat point oriented to its normal. Because we are specifying the corners of a square, we use a distance equal to the splat radius multiplied by $\sqrt{2}$. Each vertex receives its relative coordinates in the square, (-1,-1), (1,-1), (-1,1), or (1,1). This is passed on, along with the other attributes, to the fragment shader. In the fragment shader, we simply discard the fragments whose distance from the point (0,0) is greater than 1. For the first rendering pass, we store each fragment's depth in a depth buffer.

We then run a second rendering pass, with two key differences. First, we add one more condition to the geometry shader: the point must pass a depth test against the depth buffer created in the first rendering pass. Second, we enable blending, such that every fragment contributes to the final normal and color of the pixel weighted by its distance from the original point. The weight function that we apply is a non-normalized Gaussian weight with variance $\sigma = 1$. This function could be substituted by other functions but we found in our experiments that this approximation worked well enough. If v is the fragment's normal or color, d its distance from the center splat point, and e Euler's number, then the weighted amount w that it contributes is calculated using the following formula:

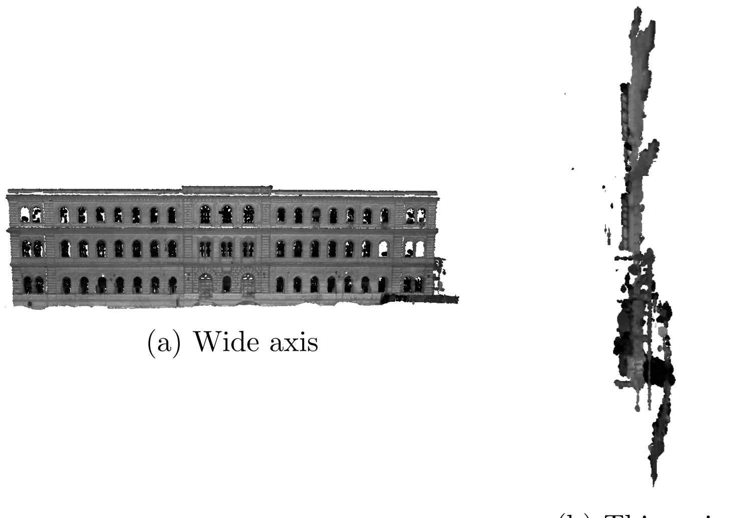
$$w = \frac{v}{e^{d^2}}$$

Chapter 5

Results

In this section, we present and discuss the results we achieved.

The total rendering time of a model can vary widely depending on how it is viewed. We observed a decrease of up to 50% on some models if viewed down their thin axis as opposed to their wide axis. In the interest of an accurate representation of the execution times, we made sure to have the entire model inside the viewport, and to constantly rotate the model so that no one view was favored.


(a) Wide axis

(b) Thin axis

Figure 5.1: The DaFeltre model is an extreme example of the difference the viewpoint can make in the execution time of the method. Rendering the model's wide axis (a) takes around 280ms, twice more than the 140ms it takes to render the thin axis (b).

Tables 5.1 and 5.2 list the frames per second achieved and the rendering times, respectively. Table 5.1 is sorted by increasing number of vertices while table 5.2 is sorted by total execution time. Figures 5.2-5.18 show a variety of models being rendered by our method with a side-by-side comparison of a simple projection of the points.

The smallest model we tested was Cube with 1,538 points, and the largest was Tempietto with 59,479,779 points. On Cube we achieved an average of 30.76 FPS, while on Tempietto we achieved 0.98 FPS. In other words, despite Tempietto having almost 39,000 times as many points as Cube, it was only 31 times slower. With a model as dense as Tempietto, culling certainly helps, but it is an example of the method scaling extremely well.

If we examine the breakdown of the execution times by steps, we see that the Layer Search and Closest Neighbor Search times increase slowly. In fact, for large models with millions of points, it represents only a fraction of the total execution time. In those cases, Projection and Rendering take the longest, but even those grow sublinearly. The reason that the Layer Search and Closest Neighbor Search times increase despite operating in image-space is that more projected points in the image leads to more texture fetches.

Figure 5.18 shows our method rendering a visualization of the Esportazione model containing varying point densities. In figure 5.18b, the columns of the central temple become sparse when zoomed in, especially compared to the dense walls in the background. Figure 5.18d has a similar situation with the temple's wall. The method is still able to find good closest neighbors for the points, though, and render splats that are reasonable.

In figure 5.12d, there are small gaps in some parts of the rendering. This happens in some areas when points find neighbors that are so close together, the splats that they generate are not big enough to close the surface. This can also be observed to a lesser extent in figures 5.3b and 5.15b. However, this is an issue with any point-based rendering technique that does not have prior knowledge of the point cloud density (i.e. radius-per-point). This is because there is not enough information to determine if a gap in image-space should be filled or is an actual hole in the surface. In this work, we opted for a conservative approach, preferring to leave a few holes rather than incorrectly filling them.

The Blade dataset in figures 5.9 and 5.10 contains layers that are very close together. Figure 5.9c shows the dataset rendered with triangles in the Meshlab program. Our rendering in figure 5.9b incorrectly brings out interior details in the model. This is because those layers have similar normals and are close to each other, so they pass both the candidate test and the max distance threshold.

Model	Vertices	FPS
Cube	1,538	30.76
Stay Puft	97,078	20.23
Armadillo	172,974	21.51
Hand	327,323	30.33
Buddah	543,652	21.15
Tempietto	753,662	16.63
Blade	882,954	25.56
Strada Ruffini	1,578,898	14.49
Asian Dragon	3,609,600	10.72
Esportazione	10,036,978	5.08
DaFeltre	10,937,594	3.97
Arco	19,897,657	2.33
Schimele	26,245,666	1.94
Tempietto 50M	59,479,779	0.98

Table 5.1: Number of vertices of each model used for testing and average FPS achieved with them.

Model	Projection	Layer Search	Closest Neighbor Search	Rendering	Total
Cube	11.01	16.51	3.79	1.18	32.50
Hand	5.39	19.02	4.40	4.14	32.96
Blade	8.82	16.93	4.12	9.22	39.11
Armadillo	13.14	24.68	6.06	2.57	46.47
Buddah	13.47	21.66	5.64	6.49	47.27
Stay Puft	8.70	30.97	7.55	2.17	49.42
Tempietto	11.49	31.23	7.86	9.52	60.12
Strada Ruffini	10.88	29.90	8.11	20.10	69.00
Asian Dragon	23.60	20.40	4.72	44.52	93.24
Esportazione	48.48	34.76	9.20	104.23	196.69
DaFeltre	46.91	23.40	5.83	175.31	251.46
Arco	134.94	29.26	7.60	256.87	428.68
Schimele	140.09	35.19	8.30	329.54	513.13
Tempietto 50M	240.47	39.97	10.85	720.20	1011.52

Table 5.2: Execution times of each of the steps (in milliseconds).

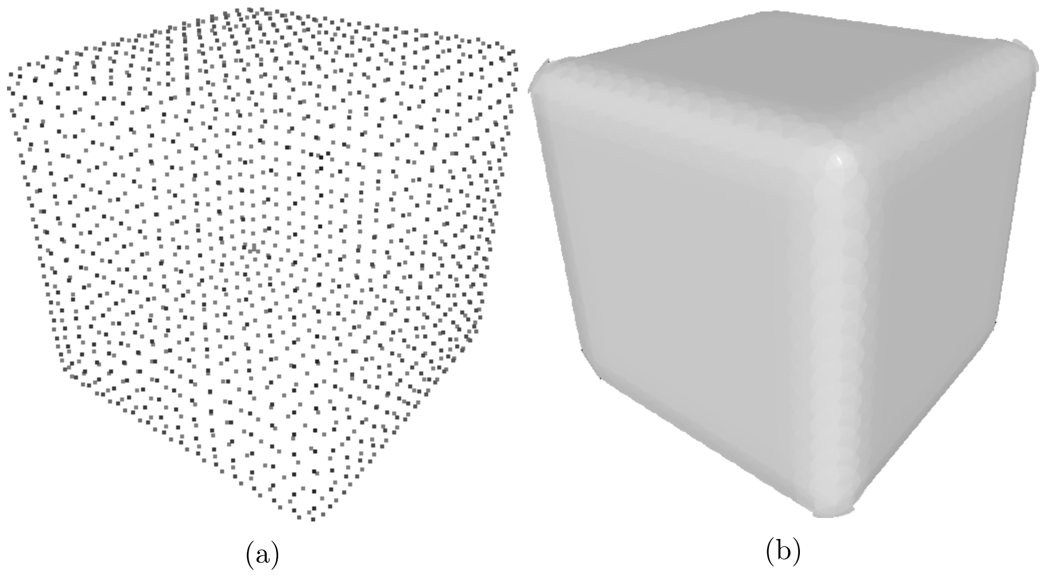


Figure 5.2: Cube

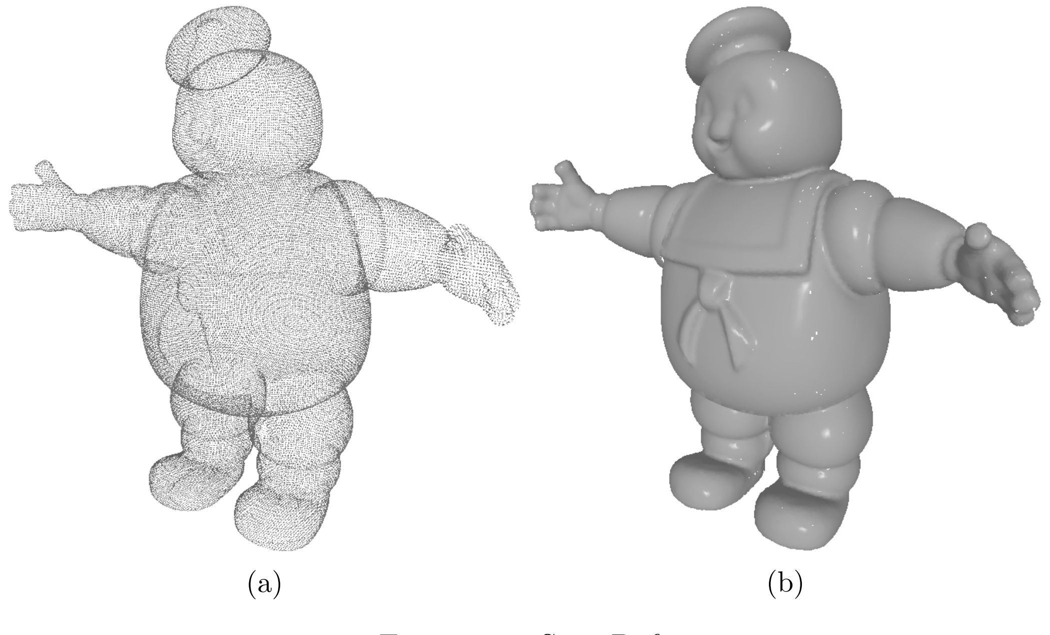
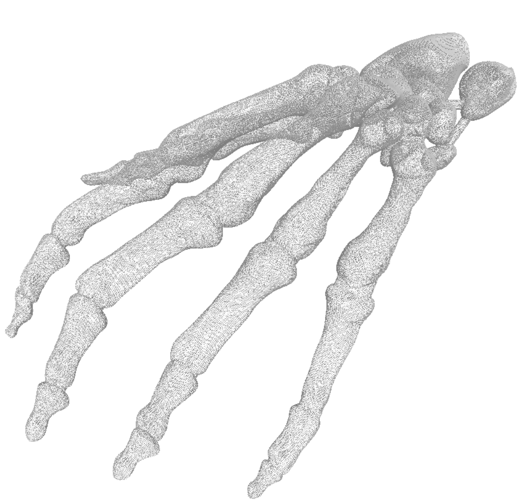


Figure 5.3: Stay Puft


(a)

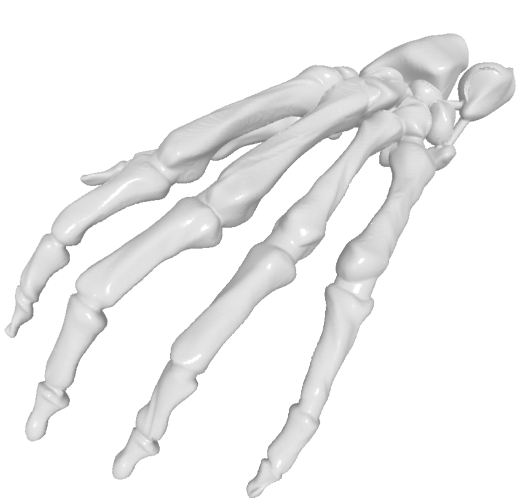
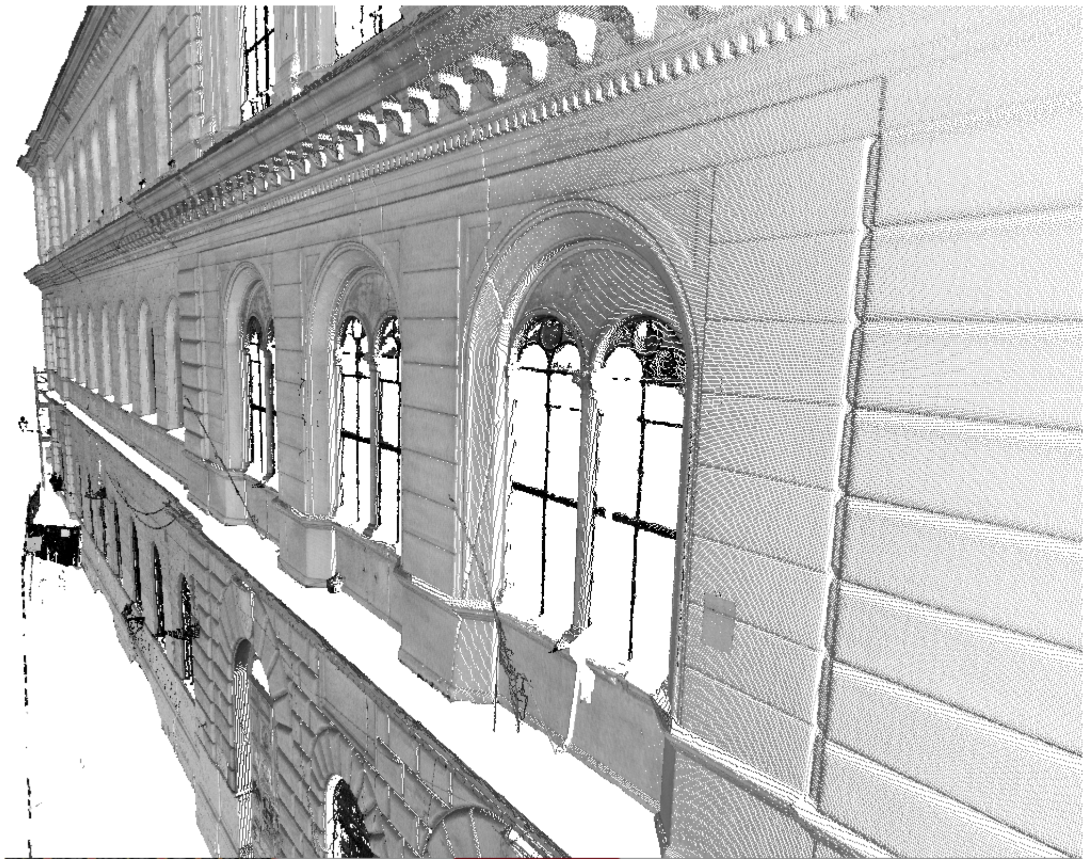
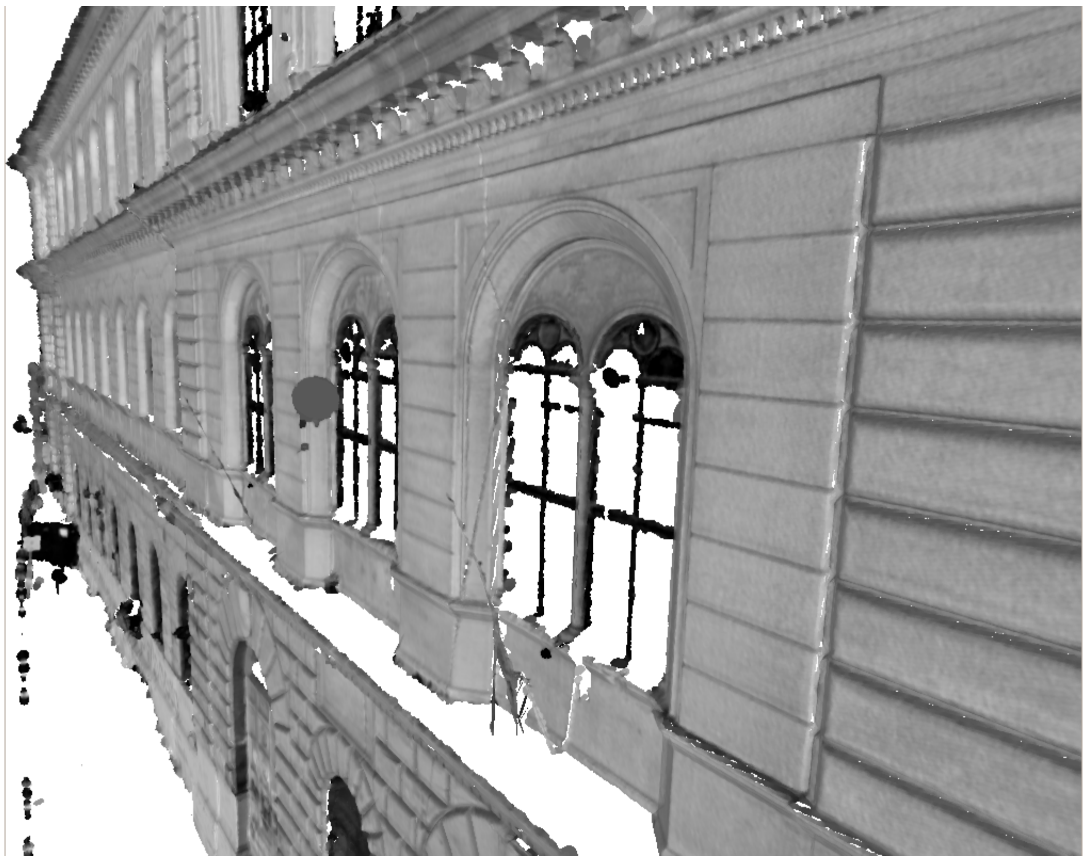

(b)

Figure 5.4: Hand

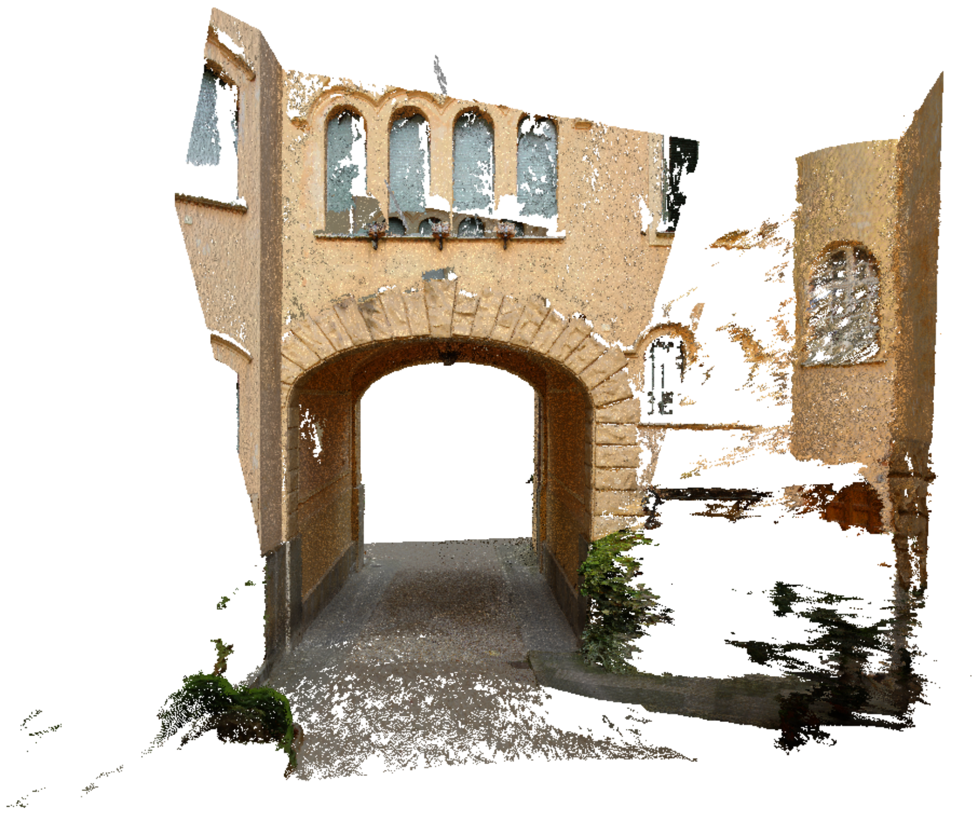


(a)

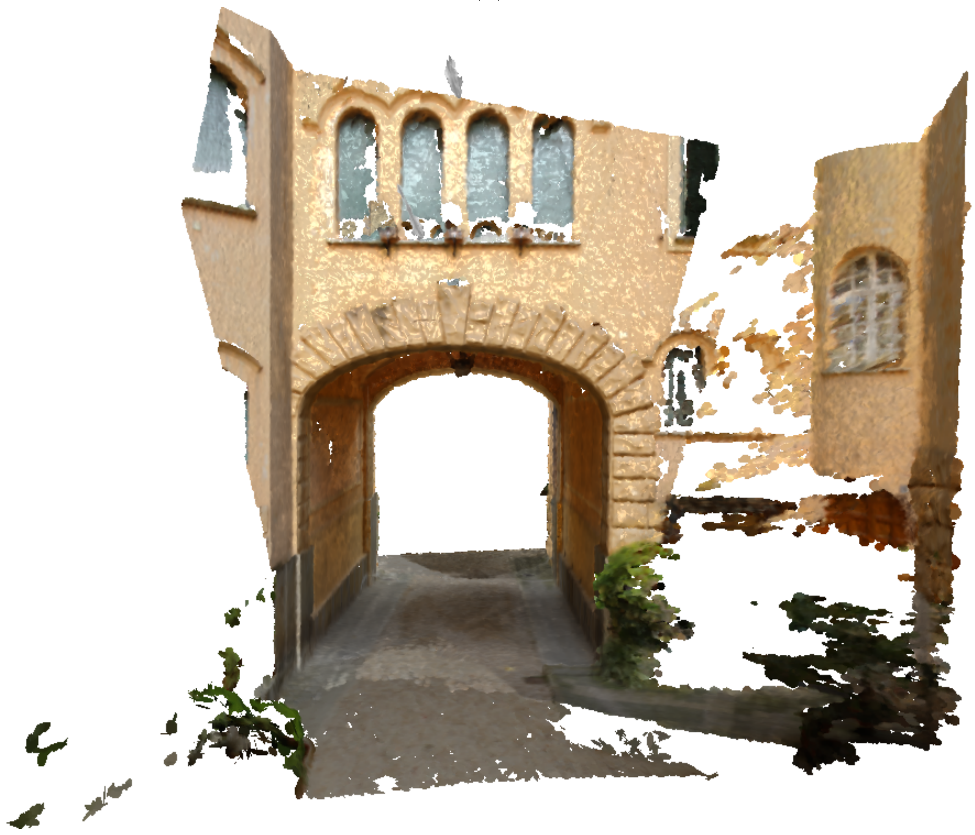


(b)

Figure 5.5: DaFeltre

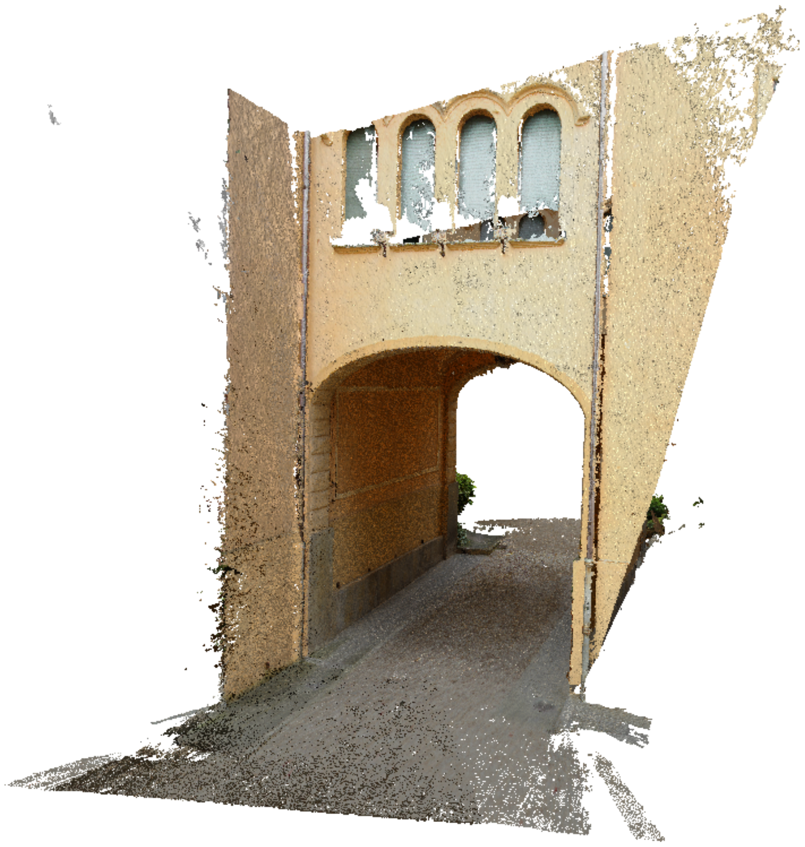


(a)



(b)

Figure 5.6: Arco (1)


(a)

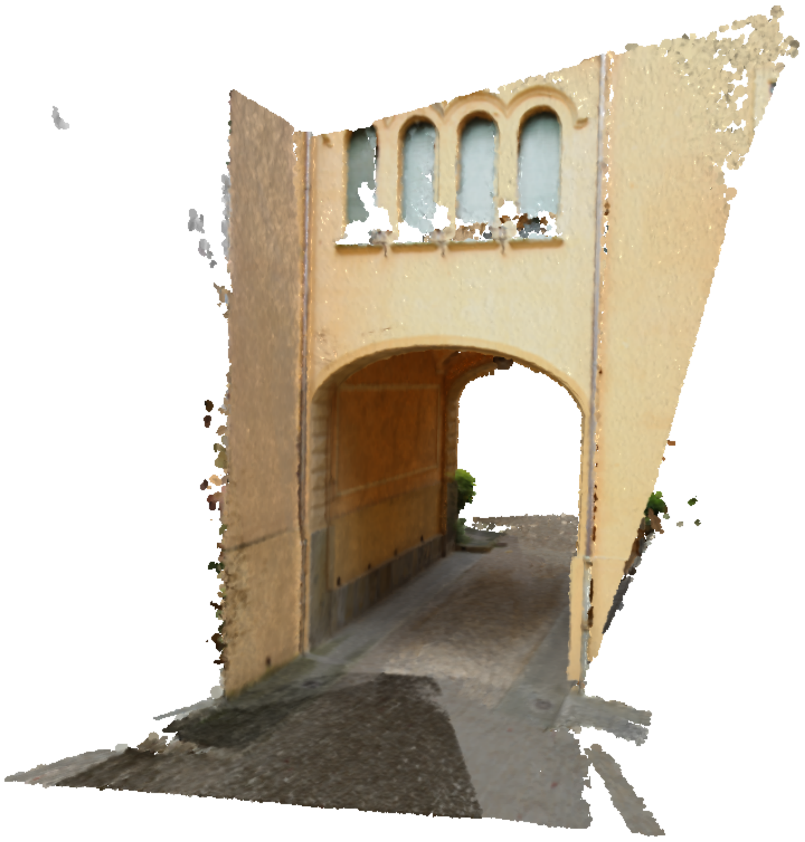
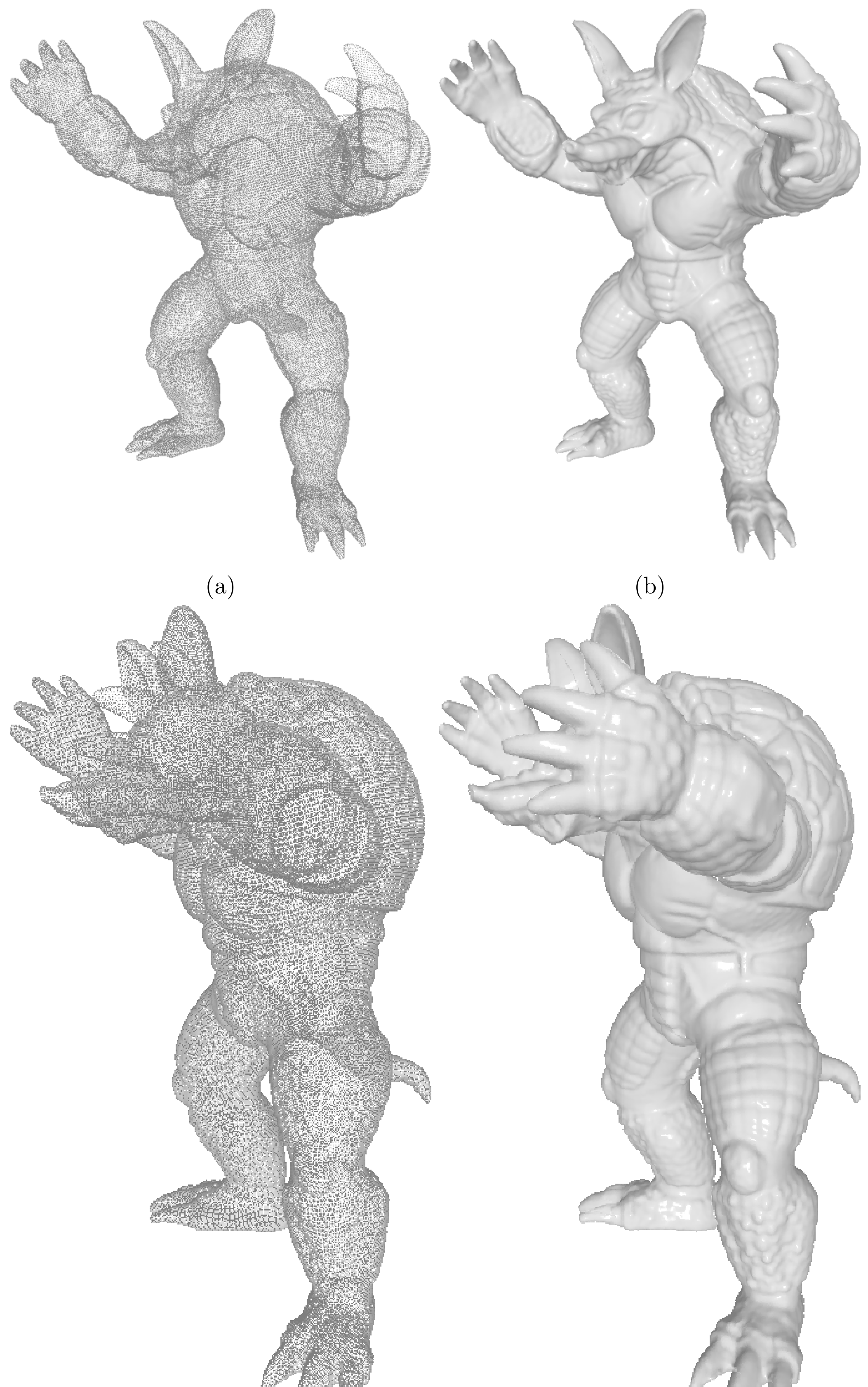
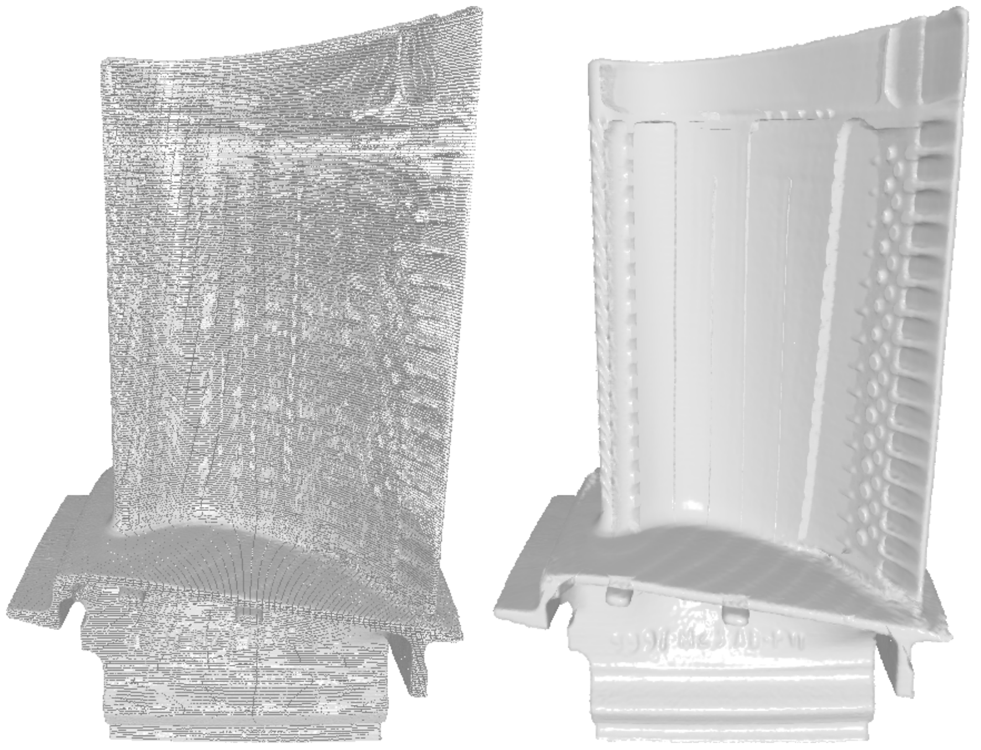

(b)

Figure 5.7: Arco (2)

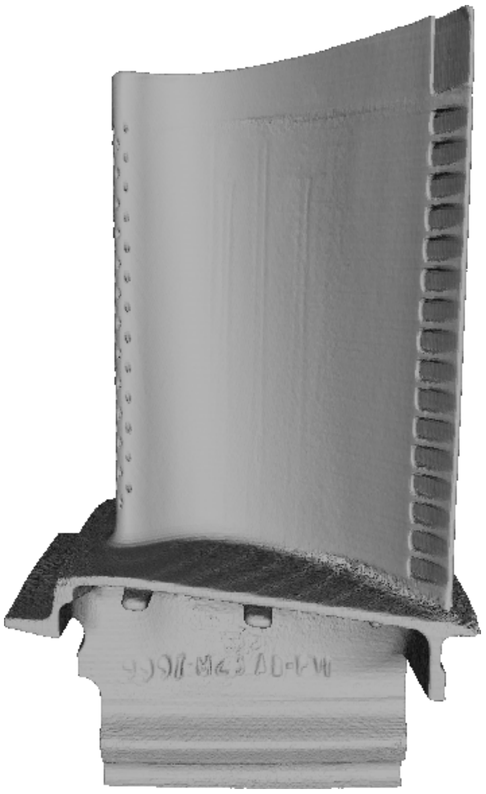

(a) (b)

(c) (d)

Figure 5.8: Armadillo

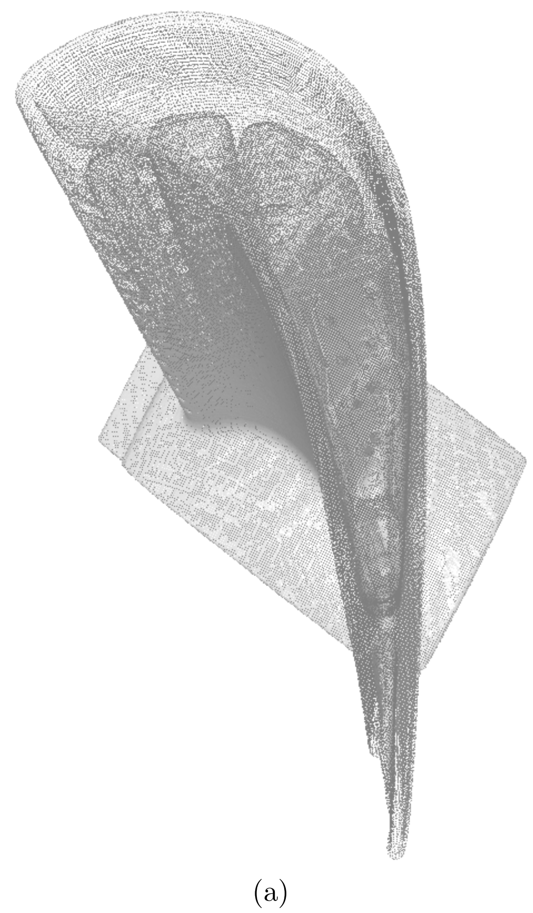


(a) (b)



(c)

Figure 5.9: Blade (1). This dataset contains layers that are very close together. Our rendering (b) exposes interior details of the model, as can be seen when compared to the Meshlab rendering with triangles (c). These layers are not separated by the candidate test because they have similar normals, nor are they eliminated by the max distance threshold because of their proximity.


(a)

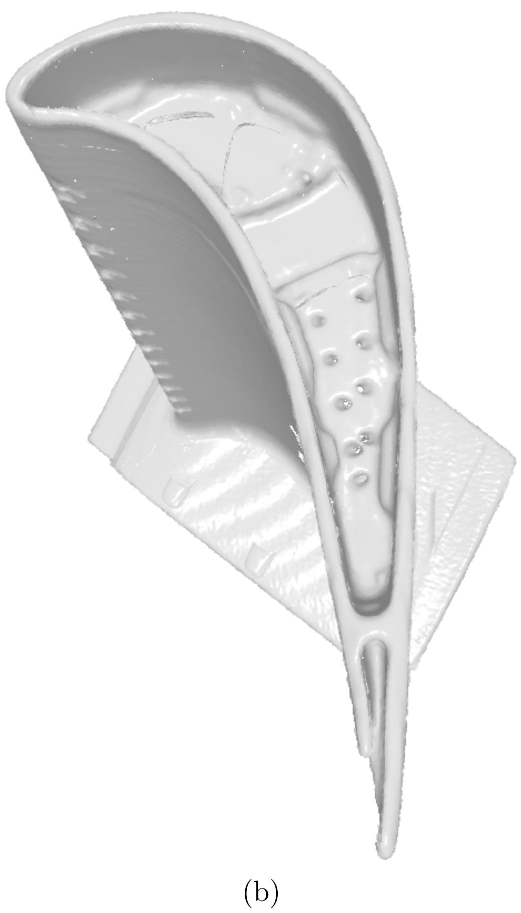

(b)

Figure 5.10: Blade (2)

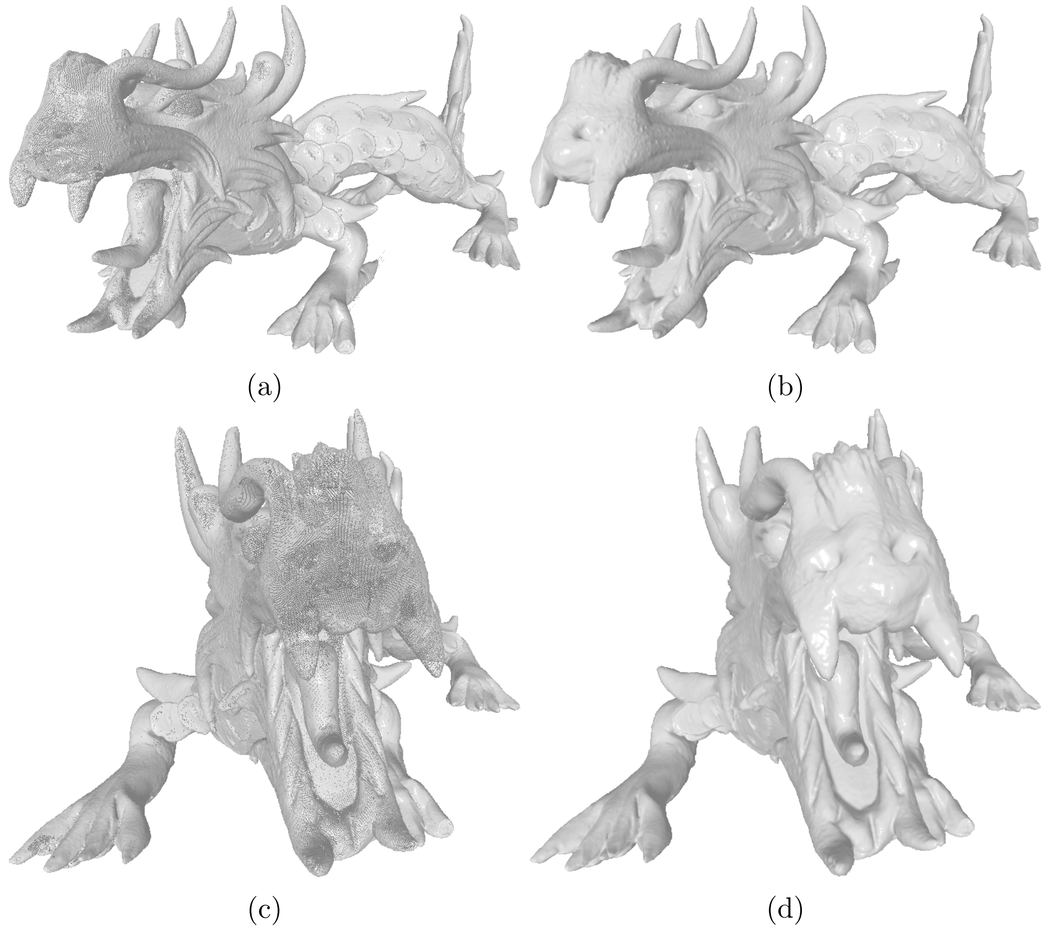


Figure 5.11: Asian Dragon

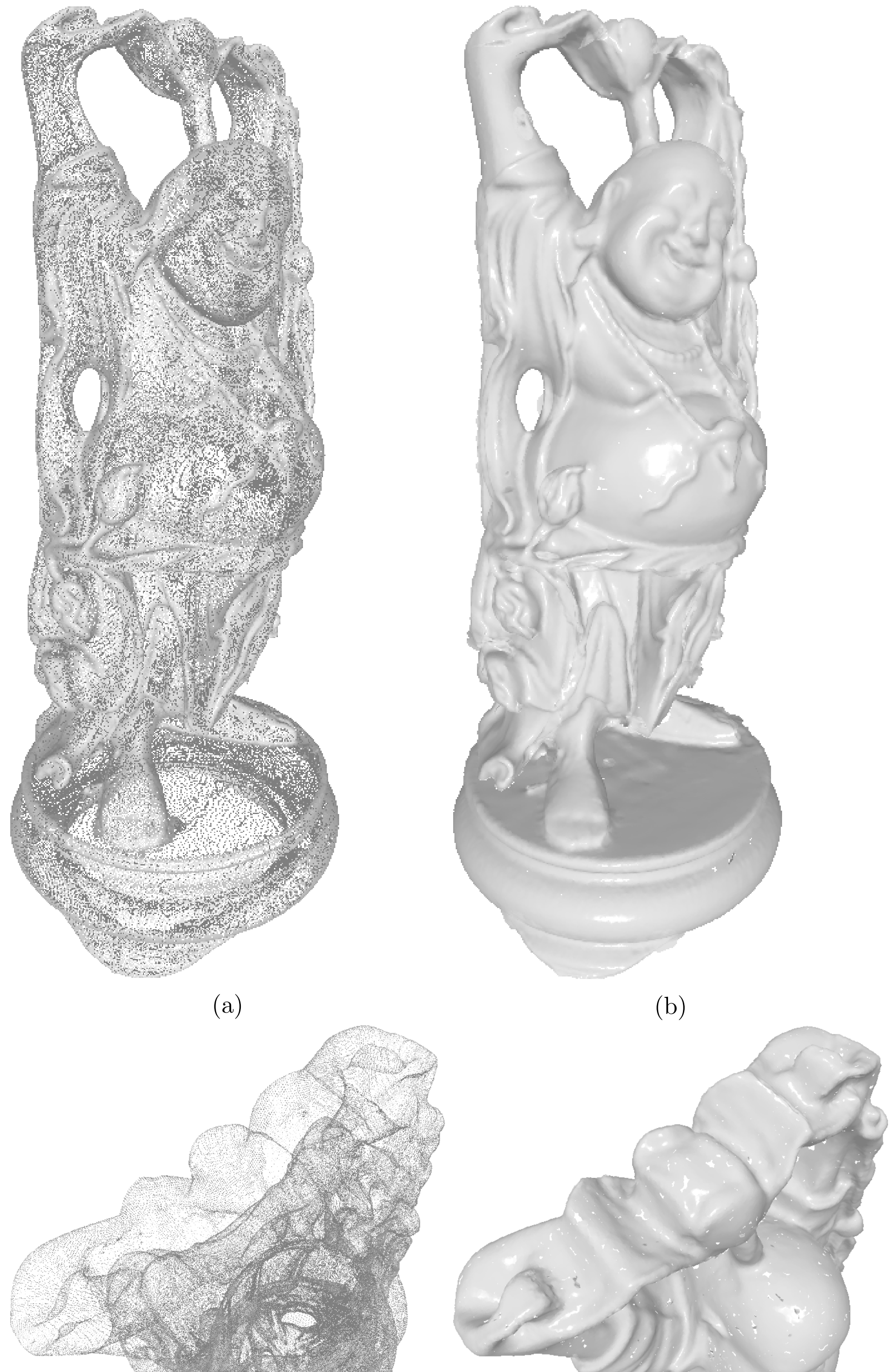
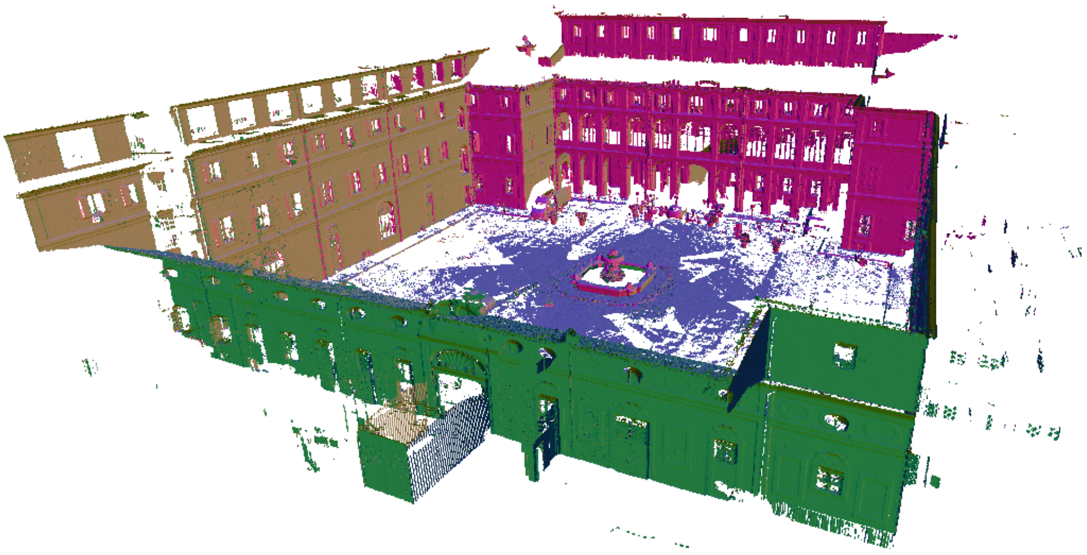
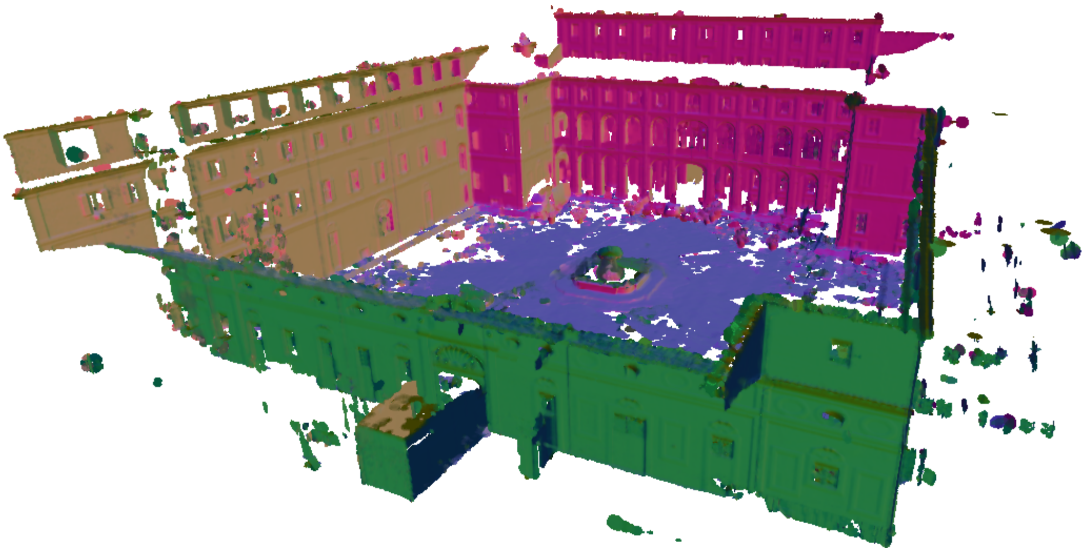


Figure 5.12: Buddha

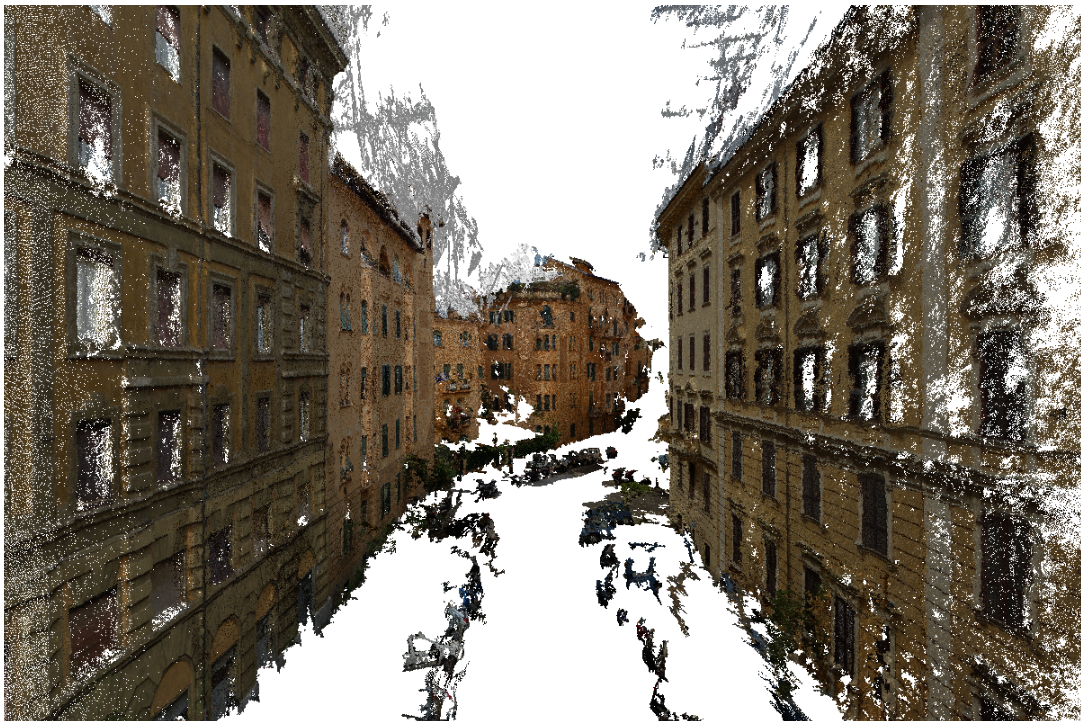


(a)

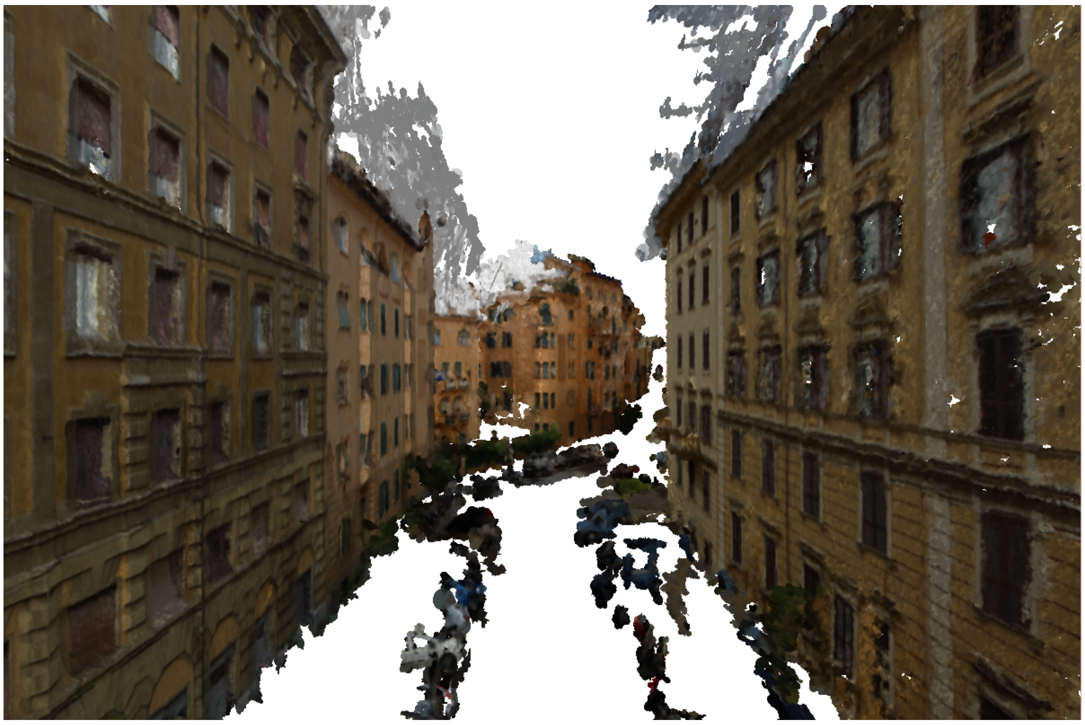


(b)

Figure 5.13: Schimele

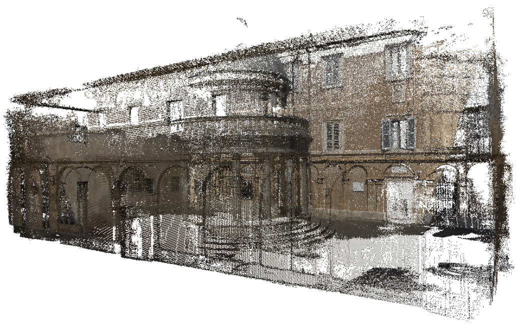


(a)

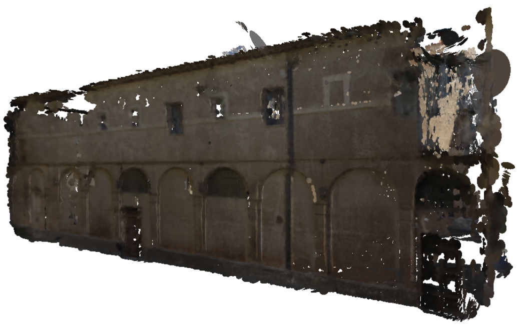


(b)

Figure 5.14: Strada Ruffini

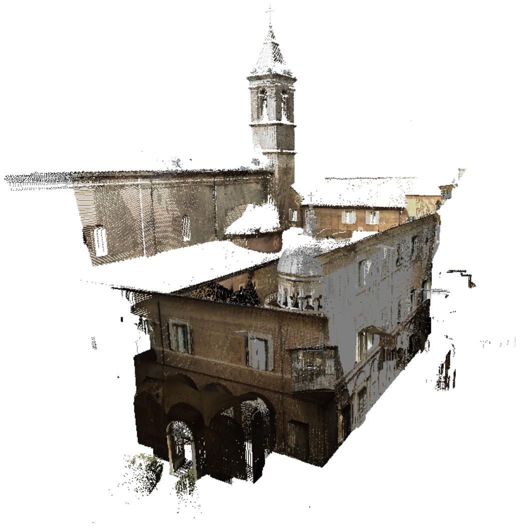


(a)

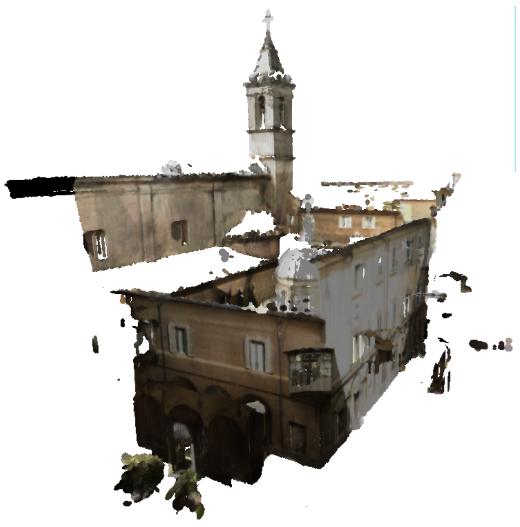


(b)

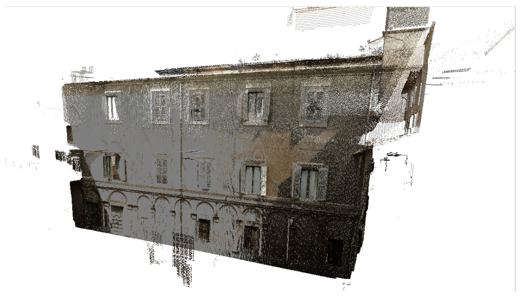
Figure 5.15: Tempietto



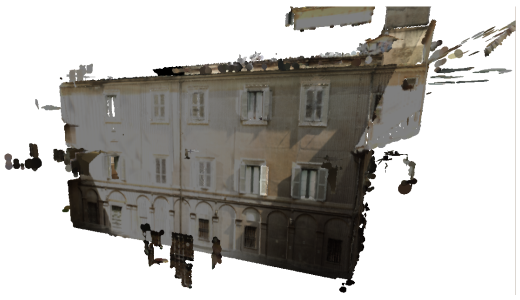
(a)



(b)



(c)



(d)

Figure 5.16: Esportazione (1)

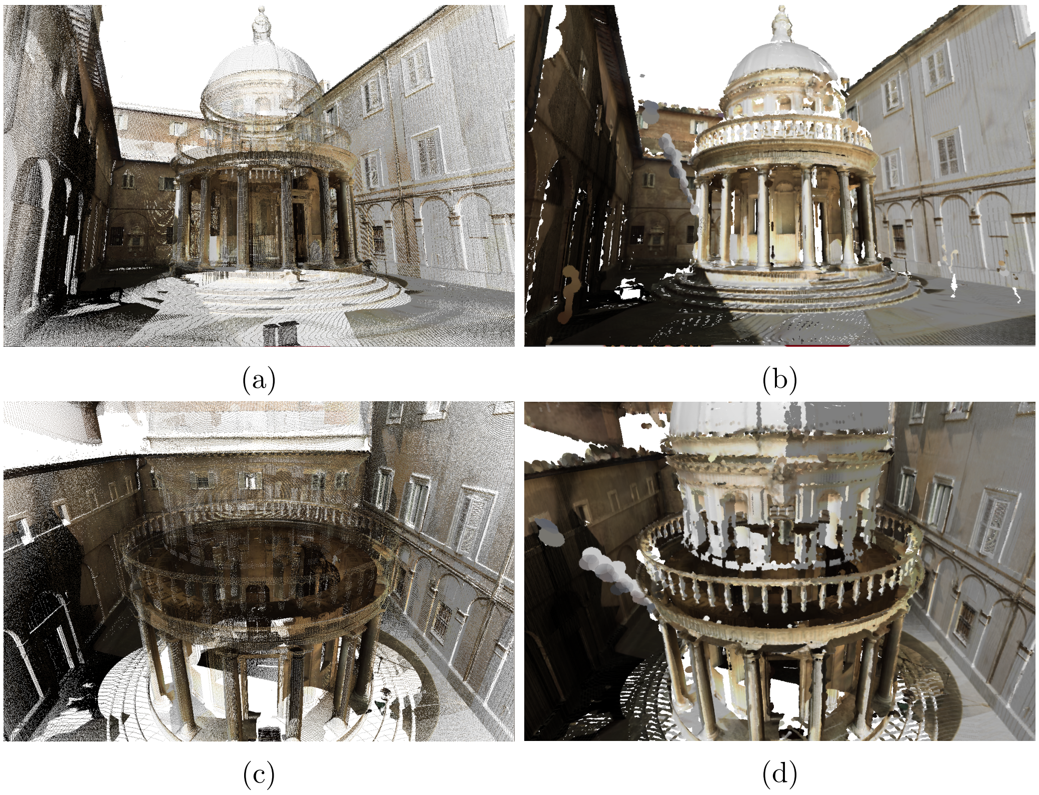


Figure 5.17: Esportazione (2)

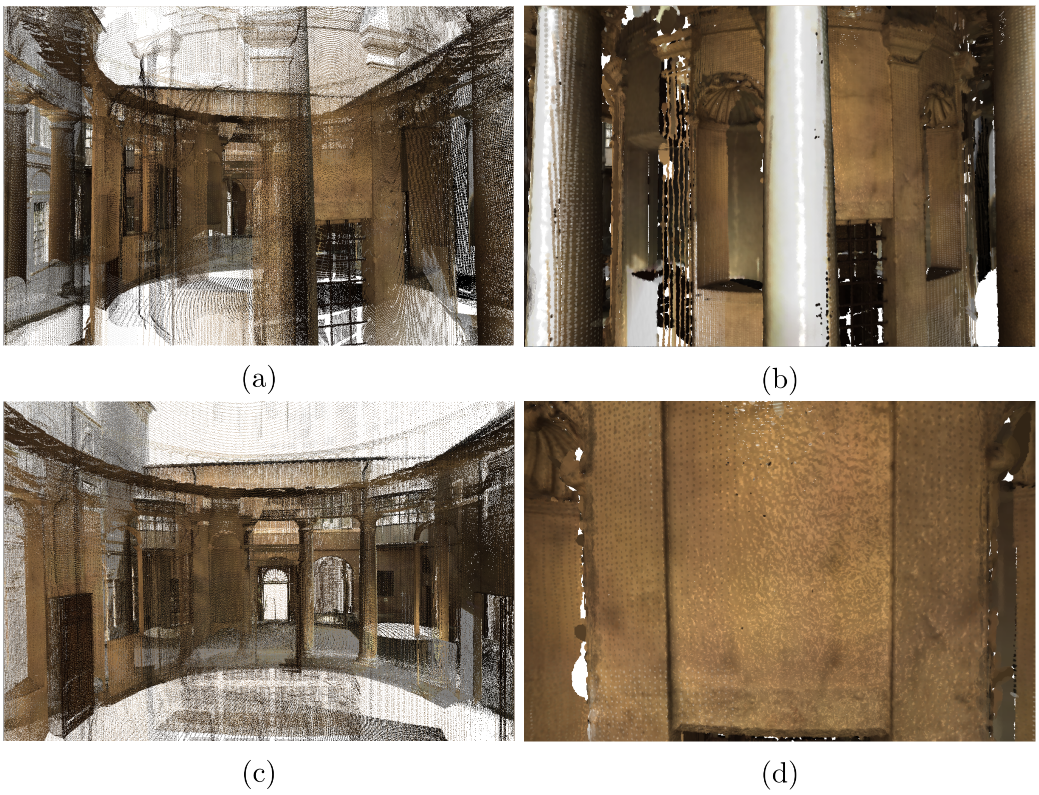


Figure 5.18: Sparse foreground and dense background for the Esportazione model.

5.1 Implementation Details

The computer used to perform our experiments was running 64-bit Ubuntu release 13.10, with an Intel Core i7 processor, a GeForce GTX 770, and 16GB of RAM. The Nvidia driver version was 319.32.

Our rendering resolution was 1200x800. The buffers used in the Projection, Layer Search and Closest Neighbor Search steps were half of this, 600x400. The depth buffer and the rendering buffers used in the final step had the same resolution as the rendering resolution.

In our implementation, we used $n = 4$ layers and $k = 4$ neighbors. Initially this was because of the limitation of attaching at most 8 color attachments in the GPU and our use of the ping-pong strategy. However, we found this to be sufficient for good results on the models we tested, since most projected scenes do not have more than four layers. Further, because we are working with screen-space coordinates, (x,y) pairs, during the JFA executions, we were able to pack the 4 pairs into a single float using the GLSL packHalf2x16 function for writing, and unpacking them with unpackHalf2x16 for reading. This allowed us to use just two Layer buffers and two Closest Neighbor buffers instead of eight buffers each.

The following parameters were settled on through a process of trial-and-error. We tested many different values and settled on a combination that produced what we believe to be the best results.

In our tests, an angle threshold of 0.5 and continuity threshold of 0.05 for the candidate test yielded good results.

The optimal max distance threshold is dependent on the density of the model. We used a max distance threshold as low as 0.01 (1% of the model's scale) for dense models and as high as 0.1 (10%) for sparse models. Setting these thresholds too low eliminates splats and may leave gaps in the model. Setting them too high may allow large splats to appear, usually those formed by outliers.

The max radius threshold was equal to the max distance threshold. The max relative radius threshold was set to 6.

We found that limiting the JFA's initial step to a degree did not have any averse effect on the results. This is reasonable, because the process tends to end up with points close to it, anyway. Even with smaller initial jumps, the information was able to be propagated. We settled on an initial jump step of 2^6.

Chapter 6

Conclusions

In this work, we presented a method of rendering raw point clouds without any pre-processing, auxiliary data structures, or density information. We applied the Jump Flooding Algorithm to the problem of finding different surfaces of the projected model. We used this information in a second search, also using the Jump Flooding Algorithm, to find nearest neighbors for each projected point of the model. With each point and its local neighborhood, we determined a radius for each splats and blended them together to produce the final rendering.

The main advantage of our method is that it does not require any prior knowledge of the density of the model, and works regardless of if it is sparse or dense. It also works when dealing with complex densities, such as a sparse foreground and dense background. Because we separate the layers and give every surface a chance to propagate its information, points correctly build their local neighborhoods and are able to render reasonable splats. Other methods attempt to solve the density problem using different forms of culling, such as masks. The problem with this approach is that it does not deal with all densities well. Further, it usually requires a small window of pixels to have a decent amount of projected points inside of it to have enough information to work with. This does not always happen, especially with very sparse surfaces.

Because we do not need to do any pre-processing, our method is reasonably fast. It scales well with the size of the dataset and the rendering resolution, as discussed in the Results chapter.

We demonstrated the robustness of our method on datasets of greatly varying characteristics. Some of them have little or no noise nor outliers, such as Armadillo, Asian Dragon, and Stay Puft. This is because these models are in fact triangle meshes, and we simply discarded the connectivity information when processing them. Other models, like Tempietto, Esportazione, and Strada Ruffini, are scanned, and thus contain noisy normals and outliers. We were still able to separate layers and achieve good or at least reasonable visual results on all datasets.

However, our method does not work as well with datasets that have very noisy normals. This can cause the results of the Layer Search to become somewhat arbitrary, decreasing the robustness of the Closest Neighbor Search. Another limitation is that, aside from the max distance thresholds we apply in our method, we don't explicitly deal with outliers. Groups of outliers are commonly present in scanned data and can spawn splats that clearly do not belong in the rendering.

Chapter 7

Future Works

There are several directions for future works.

Currently, we limit points from pairing up with distant points by manually setting a global threshold. A better solution would be to determine this threshold dynamically based on the model. This would require either some pre-processing or access to information that we do not assume we have at the moment (density information).

Another possibility is the use of a subdivision data structure for level of detail. By determining points we can quickly and safely eliminate based on the current view, we might be able to improve the overall quality of results. This is because less problem points leads to better neighbors for the points, resulting in splats that more closely resemble the surface we are trying to approximate. It should also considerably lower projection and rendering times. Because these two steps make up the majority of the execution times on large datasets, that could result in considerable speed gains.

Another idea would be to eliminate the necessity of normals altogether. At the moment, we use normals to perform the Layer Search candidate test, as described in Section 4.2. We use them in our lighting calculations and to orient the splats during rendering, as well. However, these are not strictly necessary, and in any case the normals could be calculated using the point's neighborhood. Without requiring normals, the method could work with the simplest of point clouds, those with only positional information. The challenge would lie in coming up with a replacement candidate test that does not use normals but is still robust enough to separate the layers.

Finally, there are many other small improvements that could be made. The splats that we draw are currently perfect circles. With more neighbors, the method could allow for elliptical splats. Our blending considers only distance when weighing fragment contributions; colors could also be taken into account. Another possibility is to apply smoothing filters to our rendering solution.

Bibliography

[1] 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*, pp. 109–116, 2006. doi: http://doi.acm.org/10.1145/1111411.1111431. Disponível em: <http://doi.acm.org/10.1145/1111411.1111431>.

[2] RONG, G., TAN, T.-S. “Variants of Jump Flooding Algorithm for Computing Discrete Voronoi Diagrams.” In: *ISVD*, pp. 176–181. IEEE Computer Society, 2007. Disponível em: <http://dblp.uni-trier.de/db/conf/isvd/isvd2007.html#RongT07>.

[3] RONG, G., TAN, T.-S. “Utilizing Jump Flooding in Image-based Soft Shadows.” In: Slater, M., Kitamura, Y., Tal, A., et al. (Eds.), *VRST*, pp. 173–180. ACM, 2006. ISBN: 1-59593-321-2. Disponível em: <http://dblp.uni-trier.de/db/conf/vrst/vrst2006.html#RongT06>.

[4] CARR, J. C., BEATSON, R. K., CHERRIE, J. B., et al. “Reconstruction and Representation of 3D Objects with Radial Basis Functions”. In: *Computer Graphics (SIGGRAPH 01 Conf. Proc.), pages 6776. ACM SIGGRAPH*, pp. 67–76. Springer, 2001.

[5] OHTAKE, Y., BELYAEV, A., ALEXA, M. “Sparse Low-degree Implicit Surfaces with Applications to High Quality Rendering, Feature Extraction, and Smoothing”. In: *Proceedings of the Third Eurographics Symposium on Geometry Processing*, SGP ’05, Aire-la-Ville, Switzerland, Switzerland, 2005. Eurographics Association. ISBN: 3-905673-24-X. Disponível em: <http://dl.acm.org/citation.cfm?id=1281920.1281944>.

[6] KAZHDAN, M., BOLITHO, M., HOPPE, H. “Poisson Surface Reconstruction”. In: *Proceedings of the Fourth Eurographics Symposium on Geometry Processing*, SGP ’06, pp. 61–70, Aire-la-Ville, Switzerland, Switzerland, 2006. Eurographics Association. ISBN: 3-905673-36-3. Disponível em: <http://dl.acm.org/citation.cfm?id=1281957.1281965>.

[7] MULLEN, P., DE GOES, F., DESBRUN, M., et al. “Signing the Unsigned: Robust Surface Reconstruction from Raw Pointsets”, *Computer Graphics Forum*, v. 29, n. 5, pp. 1733–1741, 2010. doi: 10.1111/j.1467-8659.2010.01782.x. Disponível em: <`http://dx.doi.org/10.1111/j.1467-8659.2010.01782.x`>.

[8] HOPPE, H., DEROSE, T., DUCHAMP, T., et al. “Surface Reconstruction from Unorganized Points”, *SIGGRAPH Comput. Graph.*, v. 26, n. 2, pp. 71–78, jul. 1992. ISSN: 0097-8930. doi: 10.1145/142920.134011. Disponível em: <`http://doi.acm.org/10.1145/142920.134011`>.

[9] KAWATA, H., KANAI, T. “Direct Point Rendering on GPU”. In: *Proceedings of the First International Conference on Advances in Visual Computing*, ISVC’05, pp. 587–594, Berlin, Heidelberg, 2005. Springer-Verlag. ISBN: 3-540-30750-8, 978-3-540-30750-1. doi: 10.1007/11595755_71. Disponível em: <`http://dx.doi.org/10.1007/11595755_71`>.

[10] DIANKOV, R., BAJCSY, R. “Real-time adaptive point splatting for noisy point clouds”. In: *GRAPP (GM/R)’07*, pp. 228–234, 2007.

[11] PREINER, R., JESCHKE, S., WIMMER, M. “Auto Splats: Dynamic Point Cloud Visualization on the GPU”. In: Childs, H., Kuhlen, T. (Eds.), *Proceedings of Eurographics Symposium on Parallel Graphics and Visualization*, pp. 139–148. Eurographics Association 2012, maio 2012. ISBN: 978-3-905674-35-4. Disponível em: <`http://www.cg.tuwien.ac.at/research/publications/2012/preiner_2012_AS/`>.

[12] DOBREV, P., ROSENTHAL, P., LINSEN, L. “Interactive Image-space Point Cloud Rendering with Transparency and Shadows”. In: Skala, V. (Ed.), *Communication Papers Proceedings of WSCG, The 18th International Conference on Computer Graphics, Visualization and Computer Vision*, pp. 101–108, Plzen, Czech Republic, 2 2010. UNION Agency – Science Press. dobrevrosenthallinsenvcgl.

[13] PINTUS, R., GOBBETTI, E., AGUS, M. “Real-time Rendering of Massive Unstructured Raw Point Clouds Using Screen-space Operators”. In: *Proceedings of the 12th International Conference on Virtual Reality, Archaeology and Cultural Heritage*, VAST’11, pp. 105–112, Aire-la-Ville, Switzerland, Switzerland, 2011. Eurographics Association. ISBN: 978-3-905674-34-7. doi: 10.2312/VAST/VAST11/105-112. Disponível em: <`http://dx.doi.org/10.2312/VAST/VAST11/105-112`>.

[14] YANG, R., GUINNIP, D., WANG, L. "View-dependent Textured Splatting", *The Visual Computer*, v. 22, n. 7, pp. 456–467, 2006. ISSN: 0178-2789. doi: 10.1007/s00371-006-0015-5. Disponível em: <http://dx.doi.org/10.1007/s00371-006-0015-5>.

[15] GUENNEBAUD, G., BARTHE, L., PAULIN, M. "Real-time Point Cloud Refinement". In: *Proceedings of the First Eurographics Conference on Point-Based Graphics*, SPBG'04, pp. 41–48, Aire-la-Ville, Switzerland, Switzerland, 2004. Eurographics Association. ISBN: 3-905673-09-6. doi: 10.2312/SPBG/SPBG04/041-048. Disponível em: <http://dx.doi.org/10.2312/SPBG/SPBG04/041-048>.

[16] MARROQUIM, R., KRAUS, M., CAVALCANTI, P. R. "Efficient Point-Based Rendering Using Image Reconstruction". In: *Proceedings of the Third Eurographics Symposium on Geometry Processing*, pp. 101–108, 2007.