DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics: Games Engineering

Level of Detail Generation for Point Clouds

Patrick Radner

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics: Games Engineering

Level of Detail Generation for Point Clouds

Level of Detail Generierung für Punktwolken

Author:	Patrick Radner
Supervisor:	Prof. Dr. Rüdiger Westermann
Advisor:	M.Sc. Henrik Masbruch
Submission Date:	15. August 2018

I confirm that this bachelor's thesis in informatics: games engineering is my own work and I have documented all sources and material used.

Munich, 15. August 2018 Patrick Radner

Acknowledgments

Firstly I would like to thank my adviser, Henrik Masbruch, for guiding me through this thesis and providing me with the necessary materials and motivation. And for his fast response times.

I would also like to thank Prof. Westermann for posting this thesis work and getting me started with a short recap of the entire point cloud thematic.

Thanks to Markus Schuetz and the team behind Potree. A large part of this thesis is based on their work.

Thanks to the Stanford Computer Graphics Laboratory for providing and maintaining their 3D scanning repository [Lab18]. It provides a great variety of 3D models, which were used to test and demonstrate the algorithms shown in this paper.

Abstract

With the increasing quality of 3D models the conventional approach of representing 3D models as a closed polygonal surfaces looses efficiency. Point Clouds offer an alternative by only using point samples of the 3D surface. These samples do not require connectivity information, drastically simplifying the rendering process. In addition, data from 3D Scans, LIDAR, etc. is often only available as point data, and would otherwise have to be converted into a polygonal model. Point clouds can often consist of hundreds of millions or even billions of samples. It is therefore necessary to find a way to simplify regions that are further away from the camera in order to not draw an entire dataset every frame. This is commonly known as Level of Detail.

This Thesis revisits the sub sampling approach presented in Potree. Afterwards we explore clustering based simplification techniques as a way to not only use the position of samples, but also take advantage additional information such as curvature and color. Human-made objects often contain planar surfaces. By using geometric information as a criterion for creating our level of detail hierarchy, we manage to drastically reduce the sample density in such planar regions, while still being able to conserve complex geometries.

Contents

1 Introduction

Point clouds represent three-dimensional surfaces as a set of points, so-called surfels (surface elements) [Pfi+00]. In contrast to conventional triangle-based meshes, these surfels do not contain any form of neighborhood or connectivity information. This makes them preferable in regions of high geometrical complexity, where a large number of primitives is needed to adequately represent the given surface.

The rendering pipeline is further simplified by storing surface information such as color or normals for each point, thereby removing the need for texture mapping.

In addition various 3D scanning methods, such as laser scanning, LIDAR, some 3D camera reconstruction techniques, etc. only output point clouds. It can therefore be preferable to draw the point cloud directly instead of adding other – costly – preprocessing steps before displaying the scanned object. [Sch16; Pfi+00; RL01; Bot+05]

The constantly improving precision of these scanning techniques leads to a higher sample count per surface area which in turn results in point clouds becoming larger and larger. Modern point clouds often consists of hundreds of millions or even billions of point-samples. This means it is not possible to draw the entire point cloud each frame. It is therefore necessary to simplify them in areas of lower interest to the user, i.e. further away from the camera, as well as enable efficient ways to decide which parts of the point cloud are actually visible on screen (culling).

1.1 Problem Definition

Modern point clouds can contain up to several billion samples. Current graphics hardware is not capable of drawing such a large amount of primitives at interactive frame rates. The goal of this thesis is to present different ways to implement a Level of Detail (LoD) hierarchy, in order to manage point clouds.

In a modern human-made world, there are lots of flat areas. Point clouds usually fall short, when it comes to representing planes, as a large amount of almost identical samples is required, to represent planar surfaces. It is therefore desirable, to find an approach, that can deal with planes in a point cloud, while still preserving interesting features, like geometry or color.

1.2 Structure

Chapter 2 will provide an overview of related work. An explanation of the octree and nested octree data structures used will be given in Chapter 3. In Chapter 4 we will

explain how point primitives are rendered. Chapter 5 will review the Level of Detail approach presented inSchütz [Sch16] and Chapter 6 will present a clustering based alternative. In Chapter 7 we will mention implementation specific details, such as used libraries. Chapter 8 will compare performance in generation of the LoD hierarchy, as well as performance and quality during rendering. Finally, we will offer a conclusion and mention further possible improvements and additions to our implementation in Chapter 9.

2 Related Work

Using points as primitives was first introduced by Levoy and Whitted [LW85]. QSplat, by Rusinkiewicz and Levoy [RL01], was the first point based system capable of rendering hundreds of millions of points, by utilizing a bounding sphere hierarchy. A high quality approach for surface splatting was proposed in Zwicker, Pfister, Baar, and Gross [Zwi+01]. A summary of point cloud rendering techniques was provided by Alexa, Darmstadt, Gross, et al. [Ale+02].

Pauly, Gross, and L. P. Kobbelt [PGK02] compared simplification techniques for point clouds based on subsampling, clustering or particle simulation. In Pfister, Zwicker, Baar, and Gross [Pfi+00] surfels are stored in an octree hierarchy. A hybrid approach using both point and rasterization based rendering was developed by Reichl, Chajdas, Bürger, and Westermann [Rei+].

2.1 Octrees

The concept of octrees was introduced by Meagher [Mea80]. Wimmer and Scheiblauer [WS06] then introduced nested octrees, which build the basis for the level of detail hierarchies implemented in this thesis. Nested octrees are also used in Schütz [Sch16]. Modifiable nested octrees (MNOs) were developed by Scheiblauer [Sch14] and allowed to select, insert and delete samples from a point cloud.

2.2 Subsampling

The subsampling approach presented in this thesis is mainly based on Schütz [Sch16]. Approaches with a similar concept were also used in Scanopy [Sch14]. Stochastic sampling for computer graphics was introduced by Cook [Coo86] in the context of ray tracing. Possion disk sampling, which is used in [Sch16], was presented by McCool and Fiume [MF92].

2.3 Clustering

Next to simplification and level of detail generation, clustering approaches can also be used for feature extraction. Feature extraction algorithms generally look for a lot fewer clusters than simplification approaches and can thus not be used for our purposes.

Clustering approaches for generating a level of detail hierarchy in point clouds were implemented by, for example, Fan, Huang, and Peng [FHP13] and Moenning and

Dodgson [MD03]. In Shi, Liang, and Liu [SLL11] the kmeans algorithm was used to simplify point clouds, a maximum normal deviation was also considered in the clustering approach. Song and Feng [SF08] proposed an approach in which the goal is to simplify a point cloud a set maximum number of points. A globally optimal clustering approach, as well as a hole-filling method, was proposed in Wu and L. Kobbelt [WK04]. The Extremal Points Optimal Sphere (EPOS) algorithm was introduced by Larsson [Lar08] and provides a fast method for finding bounding spheres for a given set of points.

3 Data Structures

Our algorithms use different variations of the octree data structure to create a spatial hierarchy, which is in turn used to determine the level of detail in a specific region. We will show, how such a data structure is created and how it can be accessed.

3.1 Octrees

Octrees are a simple way to partition 3D space. They were first introduced by Meagher [Mea80]. Each node in an octree represents an axis-aligned cube. The root node hereby is equal to the bounding cube of a given object. As long as a certain criterion, such as a minimum number of points per node, is met, nodes are split into 8 children. This is shown in Figure 3.1.

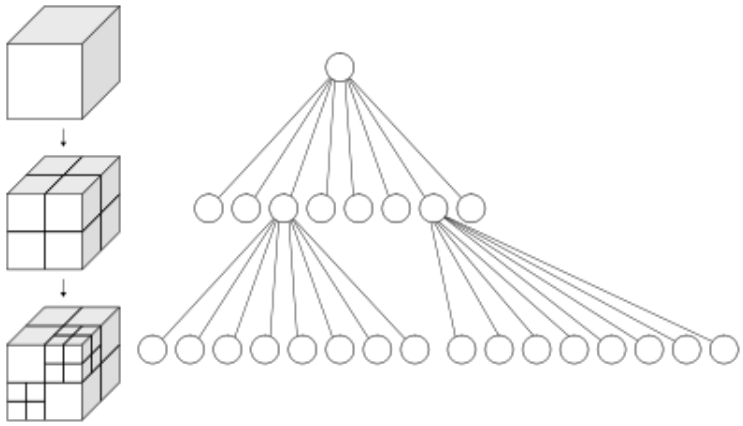


Figure 3.1: Left: Bounding cube divided into octree. Right: Tree representation. Source: [Wik18]

3.1.1 Nested Octrees

Each node in a nested octrees contains a 3-dimensional grid with at most one sample per grid cell. Points that would be mapped to the same grid cell will be stored in the child node. The concept of was introduced by Wimmer and Scheiblauer [WS06] and is also the principle structure used in Schütz [Sch16], who also showed, that in practice a resolution of 128^3 tends to provide a good balance between points per node and overall node count. Algorithm 1 shows, how a nested octree data structure would be created

for a given point cloud. Since the grid structure will only be sparsely occupied it is implemented using a hash map [Sch14; Sch16]. The key is created by calculating the 3D index of the cell occupied by a given point and then flattening it to one dimension. This is shown in Line 6 and 7. Insert and delete options for additional samples are not provided at this point, seeing as we are only working with static point clouds.

As to not unnecessarily create lots of small nodes, all points that would be moved to the child nodes are stored in temporary buffers corresponding to the child node they would be added to [Sch16]. A node is only expanded, if the number of points in its buffer exceeds a certain threshold. Assuming a somewhat even distribution of samples a value of 500 has been found to give the desired results.

Algorithm 1 Nested Octree Create

```
1:  procedure CREATE(Node node, Point[] pointCloud, BoundingCube bounds)
2:      if node = root then
3:          bounds ← boundingCube(pointCloud)
4:      for each v in pointCloud do
5:          vc ← v − bounds.start
6:          indexVector ← floor(vc/bounds.length/gridResolution)
7:          gridIndex ← indexVector.x + 128 ∗ indexVector.y + 128^2 ∗ indexVector.z
8:          if canInsert(node.data, gridIndex, v) then
9:              node.data[gridIndex] ← v
10:         else
11:             childIndex ← 0
12:             if vc.x < bounds.length/2 then
13:                 childIndex ← childIndex + 1
14:             if vc.y < bounds.length/2 then
15:                 childIndex ← childIndex + 2
16:             if vc.z < bounds.length/2 then
17:                 childIndex ← childIndex + 4
18:             buffer[childIndex].insert(v)
19:     for i ← 1 . . . 8 do
20:         if size(buffer[i] > threshhold then
21:             childBounds ← subCube(bounds, i)
22:             CREATE(node.children[i], buffer[i], childBounds)
23:         else
24:             node.data.insert(buffer[i])
```

After all points have been hashed into the grid or passed onto child nodes, they are then stored in a vector, in order to save space and allow a simpler handling for tasks, such as uploading them to video memory. Once the entire structure is created, it is stored breadth first into a vector for increased memory efficiency during traversal. This is shown in Algorithm 2.

Algorithm 2 Nested Octree to Vector

```
1:  procedure toVector(Vector out, dataFunction)
2:      curentIndex ← 0
3:      nodeBuffer ← root
4:      while ¬empty(nodeBuffer) do
5:          currentNode ← nodeBuffer.popFront()
6:          vectorNode.data ← dataFunction(currentNode.data)
7:          vectorNode.firstChildIndex ← size(nodeBuffer)
8:          vectorNode.children ← 0
9:          for i ← 1 . . . 8 do
10:             if exists(currentNode.children[i]) then
11:                 vectorNode.children ← vectorNode.children OR i
12:                 nodeBuffer.pushBack(currentNode.children[i])
13:         out[currentIndex] ← vectorNode
14:         curentIndex ← curentIndex + 1
```

4 Rendering

This chapter will explain how point clouds are rendered. As already mentioned, point clouds do not contain any form of neighborhood context. Instead every sample in a point cloud represents a small piece of surface area, also referred to as **surfel**. During rendering, surfels are drawn as small surfaces via a method called splatting [Bot+05; Zwi+01]. We will also determine, which nodes of a level of detail hierarchy, need to be rendered.

4.1 Point Attributes

Triangle-based rendering often uses techniques such as texture or normal mapping, where surface information is stored in external textures and sampled per frame. Point clouds simplify the process of gathering such information by storing it directly with each sample. This radically simplifies the entire rendering pipeline. Per sample attributes can contain:

- **Position:** A 3D vector placing the point in 3D space. This is the only necessary attribute. It ultimately determines where the point will be seen on screen.
- **Color:** The surface color of the sample. Not every point cloud possesses a color value. Results from laser scans, for example, usually don't have color values assigned to them. In our implementation a points color is stored in a 4 float RGBA vector. The alpha channel is not used.
- **Normal:** The normal vector to the tangent plane at the samples location. Most scanning methods do not provide normal information. Normals can however be approximated, as mentioned in subsection 4.1.2. Normals are stored in a 3D float vector. They are per definition always have unit length.
- **Radius:** In our clustering implementation different surfels at the same level of detail can have different sizes. Therefore each surfel has to store its size. Our implementation uses one float to store this value. Size information could be encoded, by scaling the normal vector. This would however require a normalization step per surfel to recover the Radius, which was deemed to costly.
- **Major and Minor:** When dealing with elliptical surfels, these two 3D vectors define direction and length of the major and minor semi-axis.

- **Level of Detail:** The sub-sampling approach presented by Schütz [Sch16] and shown in chapter 5 requires the level of detail to be calculated per sample per frame.

4.1.1 Additional Shader Data

In order to render a point cloud, additional data is needed:

- **World View Projection Matrix:** A 4×4 matrix used to transform objects from local space to screen space. It combines the world matrix, which performs rotation and translation, the camera matrix, which transforms the space so, that the camera is at the origin, looking along the Z-axis and finally the projection matrix used for perspective.
- **World Matrix:** A points position in world space needs to be known, if we want to illuminate our scene. It can be calculated by applying this 4×4 matrix. The world matrix is contained in the World View Projection matrix. If no shading is performed, this matrix is not required.
- **Light Direction:** Our demo world is illuminated by a single directional light source. This vector describes, from which direction the light comes will hit a surface.
- **Camera Position:** For specular lighting, as used in the Phong illumination model, the position of the camera needs to be known.
- **Max LoD:** The maximum depth of the level of detail hierarchy described by our octree.
- **Splat size:** The subsampling approach described in chapter 5 assumes globally consistent sample density. This means each sample of the original dataset has the same radius.
- **Bounding Box:** The bounding box of the entire object. It is used in the subsampling approach described in chapter 5, when determining the splat size in a given region.

4.1.2 Normal Generation

Most point clouds do not contain normal information. Our implementation uses normals for illumination. More importantly they can be used to determine surface complexity, which is considered in our clustering algorithm. Normals can be computed by fitting a plane to the k-nearest neighbors for each sample. This can be done by minimizing the least squares error given in Equation 4.1, representing the sum of projected distances (see also [WK04]). Our implementation is not capable of normal generation. Instead

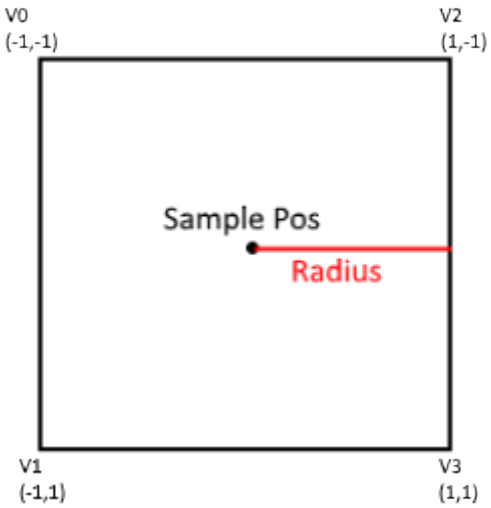


Figure 4.1: Image depicting, how a splat can be generated, given a position and a radius

point clouds without normal information are pre-processed in MeshLab, where per point normals can be computed.

$$E_{LS}(n) = \sum_{v \in k-NN(c)} n \cdot (v - c) \tag{4.1}$$

4.2 Splatting

Splatting [Bot+05; Zwi+01] describes the process of turning a point sample into a small surface area, a so-called splat. For quadratic and spherical splats the size of the area is either given by the radius attribute or calculated via Equation 5.2. For elliptical splats it is given by the length of the major and minor semi-axis. Splats are generally rendered as squares or quads. In the pixel shader they can be turned into circles or ellipses by discarding certain pixels. Splatting is performed in the geometry shader by creating four points in a triangle strip. This can be done for example with the code given in Listing 4.1. The resulting quad splat can be seen in Figure 4.1.

Listing 4.1: HLSL code to generate a splat from position and radius

```
output.pos = mul(input[0].pos, wvp);
output.pos.xy += float2(-1, 1) * radius;
output.tex.xy = float2(-1, -1);
OutStream.Append(output);
output.pos.y -= radius * 2;
output.tex.xy = float2(1, -1);
OutStream.Append(output);
output.pos.xy += float2(radius * 2, radius * 2);
```

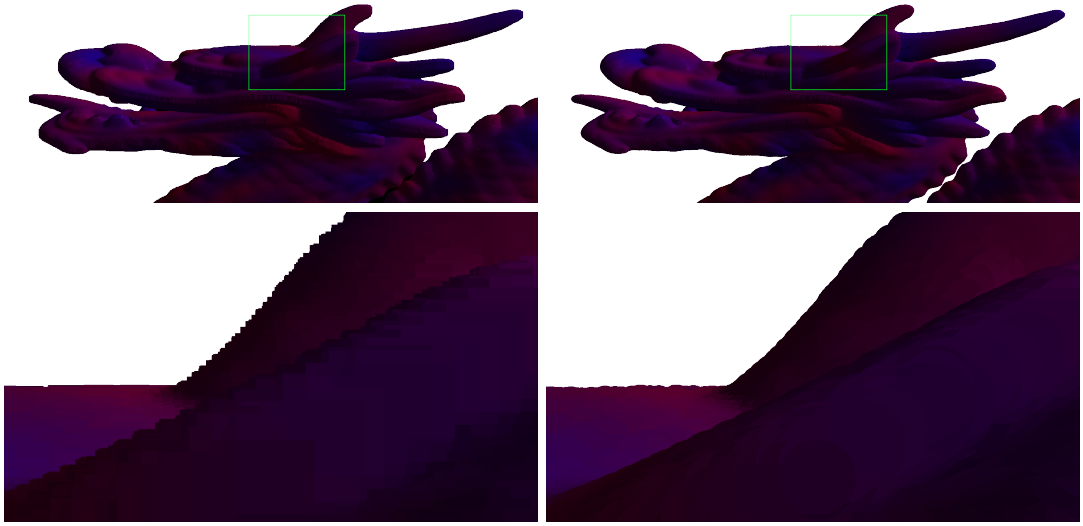


Figure 4.2: Left: Quad Splats, Right: Circle Splats, Model: Dragon_perlin_color

```
output.tex.xy = float2(-1, 1);
OutStream.Append(output);
output.pos.y -= radius * 2;
output.tex.xy = float2(1, 1);
OutStream.Append(output);
```

4.2.1 Circular Splats

Circular splats generally provide sharper edges. This can be seen in Figure 4.2. When working with point clouds in general, single samples are usually assumed to be small spheres or oriented circles in three dimensional space. For example the in chapter 6 presented clustering algorithm can compute spherical clusters (by ignoring normal information), which can be described by a center and a radius. By using circular splats, the resulting image tends to be closer to the results of the computed representatives.

In order to draw circular splats the texture coordinates, which were added in the geometry shader (Listing 4.1), are used. For a pixel to lie within the circle of its splat Equation 4.2 must be fulfilled. Otherwise the pixel will be discarded. This however results in circular splats needing to be slightly larger and some computational overhead.

$$tex.x^2 + tex.y^2 < 1 \tag{4.2}$$

4.2.2 Elliptical Splats

Elliptical surfels can be generated using the clustering algorithm described in chapter 6. They are described by a major and a minor semi-axis. During splatting the four

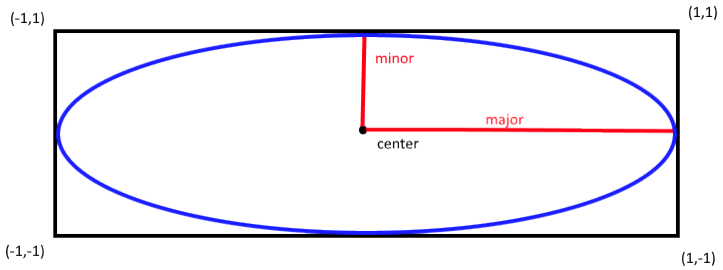


Figure 4.3: Elliptical splat (blue) drawn on a rectangle(black) determined by a surfels major and minor semi-axis (red). The texture coordinates in the corners are used to determine if a pixel lies within the ellipse.

points describing the quadrilateral, upon which the ellipse is drawn, are calculated by respectively adding or subtracting both of the two semi-axis from the surfels center. This is shown in Figure 4.3. Equation 4.2 can be used to determine whether a pixel lies within the desired ellipse. This is possible since the space defined by our texture coordinates is implicitly stretched in the direction of the major semi-axis.

In our implementation ellipses are only drawn as oriented splats, since their orientation plays a key role during clustering (see chapter 6).

4.3 Oriented Splats

Splats can be oriented along a points normal. This results in a better surface representation and allows surface geometry to be better recognized [Zwi+01; Bot+05]. The results of orienting splats can be seen in Figure 4.4.

In order to orient a splat by its normal we need two tangent vectors that span the plane defined by the normal. The first vector can be computed by either solving Equation 4.3 or by taking the cross product of the normal and any vector, that is not a scalar multiple of it. The second tangent is then computed by taking the cross product of the normal and the first tangent one (Equation 4.4). Both tangent vectors are then normalized and scaled by the splat radius. Afterwards they are added or subtracted from the points center to determine the four points of our quad in local space. This is similar to creating an elliptical splat, as shown in Figure 4.3. The major and minor are hereby replaced with the two computed tangent vectors. The transformation to screen space is then applied separately to each of those points.

$$tangent1 \cdot normal = 0 \tag{4.3}$$

$$tangent2 = normal \times tangent1 \tag{4.4}$$

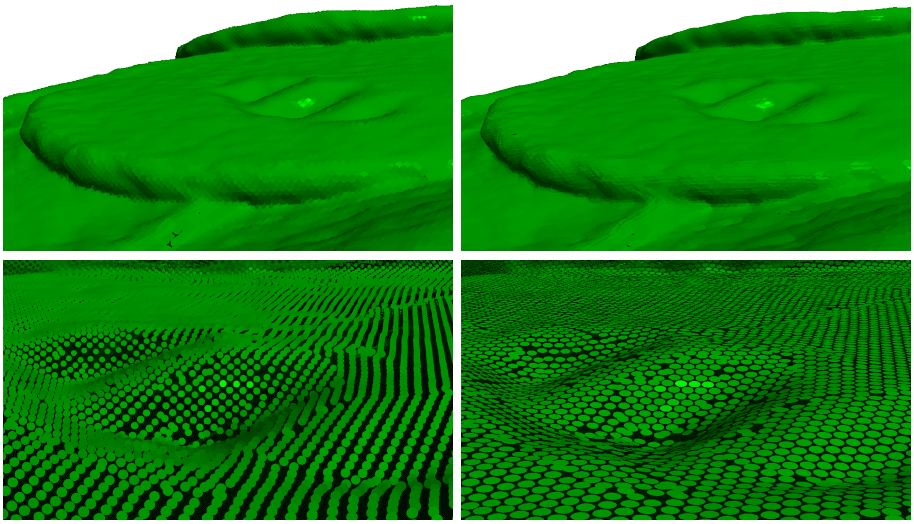


Figure 4.4: Left: Screen aligned splats, Right: oriented splats, Top: closed surface, Bottom: close up and artificially shrunk splats, Model: xyzrgb_dragon

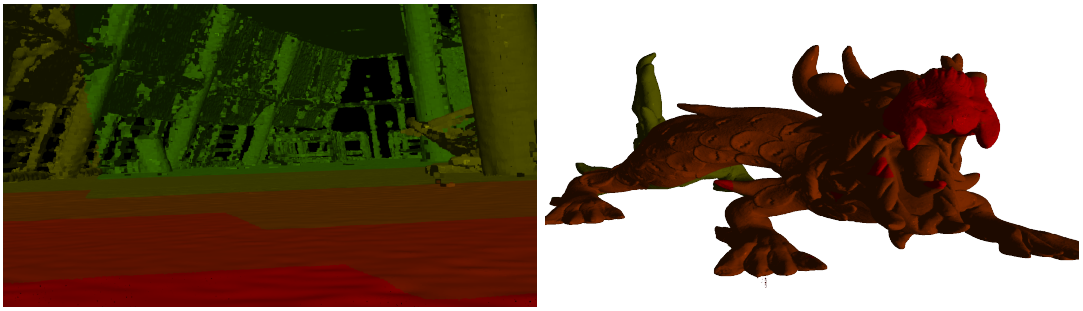


Figure 4.5: Objects colored based on their LoD. red: high, green: low. left: navvis_tum_audimax_half, right: xyzrgb_dragon

4.4 LoD Determination

During rendering the decision to draw a node of the objects octree hierarchy is met by determining the number of pixels covered by a single surfel. If that number lies above a certain threshold the octree is traversed further, otherwise it is not[Sch16]. Nodes in the octree that are explored during this traversal are referred to as **visible nodes**. The entire traversed tree structure is called **visible tree** and the leaves of the visible tree are named **visible leaves**. The traversal of the octree structure, which is stored in a vector, is given by Algorithm 3.

The calculation of the number of pixels covered is given by Equation 4.5 and Equation 4.6 as described in [Sch16]. The radius of a single surfel is either calculated using Equation 5.2 for sub-sampling approaches or stored per octree node in the case of clustering (seechapter 6). In order to calculate the distance, one can either transform

the nodes center into world space and calculate the distance to the camera position (Equation 4.7) or take the Z-component of the nodes center after transforming it into (projected) view space, as is shown in Equation 4.8. The result of applying such a LoD calculation per node can be seen in Figure 4.5, where surfels are colored based on the level of detail they are drawn at.

Assuming a freely moving camera or scene, the visible hierarchy will have to be recomputed for every frame. If scene and camera are both static, the level of detail hierarchy could be computed only once and used repeatedly. Alternatively, as shown in [Sch16] the resulting image could be continuously refined, if the scene and camera are both static. This is subject of future improvements and not used in the current implementation.

$$slope = tan(\frac{fov}{2}) \tag{4.5}$$

$$projectedSize = \frac{screenHeight}{2} * \frac{radius}{slope * distance} \tag{4.6}$$

$$distance = \|cameraPos - WorldMat \cdot NodeCenter\|_2 \tag{4.7}$$

$$distance = Zcomponent(WorldViewProjectionMat \cdot NodeCenter) \tag{4.8}$$

Algorithm 3 Determine Visible Nodes

```
1: procedure DETERMINEVISIBLENODES(Index nodeIndex = 0, int depth = 0, Vector3
   nodeCenter = ObjectBoundingCubeCenter)
2:     currentNode ← octreeVector[nodeIndex]
3:     performTasksForVisibleNodes()
4:     if failed(visibilityCheck(depth,nodeCenter,cameraPos)) OR currentNode.isLeaf()
   then
5:         performTasksForVisibleLeaves() return
6:     childCount ← 0
7:     for i = 0 · · · 8) do
8:         if currentNode.hasChildAtIndex[i] then
9:             childIndex ← currentNode.firstChildIndex + childCount
10:            childCenter ← calculateCenterOfChildNode(nodeCenter,depth,i)
11:            DETERMINEVISIBLENODES(childIndex,depth + 1,childCenter)
12:            childCount ← childCount + 1
```

4.4.1 Frustum Culling

As each octree node represents the bounding cube of the points contained within it, one can also perform frustum culling on a per node basis. Frustum culling describes

the process of determining whether an object lies within the region seen by camera, the so-called view frustum. Especially when close or even inside a point cloud this can provide a huge performance boost, as large parts of the scene will not be visible. There exist various algorithms for frustum culling, reviewing them, however, is not in the scope of this thesis. The reader is referred to Assarsson and Moller [AM99], which contains a review and improvements to commonly used algorithms.

4.5 Shading

Especially when no color information is available shading is important to recognize more than just the silhouette of an object. Our implementation uses Phong shading, based on the Phong's illumination model, which combines ambient, diffuse and specular lighting. It was introduced by Phong [Pho75]. We further assume only a single directional light source is placed in the scene. Phong shading is performed per pixel and requires the pixels position in world space, as well as the objects surface normal in world space. HLSL pixel shader code for a Phong shading implementation is presented in Listing 4.2. The difference between a lit and an unlit scene can be seen in Figure 4.6.

There are more advanced lighting implementations, for example Eye-Dome Lighting [Bou09], which was also implemented in [Sch16]. It has the huge advantage of not requiring normal information to be present in the original data and works by approximating local curvature during rendering.

Listing 4.2: HLSL implementation of Phong Shading

```
float4 lightning_phong(float3 worldPos, float3 normal)
{
    float3 r = reflect(-g_lightDir.xyz, normal);
    float3 v = normalize(g_cameraPos.xyz - worldPos);

    //lighting constants
    float cdiff = 0.5f;
    float cspec = 0.4f;
    float espec = 200; // specular exponent
    float camb = 0.15f;

    return (cdiff * saturate(dot(normal, g_lightDir.xyz))
      + cspec * pow(saturate(dot(r, v)), espec)
      + camb) * g_lightColor;
};
```

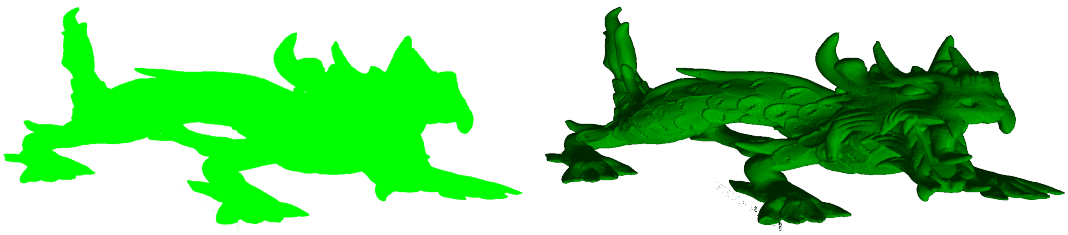


Figure 4.6: Model: xyzrgb_dragon; Left: no illumination; Right: Phong shading

4.6 Blending

To further improve the resulting image, and get rid of aliasing effects, one can perform blending operations on overlapping splats. The general idea of blending splats is to average all the color values in a small ϵ-depth behind the pixel closest to the camera. Those can further be weighted by their distance to the center of their respective surfel. Commonly a Gaussian weight is used. A blending algorithm for splats was introduced in Botsch, Hornung, Zwicker, and L. Kobbelt [Bot+05] and is also implemented and explained in detail in [Sch16].

5 Subsampling

This chapter is largely based on the level of detail method implemented in **Potree** [Sch16]. Subsampling describes methods, that use a subset of the original samples to approximate the model. A nested octree structure (subsection 3.1.1) is used to create the level of detail hierarchy. Algorithm 1 shows, how the octree is created. The actual subsampling technique is defined by the function $canInsert(data, index, point)$ in Line 8. Our implementation uses Possion-disk subsampling, following the results from Schütz [Sch16]:

> We also decided to replace subsampling on a grid with Poisson-disk subsampling. Poisson-disk subsamples are evenly spaced subsamples with a minimum distance between points, and they exhibit more naturally looking and pleasant patterns.

5.1 Possion-Disk Subsampling

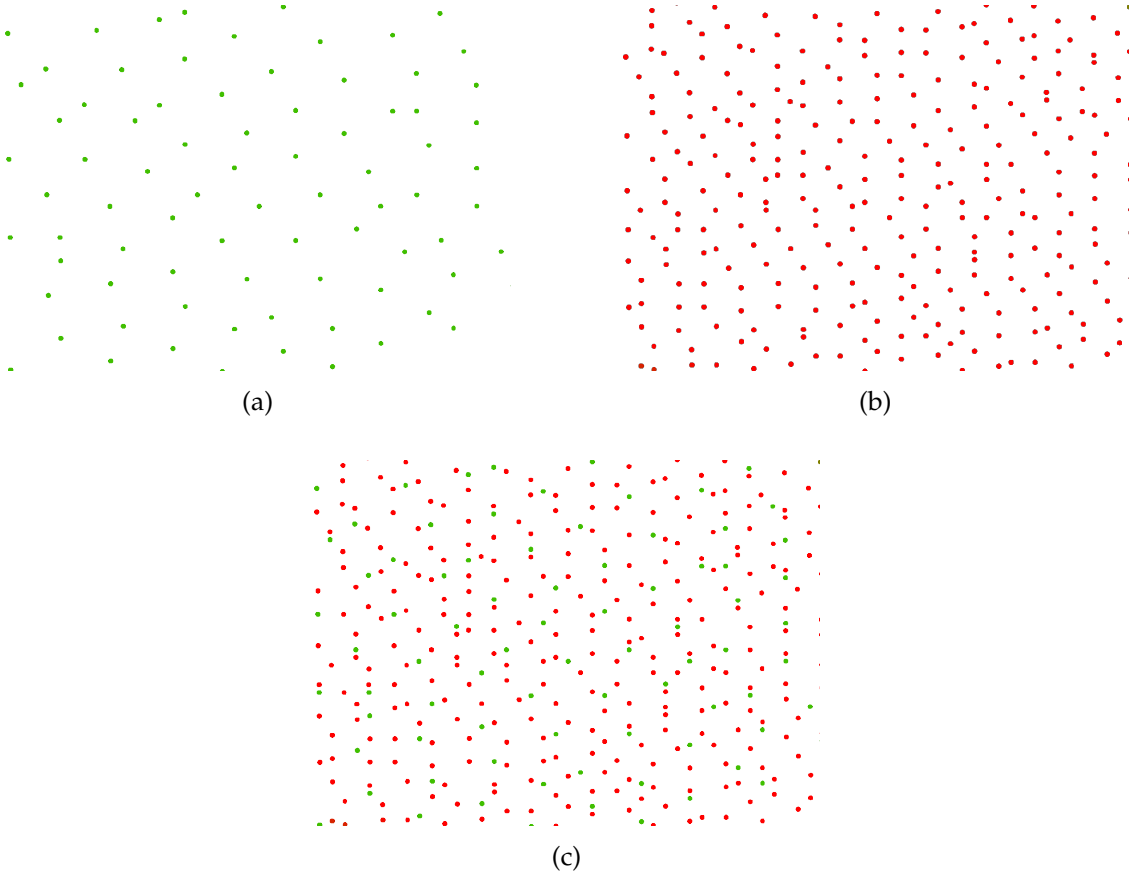
Possion-Disk subsampling describes a method, in which a minimum distance between all samples is enforced. To create a level of detail hierarchy, this minimum distance is then halved at each level, which approximately doubles the sample density. Possion-disk sampling could be achieved by simply taking a given sample and checking its distance against all other selected samples. Due to its quadratic complexity however, this method becomes intractable with larger point clouds.

Instead, as is shown in [Sch16], we can use the inscribed grid of our nested octree structure. By choosing the minimum sample distance to be the side-length of one grid cell, we simply have to check the cell, in which the sample would fall in, and its 26-connected neighbors. Such an implementation of the $canInsert()$ function mentioned by Algorithm 1 is shown in Algorithm 4. The 26-connected neighborhood in a 3D grid is defined by Equation 5.1. We again choose a grid resolution of 128^3 at each level, as it gives good results in practice. The result of iterative Possion-Disk sampling is shown in Figure 5.1.

$$\texttt{26-Neighborhood}\left(\begin{pmatrix} i_x \\ i_y \\ i_z \end{pmatrix}\right) = \{n \in \mathbb{N}^3 | n = \begin{pmatrix} i_x + \{-1,0,1\} \\ i_y + \{-1,0,1\} \\ i_z + \{-1,0,1\} \end{pmatrix} \wedge n \neq i \wedge \sum_i n_i < gridResolution^3\} \tag{5.1}$$

Algorithm 4 Insert Possion Disk

```
1: procedure CANINSERT(Grid[] grid, GridIndex index, Point v)
2:     if ¬empty(grid[index]) then return false
3:     for each neighborIndex in neighborhood(index) do
4:         if distance(grid[neighborIndex],v) < minimumDistance then return false
       return true
```



(a) (b)

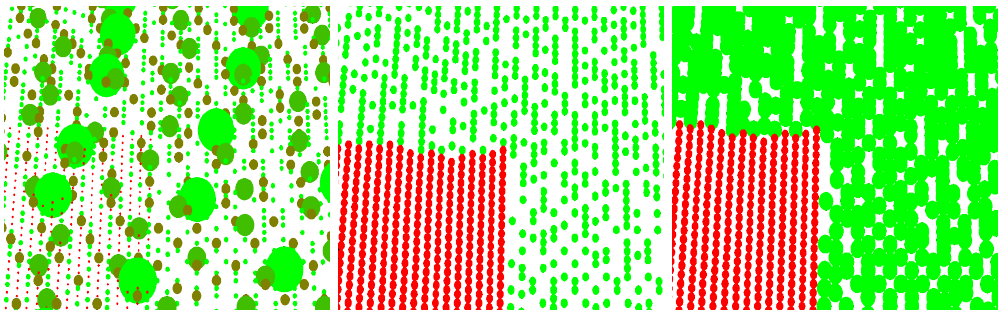
(c)

Figure 5.1: (a,b)Possion-Disk subsamples at LoD = 0/1; (c) final point cloud at LoD = 1, created by combining level 0 and 1

5.2 Splat Size Determination

Figure 5.1 shows, how a certain level of detail is formed by drawing all nodes in the octree hierarchy up until the lowest visible node. Since the main idea behind level of detail is to draw more samples in regions closer to the camera the maximum level of detail will inevitably vary across an entire object. The radius of a spat is given by Equation 5.2. The problem is now, to determine the level of detail for a given splat. Three methods will be explored in this section. Their effects can be seen in Figure 5.3.

$$splatSize(LoD) = originalSamlpeSize * 2^{Max\ LoD - LoD} \tag{5.2}$$



(a) (b) (c)

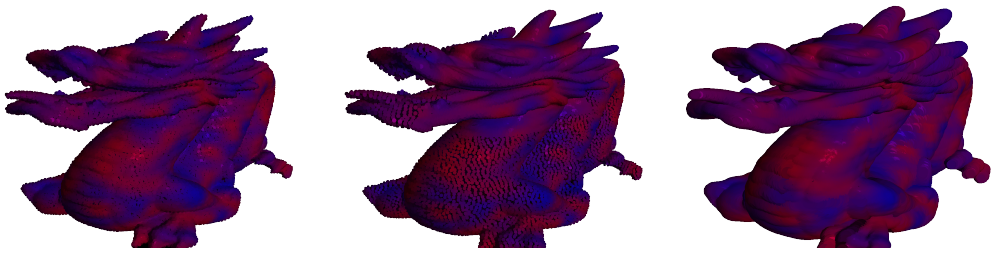
Figure 5.2: Different splat size determination methods: (a) Splat size is determined based on the level of the node in which the point is stored, lower level surfels are covered by higher level ones; (b) A global splat size is imposed, holes emerge at lower levels of detail; (c) Splat size is determined, based on the level of detail in an area.

5.2.1 Fixed Size per Octree Level

The the simplest approach would be to determine the level of detail for Equation 5.2 dependent on a nodes depth in the octree. This has the effect, that samples contained in lower nodes (higher level of detail) will be a lot smaller than samples in higher nodes and might even get occluded. This is very undesirable, as it causes the effect of a lower level of detail being used. The result of a fixed size per node approach can be seen in Figure 5.2a.

5.2.2 Globally Fixed Size

Another simple approach would be to globally fix the size of samples. Depending on the choice of size, however, their either holes will emerge at lower levels of detail, as shown in Figure 5.2b, or there will be significant overdraw in regions with a high level of detail.



(a) (b) (c)

Figure 5.3: Demo for different methods of splat size determination: (a) Fixed Size per Octree Level; (b) Globally Fixed Size; (c) Adaptive Splat Size; Model: dragon_perlin_color

5.2.3 Adaptive Splat Size

Since neither of the above approaches gives good results in practice, the adaptive size approach developed in [Sch16] has to be used. The idea behind this adaptive approach is to determine the level of detail for each sample based on its position. For this operation to be computationally viable it has to be performed on the GPU.

Before rendering, the octree is traversed in a breadth-first manner and the visible hierarchy is stored in a one-dimensional Texture using 32 bit per node entry using the following layout:

- **bit 0-7**: Mostly padding. Set to $0xFF$ if the node is a leaf in the original octree, $0x00$ otherwise. This is used for nodes, that were not expanded due to not having enough samples overall. These nodes contain samples at the density of the original model and should thus be drawn at the maximum LoD.
- **bit 8-15**: Each bit indicates, whether there exists a child node corresponding to one of the 8 sub-cubes of the nodes bounding cube
- **bit 16-31**: Number of entries between the current node and its first child. Per definition of breath-first, all children of the same node are direct successors of the first child in the generated array.

Visible nodes are determined according to section 4.4. During this traversal all visible nodes are marked. This texture is then uploaded to the GPU and in a second traversal on the CPU side a draw call is issued for each marked node and the nodes are unmarked to prepare for the next frame.

On the GPU side then, the octree structure is traversed for each sample in order to determine its level of detail. This is done by following Algorithm 5. The traversal process is also visualized in Figure 5.4.

Algorithm 5 GPU Octree Traversal per Sample

```
1:  procedure calcDepth(Position pos)
2:      depth ← 0
3:      nodeIndex ← 0
4:      center ← bounding Cube Center
5:      sideLength ← bounding Cube Side Length
6:      node ← LookUpInTexture(nodeIndex)
7:      while true do
8:          center, childIndex ← subCubeCoveringPosition(pos, center, depth)
9:          if ¬bitSet(node.children, childIndex) then return depth
10:         childOffset ← 0
11:         for i ← 1 . . . childIndex do
12:             if bitSet(node.children, i) then
13:                 childOffset ← childOffset + 1
14:         nodeIndex ← nodeIndex + node.firstChildOffset + childOffset
15:         node ← LookUpInTexture(nodeIndex)
16:         if First8BitsSet(node) then return maxDepth
        return depth
```

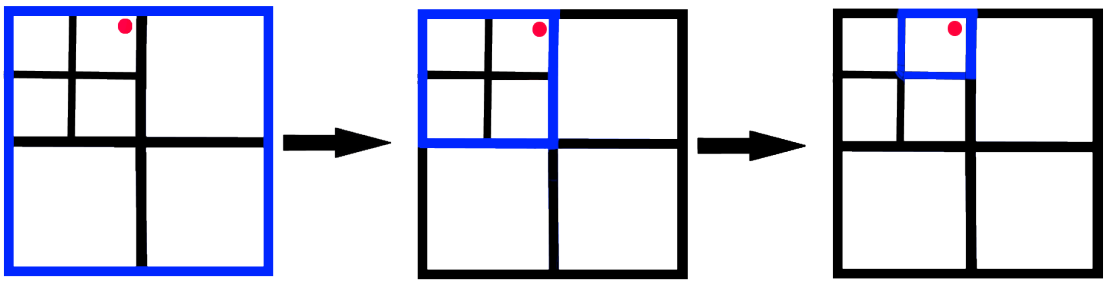


Figure 5.4: Depth determination for a single sample (red); tree is traversed until selected node (blue) is a leaf

6 Clustering

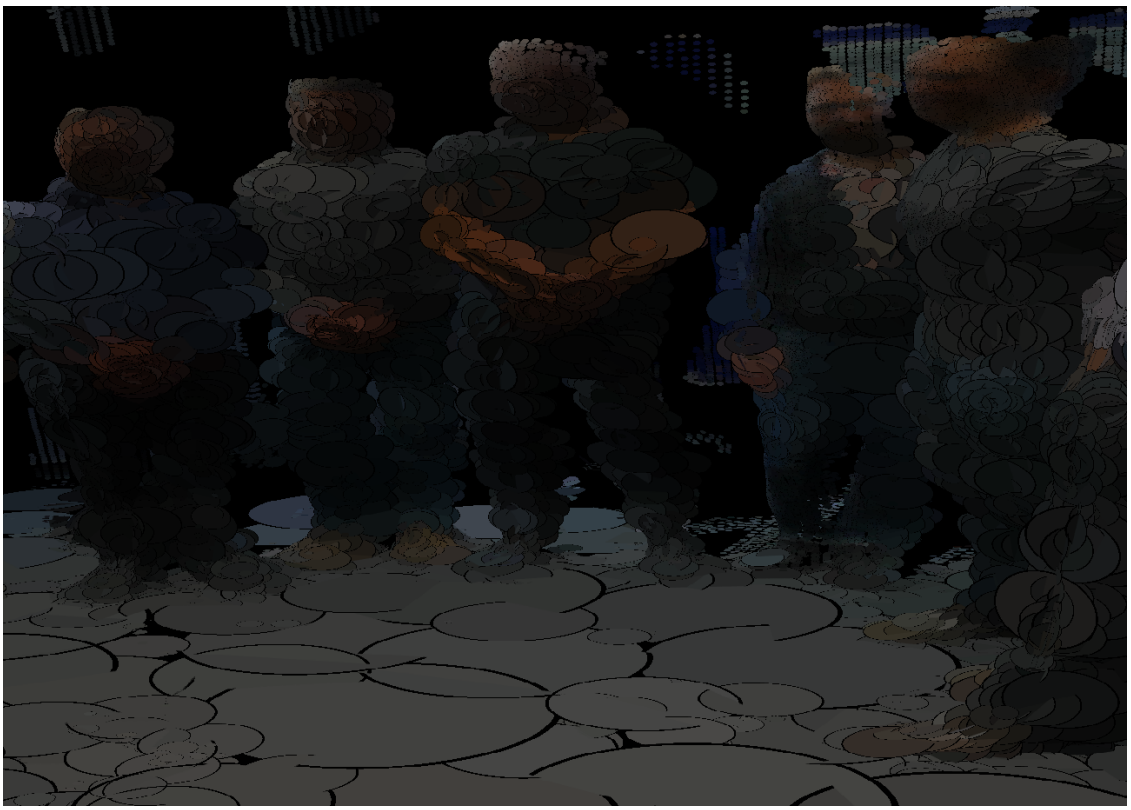


Figure 6.1: Simplified for3d_januartreffen point cloud. People are approximated by relatively small surfels, while the floor has larger uses fewer large surfels. Splats are outlined in black for better identification.

In statistics the term clustering describes the process of finding data points that, in some sense, have similar features. For the applications on point clouds this means, that we do not have to be limited to selecting points based on their distance, but we can also consider other attributes, namely color and curvature (normal) information.

Most clustering algorithms for point clouds are only intended for simplification tasks, which is the process of approximating a point cloud with a smaller set of points while staying within a maximal error. By iteratively increasing this maximum error and imposing a spacial partitioning structure, such as octrees, simplification methods can be used to create a level of detail hierarchy [FHP13; PGK02].

The main property of clustering based approaches is that there is no global sample density. Instead the number of surfels depends on the local complexity in terms of geometry as well as color. As a result we can no longer calculate the size of a surfel on-the-fly, but we have to compute and store it during the creation of our hierarchy. The result of having different sized surfels at the same level of detail can be seen in Figure 6.1.

6.1 Creation

In order to provide a better comparison between our clustering and our subsampling approach, and since a lot of code from our subsampling implementation could be reused, the creation of a level of detail hierarchy using clustering was implemented as follows:

1. Create the octree hierarchy according to Algorithm 1, but instead of storing the data inserted into the grid at the node, also push it down to the children. This way at the end of this step all samples will be stored at the leaf nodes of our octree and the inner nodes will be empty.

2. Select a node, who's children are either leaves or have been marked.

3. Sort all the surfels stored in the nodes children into a grid. As resolution we picked the same resolution that was used in step (1). In practice 128^3 has proven to be a reliable choice. This is used to accelerate the finding points close to a given seed point.

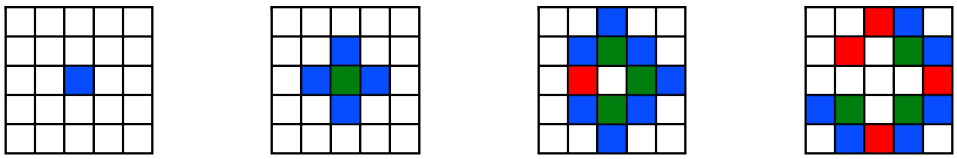
4. Perform the actual clustering approach. For our implementation this is described in the next section (subsection 6.1.1).

5. Mark the selected node and continue with step (2) until you the octree's root node is reached.

A lot of computation time is essentially wasted in the first step. For the overall performance of the clustering algorithm, this can be neglected, as the time required by the clustering itself drastically outweighs it. Still one could try sorting the entire point cloud into a giant grid and then iteratively applying the clustering implementation. The octree structure would then have to be created afterwards. This approach is, however, seems not suitable for out-of-core approaches.

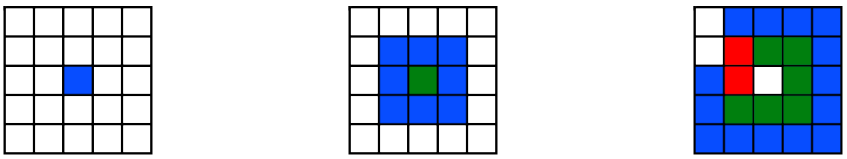
6.1.1 Region Growing

Our clustering implementation uses a grid-based, greedy region growing approach similar to the one described in Wu and L. Kobbelt [WK04]. This means, that we start at a randomly chosen point, a so called **seed point**. We then determine the cell in our 3D grid, in which this seed point lies. We search that cell, and its neighborhood, for surfels



(a) Step 0 (b) Step 1 (c) Step 2 (d) Step 3

Figure 6.2: Grid-accelerated region growing using 16-connected neighborhood. Blue: nodes currently Being explored, Green: node has been explored and at least one fitting point was found. Red: previous search did not find a fitting point



(a) Step 0 (b) Step 1 (c) Step 1

Figure 6.3: Grid-accelerated region growing using 26-connected neighborhood. Blue: nodes currently Being explored, Green: node has been explored and at least one fitting point was found. Red: previous search did not find a fitting point

that can be added to our cluster, meaning they match a certain similarity criterion. If and only if we find a point, that can be added to the cluster, we continue to explore all nodes in the 18- or 26-connected neighborhood of that point, if they have not already been explored. This is done by storing all explored nodes in a set (in C++ std::unordered_set) and each time we want to explore a new cell we check, if this cell is not already in the set.

A key assumption hereby is, that if there are two surfels P_1, P_2, where P_1 can be added to the cluster, while P_2 cannot, the following has to hold:

$$dist_{pos}(P_1, seedPoint) < dist_{pos}(P_2, seedPoint)$$

By choosing a grid resolution of 128^3 this is constraint almost always satisfied in practice.

A 2D example of such an outwards going grid-search is depicted in Figure 6.2 and Figure 6.3. As one can gather from the figures 26-connected grid-search tends to take fewer steps, while 18-connected grid-search generally explores fewer nodes in total. In practice the performance difference between both methods is negligible.

While a greedy approach does per definition not provide optimal results, the findings of [WK04] show, that it tends to come very close to an optimum. Global relaxation solutions, as presented in [WK04; SF08], take significantly more computational resources.

6.1.2 Attributes and Distance Functions

In order to decide, if a point gets added to a cluster or not, we need some form of distance or dissimilarity measure. As mentioned in the introduction to this chapter, in

a clustering setting we can consider more than just the spacial distance between two points. If we were to cluster samples only based on spacial distance, we would end up with a globally consistent density and the results of our clustering approach would end up very similar to those of Possion-Disk sampling [MF92; Sch16]. In fact our algorithm would be far worse, seeing how we create new surfels for each inner node of our LoD hierarchy instead of choosing representatives from the existing dataset.

Our algorithm considers three attributes, which will be described in the following sections. For our implementation we assume all of these attributes are present. If a loaded point cloud does not contain one of these attributes identical values are set for each sample. While this is not recommended for practical use it simplifies changing and improving our algorithm.

Position:

The key idea behind clustering is to use all points, that lie on the same plane and have the same color, or at least within a small margin of error. By not imposing a maximal radius for a single cluster, we have found, that simplifications of planar surfaces with a gradient change in color, do not tend to look very appealing to the human eye. This could be at least partially remedied by blending the resulting large splats together in order to provide a smoother transition form one surfel to the next. Blending however was not implemented as part of this thesis and is left open for future examinations [Bot+05].

Instead we chose to impose a maximum cluster size, which is equal to a multiple of the side-length of a grid cell in the inscribed search grid. Five to ten times the size of one such cell has been found to provide decent results. This can be seen in Figure 6.4, where the edges of the large splats, generated by not having a maximum radius, can be seen clearly in the left picture.

The distance between two points is calculated using the standard, euclidean distance shown in Equation 6.1.

$$dist_{euclid}(p_1, p_2) = \|p_1 - p_2\|_2 = \sqrt{(p_1 - p_2) \cdot (p_1 - p_2)} \tag{6.1}$$

Normals:

Especially when it comes to large, mostly-flat surfaces, such as walls or floors, point clouds fall short. By allowing points in a large area to be added to the same cluster, as long as their normals don't diverge too much, one can represent such flat surfaces easily by only using a few representing surfels [WK04; PGK02].

The angle between two vectors u and v is defined by:

$$cos(\theta) = \frac{u \cdot v}{\|u\| \cdot \|v\|} \tag{6.2}$$

Figure 6.4: Left: surfels generated by not having a maximum radius. Edges and intersections of large splats can be seen clearly; Right: surfels generated with a maximum radius. Transitions between splats are no longer eye-catching; Model: navvis_tum_audimax_half

Normals have per definition length one, which means Equation 6.2 can be simplified to:

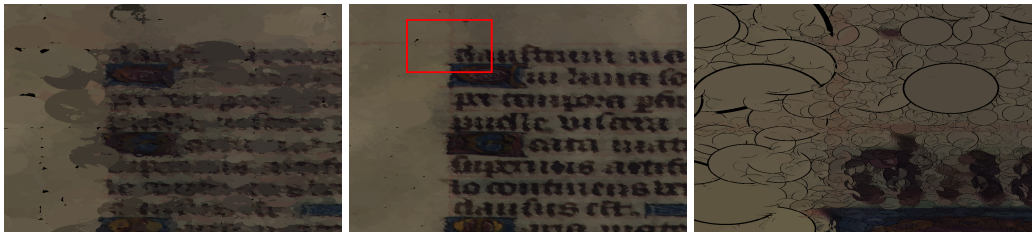
$$dist_{normal}(n_1, n_2) = angle(n_1, n_2) = |acos(n_1 \cdot n_2)| \tag{6.3}$$

The absolute of the angle is taken, since distance functions are per definition symmetric and not negative.

When calculating the angle between two normal vectors it is very important to check, that the input vectors do in fact have unit length. During testing various datasets we discovered, that some contained a small number of samples with the vector $(0, 0, 0)^T$ as normal entry. As we know from linear algebra, and as can also be seen in Equation 6.3, the zero-vector is perpendicular to all vectors, including itself, meaning the distance function would always return $\frac{\pi}{2}$ or 90°. Depending on the implementation, this could result in an endless loop, where a sample is not added to a cluster that was generated using that same sample as seed point. We therefor decided to check the normal vector of each sample during the creation of the octree hierarchy and simply discard all samples, that did not have a unit length normal vector.

Colors:

Of course for clustering objects, like paintings, which generally tend to have a flat surface, it would not be desirable to simplify a point cloud only by its geometric complexity. This would result in an essentially useless approximation. Instead, if color information is present, we have to consider it, when building our clusters. As distance measure for colors we chose the Euclidean distance (Equation 6.1). The results of also using color information can be seen in Figure 6.5.



(a) (b) (c)

Figure 6.5: Clustering with color constraint; (a) No color constraint. The image is basically unusable; (b) With color constraint: The writing is still recognizable, while the areas with no text are approximated by larger surfels; (c) Close-up image of (b). The splats are outlined in black. One can see larger splats in areas with no writing and smaller ones where writing is present; Model: xyzrgb_manuscript

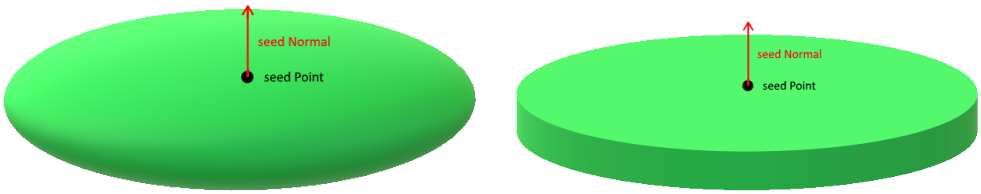


Figure 6.6: Decision boundaries for different notions of distance. Left: Ellipsoid-shaped cluster as a result of using a combined distance function; Right: Cylinder-shaped cluster as a result of using separate conditions

6.1.3 Feature-space Distance Function vs. per Attribute Threshold

While one could unify all attributes into one 9-dimensional distance function, as shown in Equation 6.4 [SLL11], we chose to separately impose a threshold for each attribute [WK04]. In theory Equation 6.4 would allow points spatially closer to the seed point to have a higher variation in color and normal dissimilarity, while points further away would have to be similar in color and normals. This would lead to more ellipsoid shaped clusters, while separate constraints will lead to cylinder shaped ones (shapes shown in Figure 6.6). When ultimately determining a representative surfel for the computed cluster both approaches deliver similar results. The differences between both approaches are listed in Table 6.1.

$$dist(\begin{pmatrix} pos_1 \\ nor_1 \\ col_1 \end{pmatrix}, \begin{pmatrix} pos_2 \\ nor_2 \\ col_2 \end{pmatrix}) = \lambda_{pos}\|pos_1 - pos_2\| + \lambda_{nor}|acos(n_1 \cdot n_2)| + \lambda_{col}\|col_1 - col_2\| \tag{6.4}$$

per Attribute Threshold	Unified Distance Function
• Cylinder-shaped clusters	• Ellipsoid-shaped clusters
+ Attributes of a maximum-size cluster are easily defined.	- One has to determine a maximum size in 9 dimensional feature space.
+ Can be used even if normal/color information is not present.	- If attributes are missing distance function or maximum distance has to be adapted.
- Thresholds only determine, whether a point should be added to the cluster.	+ Gives an actual notion of distance between two points (useful when attempting global relaxation)

Table 6.1: Differences between separately thresholding attributes and choosing a unified distance function.

6.2 Computing Representatives

While adding samples to a cluster, two normalized vectors and two lengths corresponding to those vectors are used to track the size of resulting bounding volume. This is similar to the *Extremal Points Optimal Sphere (EPOS)* algorithm introduced by Larsson [Lar08]. These two vectors will be referred to as **major** and **minor**, the major being the longer of the both. Both vectors are initialized as the zero vector $(0,0,0)^T$, the lengths are initialized with ∞. For each sample found during our grid-accelerated region growing, Algorithm 6 is executed. It adds the input sample, if the distance conditions are met. If the conditions are not met the major and minor vectors are updated, aiming to maximize the angle between them. After recomputing the cluster center (see subsection 6.2.1) the size of the final representative surfel is computed:

- For circular surfels the radius is simply set to the length of the major.
- For elliptical surfels the major semi-axis is taken as the major vector scaled by its length. The minor semi-axis is recomputed as the cross product of the major and the centers normal and then scaled by the maximum projected distance [WK04] in the direction of the computed vector. This can be seen in Equation 6.5 and Equation 6.6. This ensures that orthogonality between major and minor semi-axis, as well as the correct orientation of the ellipse.

$$minorSemiAxis = (major \times center.normal) \tag{6.5}$$

$$minorLength = max_{v \in Cluster} |minorSemiAxis \cdot (v - center.position)| \tag{6.6}$$

Algorithm 6 Try Insert Point To Cluster

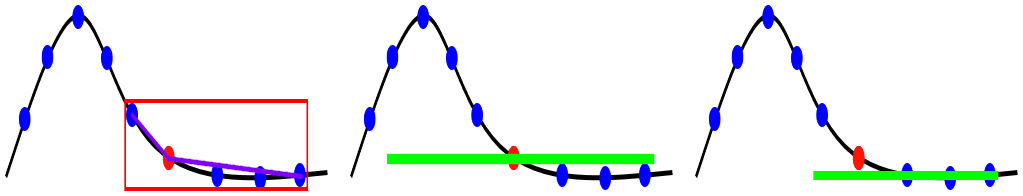
```
1:  procedure tryInsert(Point v)
2:      majorDist ← major · v.pos
3:      minorDist ← minor · v.pos
4:      if majorDist > majorLengthORminorDist > minorLength then return false
5:      if SUCCESS(checkDistanceConstraints(v,seedPoint)) then
6:          addPointToCluster(v) return true
7:      else
8:          angPointMajor ← dist_normal(major, normalize(v.normal))
9:          angPointMinor ← dist_normal(minor, normalize(v.normal))
10:
11:         if angPointMajor > dist_normal(major, minor) then
12:             majorLength ← dist_euclidean(v.pos, seedPoint.pos)
13:             major ← normalize(v.pos − seedPoint.pos)
14:         else
15:             if angPointMinor > dist_normal(major, minor) then
16:                 minorLength ← dist_euclidean(v.pos, seedPoint.pos)
17:                 minor ← normalize(v.pos − seedPoint.pos)
        return false
```

6.2.1 Cluster Centers

The presented algorithm can potentially "grow" clusters differently far in different directions. As one can see in Figure 6.7b, by choosing the seed point as center, the resulting surfel will be a lot larger than what will be visible during rendering. This not only wastes pixel shader executions, but, in extreme cases, could also cause parts of a splat to be drawn in areas, where it is not supposed to be. To prevent this we have to choose a different center for the final surfel. Two methods of choosing that new center present themself: the arithmetic mean and the center of the Axis-Aligned Bounding Box (AABB). Both can be computed very simply by Equation 6.7 and Equation 6.8. The results from [Lar08] show, that the AABB-center tend to work better when approximating the center of the minimum bounding sphere. To calculate the representative surfels normal and color we decided to use the arithmetic mean. The results, however, are indistinguishable from those obtained using the AABB-center.

Before centering the position of we also have to save our major and minor vector by computing the points they reference, which can be done using Equation 6.9. After recomputing the center, major and minor can be recovered using Equation 6.10.

$$Center_{mean}(Cluster) = \frac{1}{size(Cluster)} \sum_{v \in Cluster} v \tag{6.7}$$



(a) Selected samples (b) Seed point center (c) Re-calculated center

Figure 6.7: Choosing the seed point as center vs. re-calculating the clusters center; Blue: Surface samples, Red: seed point and cluster, Purple: major and minor, Green: resulting surfel

$$center_{AABB}(Cluster) = \frac{componentWiseMin(Cluster) + componentWiseMax(Cluster)}{2} \tag{6.8}$$

$$\begin{aligned} majorWorld &= center + majorLength * major \\ minorWorld &= center + minorLength * minor \end{aligned} \tag{6.9}$$

$$\begin{aligned} majorLength &= \|majorWorld - center\| \\ major &= \frac{majorWorld - center}{majorLength} \\ minorLength &= \|minorWorld - center\| \\ minor &= \frac{minorWorld - center}{minorLength} \end{aligned} \tag{6.10}$$

6.3 Rendering

Rendering is pretty straight forward compared to subsampling approaches. The octree is traversed according to section 4.4. Draw calls are only issued for the **visible leafs** determined by that traversal. The splat size is also stored per surfel and does not need to be computed. As the radius of a surfel in the original point cloud is not available in most datasets it can be choosen by the user. The user input radius is then added to the value stored in the surfel each frame during shader execution.

In a point cloud each sample can be seen as a small sphere and thus drawn as screen-aligned splats as shown in Listing 4.1. Since the computed clusters are ellipsoid- or cylinder-shaped the surfels representing these clusters must be considered as oriented circles or ellipses in 3-dimensional space. This means they should be drawn using oriented splats.

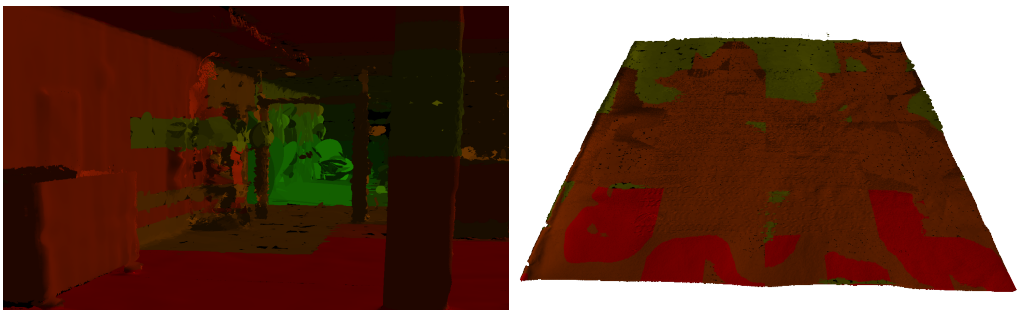


Figure 6.8: Level of Detail using average surfel size per node. red: high LoD, green: low LoD; left: for3d_januartreffen, right: xyzrgb_manuscript

6.3.1 Level of Detail

The level of detail is calculated according to section 4.4. Since the size of our surfels is no longer constant per level or even per node in our octree. In order to determine the level of detail efficiently, however, the size of a surfel for a given octree node needs to be known. Our implementation simply stores the average radius for circular surfels and the average length of the major semi-axis for elliptical ones. The result of doing so can be seen in Figure 6.8. As one can see, the general notion of lower level of detail for regions further away from the camera is preserved, however the decision boundary is not as clear, as it is for subsampling methods.

7 Implementation

This chapter will briefly mention the technologies used to implement the algorithms presented in this thesis. Our program was implemented in C++ using the *Microsoft Visual Studio 2017* IDE, as it offers a great amount of debugging tools, including a graphics debugger. For rendering the *DirectX 11* graphics API was used. Our implementation supports point clouds provided in the PLY file format. Some external libraries were used to accelerate development. The program is meant to be run on a *Microsoft Windows* operating system.

7.1 DirectX 11

DirectX 11 is a graphics API developed by Microsoft. It allows the user to access graphics hardware. The Direct X 11 rendering pipeline consists of 10 stages. Five of those are programmable stages, so called **shader**-stages. Our program implements only three of those:

- **Vertex Shader:** The Vertex Shader is meant to perform operations per point. It takes exactly one point as input and returns also exactly one. This stage must be specified whenever a draw call is issued. Our implementation uses this stage only to pass on the input to the next stage.
- **Geometry Shader:** The Geometry Shader takes as input a single point and outputs four points, defining the surface our splats will be drawn on. This is the workhorse of out rendering algorithms. It performs the following actions:
  1. (optional) Transform the input position and normal to world space, if illumination is required. The objects world matrix is stored in a constant buffer.
  2. Determine the size of a splat (for subsampling).
  3. Transform the input position to screen space.
  4. Compute and return the four points, that define the surface our splat will be drawn on. Additionally texture coordinates are set to $(\pm 1, \pm 1)$ or $(\pm 1, \mp 1)$, so that the center of the surface will have coordinate $(0, 0)$. Which point is assigned which texture coordinate is shown in Figure 4.1.

  Afterwards the **Rasterizer Stage** is executed and the computed surface is discretized into pixels.

- **Pixel Shader:** After the rasterisazion this stage is executed for each pixel. If circular splatting is used, all pixels not fulfilling the radius condition Equation 4.2 are discarded. This causes the depth check to be performed after the pixel shader is executed. For the remaining pixels shading is calculated. The Pixel Shader then returns the final color of that pixel.

7.2 Memory Management

Modern point-clouds can contain several billion samples. These, of course, can not all be stored in video memory, in fact even system memory might not suffice to hold all of them. For this reason a memory management strategy needs to be developed. This was however not in the scope of this thesis. The reader is referred to [Sch16], which implements an out-of-core algorithm for large point clouds.

7.3 PLY Format

Our implementation supports loading point clouds stored in the PLY file format, also known as Stanford Triangle Format. A PLY-file consists of two parts:

- The header defines the layout of a single sample, as well as, the total number of samples. PLY can also be used to store polygonal models. For such files indices and faces are also defined in the header.

- In the body the actual samples are stored according to the layout defined in the header. The information can either be stored in ASCII or binary.

For more information on the PLY-format the reader is directed to [Bou18].

7.4 MeshLab

MeshLab is an open-source tool for editing various kinds of 3D models, including conventional triangle meshes and point clouds. It also provides varous algorithms for computing per-sample normals. Those were used to compute normals for point clouds in which normal information was not present. MeshLab can be found on [Mes18].

7.5 External Libraries

Three external libraries were used to implement this project. They are briefly described in the next sections.

7.5.1 Eigen 3

The *Eigen 3* library provides linear algebra operations. It can be used with vectors of variable length. It is used in our clustering implementation. Eigen 3 is licensed under the *MLP2* software license. Additional information can be found on [Eig18]. While DirectX provides its own implementation for vector operations in the form of *DirectXMath*, those implementation only support up to 4-dimensional vectors. Our points are stored in a 9D vector.

7.5.2 AntTweakBar

For user input via our graphical interface the *AntTweakBar* library was used. It provides a large selection of user input techniques, such as check-boxes, drop-down menus, quaternion-inputs, etc... AntTweakBar is provided under the *zlib/libpng* license. It can be found under [Ant18]

7.5.3 tinyply

The *tinyply* library library is a light-weight library used to load and store PLY-files. It can be found on GitHub, see [PLY18].

8 Results and Comparison

This chapter provides a comparison between the Potree [Sch16] based subsampling approach and our clustering implementation, based on [WK04; Lar08; FHP13]. We will look at the performance of both algorithms during the generation of the level of detail hierarchy. Also the rendering performance will be reviewed. Finally the quality of both approaches will be showcased. Testing was done on a desktop computer with an *Intel(R) Core(TM) i7-4770K CPU @ 3.50GHz (8 CPUs), ~3.5GHz* processor, *~16 GB* system memory and a *NVIDIA GeForce GTX 780* graphics card with *~3 GB* of video memory. A constant screen-resolution of 1920×1080 was chosen. The level of detail generation was performed on a single CPU core, even though a make-shift multi-threading implementation is available for the clustering algorithm. The general benefits and caveats of both approaches are listed in Table 8.1.

Subsampling	Clustering
• Does not consider normal and color information.	• Uses normal and color information to increase sample density in complex areas.
+ Generation of LoD hierarchy is fast.	- Generating the Levels of Detail takes considerably longer.
+ No additional points need to be computed and stored; only original samples are used.	- At each LoD new representative surfels are generated.
+ Only needs grid resolution to be specified. 128^3 works for most.	- Need to determine thresholds for normals and color dissimilarity.
- Visible tree needs to be traversed twice on CPU.	- Too large thresholds will lead to oversimplification at lower LoD.
- Entire visible tree needs to be drawn.	+ Only visible leaves need to be drawn.
- Tree traversal and texture lookup per sample on GPU.	+ Fewer surfels used per frame.

Table 8.1: Differences between subsampling and clustering approaches.

Model	Samples Original	Samples Cluster	Time Subsampling	Time Cluster
bunny	35,947	38,559	0.015s	0.11s
dragon_perlin_color	437,645	730,764	0.22	3.02s
xyzrgb_manuscript	2,155,617	3,177,537	1.34s	16.56s
xyzrgb_dragon	3,609,600	5,506,217	2.72s	45.81s
for3d_januartreffen	18,616,336	19,565,000	16.38s	58.13s
navvis_tum_audimax_half	26,668,620	30,390,700	23.16s	125.16s
navvis_tum_audimax	58,859,843	67,271,989	67.58s	276.12s

Table 8.2: Statistics for creating the Level of Detail hierarchy using our subsampling and clustering implementations

8.1 Performance

This section shows the performance during generation of the level of detail hierarchy for point clouds of varying sizes. The results are ordered by the number of samples in the original point cloud. Table 8.2 and Figure 8.1 show the time required to create a LoD hierarchy, as well as the total number of points in the entire structure. Note, that in the subsampling approach, the number of surfels is equal to the number of samples in the original point cloud. The clustering implementation performs better, when flat surfaces are present in the point cloud. To see this compare the results of creating the LoD for the xyzrgb_dragon and for3d_januartreffen datasets. It is however also clear to see, that the subsampling approach takes significantly less time to create a hierarchy.

8.2 Circular vs. Elliptical Surfels

This section compares circular and elliptical surfels, which can both be generated by our clustering implementation. The general differences using between circular and elliptical surfels are listed in Table 8.3. An example is given by the pillar in Figure 8.2: circular splats generate a lot of overdraw, while elliptical ones fit the surface better.

8.3 Showcase

In this section the visual results of subsampling and clustering will be presented and compared. Each figure also mentions the approximate frames per second the image was rendered at. This serves as comparison for rendering performance. The images were drawn using oriented circular splats for subsampling and oriented ellipses for clustering.

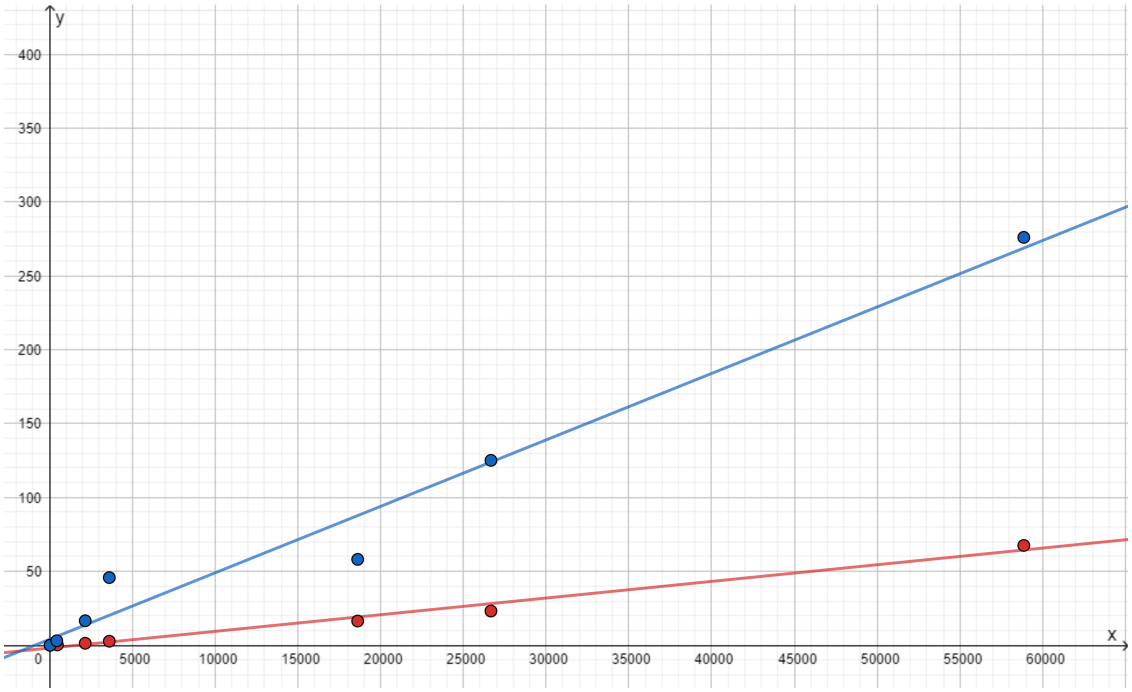
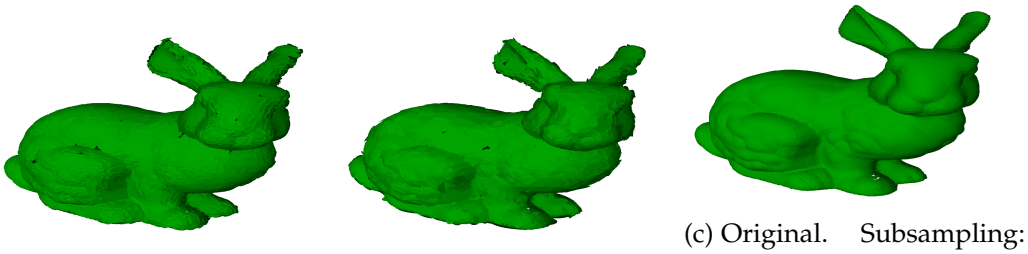


Figure 8.1: Create times for subsampling (red) and clustering(blue) plus regression lines. x-axis: number of 1000 samples, y-axis: time in seconds.

Circular Surfel	Elliptical Surfel
+ Can be defined by a single radius (1 value), which could even be stored in the length of the surfels normal vector.	- Requires at least one more vector (3 values), when storing only major and minor semi-axis and computing the normal vector as a cross product as those 2 vectors during rendering.
- Long, thin shapes require either a lot more surfels or cause substantial overdraw.	- More operations required per surfel during rendering.
	+ Tend to represent the cluster better.

Table 8.3: Differences between subsampling and clustering approaches.

Figure 8.2: Difference between circular (Left) and elliptical (right) surfels; Model: navvis_tum_audimax_half

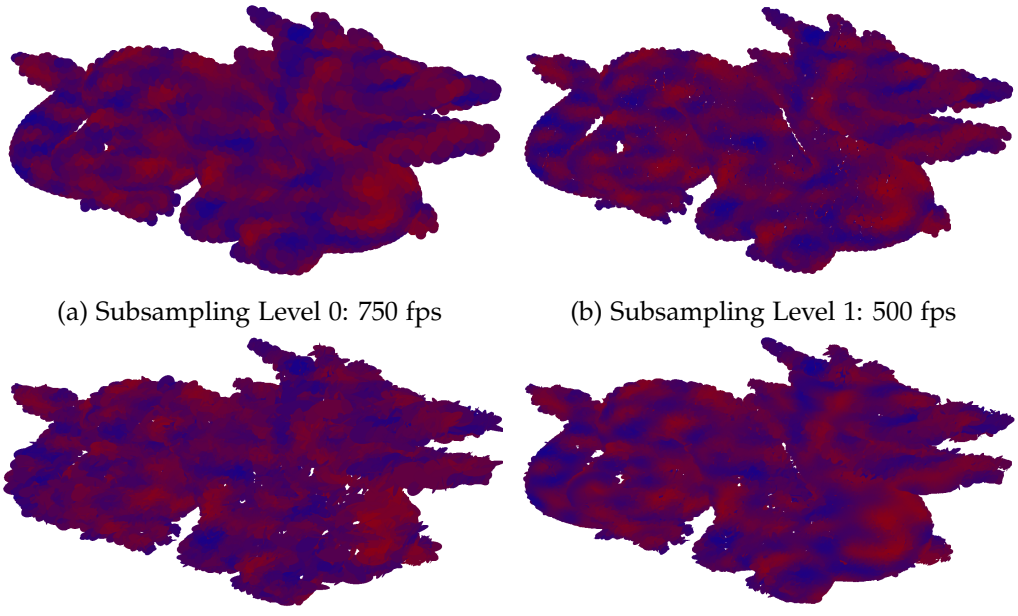


(a) Subsampling Level 0: 1100 fps

(b) Clustering Level 0: 1000 fps

(c) Original. Subsampling: 600 fps, Clustering: 670 fps

Figure 8.3: Different Levels of Detail; Model: Stanford Bunny



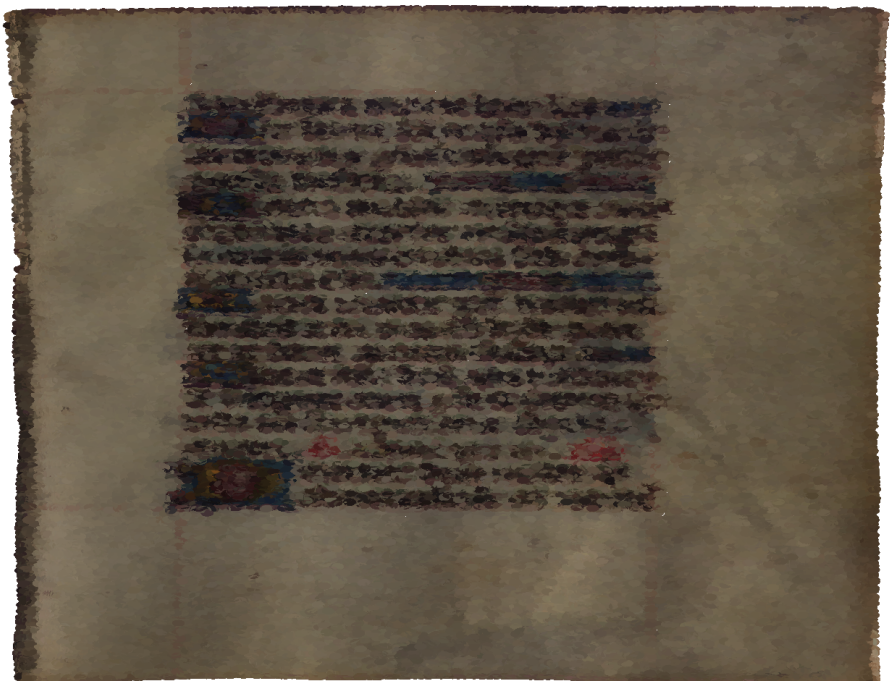
(a) Subsampling Level 0: 750 fps

(b) Subsampling Level 1: 500 fps

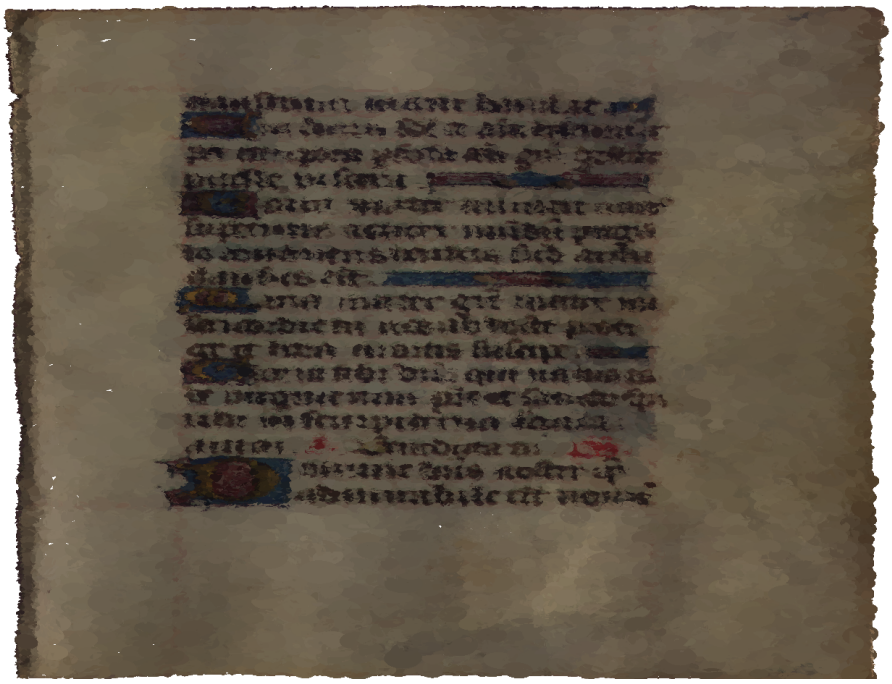
(c) Clustering Level 0: 800 fps

(d) Clustering Level 1: 600 fps

Figure 8.4: Different Levels of Detail; Model: Dragon Perlin Color

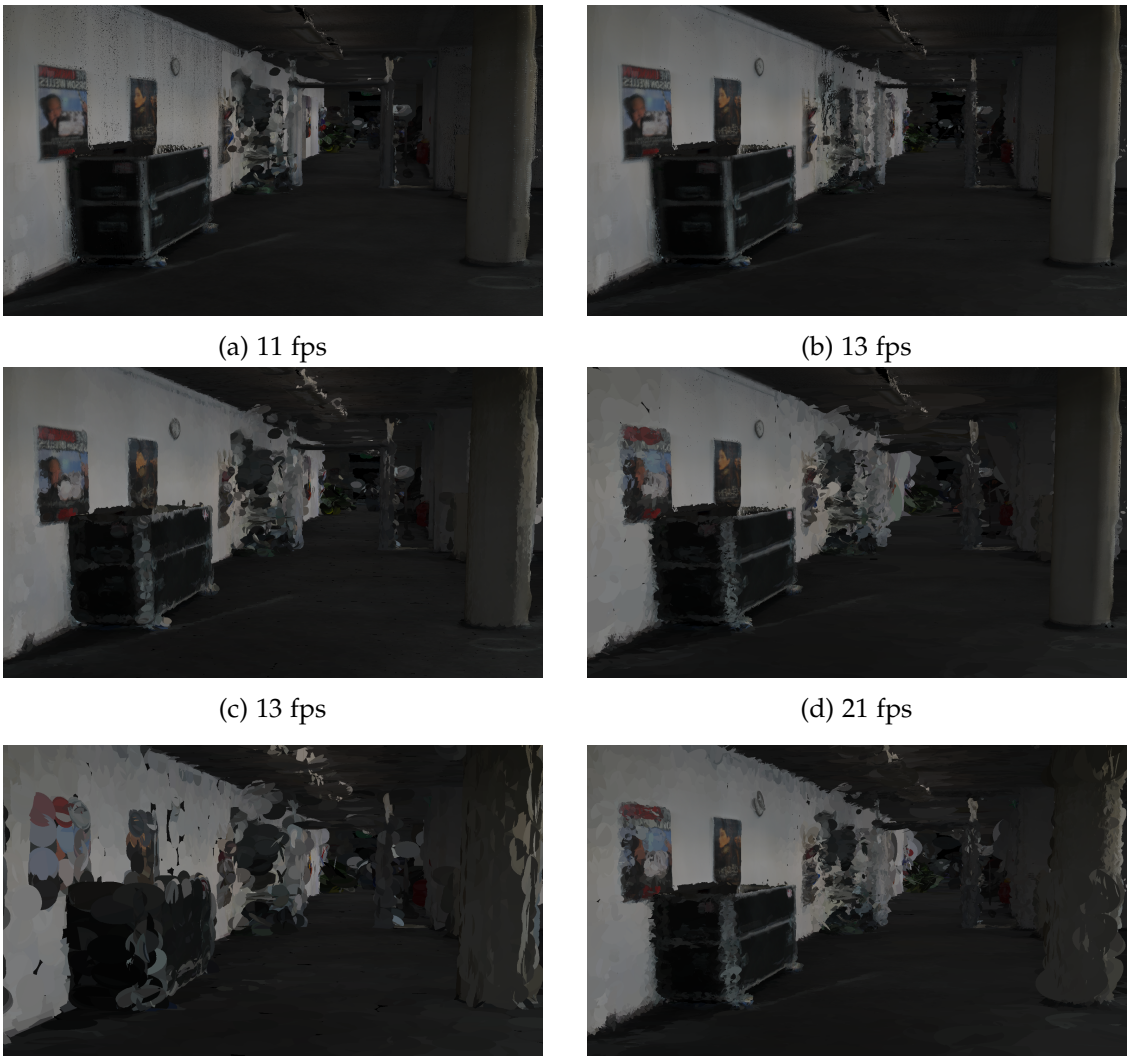


(a) Subsampling Level 1: 760 fps



(b) Clustering Level 1: 690 fps

Figure 8.5: Clustering and Subsampling applied to point cloud with complex and simple regions; Model: xyzrgb manuscript

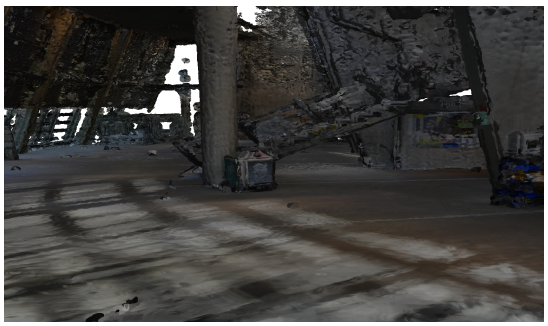


(a) 11 fps (b) 13 fps

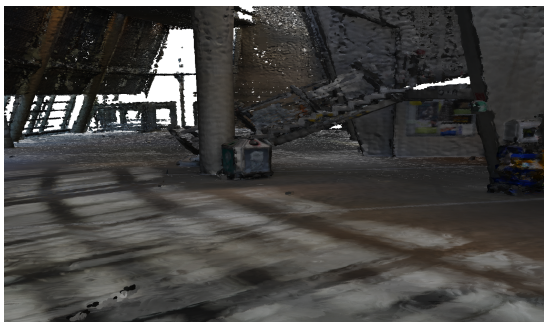
(c) 13 fps (d) 21 fps

(e) 31 fps (f) 54 fps

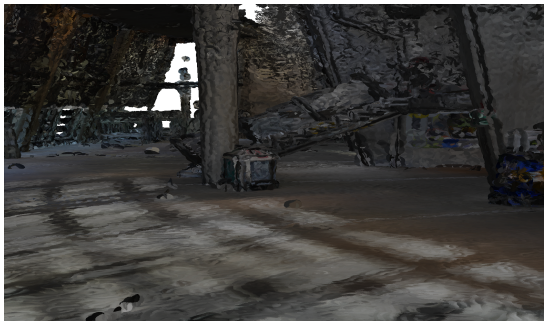
Figure 8.6: Scene rendered according to section 4.4 with pixel thresholds: 10/20/80px. Left: Subsampling, Right: Clustering; Model: for3d januartreffen



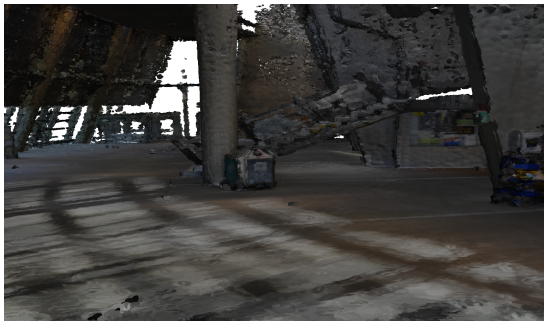
(a) 11 fps



(b) 13 fps

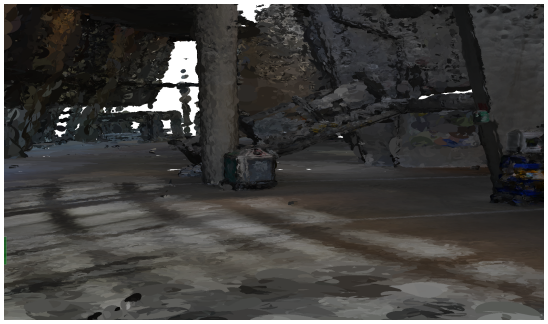


(c) 13 fps



(d) 21 fps

(e) 31 fps



(f) 54 fps

Figure 8.7: Scene rendered according to section 4.4 with pixel thresholds: 10/20/80px. Left: Subsampling, Right: Clustering; Model: audimax (half)

9 Conclusion and Future Work

We re-implemented the subsampling based Level of Detail algorithm presented in [Sch16], which only relied on reducing the point density at each level. We then used clustering based simplification algorithms, based on the works of [FHP13; WK04; Lar08], to generate a level of detail hierarchy, that also considers geometric and color complexity.

In terms of the time it takes to generate a level of detail hierarchy, our clustering approach falls short compared to subsampling techniques. An inherent characteristic of the clustering approach is, that we need to generate new surfels at each level of our hierarchy, while subsampling approaches only store the original point cloud (plus the generated control structure). In addition with the fact, that a clustering implementation requires us to store size information per surfel, this leads to larger overall memory usage, which is of course a huge downside. In addition normal information is often not present in point clouds and needs to be precomputed, for the clustering approach to be viable. A subsampling approach might get away with only using position (and color, if available) information, perform splatting in screen space and implement shading via Eye Dome Lighting [Bou09]. On the contrast, surfels in a clustering approach need normal information, since the clusters they represent are oriented disks in 3D space, and they need to be drawn using oriented splats.

When looking at rendering performance and quality, however, we determined, that our clustering algorithm substantially outperforms the subsampling approach. By utilizing normal and color information to compute splats, we were able to reduce the points the generated levels of detail. Since the size of each surfel is stored per element, we do not need to perform elaborate computations and texture lookups during rendering. Clustering is also less dependent on CPU time, as the LoD hierarchy only needs to be traversed once.

Especially human-made objects, like walls, floors, ceilings, tables, etc., tend to include lots of planar surfaces. By utilizing normal information a clustering implementation can simplify such regions using a comparably low number of surfels.

Future work includes:

- Our clustering implementation still allows for holes to exist. A hole filling step, as shown in [WK04] could be added to the processing pipeline.

- Implementing a blending approach, as mentioned in section 4.6. Right now we do not use blending for splats. A blending approach will most definitely improve the image quality. It might even allow for a larger maximum cluster size, which would

allow us to use fewer, but larger, surfels to represent a scene, without sacrificing image quality.

- Multi-threading or GPGPU support for the presented algorithms. Right now we only provide a make-shift multi-threading per node for the clustering approach. The time it takes to generate a LoD hierarchy for large point clouds, however, demands a faster implementation.
- At this time the processing pipeline includes using MeshLab to generate normals, if they are not present in a given dataset. For practical use, this feature would be a must have.
- Our implementation does not support explicit memory management. All nodes are stored in video memory and the DirectX 11 driver takes care of residency management. For our implementation to be able to handle larger point clouds, for which even system memory might not suffice, an out-of-core implementation is needed.
- Persistent storing of the LoD hierarchy. Right now, our hierarchy is created when a point cloud file is opened. Since clustering takes a substantial amount of time it might be helpful, to create the entire hierarchy offline and load it, when a model is opened.
- Currently only the PLY format is supported. In the future support for additional formats, such as CSV, VTK, PCD, should also be provided, to allow for more general use.
- If no color information is present our implementation stores the same color value for each sample. This results in larger memory usage and wasted computations during clustering. If the program is intended to frequently handle color-less point clouds, a separate function, that ignores color and sets a constant valu during rendering should be implemented.

List of Figures

List of Tables

List of Algorithms

Bibliography

[Ale+02] M. Alexa, T. Darmstadt, M. Gross, M. Pauly, E. Zrich, H. Pfister, M. Cambridge, M. Stamminger, B.-u. Weimar, and M. Zwicker. "Point-Based Computer Graphics." In: 28 (July 2002).

[AM99] U. Assarsson and T. Moller. "Optimized View Frustum Culling Algorithms for AABBs and OBBs." In: (Nov. 1999).

[Ant18] AntTweakBar. *AntTweakBar GUI*. [Online; accessed 30-July-2018]. 2018.

[Bot+05] M. Botsch, A. Hornung, M. Zwicker, and L. Kobbelt. "High-quality surface splatting on today's GPUs." In: *Proceedings Eurographics/IEEE VGTC Symposium Point-Based Graphics, 2005.* June 2005, pp. 17–141. DOI: 10.1109/PBG.2005.194059.

[Bou09] C. Boucheny. "Visualisation scientifique de grands volumes de données : Pour une approche perceptive." In: (Feb. 2009).

[Bou18] P. Bourke. *PLY - Polygon File Format*. [Online; accessed 30-July-2018]. 2018.

[Coo86] R. L. Cook. "Stochastic Sampling in Computer Graphics." In: *ACM Trans. Graph.* 5.1 (Jan. 1986), pp. 51–72. ISSN: 0730-0301. DOI: 10.1145/7529.8927.

[Eig18] Eigen. *Eigen 3 - LinAlg Library*. [Online; accessed 30-July-2018]. 2018.

[FHP13] Y. Fan, Y. Huang, and J. Peng. "Point cloud compression based on hierarchical point clustering." In: *2013 Asia-Pacific Signal and Information Processing Association Annual Summit and Conference*. Oct. 2013, pp. 1–7. DOI: 10.1109/APSIPA.2013.6694334.

[Lab18] S. C. G. Laboratory. *The Stanford 3D Scanning Repository*. [Online; accessed 30-July-2018]. 2018.

[Lar08] T. Larsson. "Fast and Tight Fitting Bounding Spheres." In: *SIGRAD 2008. The Annual SIGRAD Conference Special Theme: Interaction; November 27-28; 2008 Stockholm; Sweden*. 34. Linköping University Electronic Press; Linköpings universitet, 2008, pp. 27–30.

[LW85] M. Levoy and T. Whitted. "The Use of Points as a Display Primitive." In: 1985.

[MD03] C. Moenning and N. Dodgson. "A New Point Cloud Simplification Algorithm." In: (Nov. 2003).

[Mea80] D. Meagher. *Octree Encoding: A New Technique for the Representation, Manipulation and Display of Arbitrary 3-D Objects by Computer*. Oct. 1980.

[Mes18] MeshLab. *MeshLab*. [Online; accessed 30-July-2018]. 2018.

[MF92] M. McCool and E. Fiume. "Hierarchical Poisson Disk Sampling Distributions." In: *Proceedings of the Conference on Graphics Interface '92*. Vancouver, British Columbia, Canada: Morgan Kaufmann Publishers Inc., 1992, pp. 94–105. ISBN: 0-9695338-1-0.

[Pfi+00] H. Pfister, M. Zwicker, J. Baar, and M. Gross. "Surfels: Surface Elements as Rendering Primitives." In: (May 2000).

[PGK02] M. Pauly, M. Gross, and L. P. Kobbelt. "Efficient Simplification of Point-sampled Surfaces." In: *Proceedings of the Conference on Visualization '02*. VIS '02. Boston, Massachusetts: IEEE Computer Society, 2002, pp. 163–170. ISBN: 0-7803-7498-3.

[Pho75] B. T. Phong. "Illumination for Computer Generated Pictures." In: *Commun. ACM* 18.6 (June 1975), pp. 311–317. ISSN: 0001-0782. DOI: 10.1145/360825.360839.

[PLY18] T. PLY. *Tiny PLY - PLY file loader*. [Online; accessed 30-July-2018]. 2018.

[Rei+] F. Reichl, M. G. Chajdas, K. Bürger, and R. Westermann. "Hybrid Sample-based Surface Rendering." In: pp. 47–54. DOI: 10.2312/PE/VMV/VMV12/047-054.

[RL01] S. Rusinkiewicz and M. Levoy. "QSplat: A Multiresolution Point Rendering System for Large Meshes." In: 2000 (Oct. 2001).

[Sch14] C. Scheiblauer. "Interactions with Gigantic Point Clouds." PhD thesis. Favoritenstrasse 9-11/186, A-1040 Vienna, Austria: Institute of Computer Graphics and Algorithms, Vienna University of Technology, 2014.

[Sch16] M. Schütz. "Potree: Rendering Large Point Clouds in Web Browsers." In: (Sept. 2016).

[SF08] H. Song and H.-Y. Feng. "A global clustering approach to point cloud simplification with a specified data reduction ratio." In: *Computer-Aided Design* 40.3 (2008), pp. 281–292. ISSN: 0010-4485. DOI: https://doi.org/10.1016/j.cad.2007.10.013.

[SLL11] B.-Q. Shi, J. Liang, and Q. Liu. "Adaptive simplification of point cloud using k-means clustering." In: *Computer-Aided Design* 43.8 (2011), pp. 910–922. ISSN: 0010-4485. DOI: https://doi.org/10.1016/j.cad.2011.04.001.

[Wik18] Wikipedia. *Octree*. [Online; accessed 24-July-2018]. 2018.

[WK04] J. Wu and L. Kobbelt. "Optimized Sub-Sampling of Point Sets for Surface Splatting." In: 23 (Sept. 2004), pp. 643–652.

[WS06] M. Wimmer and C. Scheiblauer. "Instant Points." In: *Proceedings Symposium on Point-Based Graphics 2006*. Eurographics. Boston, USA: Eurographics Association, July 2006, pp. 129–136. ISBN: 3-90567-332-0.

[Zwi+01] M. Zwicker, H. Pfister, J. van Baar, and M. Gross. "Surface Splatting." In: *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '01. New York, NY, USA: ACM, 2001, pp. 371–378. ISBN: 1-58113-374-X. DOI: 10.1145/383259.383300.