Dynamic loading of 3D models

PROJ-H-402 Project Report

Tim Lenertz, INFO MA1, ULB

April 23, 2014

Résumé

résumé en français

Abstract

english abstract

Contents

1 Introduction

Point clouds are a way of digitally representing three-dimensional models using only a set of points located on the object's surfaces[1]. Each point consists of 3 coordinates (x, y, z) on an Euclidian coordinate system defined for the model, and can be attributed with additional information such as color, surface normal vector, and others.

This data is typically produced by 3D scanners, which today can capture surfaces at a very high level of detail and thus yield huge quantities of points. Full representations of large objects or environments can be generated by combining the outputs of multiple scans from different view points. For example to get a full point cloud model of an archeological site, aerial scans may be combined with close-up scans of individual landmarks. The resulting files can easily reach several GB in size and contain over 10^8 points, and so they can no longer be processed or visualized efficiently as a whole. Instead, subsets of points around an area of interest are extracted.

The goal of this project is to develop a system which dynamically loads the visible subsets of points to render, while a user moves through a large scale point cloud. This involves methods to extract the right subsets from the point cloud, the data structure which the full point cloud is stored in, and the file format using which this data structure is serialized on secondary storage. The process should appear seamless to the user.

To this end, different data structures and file formats were compared for their possibilities to extract the right subsets of points in an efficient way, and a program was developped which converts a given point cloud file into such a data structure (preprocessing stage), and then allows the user to explore the point cloud, by dynamically loading chunks of it from the preprocessed data structure file.

[1] Only non-volumetric point clouds are considered for this project. In volumetric point clouds, points are not only located on surfaces, but on the insides of objects as well.

2 Filtering the point cloud

This chapter describes the methods used to compute a smaller set of points based on the full point cloud, which visually represent the model as seen from a given view point. The data structure used to store the point cloud is not considered in this chapter.

2.1 Definitions

The following definitions are used throughout this report. A *point cloud* is an unordered set of points with an Euclidian coordinate system. Each *point* $p = \langle x, y, z, r, g, b \rangle$ consists of its coordinates x, y, z, and RGB color information. The *model* P is the full point cloud used as input. The *point capacity* C is the maximal number of points that can be outputted to the renderer.

The *view-projection matrix* $\boldsymbol{M} = \boldsymbol{P} \times \boldsymbol{V}$ is a 4x4 matrix that defines the view frustum of the camera. The 6 planes of the frustum can be derived from the matrix as described in [1]. The *view matrix* $\boldsymbol{V}$ transforms the points' coordinate system into one centered around the camera at its current position and orientation, while the *projection matrix* $\boldsymbol{P}$ is used to project the points to their two-dimensional screen coordinates. $\boldsymbol{P}$ might define both a parallel projection or a perspective projection with a given *field of view* λ.

The *filtering function* $f_P(\boldsymbol{M})$ computes a set of rendered points P' from and model P, the matrix $\boldsymbol{M}$, and additional parameters. Its main constraint is that $|P'| \leq C$ (whereas $|P|$ may be much larger than C). P' does not need to be a subset of P: Some methods (such as uniform downsampling) will add points into P' that are not in P, in order to achieve a better visual quality.

The criteria for quality of the filtering function is that the 2D projection of P' at the current view point $\boldsymbol{M}$ looks similar to that of P, that is there should be no loss of important details and no obvious visual artifacts or discontinuities. Techniques such as hidden surface removal can actually improve the appearance of P' compared to that of P.

The function f_P described in this chapter is an idealized version that operates on a set of points. The next chapters describe algorithms that calculate an approximation of $f_P(\boldsymbol{M})$ using a specific data structure for P, and with additional time complexity constraints.

2.2 Projection

When the point cloud is rendered, the points p are projected from their three-dimensional virtual space onto the two-dimensional screen, using the view frustum defined by $\boldsymbol{M}$. This can be described as a function $\text{proj}_{\boldsymbol{M}}(x, y, z) = (x_{\text{screen}}, y_{\text{screen}})$, where $x_{\text{screen}} \in [0, w[$ and $y_{\text{screen}} \in [0, h[$, with w and h being the width and height of the screen in pixels. This operation is done on the GPU to render vertices.

First a vector in homogeneous coordinates is build from x, y, z: $\overrightarrow{p} = [x, y, z, 1]^T$. The fourth component $w = 1$ indicates that this vector represents a point in space; with $w = 0$ it would indicate a direction. In general, a point in homogeneous coordinates $[x, y, z, w]$ corresponds to $[\frac{x}{w}, \frac{y}{w}, \frac{z}{w}]$ in Euclidian coordinates. This allows for building transformation matrices that distinguish between points and vectors (notably for translations), and the projection matrix $\boldsymbol{P}$.

Next $\overrightarrow{p}$ is multiplied by $\boldsymbol{M}$: $\overrightarrow{p'} = \boldsymbol{M} \times \overrightarrow{p} = \boldsymbol{P} \times \boldsymbol{V} \times \overrightarrow{p}$. The view matrix $\boldsymbol{V}$ represents the position and orientation of the camera in the virtual space, so the first multiplication puts $\overrightarrow{p}$ into a coordinate system centered around the camera. The w component remains 1. It is then multiplied by $\boldsymbol{P}$, which can change w. Finally the resulting $\overrightarrow{p'}$ is transformed back into Euclidian coordinates to yield the camera coordinates $x_{\text{cam}}, y_{\text{cam}}, z_{\text{cam}}$. In the case of perspective projection, foreshortening is done with the component-wise division by w. Because this is a non-affine transformation, it could not be done using matrix arithmetic only.

Camera coordinates are considered to be inside the view frustum only if $x_{\text{cam}}, y_{\text{cam}}, z_{\text{cam}} \in [-1, 1]$, and then the two-dimensional screen coordinates $x_{\text{screen}}, y_{\text{screen}}$ are deduced by linearly mapping x_{cam} and y_{cam} to $[0, w[$ and $[0, h[$, respectively. z_{cam} no longer affects the position of the pixel, but comparing two values for z_{cam} indicates whether one point is in front of or behind another one in camera space, and is for example used in OpenGL's depth testing.

If $\boldsymbol{P}$ is the identity matrix, it represents an orthographic projection where the view frustum is the axis-aligned cube from $[-1, -1, -1]$ to $[1, 1, 1]$. An orthogonal projection with a different cuboid frustum can be expressed by letting $\boldsymbol{P}$ be a transformation matrix that maps coordinates in that cuboid to the former cube. The perspective projection matrix for field of view λ, screen aspect ratio w/h, and near and far clipping planes z_{near} and z_{far} is defined by:

$$\boldsymbol{P} = \begin{bmatrix} \frac{f}{w/h} & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & \frac{z_{\text{far}}+z_{\text{near}}}{z_{\text{near}}-z_{\text{far}}} & \frac{2\times z_{\text{far}}\times z_{\text{near}}}{z_{\text{near}}-z_{\text{far}}} \\ 0 & 0 & -1 & 0 \end{bmatrix} \text{ with } f = \frac{1}{\tan(\frac{\lambda}{2})}$$

2.3 Frustum culling

The simplest and most effective filtering done by f_P is *view frustum culling*, which removes all points from P that are not within the view frustum defined by $\boldsymbol{M}$. This usually eliminates more than half of the points from the model: Those behind the viewer, those outside his field of view, and those too far away (beyond the far clipping plane z_{far}). It is done implicitly by the GPU, but the goal of the filtering is to reduce the number of points before they are sent to the GPU.

Frustum culling can be done per point by clipping the camera coordinates as described above, but depending on the data structure used, entire regions of the model space will be tested to be inside or outside the frustum instead.

2.4 Downsampling

Downsampling reduces the density of points. Because of foreshortening in perspective projection, sections of the model that are farther away from the camera will become smaller in the two-dimensional projection, and as a consequence their amount and density of points increases. Since a smaller density is sufficient to visually represent the model, it makes sense to apply downsampling on regions of the point cloud, depending on their distance from the camera.

Because the point cloud is non-volumetric, points are distributed on two-dimensional surfaces of the three-dimensional model. So the density can be defined as the number of points per surface area $\rho = n/A$. Because of the way the point clouds are generated, ρ will remain approximately constant on small scale, but on composite point clouds that combine different scans, ρ can vary greatly in different regions of the point cloud. Some objects could have been scanned close-up, while the surrounding environment has a much lower resolution.

The level of downsampling is determined by a function $r(d)$ which gives a ratio in function of a distance to the camera. A downsampling algorithm will transform the point cloud such that at any surface position $\overrightarrow{p}$, the density becomes $\rho' = r(d(\overrightarrow{p})) \times \rho$ (with $r \in [0, 1]$). The algorithms used here operate without knowledge of the shape of the object's surface or of the concrete value of ρ.

2.4.1 Weighted points

One method of doing downsampling is to assign a weight $w \in [0, 1]$ to each point $p \in P$, and let the downsampled set of points be the subset $P' = \{p \in P : w(p) < r(d(p))\}$. For this to work the weights w need to be uniformly distributed among the points.

This leads to a continuous ρ', so no visual discontinuities are produced. Also, if the data structure contains the weighted points P in a list ordered by ascending w, then it is possible to dynamically extract P' in time $O(|P'|)$.

A simple way to define the weights is to set each weight to a random value. However, this leads to an irregular pattern in the way downsampled points are placed on the surface, as seen in figure 2.1. The left-hand side of the point cloud is not downsampled and retains a more regular pattern.

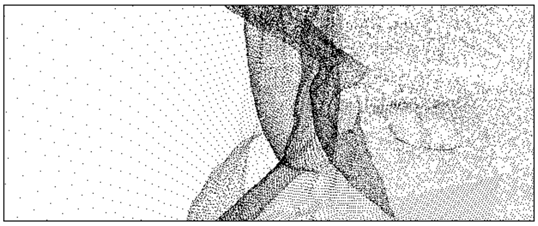

Figure 2.1: Weighted points downsampling using random weights

2.4.2 LOD regions

Another possibility is to first compute several downsampled point sets P'_r with fixed downsampling ratios r, during preprocessing. $P'(\overrightarrow{p})$ is then produced out of pieces of these sets, by choosing the set whose value r is closest to $r(d(\overrightarrow{p})$. The resulting P' is thus composed of several *level of detail* (LOD) regions[1], so $\rho'(d)$ becomes a non-continuous staircase function, and visual discontinuities are produced. However, this method allows for producing the sets P'_r with no time constraints and with a constant r.

In figure 2.2, the region farther away was downsampled this way (using uniform downsampling). A visual discontinuity can be seen in the lower-right corner.

2.4.3 Uniform downsampling

Uniform downsampling is a downsampling algorithm that aims to produce a more regular pattern in the distribution of points on P', as opposed to that produced by random downsampling (figure 2.1). It operates using a constant ratio r.

The input point set P is divided into a regular grid of cubes $C_{i,j,k}$ with side length s. For all cubes that contain at least one point, exactly one single point p' is outputted into P', whose position is the mean of that of the points in C, and whose color is that of the point $p \in P$ closest to that position. Figure 2.3 illustrates the procedure in two dimensions. The more regular aspect of P' is because its points are constrained by the regular grid. The algorithm also does not require P to have a regular pattern to begin with.

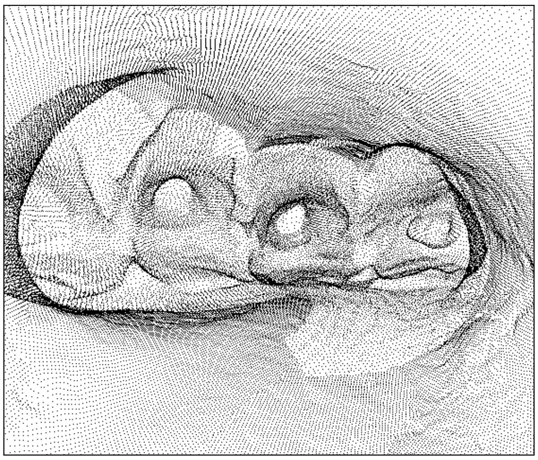

Figure 2.2: Example of uniform downsampling

The main difficulty is to find the value for the side length s, such that $n = |P'|$ is approximately equal to $r \times |P|$. Figure 2.4 shows how n varies in function of s for some example models: In general, n gets larger as the side length s gets smaller, because there are more cubes available in the region of the model. Because the cubes are non-overlapping and each point $p \in P$ belongs to exactly one cube, n is at most $|P|$. It reaches this value once s gets reaches a value (depending on the density) where each point belongs to a different cube.

However, the function is not entirely monotonic, as the close-up view in figure 2.5 shows. This is because when s gets changes, all cubes on the grid (except those at the origin) get displaced. So it is possible that two neighboring points fit into one cube for one value of s, but for $s' > s$ the grid becomes such that a boundary lies between these points. This effect only occurs at the small scale because as s gets even larger, the boundary moves further so the two points again fit into one cube, and many more points in the model fit into the same cubes.

Because the points are not distributed evenly in space but rather on the object's surfaces, but the algorithm operates in three-dimensional space, there is way to effectively estimate a value for $n(s)$. Instead the algorithm does a dichotomic search to find a value s such that $n \approx r \times |P|$, within a certain tolerance. The small-scale ascending intervals of $s(n)$ are taken into account.

[1] These are called *mipmaps* in the implementation

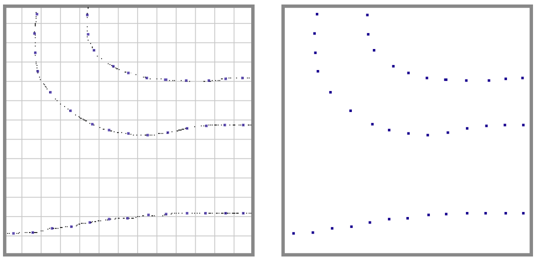

Figure 2.3: Two-dimensional visualization of uniform downsampling algorithm

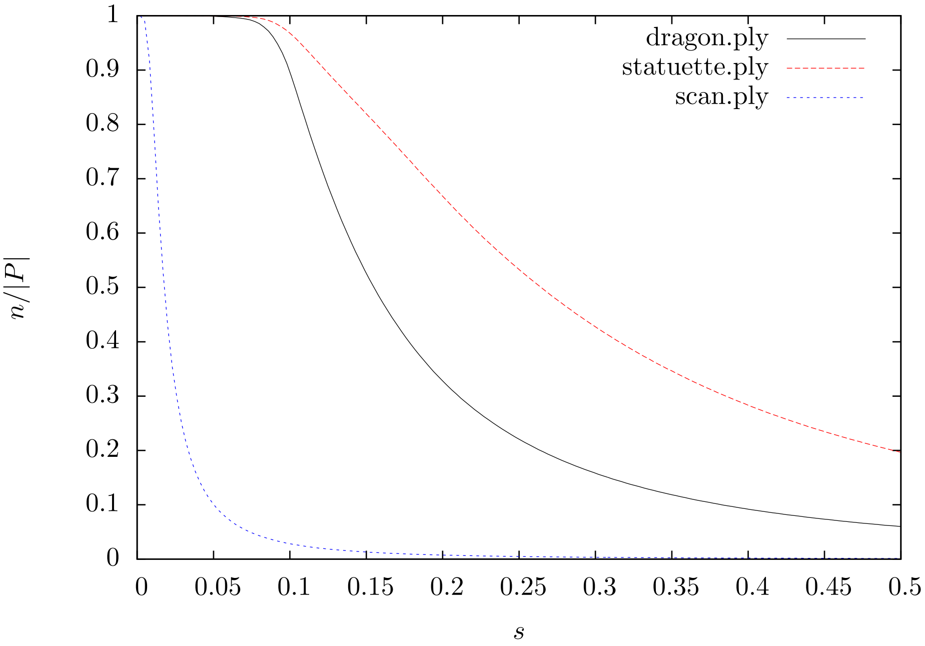

Figure 2.4: Number of output points n VS cubes side length s in uniform downsampling; normalized by number of points in model $|P|$

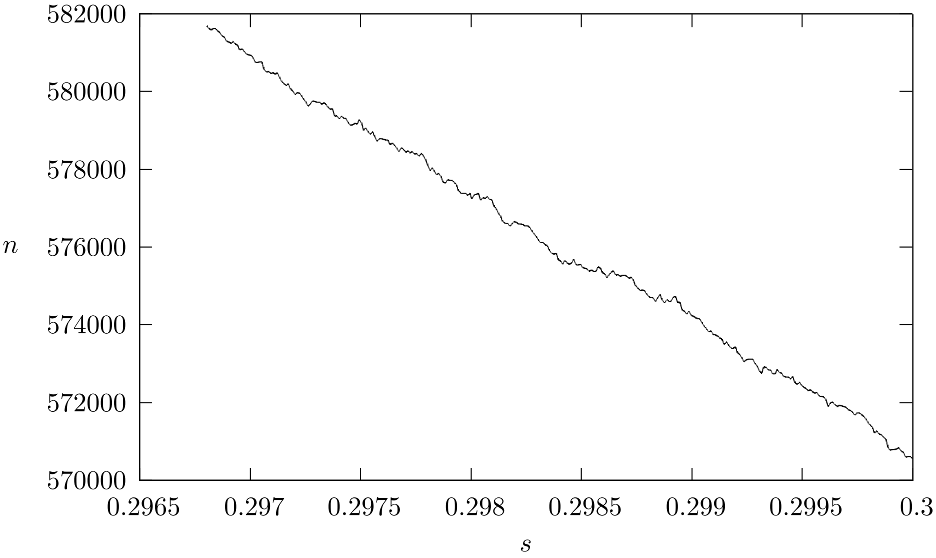

Figure 2.5: Close-up view of figure 2.4

2.4.4 Choosing the downsampling ratio

The function $r(d)$ determines that ratio of downsampling that should be applied depending on the distance d to the camera. If *LOD regions* are used, it is a staircase function and can only take a small number of different values, otherwise (*weighted points*) it is continuous.

Let L be the number of *LOD regions* ("levels"). In the program, it is usually 1, 4, 8 or 16. For *weighted points* downsampling, L can take any value and can be changed at runtime. In both cases, $r(d)$ is defined the same way:

$r_\ell(\ell, a, L, n, n_{\text{min}})$ gives the downsampling ratio for level $\ell \in [0, L-1]$. n is the total number of points used as input to the downsampling algorithm, n_{min} the minimal number of points the should be outputted. If the algorithm is applied to the full model $n = |P|$ (number of points in model), and $n_{\text{min}} = C$ (capacity of renderer) is used, since the goal is to stay as close as possible but inferior to C, so as to not lose more points than necessary. The function is defined such that $r_\ell(\ell = 0) = n$ (level 0 means no downsampling), $r_\ell(\ell = L-1) = n_{min}$ (maximal possible downsampling must be reached on last level), and it is stictly decreasing. The *amount* parameter a sets how strongly the function should be decreasing, and must be adjusted to fit the input model.

If *LOD regions* are used, this function is evaluated in the preprocessing stage, when the $L-1$ downsampled point sets are created. For *weighted points*, it is evaluated at runtime. Then, a and L also can be changed at runtime, and ℓ can take real values (as opposed to integers only). This allows for continuous downsampling.

The level to use at a given location is chosen by the function $\ell(d, s)$, where d is the distance to the cemera, and the *setting* s a parameter that can always be adjusted at runtime (unlike a). The function is defined such that $\ell(d = 0) = 1$ (no downsampling when very close to camera), and $\lim_{d \to \infty} = L$ (eventually reach maximal level). Also, it increases with d, where s determines how quickly. $\ell(d, s)$ is defined as a real function, but when *LOD regions* are used, the floor value $\lfloor \ell(d, s) \rfloor$ is always taken.

In short, the process of choosing the downsampling ratio $r(d)$ in function of distance d works as follows: For *weighted points*, $r(d) = r_\ell(\ell(d, s), a, L, n, n_{min})$ is evaluated at runtime, and s, a, L can all be adjusted at runtime. Thus $r(d)$ is continuous.

For *LOD regions*, L point sets are created in the preprocessing stage which are downsampled with ratio $r_{ell}(i, a, L, n, n_{min})$. The integer $i \in 0 \dots L-1$ is the level of the point set, and a, L, n, n_{min} all need to be known in this stage. The program stores the downsampling ratios associated with the L levels with the generated data structure. At runtime, the downsampled point set for level $i = \lfloor \ell(d, s) \rfloor$ is chosen depending on d. Only s can be adjusted at runtime.

2.4.5 Definition of the downsampling functions

The functions ℓ and r_ℓ are defined as follows:

$$\ell(d, s) = \begin{cases} \min\{\frac{d-d_0}{\Delta d}, L-1\} & d > d_0 \\ 0 & d \leq d_0 \end{cases}$$

with $d_0 = s^e$ the *start distance*, i.e. the distance from the camera where downsampling should start, and $\Delta d = s$ the *step distance* after which the next level is chosen.

So the level is chosen linearily with the distance, with an offset d_0. The *setting* s controls the intervals between levels and the start distance. In the program, the e is fixed to the value $e = 1.3$. It makes sure that d_0 scales faster than linearily with s, because the jump from no downsampling to the first level of downsampling represents a greater loss in output quality than the jump between downsampling levels.

The following graph shows the function in its staircase form, with different values for s.

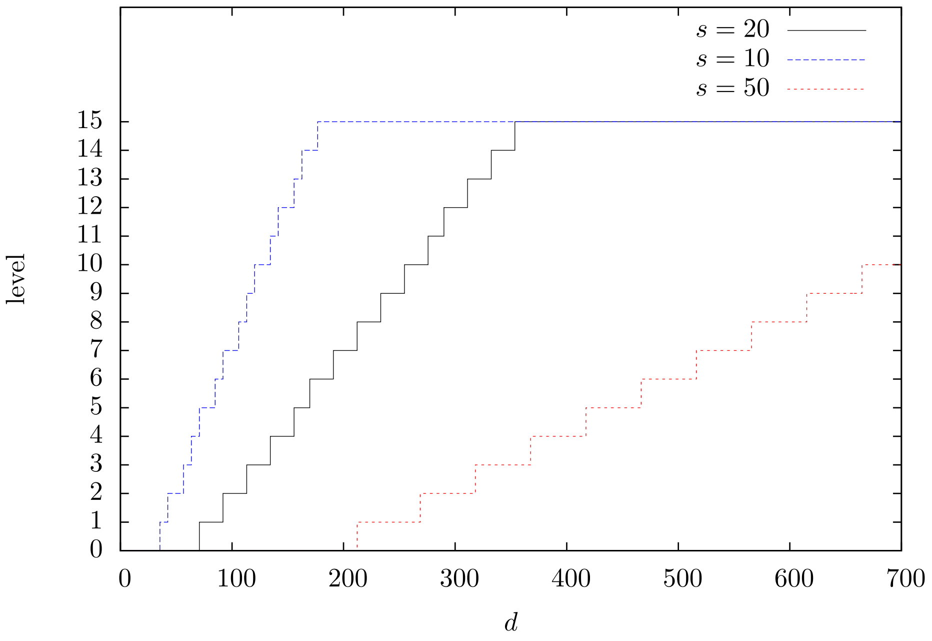

Figure 2.6: Downsampling level for distance function $\ell(d, s)$

The downsampling ratio function is defined as follows:

$$r_\ell(\ell, a, L, n, n_{\min}) = 1 - (1 - r_{\min}) \left(\frac{\ell}{L - 1} \right)^a$$

where $r_{min} = \max\{\frac{n_{\min}}{n}, 1\}$ is the minimal downsampling ratio. Note that when $n \leq n_{min}$, the function always yields 1 and no downsampling will be applied.

The following graph shows the function with different values for the downsampling amount a:

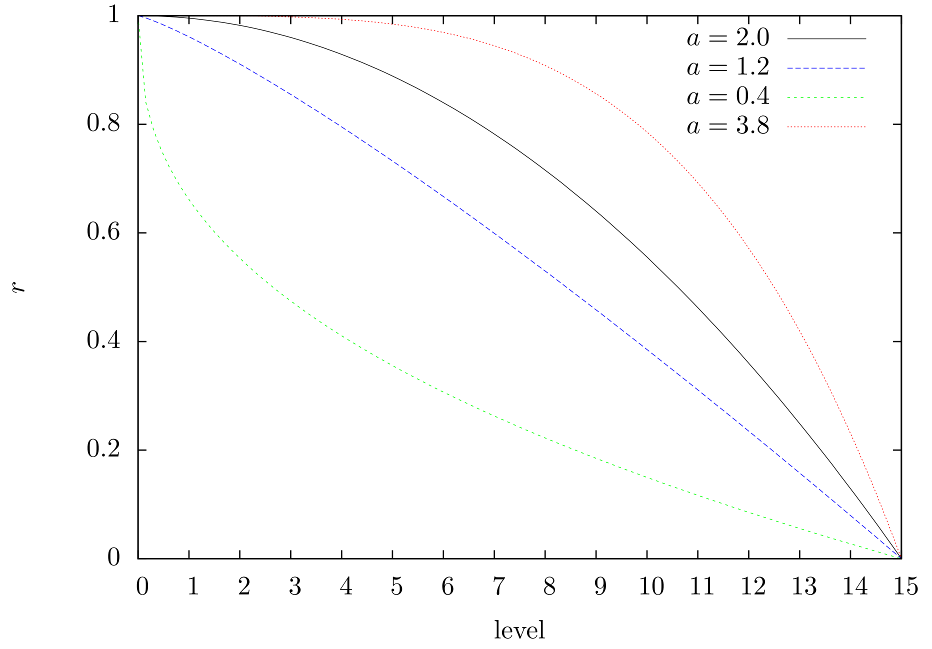

Figure 2.7: Downsampling ratio for level function $r_\ell(d, a)$

Figure 2.8 shows visible areas[2] with distances d, $2d$ and $3d$ to the camera in a perspective projection. Their area is proportional to d^2. Supposing that surfaces of the point cloud is located exactly on these areas, the point density of the projection ρ' would also be proportional to d^2.

Since ρ' should be more or less constant, setting $a = 2$ makes sense as it cancels out the effect. In practice, it needs to be adjusted to fit the particular point cloud.

Finally, this graph shows $r_{d,s}$ $(r_\ell \circ \ell)$ in piecewise form, with $a = 2$, $L = 16$ and $r_{\text{min}} = 0.2$:

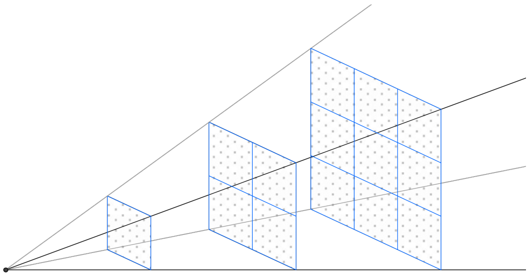

Figure 2.8: Perspective projection and area

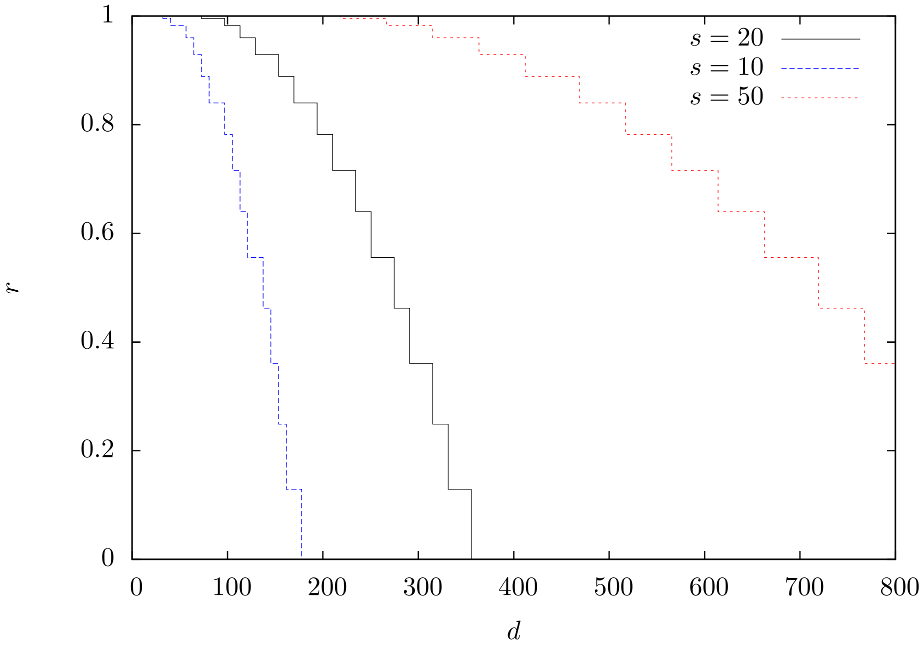

Figure 2.9: Downsampling ratio for distance $r(d, s)$

2.4.6 Adaptive downsampling

2.5 Occlusion culling

[2] In reality, the areas would be curved (intersection of view frustum and sphere of radius d), but the simplification holds because the field of view is small

3 Data structures

3.1 Cubes structure

3.2 Tree structures

3.2.1 Octree

3.2.2 KdTree

4 Application

5 Results

6 Conclusion

Bibliography

[1] Klaus Hartmann Gil Gribb. Fast extraction of viewing frustum planes from the world-view-projection matrix. 2001.