Dynamic loading of 3D models

PROJ-H-402 Project Report

Tim Lenertz, INFO MA1, ULB

May 10, 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 synthesized 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 subset of points to render, while the user moves through a large point cloud. This involves filtering 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 the storage device. 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, 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.

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 reproduce 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 **??**. 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 the 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 could 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 implement $f_P(\boldsymbol{M})$ 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 3x3 matrix arithmetic alone.

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 number 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 $\rho = n/A$. This value remains unknown, because the program does not know or try to find the shapes of the surfaces. 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 position $\overrightarrow{p}$ (assumed to be located on a surface), the new density becomes $\rho' = r(d(\overrightarrow{p})) \times \rho$ (with $r \in [0, 1]$).

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 distribute the weights is to set each weight to a random value. However, this leads to an irregular visual pattern in the way downsampled points are placed on the surfaces, as seen in figure 2.1. The left-hand side of the point cloud is not downsampled and retains the original regular pattern from the model.

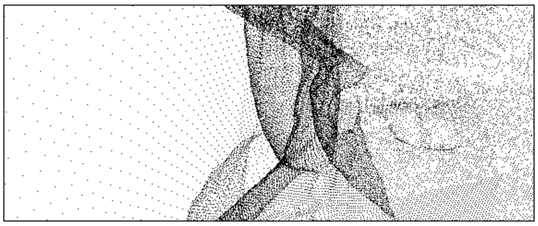


Figure 2.1.: Weighted points downsampling using random weights

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' is then produced out of pieces of these sets. The resulting P' is thus composed of several *level of detail* (LOD) regions[1], and so $\rho'(d)$ becomes a non-continuous staircase function, and visual discontinuities are produced. However, this method allows for producing the sets P'_r without 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.

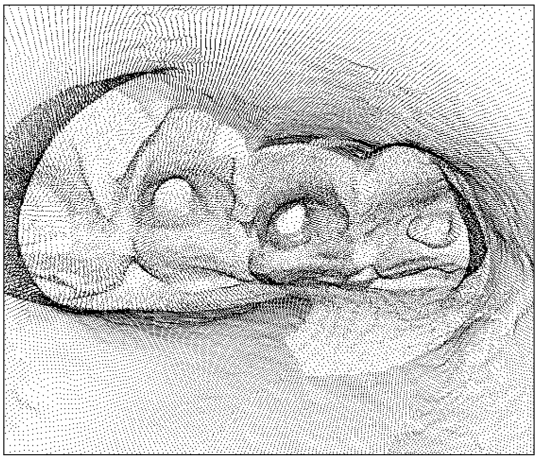


Figure 2.2.: Example of uniform downsampling

The input point set P is divided into a regular grid of cubes $C_{i,j,k}$ with side length l. 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.

The main difficulty is to find the value for the side length l, such that $n = |P'|$ is approximately equal to $r \times |P|$. Figure 2.10 shows how n varies in function of l for some example models: In general, n gets larger as the side length l 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 l 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 l 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 l, but for $l' > l$ the grid becomes such that a boundary lies between these points. This effect only occurs at the small scale because as l 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 no way to effectively estimate a value for $n(l)$. Instead the algorithm does a dichotomic search to find a value l such that $n \approx r \times |P|$, within a certain tolerance. The small-scale ascending intervals of $l(n)$ are taken into account.

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

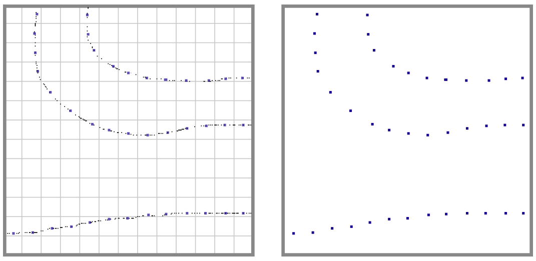


Figure 2.3.: Two-dimensional visualization of uniform downsampling algorithm

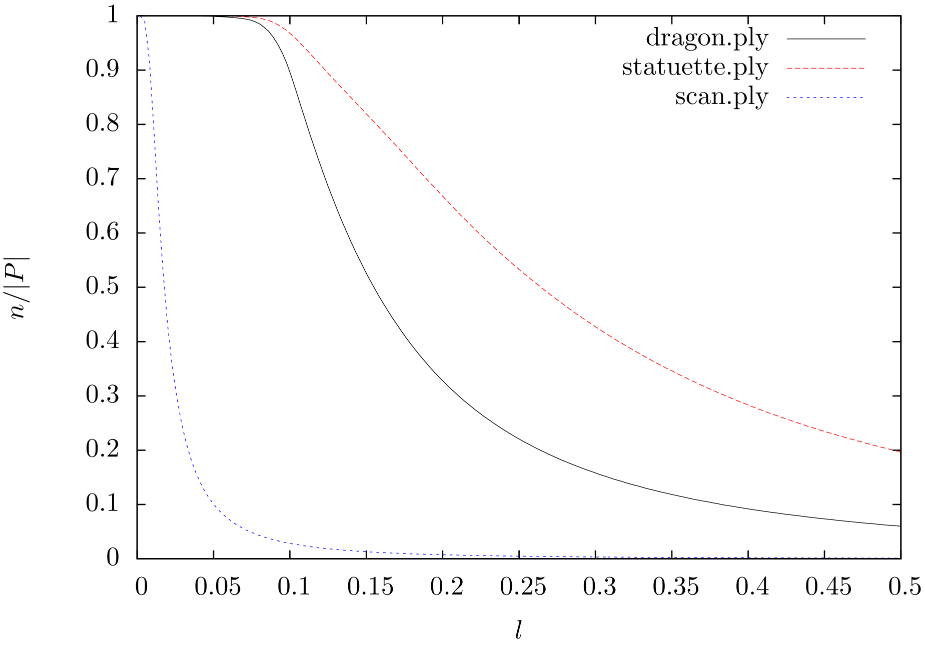


Figure 2.4.: Number of output points n VS cubes side length l in uniform downsampling

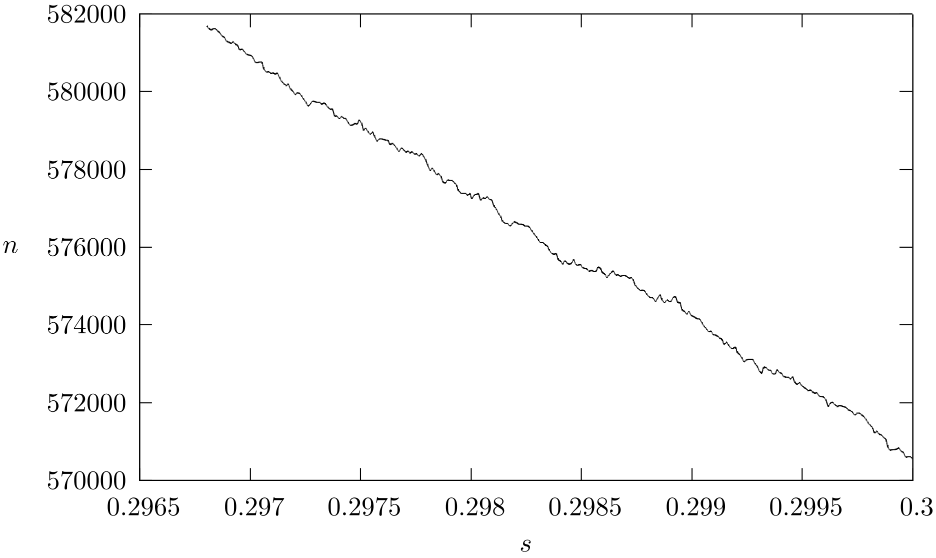


Figure 2.5.: Close-up view of figure 2.10

2.4.4. Choosing the downsampling ratio

The function $r(d)$ determines what 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. On what regions downsampling will be applied, and how d is defined depends on the data structure used. (see next chapter.)

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:

$r_\ell(\ell, a, L, n_{\text{total}}, n_{\text{min}})$ gives the downsampling ratio for level $\ell \in [0, L-1]$. n_{total} 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 $n_{\text{total}} = |P|$ (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 $r_\ell(\ell = 0) = n_{\text{total}}$ (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 during 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 camera, and the *setting* s a parameter that can always be adjusted at runtime (unlike a). The function is defined such that $\ell(d = 0) = 0$ (no downsampling when very close to camera), and $\lim_{d\to\infty} = L-1$ (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*, $r(d) = r_\ell(\ell(d, s), a, L, n_{\text{total}}, n_{min})$ 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 $r_{ell}(i, a, L, n_{\text{total}}, n_{min})$. 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 downsampling level function ℓ is defined as follows:

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

with $d_0 = b^{1.3}$ the *start distance*, i.e. the distance from the camera where downsampling should start, $\Delta d = b$ the *step distance* after which the next level is chosen, and $b = \frac{250}{s}$.

So the level is chosen linearly with the distance, with an offset d_0. The *setting* s controls the intervals between levels and the start distance. The exponent 1.3 for d_0 keeps d_0 larger than Δd, because the jump from no downsampling to the first level of downsampling represents a greater loss in output quality than a jump between downsampling levels.

The definition of b makes sure $\ell(d)$ increases faster when s is larger: If $s = 0$, ℓ is always zero, and no downsampling will be applied. If $d \gg 0$, eventually $\ell(d)$ will always yield the maximal downsampling level $L-1$.

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

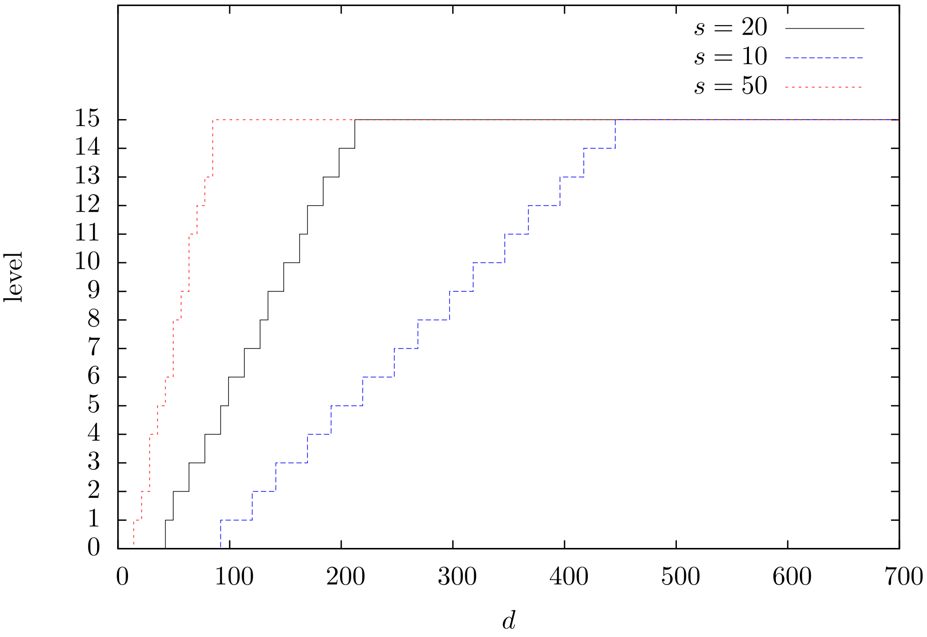


Figure 2.6.: Downsampling level for distance function $\ell(d, s)$

The downsampling ratio function r_ℓ is defined as follows:

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

where $r_{min} = \max\{\frac{n_{\text{min}}}{n_{\text{total}}}, 1\}$ is the minimal downsampling ratio. Note that when $n_{\text{total}} \leq n_{min}$, 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:

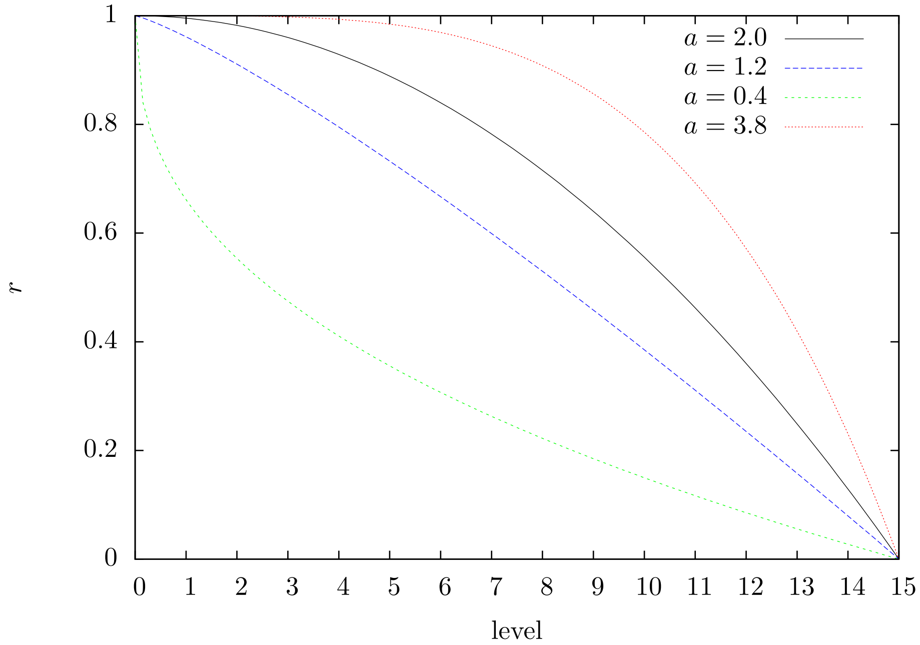


Figure 2.7.: Downsampling ratio function $r_\ell(d, a)$

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 $r_{\text{min}} = 0.2$:

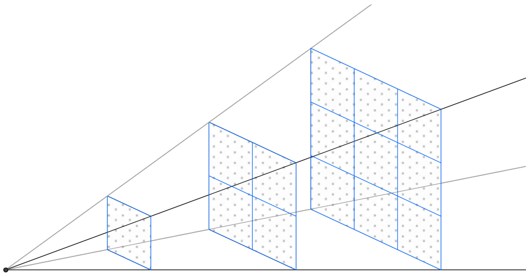


Figure 2.8.: Perspective projection and area

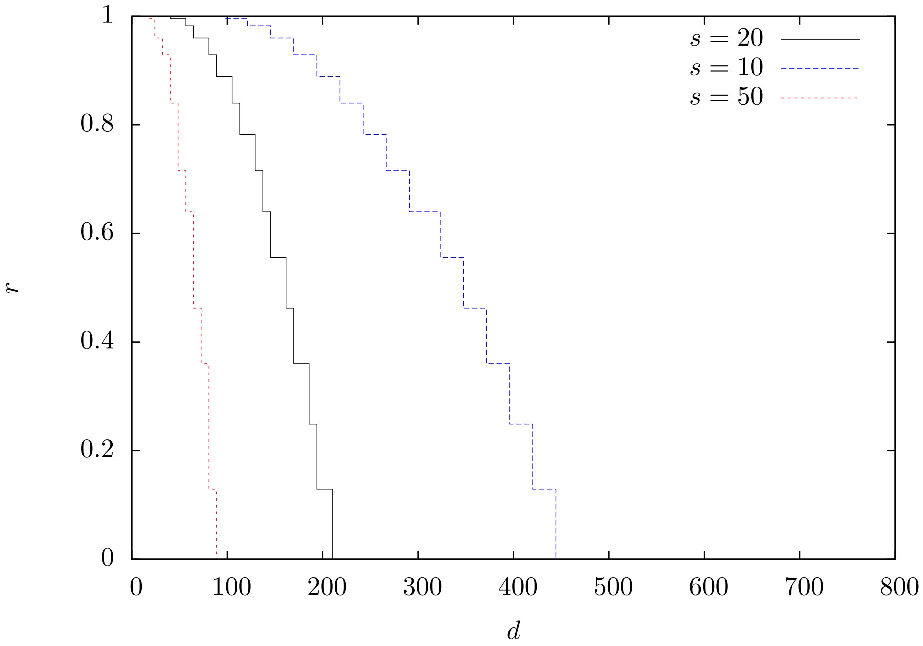


Figure 2.9.: Downsampling ratio for distance $r(d, s)$

2.4.6. Adaptive downsampling

The following graph shows how the downsampling setting s affects the number of outputted points n, on different camera view points and for different models. Since downsampling is applied on the point set that was already filtered by view frustum culling, the camera position and orientation have a great effect on the output range.

[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

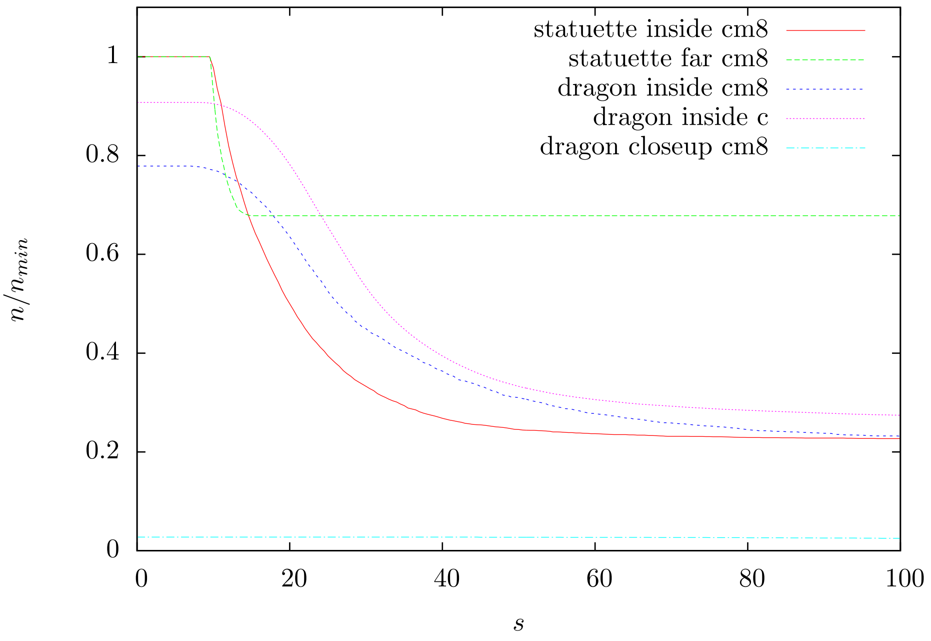


Figure 2.10.: Number of output points n on downsampling VS downsampling setting s

The outputted number of points n never gets larger than n_{min} because in this example $n_{\text{min}} = C$ the capacity of the renderer, and the algorithm that extracts points from the data source never outputs more than C points. It is chosen so that in the extreme case where all points from P lie in the view frustum, it can still be rendered by choosing the highest downsampling level $\ell = L - 1$ everywhere. This is the case in `dragon far cm8`. frustum culling makes the actual outputted number of points n smaller than n_{min} as seen in the graph.

For view points where only a small portion of the model is visible, n can remain smaller than n_{min} for any setting. An extreme case is the a close-up view list in `dragon closeup cm8`.

The optimal setting s is one where n is as large as possible, but below n_{min}. If it is smaller than is could be, the renderer capacity would still allow for a higher number of points and thus higher quality. If it hits n_{min}, then the program attempts to extract more than C points and breaks up in the process, leading to missing parts in the output.

Also the aspect of the curve varies greatly with the camera position and orientation, especially for composite models where different parts have different point densities. So the user would have to keep adjusting the setting s in order to get good results. Adaptive downsampling is an attempt to implement a controller system that automatically adjusts the setting as the user is moving through the point cloud.

...

2.5. Occlusion culling

Occlusion culling, or hidden surface removal is the process of removing points from P that are occluded by another points surface in front of it. The algorithms have no knowledge about the shape of the surfaces, but only operate on the assumption that the model is such that all points lie on a surface with a density ρ that remains locally more or less constant. Removing the occluded points aids in reducing the size of P', and also makes the output look better.

Different algorithms exist to approximate occlusion culling on a point set without knowledge of the surfaces.

2.5.1. Point shadows

A simple method that is implemented in the application is do draw a black *shadow* behind each rendered point. The shadow is a disk or square a few pixels wide centered around the point. If a surface is at a reasonable distance from the camera, its points will all be just a few pixels apart on the projection, and the shadows completely cover the points behind it. Thanks GPU's *depth testing*, this still works when those points are rendered after those from the surface.

The method can improve visual quality, but it no longer works when the camera is too close to the surface, and as a side effect it produces black contours around the edges of the model, since the shadows don't take the (unknown) normal vectors of the points on their surface into account. Also, it increases the load on the GPU too much since every pixel of the shadow it processed the same way as a point from P', and there can easily be many unnecessary overlapping pixels produces this way. Because it is implemented on the GPU (as an OpenGL shader), it does not contribute to the goal of reducing the size P' to fit C.

2.5.2. HPR Operator

...

Approximation

3. Data structures

The f_P function described in the previous chapter takes a unordered set P of three-dimensional points as input. In an implementation, P will be stored in a certain *data structure*, which is essentially a way of representing P using a one-dimensional array of bits. The data structure will hold the points P, along with weights or LOD sets for downsampling, and possibly additional information. The criterion for choosing a data structure is that f_P can be implemented in an efficient way.

3.1. General

Since the data structure must contain all the n_{total} points of P, its size must always be at least in $O(n_{\text{total}})$. The algorithm which extracts $P' = f_P$ will typically only need to read a small portion consisting of $n \ll n_{\text{total}}$ points. To make this efficient the data structure must be designed such that:

- It must be possible to determine the portions of data to extract without searching through the whole structure. The worst case would be a data structure that consists simply of an array of points in any order, because in order to find the points that lie inside the current view frustum the algorithm would have to check every point in the array, leading to a time complexity $O(n_{\text{total}})$. This would defeat the goal of enabling efficient loading of large models.

- The data structure is always serialized to storage in one dimension. It is generally much more efficient to read a small number of long segments, rather than many small segments: Seeking times on the storage device are avoided, and various caching mechanisms from the underlying file system are used.

  No order can be defined for three dimensional points, but their one-dimensional representations always have an order. So a compromise needs to be found by which the serialization is ordered such that for the majority of cases, points that are close to each other in three dimensional space remain close to each other in the serialization.

Two techniques are used to achieve this: The model is split into smaller *regions*, and the points within that region remain grouped together in the data structure. Points within the same regions have the property that their distance is upper bounded by the size of the region, and so this contributes to the ordering issue. Also, *pointers* is used in various ways, meaning reading algorithm will directly jump to a given offset in the data independently of its distance.

3.2. File types

The data structure is built on top of a *file type*. Several possible file types were compared:

Raw file Essentially no file type is used, and the data structure is implemented on the lowest level. The implementation needs to write and read the structure as a single continguous segment of binary data. This allows for the highest level of control, but requires an ad-hoc definition of helper structures such as a way to split the file into different parts for points, downsampled points, regions, additional file information... Also the precise way that data values serialized (floating point and integer data types, alignment, endianess) needs to be taken care of in the implementation of the data structure.

ASCII-based file Storing data as an ASCII-based text file instead of as raw binary data removes the latter problems of data serialization. The remaining problems stay the same. The files get much larger: If the file contains a set of 1000000 points with x, y, z coordinates, storing them as 4 byte floating point values would require 11.72 MB. If the values are written in decimal representation with 8 digits, plus separation characters, a size of about 29.56 MB would result. This additional space requirement, and the parsing overhead when reading from the file, makes such file types impractical for large amounts of data.

XML XML would provide a way to represent structured data, and so it would be possible to build the data structure in terms of XML tags, containers and attributes. However, it suffers the same problems as any ASCII-based format, to an increased extent since it tends to be very verbose compared to simpler formats. Also, the purpose of XML is not to deal with large amounts of data that is dynamically read, but rather to deal with relatively small, deeply structured files such as computer programs or web pages, and it provides tools to validate, transform, interlink, etc. such files.

HDF HDF (*Hierarchical Data Format*) is the main format used in the program. It is a binary designed designed for storage of large quantities or scientific data in such a way that parts of it can be extracted efficiently. It allows for the definition of simple or combound data types (such as 3 floating point values and 3 bytes for an XYZRGB point), and can store several data sets or such values in the file. These data sets are ordered arrays of one or more dimensions, and the *hyperslap* mechanism provides a way to efficiently read or write in it. HDF stores the data sets either directly as sequential data on the storage device, but can also internally store the data sets in multiple chunks, and provide additional optimizations such as internal caches for dealing with large data sets.

SQL database Using a relational database database such as *SQLite* allows for implementing the structure on a higher level, while the database takes care of arranging the data internally so as to allow for fast retrieval. This includes its internal data structures and algorithms used to implement indexing, executing *SELECT* queries, caching...

In general, it seems a much better choice to stay with a lower level file format like *HDF* that still provides basic facilities for separating data sets, defining data types, etc. The definition of the *data structure* should include how it is serialized to a one-dimensional representation, since this is critical for its performance.

3.3. HDF

The *Hierarchical Data Format* HDF, was designed to allow for storing large quantities of multi-dimensional numerical data, such as fields recorded by scientific instruments. Using to serialize a data structure is not its main purpose, but the features *HDF* provides still make it a good option. Here is a brief overview of some building blocks of *HDF5*:

data type Usually numeric (integer or real), possible to have fixed length arrays for any data type. The precise representation (endianness, bit size) can be defined.

combound data type Similar to a `struct` in C: Data type composed of several named members of different types.

data set An HDF5 file contains one or more data sets, and these contain the actual data. It has a data type (can be combound), and can be one- or multi-dimensional. Usually its length needs to be specified at creation, but *chunked data sets* are stored internally in a way that allows for a dynamic unlimited extend.

data space A selection of data in a data set. Can be a simple one-dimensional interval, or a *hyperslab* that can represent a selection on a serialized multi-dimensional data set.

attribute Single named data values that can be attributed to a data set of another object. Used to attach additional information to the file.

Here, only one-dimensional data sets are used. The `points` data set would be a data set of a combound type *point*, which has data members for the coordinates and color.

3.4. Cuboid regions

All of the data structures used in the projet rely on subdividing the three dimensional space in axis-aligned cuboid regions. A cuboid $\mathcal{C}$ is defined by its *origin* (x, y, z) and its *extremity* (x', y', z'), and a point $\overrightarrow{p}$ is considered to be inside the cuboid if and only if $p_x \in [x, x'[$, $p_y \in [y, y'[$ and $p_z \in [z, z'[$.

Because the intervals are open on one side, points that lie on the border of two adjacent cuboid can never fall into both cuboids. However, when defining a bounding cuboid that should enclose a given set of points, it becomes necessary to make the coordinates of its extremity slighly larger than the point with maximal coordinates.

3.4.1. Floating point issues

Cuboids are defined in terms of origin and extremity, instead of origin and side lengths, in order to deal with floating point imprecision problems that would otherwise occur in the implementation: For the *tree structures*, cuboids will be recursively subdivided into smaller cuboids, out of which some share extremitiy coordinates with the parent cuboid. If side length were to be used, those floating point values would not be copied, but rather need to be recomputed from the new origins and side lengths of the child cuboids. This could make their values slightly different, and as a consequence the algorithms would be incorrect, as they could for instance encounter points that are inside a child cuboid, but outside the parent cuboid. These edge cases are significant because the coordinates are real values, whereas quantities of points are discrete. Parts of the program rely on having exact knowledge of these quantities, for example to align data segments in files the right way.

In other situations where floating point imprecisions are tolerated, the program uses a small threshold value ϵ to test two floating point values for "equality". That is, $f_1 \approx f_2 \iff |f_1 - f_2| < \epsilon$.

3.4.2. Point-to-cuboid distance

Also for all of the data structures, the implementation of f_P will apply downsampling to entire cuboid regions with a constant downsampling ration r. This ratio is determined by $r(d, s)$ as described in the previous chapter, where $d(\overrightarrow{p}, \mathcal{C})$ is a *point-to-cuboid distance* from the camera position to the cuboid. The program allows the user to choose between several definitions for d:

center The distance from $\overrightarrow{p}$ to the centroid $(\frac{x+x'}{2}, \frac{y+y'}{2}, \frac{z+z'}{2})$ of $\mathcal{C}$.

minimal The minimal distance from $\overrightarrow{p}$ to any position $\overrightarrow{q}$ on the border of $\mathcal{C}$. The $\overrightarrow{q}$ which gives the minimal distance may be a corner of $\mathcal{C}$, or an orthogonal projection of $\overrightarrow{p}$ on either an edge or a side of $\mathcal{C}$.

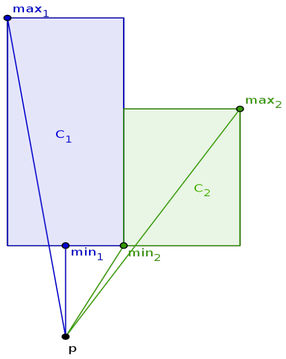
maximal The maximal distance from $\overrightarrow{p}$ to any position $\overrightarrow{q}$ on the border of $\mathcal{C}$. In this case, $\overrightarrow{q}$ is always a corner of $\mathcal{C}$.

mean The mean value of the minimal and maximal distances.

Assuming that any definition of d gives a value $\geq$ the maximal distance and $\leq$ the minimal distance, it is not possible in general to define d such that if $d(\overrightarrow{p}, \mathcal{C}) > d(\overrightarrow{p}, \mathcal{C})$, then for all $\overrightarrow{q_1} \in \mathcal{C}$ and all $\overrightarrow{q_2} \in \mathcal{C}$, it remains true that $d(\overrightarrow{p}, \overrightarrow{q_1}) > d(\overrightarrow{p}, \overrightarrow{q_2})$.

The figure on the right illustrates this: $d(p, \text{min}_1) < d(p, \text{min}_2)$, whereas $d(p, \text{max}_1) > d(p, \text{max}_2)$. So no matter how the *point-to-cuboid distance* is defined in that case, there will always be points in one of the cuboids that don't fullfill the condition. When such points are inside the view frustum, is leads to visual discontinuities, as there are places where the downsampling ratio gets lower at a greater distance instead of higher.

As a compromise, one of the proposed definitions needs to be chosen chosen that reduces this effect. Generally, better results are produces when the regions are more or less cubic, and in that case the **center** definition of d yields good results. Otherwise, **minimal** tends to be better.



3.5. Cubes structure

The *cubes structure* is a first attempt at a simple way to organize the set of points such that the algorithm is faster than a naive implementation which would iterate through the whole set of points at each evaluation of f_P.

Space is subdivided into a regular grid of axis-aligned *cubes* $\mathcal{C}_{i,j,k}$ with a common side length l.[1] These cubes are the cuboid regions for this structure. Because of the way their boundaries are defined, every point $p \in P$ will belong into exactly one cube. The side length can be configured when the structure is created, and is by default set to $l = 5$. i, j, k are the coordinates for the cubes in the grid. For the cube $\mathcal{C}_{0,0,0}$, its *origin* corresponds to the origin of the space.

Only cubes that contain at least one point are created, so their total number will be finite. In the first stage of the structure creation progress, the algorithm iterates through the full set of points P, for each point p and determines the coordinates of the cube into p will belong by floor division:

$$(i, j, k) = (\lfloor p_x/l \rfloor, \lfloor p_y/l \rfloor, \lfloor p_z/l \rfloor)$$

This is correct because the definition of the floor function coincides with the way that the cuboid regions' boundary intervals are defined. Similar relations will occur for the tree structures, where it is critical for the correct functionning of the algorithm that the relations always hold, despite possible floating point imprecision errors.

There are two variants for the cubes structure, one with *weighted points* and one with *LOD regions.*

3.5.1. Weighted point variant

The algorithm proceeds by attributing random weights $w \in [0, 1]$ to each point $p \in P$ in an uniform distribution.

When a *HDF* file type is used, two one-dimensional data sets `points` and `cubes` are created. The `points` data set is contains $|P|$ entries of the `point` combound type. This type consists of four floating point values for the point coordinates p_x, p_y, p_z and the weight w, plus three byte values for the RGB color information. The points in P are added into this data set, grouped by the cube in which they belong. Inside each cube group, the point entries are sorted by ascending w. The outer order of the cube groups does not matter.

The `cubes` data set is filled with an entry for each cube. For each cube entry, it contains the three incides of the cube i, j, k, the number of points in the cube $N(\mathcal{C})$, and the offset $\text{off}(\mathcal{C})$ in the `points` data set of the first point from that cube. So given a `cubes` entry the corresponding array of points in the `points` ranges from $\text{off}(\mathcal{C})$ to $\text{off}(\mathcal{C}) + N(\mathcal{C}) - 1$.

This data structure is also implemented with an *SQLite database* as backend. Here, two tables for the cubes and for the points are created, except that instead of storing offsets, each cube has a unique *primary index*, the points table contains an *index* column that gives is set to the cube's index. The points corresponding to one cube are loaded using SQL queries, and *SQLite*'s indexing algorithms take care of the internal representation of the data and then necessary optimizations.

3.5.2. LOD regions variant

For the *LOD regions variant*, the algorithm instead proceeds by generating, for the sets of points in each cube $P_\mathcal{C} = \{p \mid p \in \mathcal{C}\}$, the $L - 1$ downsampled point sets $P'_r(\mathcal{C})$.[2]

In the *HDF* file, L different point sets `points#` are created which all the points for the downsampled point sets. Again, the points are grouped by their group. Here, the points don't have a weight, and the order of the point entries inside the groups, just like the order of the groups, doesn't matter. The `cubes` data set now needs to store the cubes' point offsets and counts for each downsampling level, because they will be different.

There is no implementation for a *SQLite database* file type for this variant.

[1] This is unrelated to the cubes used for uniform downsampling in the previous chapter.

[2] An alternative implementation would be a generate the downsampled point sets for the entire model P, and then reattribute the resulting points $p' \in P'_r$ to their respective cubes.

3.5.3. Extraction algorithm

The *extraction algorithm* of a data structure is executed repeatedly as the used moves through the points cloud and his camera position and view frustum changes. One of the main goals of the project is for this algorithm do be as efficient as possible, even when the model is very large.

For the cubes structure, the algorithm iterates through the full list of cubes each time, and for each cube, checks whether it is completely or partially inside the view frustum. This test is done on the cube alone and without considering the points in that cube, using the test described in A.5. If the test is positive, it determines a downsampling level ℓ for that cube using $\ell(d, s)$ where the point-to-cuboid d is calculated by one of the schemes described above.

For the *LOD regions* variant, it then copies the points of the cube for level ℓ from the file into the output buffer. This always a single segment of data. For the *weighted points* distance, it extracts the points for which $w < r$. So this requires iterating through the points, but because the points are sorted by ascending w, the procedure can stop when $w \geq r$ is reached. So the *weighted points* variant is slower, but requires less storage space. It also allows for r of a cube to take any value as opposed to only L possibilities, but small discontinuities still occur at the boundaries of the cubes grid.

The complexity of the algorithm is linear with respect to the number of cubes, and so it scales too much with the size of the model.

3.6. Tree structures

The *tree structures* recursively subdivide space into cuboid regions. A *root cuboid* is taken that encloses the entire model[3] Then, according to a certain rule, that cuboid is subdivided into *c child* cuboids. These child cuboids must be such that they don't overlap, don't go outside the original (*parent*) cuboid, and completely cover the *parent* cuboid. This process is recursively repeated, until the number of points p that lie inside a cuboid $\mathcal{C}$ is smaller or equal to the *leaf capacity k*. These cuboids are the *leaves* of the tree formed by the hierarchy of cuboids.

3.7. Serialization

The constraints for the rule by which the cuboids are subdivided imply that it is possible to order the points in such a way that the points inside a given cuboid $\mathcal{C}$ of any depth lie on one segment of the ordered list. If $\mathcal{C}_1$ is a child cuboid of $\mathcal{C}_0$, then $p \in \mathcal{C}_1 \Rightarrow p \in \mathcal{C}_0$, and if $p \in \mathcal{C}$ and $\mathcal{C}$ is not a leaf, then there must exist exactly one child cuboid of $\mathcal{C}$ that also contains p.

This can be reproduced in the one dimensional representation by recursively sudividing segments in the same manner. So, each cuboid will be associated an *offset* $\mathrm{off}(\mathcal{C})$ in the points list, and a *number of points* $N(\mathcal{C})$.

Just like with the *cubes structure*, downsampling can be added through the use of *LOD regions*, where there are multiple offsets and numbers of points for each point. However, it is no longer possible to effectively implement a *weighted points* variant: The points belonging to a cuboid cannot be ordered by their weight w because subsegments would also have to be ordered the same way. Without this, the program would be forced to iterate through all points in the view frustum.

Similarily to cubes in the *cubes structure*, cuboids "nodes" are stored in a separate list, with their link (offset, count) to the points sets. Also the offsets (in the same list) of a node's child nodes are stored with it, as well as the node's cuboid. There is no need for the nodes to be ordered in a specific way.[4] Storing the cuboid with the node slightly increases the file size, but it simplifies implementation, avoids the need to store the *node points information*, and allows for trees formed by several different rules. (see later)

In the implementation, *splitters* define the rule by which the cuboids are chosen and subdivided. The following three are implemented:

3.7.1. Octree

All cuboids must be cubic, i.e. have equal side length. The *root cuboid* will be the smallers cube that encloses the entire model, such that the model is placed at the center of the cube. Each cube will be subdivided into 8 child cubes, that are on the 8 sides of the cube's center point.

This is a good alternative in most cases because the regions are cubic, and so choosing the distance from the camera to the center of a cube as *point-to-cuboid* distance yields good results.

3.7.2. KdTree

For the *KdTree*, the cuboids no longer need to be cubes. Each cuboid is split into 2 child cuboids. The *split plane* that separates them is alternately orthogonal to the $X, Y, Z, X, \ldots$ axis, depending on the *depth*[5] of the cuboid modulo 3. Two different ways of choosing the position of the split plane on that axis are implemented:

Half

The split plane is always at the center of the cuboid. That way if the *root cuboid* is near-cubic, and cuboids won't diverge from an near-cubic shape. Its advantage over the *Octree* is that there are only

[3] Due to the definition of the cuboid boudaries, its extremity coordinates need to be made slightly larger than the maximal point coordinates.

[4] This is useful in a parallelized implementation of the structure creation algorithm; see further.

[5] The *depth* of the *root cuboid* is set to 0. For each cuboid $\mathcal{C}$, its depth is that of its parent plus one.

two children per node, which useful for the *piecewise tree structure* (see further).

Median

In the proper definition of a *KdTree*, the position of the *split plane* is at the median of the coordinates for the given axis, of all the points in the cuboid. This implies that there will be the same number of points in both children, and so the entire tree will be perfectly balanced.

This could in general accelerate descent into a tree. But it does not present an advantage here, and leads two problems: The cuboids tend to become strongly non-cubic, making it impossible to define a good *point-to-cuboid distance*. And in order to calculate the median, the entire array of coordinates needs to be stored temporarily in memory and then sorted. This is not practical for large point sets.

There are algorithms for approximating a median value without this. For example, one way would be to build a histogram on intervals, find its median, and repeat the same on that interval. This is not implemented in the application.

3.7.3. Splitter

The *splitter* of a *tree structure* consists of these elements:

adjust root cuboid For example making the root cuboid cubic for the *OcTree*.

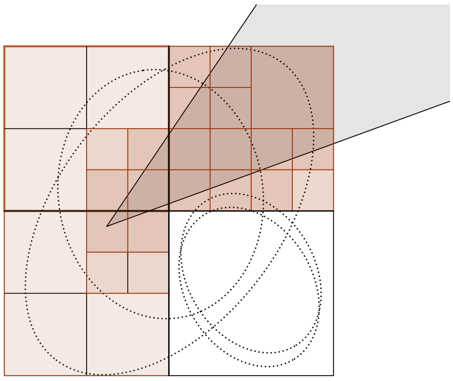
node points information Information associated to each node that is deduced from the points in the cuboid. For *KdTree*, it is the position of the split plane.

child cuboid function Calculates the cuboid of a child node, using the child index i, the *node points information*, and the *depth* and *cuboid* of this node.

child index function Calculates the index of the child (of this node) that contains a given point p. Also uses the *node points information*, and the *depth* and *cuboid* of this node. This can be implemented faster than checking every child cuboid, for example using just a floating point value comparation with the *KdTree split plane* position.

As described previously, the implementation must take care of floating point imprecisions for these two functions.

3.7.4. Extraction algorithm



The recursive algorithm starts with the *root cuboid*. If it is completely inside the view frustum, the full set of points for that cuboid is extracted. If it is partially inside the view frustum and not a leaf, the algorithm recursively descends into the child cuboids. It it is completely outside the view frustum, the cuboid is discarded. So unlike for the *cubes structure*, large parts of the model can be immediately either included or excluded. The sizes of the cuboids change exponentially with the depth in the tree, so it is very well possible to immediatly exclude over half of the model, but be very precise inside the frustum.

Extra splits

Including a cuboid means that the set of points inside that cuboid is copied to the output buffer. As described before, these points are always on a single data segment. The downsampling level ℓ for this cuboid is chosen using a *point-to-cuboid* distance, as described before. However there is the problem that if large cuboids are to be included, choosing a single downsampling level for it will not be appropriate.

Therefore, the algorithm further recursively splits such cuboids, as long as the difference between its *maximal* and *minimal point-to-cuboid* distance is larger than a given threshold. This difference corresponds to the two extrema where $r(d)$ would be most different.

Load order

The algorithm described here does not specify in which order the children of a cuboid should be traversed. This becomes a problem when the *downsampling setting* is not sufficient, and the algorithm stops at some instant during the recursions because the capacity C has already been filled. Then it can happen that parts of the model farther to the camera are rendered, but closer parts are no longer included.

If such an overflow occurs, it is preferential to first render the closer parts, and maybe drop the points in the background.[6] A modified, "ordered nodes" version of the algorithm does this.

Firstly, when traversing the child cuboids of a cuboids, the algortihm first sorts them by their *point-to-cuboid* distance to the camera, and recurses into the closer ones first.

In addition to this, it also must no longer begin with the *root cuboid*, but rather begin at the *leaf* containing the camera, and then move outward. This is mostly an optimization, but in somecases, for instance when the *minimal point-to-cuboid* distance is chosen, prevents the algorithm from moving down the wrong node.

Also as a small optimization, the algorithm stores the current node containing the camera and tracks as the camera moves into parent/neightboring nodes.

3.7.5. Creation algorithm

The following describes the procedure taken to create a tree structure. First the bounds of the model are determined by finding the minima and maxima of the points' coordinates. An enclosing cuboid (adjusted to *include* all points, and to fit needs of splitter) is created.

Recursively, starting with the root node, these steps are taken: The list of points from the current node $p \in \mathcal{C}$ is used to calculate the *node points information*. Then, the points are distributed into c[7] lists according which child node they belong. At this point, the algorithm requires memory for two times the full list if points. This is necessary because the algorithm that calculates *node points information* for the *KdTree-Median* requires a list of points. It also simplifies things because the numbers of points are known.

The full points list is now no longer needed and memory can be released. Using the *node points information*, cuboids for the child nodes are computed. Then the child node points lists are used to recursively execute the same process in each child node. The recursion stops when the number of points is smaller than the *leaf capacity*, and the *leafs* store their points in an internal buffer. So at the end, each point is exactly once in memory, a leaf's buffer.

The points used as input for the whole algorithm don't come in any specific order. Their required final order (for extraction of node points as segment) can be known only after the full tree, with numbers of points in each node, is determined. So the second phase is to *move out* the points from the leaves' buffers into a global list of points.

A depth-first descent into the tree is done, and for each leaf its points are appended to the global list. During traversal, each node stores the current length of the global list: It will become the offset in the list that contains the points for that node. The recursive descent is also used to sum up the numbers of points for non-leaf nodes.

Downsampled points

Adding downsampled points to the tree structure is easier because the nodes of the tree and their cuboids have already been determined.

Unlike with the *cubes structure*, the downsampling algorithm is applied to the entire set of points. The leaves likely have not enough points to allow for good quality downsampling. Then each point in the resulting set P' is added into an internal buffer (for that downsampling level) of a leaf.

[6] This is especially true when occlusion culling is added.

[7] number of children per node

Even though the downsampled point set always contains fewer points than the full points set, it is possible that within a given cuboid there will be more points: Uniform downsampling will produce different points than there were in the input, so this is unpredictable on a local level. As a consequence it occurs sometimes that points don't fit into the capacity of the leave's internal buffer. In that case the buffer is increased.

The algorithm must not delete points because the total number of downsampled points in the structure must be predictable. (explained further)

After this the points are *moved out* the same way as for the original point set, so as to create the ordered list.

3.8. Piecewise tree structure

The entire algorithm described before aims at creating the tree structure *in memory*, and so the maximal model size it limited. Also, it required much more memory during the process than the final structure will take (internal/temporary buffers, etc.)

Exporting the *in-memory* tree structure into an HDF file is relatively straightforward. In order to allow for huge models to be "streamed" from the raw point set to an HDF file containing the tree structure, the idea used is to construct several temporary tree structures in memory that contain only a *piece* of the model, and to combine them into a complete tree structure in the HDF file.

Since the goal is to end up with a large *tree* structure, the pieces will be cuboids of leaves on a outer "meta-tree" created from the full point set. In this text, the term *outer tree* denotes this tree, whereas the term *inner tree* the temporary tree structure built from a *piece*.

As described in 3.7, the structure of the *HDF* representation allows the *outer* and *inner* trees to be created by different rules. This is useful because the *inner* trees will subdivide the model into very small regions and aim for high visual output quality, making the *Octree* a good choise.

In contrary, the *outer* tree will have very large leaves, and the number of points in the leaves should be controllable: It should be as small enough so that the *inner* tree structures will still fit in memory, but there should not be too many small pieces. The *KdTree-Half* is a good choice here because it subdivides by two instead of by eight, so there is no explosion in the number of pieces.

The *KdTree-Median* might be a better choice since it equalizes the number of points in its children, but here the entire points set can't be loaded into memory to calculate a median.

3.8.1. Building the outer tree

The outer tree will not store any points in memory, but just the nodes and cuboids information. It cannot be built the same way as described before (normal tree structure, aka *inner* trees), because here temporary lists of points are kept and distributed. An analogue algorithm that does not keep lists in memory would need to read through the entire input file (containing the full point set) at each recursion, and consider only the points in its cuboid. Also an algorithm in which leaves are turned into nodes as new points get added is not possible because no information is kept about the points in the nodes.

Instead, the algorithm begins by builing a tree where every node has c children, down to a predefined depth $d_{\max}$. All nodes have an internal counter initialized at zero. Then the program reads through the file, and for each point, increments the counters in *all* of the nodes down to the leaf in which the point belongs. In case the counter of a leaf gets greater than k[8], the entire algorithm must restart, this time with a larger maximal depth $d_{\max}$.

After this has succeeded, the program traverses the nodes, and if and non-leaf node has less than or exactly k points, it removes its children and turns it into a leaf. The resulting tree now represents outer tree. The sum of the numbers of points in the leaves necessarily corresponds to the total number of input points and so the (theoretical) data offsets for an ordered points list can be calculated. The leaves of this outer tree will be the *pieces* or the piecewise tree structure.

[8] The leaf capacity of the outer tree, aka the maximal number of points in a *piece*.

3.8.2. Building the inner trees

For each *piece*, an inner tree is built the same way as described for the “normal” tree structure; except that only the points that lie inside the cuboid of the piece are taken.

3.8.3. Combining the trees

After the outer tree is built, the program starts by writing its nodes information into the HDF file, by recursively traversing it. When a leaf is reached, it builds the temporary inner tree, and then writes its nodes and points into the file.

The offsets by which a node in the inner tree refers to its child nodes in the `nodes` *HDF* data set must be ajusted by adding the offset in the global nodes list where the inner tree’s nodes list is appended.

For the points data offsets associated to outer tree nodes, the algorith just needs to keep track of the current length in the points list. The points sets from the different inner trees are concatenated one after the other.

3.9. Parallelized piecewise tree creation

Because the inner trees are independent from each other (up to the absolute offsets after adjustment), the same mechanism can be also be used to parallelize the creation of a tree structure, by constructing several inner trees simultaneously. This again increases memory usage, but can more than halven the execution time.

The main problem is that both the number of nodes and the number of points in the inner trees are only known after the tree is built, but they need to be concatenated in the final *HDF* file.

The nodes may be listed in the *HDF* file in any order, as explained in 3.7. So the tasks that create inner trees will just store the inner tree’s nodes list separately in memory. After all inner trees have been created, their nodes lists are adjusted (offsets, etc.) and concatenated into the final list, in a sequential postprocessing stage.

However, the points data needs to be in the correct order. There also cannot be gaps in the data because then the points corresponding to outer tree nodes would no longer be in a contiguous segment. The reason why the length of the inner tree’s point sets is not known in advance is because both they depend on the random and uniform downsampling algorithms (see 2.4).

Because of this random downsampling is implemented in a manner so that the exact number of points to output can be specified. For uniform downsampling, the program resorts to making the uniform downsampling algorithm yield a slightly smaller-than-requested output quantity, and then fills the set up with duplicates of randomly chosen points from itself.

With the number of downsampled points that the inner tree structures will contain being known advance, it is possible to allocate the right segment in the complete *HDF* points set for the results to be produced by the different inner structures.

4. Implementation

The implementation consists of two parts:

Library The *Dynamic Point Cloud Library* (`dypc`) implements all of the data structures and algorithms described here. It provides an interface to load models from *PLY*[1] files or generate example models, to create data structures containing a model, store it in a file, and to dynamically load points out a data structure stored in memory or in a file.

Viewer An executable that renders the point cloud and provides a graphical user interface for travelling through the point cloud, and to create and load different data structures. It uses the `dypc`.

4.1. Library

The shared library is implemented in *C++*, using the *STL* standard template library, and *GLM* for mathematical helper functions. It also uses the C/C++ libraries for *HDF5* and *SQLite*. Because of its use of template metaprogramming, a lot of symbols are generated when compiling. All of the *C++* symbols have *hidden* visibility and are for internal use inside the library only. It provides a small pure *C* interface for usage. This avoids problems with linking different *C++* binaries, minimizes coupling, and allows for the library to be used from other programming languages. It is implemented in a mix of object-oriented, procedural and generic programming similar to the *STL*.

4.1.1. Overview

The following main classes or components compose the library:

model A model is a source of input for points. There are subclasses for reading points from a *PLY* file, or for generating random example models. Points can be read from it via an iterator interface. Additionally, it is possible to read from two different iterators simultaneously, for multi-threading use. For the *PLY* file *model*, this is implemented using multiple file handles. The *model* itself does not store any points in memory (apart internal buffers).

ply_model Reads from a *PLY* file. Limited support for *PLY* (e.g. no ASCII files), optimized for speed.

random_model Abstract base class for model generated randomly. The *seed* of the random number generator acts as "file handle". Used for example models **concentric_spheres_model** and **torus_model**.

structure Abstract base class for a data structure. Object of this class contain the data structure in memory and provide an interface for *loaders* to access it. Remain linked by reference to its *model*. The subclass constructors implement the creation of the data structure from the model.

mipmap_structure Abstract base class for a data structure that uses *LOD regions*, aka *mipmaps*. Provides access to downsampling algorithms and functions to determine downsampling ratios/levels.

loader Abstract base class for a **loader**. See below.

cubes_structure Cubes stucture, *weighted points* variant.

cubes_structure_loader Abstract base class for *weighted points* cubes structure loader. Subclasses for *HDF*, *memory*, *SQLite* exist.

[1] File format used to store output from 3D scanners

cubes_mipmap_structure Cubes stucture, *LOD regions* variant.

cubes_mipmap_structure_loader Abstract base class for *LOD regions* cubes structure loader. Subclasses for *HDF* and *memory* exist.

tree_structure, etc Classes for tree structures, see below.

downsampling The downsampling algorithms, and level/ratio functions.

4.1.2. Loader

A loader exists for each data structure, and possibly both for reading from a file and from memory (aka from the *structure* object). It implements the *extraction algorithm*, which generates the filtered point set P' from its source. The *request* to the loader consists the current camera position, orientation, view frustum. Movement velocity is also included but not used by and loader. In addition, the capacity C, and loader-specific parameters such as the *downsampling setting* s, *point-to-cuboid* distance mode are given as input.

It writes the outputted points into a provided buffer. In the *Viewer* implementation, this is virtual memory mapped directly to an OpenGL GPU buffer.

The *direct_model_loader* extracts points directly from a **model** without any filtering or data structure.

4.1.3. Tree structure

Tree structures are implemented by the **tree_structure** template class.[2] It is parametrized by the *splitter* class, which is a subclass of **tree_structure_splitter** with only static members. The *splitter*'s `number_of_node_children` const static member determines how many children nodes shall have. This is implemented using metaprogramming to avoid additional heap allocations for a runtime-determined number of children.

Its *point container* can either be the *STL* `vector` or `deque`. Reading and writing to `vector` is faster, but it needs to allocate large contiguous memory segments which may fail. `deque` uses an internal chunked structure.

The tree structure node template class **tree_structure_node** works in conjunction with **tree_structure**. It also provides access functions to loaders. **tree_structure** can load (and generate) and unload points for the different downsampling levels, and create the structure only for a given cuboid.

This is used for its subclass **tree_structure_piecewise**, which essentially implements outer tree creation on piecewise tree structures. The underlying **tree_structure** will be made to contain one inner tree structure. **tree_structure_piecewise** can also spawn a new **tree_structure** that contains one piece. This is used for the parallelized implementation.

There are two loader variants: **tree_structure_simple_loader** and **tree_structure_ordered_loader**. The latter implements the improved extraction algorithm by which the cuboids closer to the camera are loader first.

The *tree loader sources* **tree_structure_memory_source** and **tree_structure_hdf_source** provide a polymorphic interface for loading from either an in-memory structure or *HDF* file.

Exporting the tree structure to *HDF* is implemented in **tree_structure_piecewise_hdf_write** and the parallelized version **tree_structure_piecewise_hdf_write_parallel**. The **tree_structure_hdf_file** class provides the basic structure for reading and writing to *HDF*. There is an optimization for copying segments of points directly out of a `vector`.

4.1.4. Additional parts

structure_loader_factory Functions for creating loaders, writing to files, etc. Essentially map parameters to template parameters.

interface The *C* interface to the library. Only these functions are exported.

[2] Many other classes used are template classes as well.

geometry Implementation of geometry algorithms and objects such as cuboid, frustum, plane.

sqlite An *SQLite* wrapper.

point, weighted_point POD class types for POD *points*. Meant be copies directly across *HDF*, memory and *GPU* video memory.

progress Helper functions for displaying progress bars. Interface provides callbacks for GUI.

utils, debug, enums Some helper functions.

4.2. Viewer

The *Viewer* is also implemented in *C++*. It uses the *wxWidgets* toolkit to provide a graphical user interface. The main reason for choosing *wxWidgets* was to have a *C++* based, cross-platform GUI with support for including *OpenGL* render context in a window. *OpenGL* is used for the rendering, and the *GLM* library is also used. The *Viewer* links with the *dypc library* and provided a graphical user interface to all of its functionality. The forms are created using *wxFormBuilder*, which generates `.cpp / .hpp` files for creating the form automatically.

4.2.1. Streaming mechanism

The *Viewer* attempts to provide the user with a seamless experience in that they can move through the virtual point cloud, while the program dynamically loads visible point sets P'. The application uses two threads: The main threads operated the user interface and OpenGL renderer, while the secondary thread runs the *updater* which periodically sends request to the *loader* in the *dypc library*, and passes the new point set to the renderer.

To make this last step efficient, a *buffer swap* mechanism is used: Two OpenGL point buffers are allocated in video memory with the point capacity C. The *renderer buffer* is used by OpenGL for the rendering, while the *loader buffer* is being filled by the loader. The use of virtual memory mapping and the *POD* **point** type makes it possible to directly copy raw data from the loader to the GPU. The **point** is composed in such a way that *OpenGL* can directly extract the coordinate and color data from the raw data. In some cases, it is possible that point data is stored in the *HDF* files in exactly the same binary format, and can be copied directly to the buffer.

The *buffer swap* occurs when the updater has finished filling the *loader buffer*. It then notifies the render that new data is available via an atomic flag, and locks. Once available, the *renderer* swaps the two buffers: The *loader buffer* becomes the new *renderer buffer* and vice versa. So no copy takes place, and there is no interference between *renderer* and *updater*. Several concurrency facilities such as mutexes and atomic types are used in the implementation.

4.2.2. Renderer

The *renderer* is responsible both for making *OpenGL* render the data in the *point buffers*, and for allowing the user to move through the virtual space.

It is implemented in *OpenGL 4*, so `glDrawArrays` is used to render data and the projection math is implemented using *GLSL* vertex and fragment shaders. There also implement a visual *fog* effect, and the *point shadow* occlusion culling.

5. Results

6. Conclusion

A. Geometric algorithms

A.1. Definitions

The geometric objects used in the algorithms are defined in the following manner:

cuboid Axis-aligned cuboid $\mathcal{C}$ is defined by *origin* point x, y, z, and *extremity* point x', y', z', such that $x' > x$, $y' > y$ and $z' > z$. The coordinate intervals are open on the extremity side, i.e. a point $\overrightarrow{p}$ is considered to be inside the cuboid if and only if $p_x \in [x, x'[$, $p_y \in [y, y'[$ and $p_z \in [z, z'[$. See section 3.4 for further explanation.

plane An oriented two-dimensional plane P in three-dimensional space is defined using four real values a, b, c, d. $\overrightarrow{n} = (a, b, c)$ is a normal vector of the plane, while d is the distance from the origin to the plane, multiplied by $|\overrightarrow{n}|$.

In its *normalized* form, a, b, c, d are defined such that $|\overrightarrow{n}| = 1$. Then d is simply the distance from the origin to the plane, and each plane there is exactly one normalized form.

The planes are oriented, that is an (oriented) distance from a point $\overrightarrow{p}$ to the plane is positive when $\overrightarrow{n} \cdot \overrightarrow{p} \geq 0$, and negative otherwise.

frustum A frustum is defined using the 6 planes that delimit it: the near, far, left, right, top and bottom planes. Supposing that the normal vectors of the planes are all pointing inside the frustum, a point is considered to be inside the frustum if and only if its oriented distance to each one of the planes of positive.

This representation of a frustum not unique, and is only valid when the 6 planes are arranged so as to delimit a frustum. A frustum can be defined using a 4x4 projection matrix, and the following algorithm describes how to get the 6 planes out of it.

A.2. Projection matrix to view frustum

Let $\boldsymbol{M}$ be a projection matrix, and $\boldsymbol{M}[i, j]$ be the value in $\boldsymbol{M}$ at column i and row j, counting from 0. As described in [1], the 6 planes $P = (a, b, c, d)$ can be derived as follows:

$$\begin{aligned}
P_{\text{near}} &= (\boldsymbol{M}[0,3] + \boldsymbol{M}[0,2], \quad \boldsymbol{M}[1,3] + \boldsymbol{M}[1,2], \quad \boldsymbol{M}[2,3] + \boldsymbol{M}[2,2], \quad \boldsymbol{M}[3,3] + \boldsymbol{M}[3,2]) \\
P_{\text{far}} &= (\boldsymbol{M}[0,3] - \boldsymbol{M}[0,2], \quad \boldsymbol{M}[1,3] - \boldsymbol{M}[1,2], \quad \boldsymbol{M}[2,3] - \boldsymbol{M}[2,2], \quad \boldsymbol{M}[3,3] - \boldsymbol{M}[3,2]) \\
P_{\text{left}} &= (\boldsymbol{M}[0,3] + \boldsymbol{M}[0,0], \quad \boldsymbol{M}[1,3] + \boldsymbol{M}[1,0], \quad \boldsymbol{M}[2,3] + \boldsymbol{M}[2,0], \quad \boldsymbol{M}[3,3] + \boldsymbol{M}[3,0]) \\
P_{\text{right}} &= (\boldsymbol{M}[0,3] - \boldsymbol{M}[0,0], \quad \boldsymbol{M}[1,3] - \boldsymbol{M}[1,0], \quad \boldsymbol{M}[2,3] - \boldsymbol{M}[2,0], \quad \boldsymbol{M}[3,3] - \boldsymbol{M}[3,0]) \\
P_{\text{bottom}} &= (\boldsymbol{M}[0,1] + \boldsymbol{M}[0,1], \quad \boldsymbol{M}[1,3] + \boldsymbol{M}[1,1], \quad \boldsymbol{M}[2,3] + \boldsymbol{M}[2,1], \quad \boldsymbol{M}[3,3] + \boldsymbol{M}[3,1]) \\
P_{\text{top}} &= (\boldsymbol{M}[0,1] - \boldsymbol{M}[0,1], \quad \boldsymbol{M}[1,3] - \boldsymbol{M}[1,1], \quad \boldsymbol{M}[2,3] - \boldsymbol{M}[2,1], \quad \boldsymbol{M}[3,3] - \boldsymbol{M}[3,1])
\end{aligned}$$

A.3. Maximal point-to-cuboid distance

The maximal distance from a point $\overrightarrow{p}$ to a cuboid $\mathcal{C}$ is always the distance to one of the 8 corners of $\mathcal{C}$ (also if $\overrightarrow{p}$ is inside $\mathcal{C}$). The algorithm simply tests the distances from $\overrightarrow{p}$ to each one of the corners and returns the maximal one. As a small optimization, the algorithm can compare the squares of the distances instead, so as to do only one square root computation.

A.4. Minimal point-to-cuboid distance

The minimal distane from a point to a $\overrightarrow{p}$ to a cuboid $\mathcal{C}$ can either be the distance to one of the corners of $\mathcal{C}$, or the distance to one of its edges, or one of its sides. If $\overrightarrow{p}$ is inside $\mathcal{C}$, the distance is always 0. However there is no need to explicitly distinguish all those cases:

Let $\overrightarrow{d} = (\Delta x, \Delta y, \Delta z)$ be the vector from $\overrightarrow{p}$ (point) to the point on $\mathcal{C}$ such that $d = |\overrightarrow{d}|$ is minimal. If $p_x \in [x, x']$, then $\overrightarrow{d}$ will point straight to $\mathcal{C}$ in the x-direction, so $\Delta x = 0$. If $p_x < x$, it needs to turn towards $\mathcal{C}$, so $\Delta x = p_x - x$. Same for when $p_x > x'$, then $\Delta x = -(x' - p_x)$.

The coordinates Δy and Δz are determined the same way. Finally $d = \sqrt{\Delta x^2 + \Delta y^2 + \Delta z^2}$ is calculated. This implies that d will be set to 0 when $\overrightarrow{p}$ is inside $\mathcal{C}$. A small optimization might be to avoid calculating the square and square root when none or only one coordinate is non-zero.

A.5. Frustum-cuboid intersection

Bibliography

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