Dynamic loading of 3D models

PROJ-H-402 Project Report

Tim Lenertz, INFO MA1, ULB

February 24, 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 the distance from the camera.

A downsampling algorithm with ratio $0 \leq r \leq 1$ applied on a set of points P yields a smaller set of points P' with $|P'| \approx r \times |P|$. The set P' is not necessarily a subset of P, and it may be possible to get visually better results by introducing points in different locations than in P.

To make the downsampling ratio dependent on the distance to the camera, the full point set P can be split into n subsets $P_i = \{p | p \in P \text{ and } d_i \leq d(p) < d_{i+1}\}$

2.4.1 Random downsampling

2.4.2 Uniform downsampling

2.5 Occlusion culling

3 Data structures

Bibliography

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