Project Computer Graphics: 3D space partitioning viewer in OpenGL

Michiel Vlaminck

1 Introduction

Space partitioning is the process of dividing the 3D space, often Euclidean, in non-overlapping regions. It is often organized in the form of a hierarchical tree because of its $O(\log n)$ search complexity. In this project we will study three different tree data structures: a k-d tree, an octree and a binary space partitioning (BSP) tree. In the field of computer graphics these hierarchical trees are widely exploited to quickly perform certain kinds of geometrical queries. One of the main usages is ray tracing, aiming at finding all intersecting volumes of a casted ray. An extension of the ray traversal problem is frustum culling, the process of eliminating the polygons that are out of the camera's viewing frustum also referred to as *hidden surface determination*. Finally, the space partitioned space can also be used to perform collision detection.

2 Preliminaries

As mentioned in the introduction, we will study three different ways of partitioning a 3D space: an octree, a k-d tree and a binary space partitioning tree. In the following, each of them will be briefly introduced, but you can read more about them online.

2.1 Octree

An octree is an axis aligned tree constructed by cutting the space using regularly spaced planes into cubes. It is the 3D equivalent of a quad-tree. The construction starts with a bounding box (cube) of the entire scene which is recursively subdivided into 8 smaller sub-cubes - also referred to as voxels - of the same size. The subdivision is only done in case the voxel is not empty, in other words when it is *occupied*. The process is repeated untill the desired leaf size is obtained. In figure 1, the left image shows the concept of the octree and the right image shows an example partitioning of a 3D map.

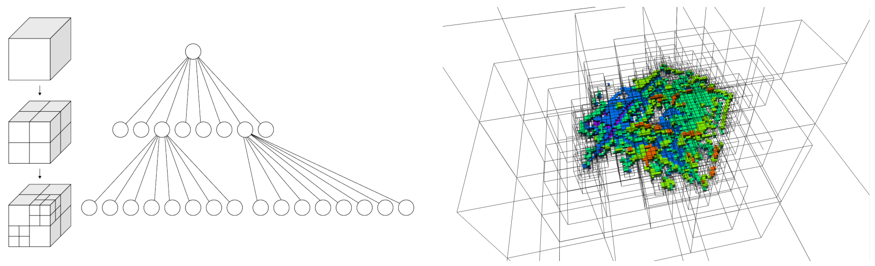

Figure 1: The concept of the octree data structure (left). The root voxel represents a bounding box of the entire 3D model or 3D scene. Subsequently, at each lower level, the *occupied* voxels are subdivided in eight smaller sub cubes of the same size. The same procedure is repeated until the desired leaf size is obtained. The right image shows an example partitioning of a 3D map.

2.2 K-d tree

A k-dimensional tree, also known as a k-d tree, represents a subdivision hierarchy that is generated by splitting a volume along one axis at a time, and changing the axis in a cyclic fashion at each subdivision step. A three-dimensional volume is commonly split first along the x-axis using a plane parallel to the yz-plane, then along the y-axis using a splitting plane parallel to the xz-plane, and then along the z-axis using a splitting plane parallel to the xy-plane. The process continues in the next step by again splitting along the x-axis. In our discussion we will assume that the splitting planes are chosen in the x-y-z order. A k-d tree is a special case of a binary space partitioning (BSP) tree where splitting planes can have arbitrary normal directions. Figure 2 shows a 2-D version of the k-d tree. Often times, the splitting is done based on the median value of the considered splitting dimension, in order to keep the tree more or less balanced.

2.3 BSP-tree

A binary spaced partitioning (BSP) tree is constructed using arbitrarily aligned planes that cut space into convex regions. As such, it is the generalization of the k-d tree which splits the volume parallel to the axis. Often, the splitting planes are based on the principal component analysis of the points or on Lloyd's relaxation. This latter is an algorithm invented by Lloyd for finding evenly spaced sets of points in subsets of Euclidean spaces. The selected partitions are well-shaped and represent uniformly sized convex cells. In figure 3 the idea of the bsp-tree is depicted and in figure 4, an example partitioning of a 3D model is given.

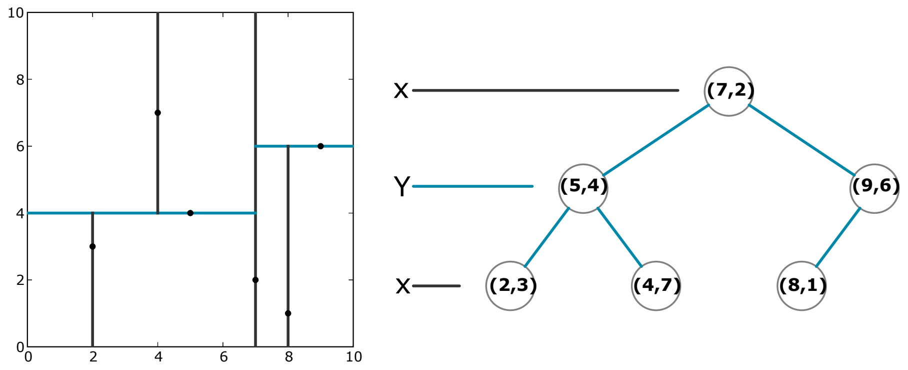

Figure 2: The concept of the k-d tree data structure (here in 2D). It is generated by splitting a volume along an alternating axis and one at a time. For example in 2D, first a subdivision is made based on the x-coordinate, then based on the y-coordinate and then the process restarts with the x-coordinate. Mostly, the point with the median value for the respective coordinate is chosen as parent.

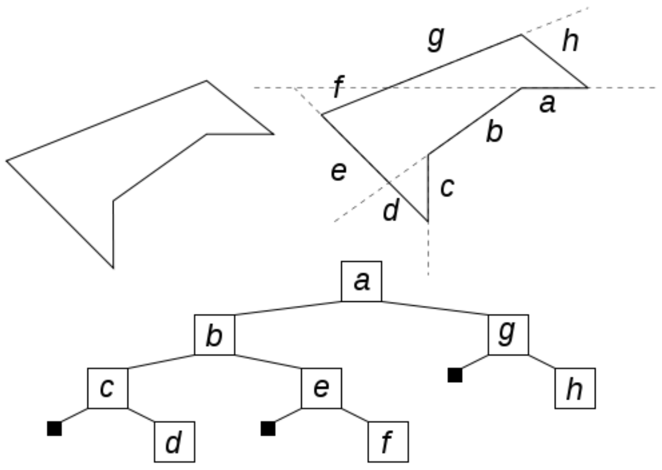

Figure 3: The concept of the binary space partitioning algorithm, a method for recursively subdividing a space into convex sets by hyperplanes, and its tree representation. On each level, each partition is divided by a a splitting line (or plane in 3D). All the lines 'in front' of the splitting line are put on one side of the parent node in the tree, while all the other lines are put on the other side of the parent node in the tree.

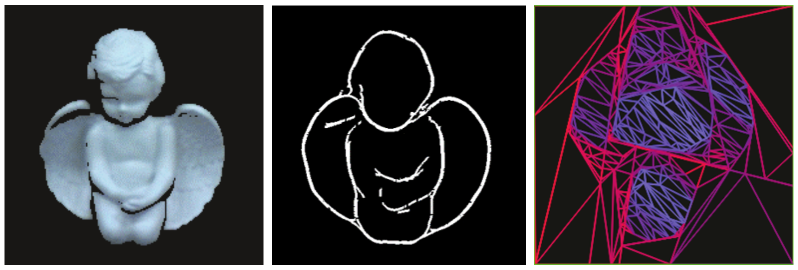

Figure 4: Example partitioning of a 3D model based on a binary space partitioning.

3 Tasks

The project consists of three main tasks. First, three different ways of 3D space partitioning have to be implemented and investigated. Second, a 3D viewer in OpenGL has to be implemented in order to visualize the algorithms and their results. Finally, a concise report should be written with the main outcomes, the answers on the questions and a thorough conclusion.

3.1 Investigation of 3D space partitioning

3.1.1 Reading the input data

The input for the project are 3D point clouds acquired by recent lidar scanners, such as the Velodyne HDL as used by Google on their autonomous car. A figure of such a 3D point cloud is depicted in figure 5. The data is provided in the *PCD* (Point cloud data) file format as proposed by the point cloud library (PCL). Information on this file format and the software to read and write these files can be found on `http://pointclouds.org/`.

3.1.2 Implementing space partitioning

Each of the three ways of partitioning the 3D space has to be implemented. To this end, existing implementations can be used, but they should be entirely understood. All implementations and algorithms should be well-organized in one project. Many questions will pop up regarding design decisions. You are encouraged to implement and investigate different possibilities for each of the occurring problems.

3.2 Building the viewer

The second task consists of building one viewer in 3D using OpenGL that is able to visualize each of the three space partitioning algorithms. It should be possible to dynamically change the view from one partitioning to another. It is up to you if you compute the partitioning of the data beforehand in an initialisation

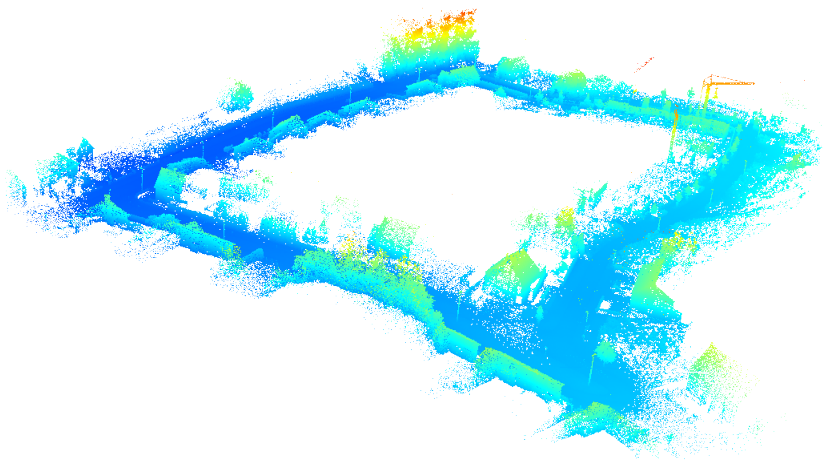

Figure 5: A 3D point cloud acquired by aligning consecutive lidar scans with each other.

phase or of you compute it when the user asks for it. In the case of the octree, the viewer should consist of cubes that have to be rendered. Thus, if an octree cell is 'occupied', it should be represented by a cube with the length of each size equal to the resolution of the leaf size. Figure 7 shows the octree for several leaf sizes. In the case of a k-d tree, the cubes will be replaced by cuboids, as the splitting plane on each dimension is not necessarily drawn in the 'middle'. Finally, in the case of the binary space partitioning tree, the building blocks can not be represented any more by a cube or cuboid. You are free to choose how you will visualize the partitions created by the binary space partitioning algorithm(s). One possibility is to color the points based on the BSP cell they are belonging to. Another option is to draw the convex hull of all the points within one BSP cell.

3.2.1 Multiple resolutions

It should be possible to change the resolution of the space partitioning dynamically. This can be done by means of a UI element, cfr Figure 6, but it is also allowed to make it happen through simple keyboard inputs such as the '+' or '-' keys. Be aware that his function implies that you should implement the trees as hierarchical data structures in order for it to be easy to select another resolution. It would be a bad thing if the tree has to be rebuilt every time the user changes the resolution.

3.2.2 Colouring the partitions

The partitioning should be clear from a visual point of view. This implies that you should use colors to make things more clear. For example, in the case of the

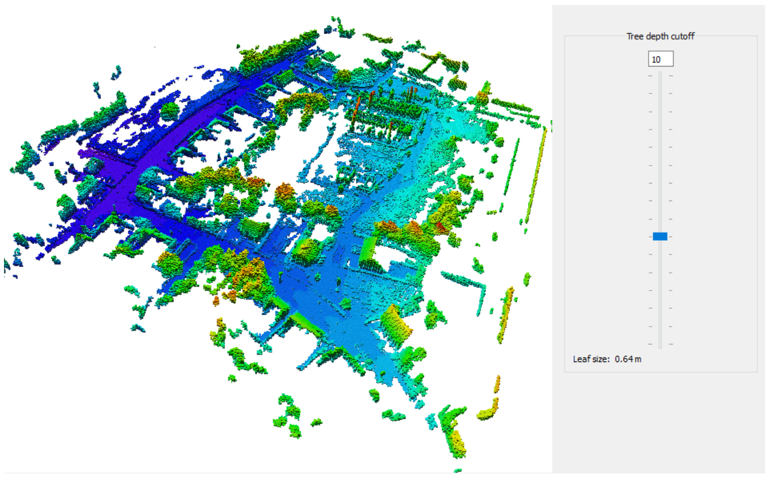

Figure 6: UI-element that can be used to adapt the resolution, i.e. the size of the leaf voxel.

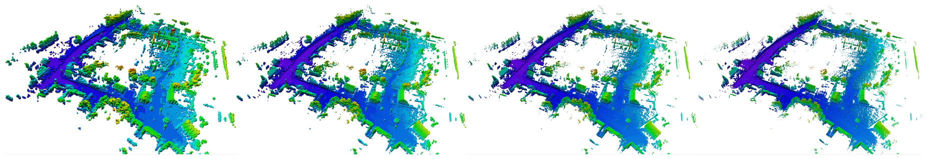

Figure 7: Picture depicting the same map using different leaf resolutions. From left to right, the voxel size is respectively 0.64, 0.32, 0.16 and 0.08 meter corresponding with resolution level 10 to 13. The map sizes are respectively 0.2, 0.8, 3.0, and 9.7 MB.

octree, voxels could be coloured by means of a 'height ramp', i.e. their colour should be according to their heigh which will most probably be denoted by the scalar value of the 'z'-coordinate. The colour can vary from purple/blue for the lower values to orange/red for the higher values.

3.2.3 Wireframe

An additional task consists of drawing a wireframe showing the contours of the cubes, cuboids or polygons that are occupied. In the right image of Figure 1 an example of such a wireframe for the octree case is depicted.

3.2.4 Fly mode [optional]

A last, optional task is to implement *flying* behaviour in the 3D world by using user input. This can be done by using the arrows on the keyboard, but can also be implemented by pressing the left mouse button and moving the mouse in the desired direction.

3.3 Writing the report

A report has to be written explaining the choices you made for each of the data structures and algorithms implemented. Recall that for many 'problems', such as choosing the right cutting plane, many solutions exist. You should at least consider each of them, implement one or more and discuss their differences. Besides, at least all of the following topics should be addressed.

- Evaluation of the time complexity to build and update the trees.
- Evaluation of the time complexity to perform ray intersection queries.
- Discussion of the balancing of a k-d tree.
- Memory efficiency for each of the trees.

And an answer on at least each of the following questions should be given.

- What are the main differences between BSP-trees, octrees and k-d trees?
- Why do BSP-trees take longer to build than octrees?
- How did you visualize the binary space partitioning?
- What are the benefits of an octree over a BSP-tree?
- What type of tree would you use in *dynamic* scenes? Why?
- What type of tree would you use to render urban scenes? Why?
- Which kind of space partitioning is best suited to perform ray casting?

4 Tips and tricks

- Describe the implemented algorithms and their evaluation thoroughly, but keep the report concise. Pseudo code is allowed and in certain situations advisable but keep it as short as possible. Do not list complete classes.
- Understanding the algorithms is important. You are allowed (and encouraged) to use existing implementations you can find on the internet, but you should know exactly how they work.
- Do not hesitate to contact your supervisor if something is unclear or if you have further questions.
- Remember that Google is your best friend.

5 Material provided

- A simple implementation of an octree can be found at `https://github.com/brandonpelfrey/SimpleOctree`.
- Example point clouds of urban and indoor scenes.