3D Geometry Processing
Exercise 3

Fall 2013

Hand in: 24.10.2013, 16:00

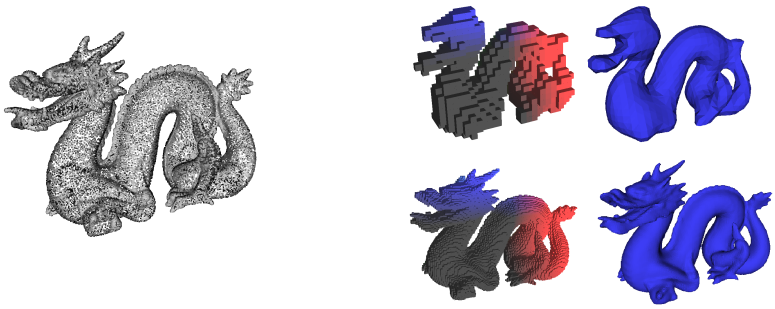

Figure 1: A pointcloud with per point normals (left, shaded) is reconstructed to a surface using an octree of depth 6 and an octree of depth 8.

Surface Reconstruction

In this assignment you will implement algorithms to produce surfaces from pointclouds. This is the first processing step after a pointcloud is obtained e.g. from scanning.

1 Dual Marching Cubes

Your first task is to implement the marching cubes / dual marching cubes algorithm to extract the triangle mesh of an isosurface from a function sampled on the vertex positions of a hashtable based octree. You can test your marching cubes implementation on the precomputed signed distance function of a sphere, which is provided in the basecode. Sample results on this signed distance function are shown in Figure 2.

1. Download the basecode_exercise3.zip package provided on the Ilias exercise page, and copy the classes `Assignment3.java`, `MCTable.java`, `MarchingCubes.java` and the `MarchableCube.java` interface into a new assignment3 package in your Eclipse project.

2. In order to treat primary and dual cube marching in a unified way, adapt the `HashCell` and `HashVertex` class to implement the `MarchableCube` interface. Through this interface a HashCell represents a primary cube with the 8 vertices being its corner elements.

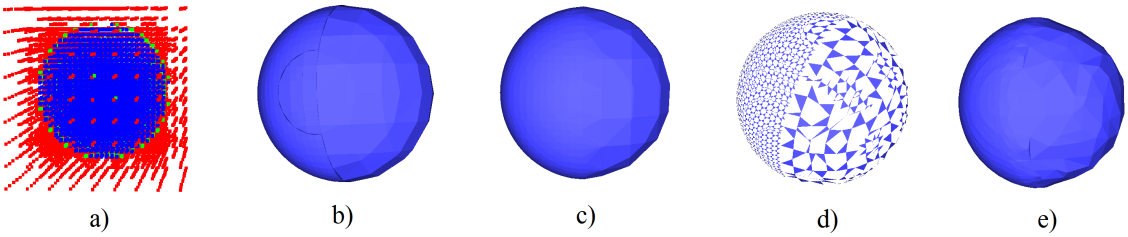

Figure 2: a) Visualization of per HashVertex function values (red; > 0, blue < 0) b) Result of primary cube marching on the octree (producing cracks). c) Result of dual cube marching. d) One iteration of smoothing on the half-edge structure obtained from c) with incorrectly captured adjacencies. e) Correct (albeit ugly) result of one smoothing step on c).

A HashVertex represents a *dual* cube, with the eight neighboring HashCells, or their centers, being the vertex's corner elements.

3. Implement the primary marching cubes algorithm, in the class `MarchingCubes` a skeleton is given. Test your algorithm by marching the tree and per tree-vertex function values provided in the `marchingCubesDemo()` method from the class `Assignment3`.

   - *Details:* Iterate over all hashtree leaf-cells and for each cell, generate triangles according to the marching cubes table provided in `MCTable.java`. To compute a position on one of the cube edges, interpolate the start and end position of the edge according to the associated function values. If a is the value associated to pos_a, and the same for b, use

   $$pos = (1 - \frac{a}{a-b})pos_a + \frac{a}{a-b}pos_b$$

   Don't bother to reuse computed vertex positions, for now compute three new positions for every triangle you create. Store the resulting triangles in a wireframe mesh.

   (**2** Points)

4. Implement the dual marching cubes algorithm and test it on the same data as the primary marching cubes algorithm.

   - *Details:* Given an array of per vertex values, you first have to compute per cell values. Logically these values are associated to the cell centers, therefore they should be computed by averaging the values associated to the vertices of the cell. Iterate over all the vertices, but omit vertices that lie on the tree boundary. You can process each dual cube (i.e. vertex) in the same way as you processed the primary cube.

   (**2** Points)

5. Refine your implementation.

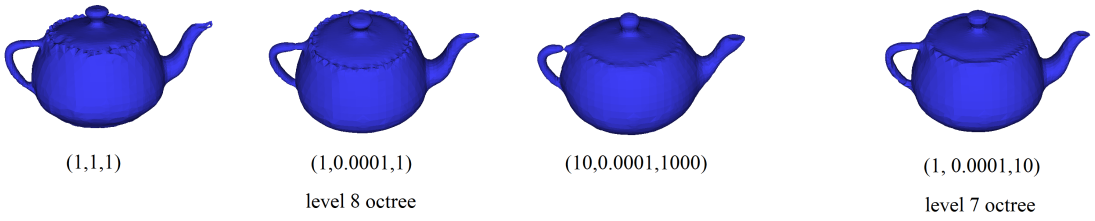

Figure 3: SSD reconstruction results with varying λ parameters and octree depths. The value tuples above describe the lambdas used. The surface was reconstructed on octrees that where refined twice and cover a volum 1.1 times as large as a tight bounding box of the teapot (`tree = new HashOctree(.,.,1,1.1f); tree.refineTree(2);`).

(a) Adapt your implementation, such that every vertex in the resulting mesh is computed once only; i.e. adjacent triangles should share vertices.

- *Details:* The marching cubes algorithms produce at most one vertex position per cube-edge. So before computing a new position on a cube-edge, check if there already is a vertex associated to this edge in your wireframe mesh. To be able to do so, you have to keep track of all $(edge, index)$ pairs, where $index$ is the index of the computed position in your wireframe mesh. You can keep track of the $(edge, index)$ pairs using a Hashmap<Point2i, Integer>; you might also use the method `key(...)` in the class MarchingCubes.

(b) During dual domain cube marching, triangles can be created which do not have three distinct vertices. These degenerate triangles will produce index tuples of the type [1,4,1] where two or more indices coincide. Check if the triangle tuples are degenerated before you add them to the wireframe mesh, and discard them if they are degenerated.

(c) To test a) and b), cast a mesh extracted by dual marching to a half-edge structure and smooth the half-edge structure using the code from assignment 1 (just 1 iteration). No holes should appear.

(**2** Points)

2 Smooth Signed Distance Reconstruction

1. Setup:

(a) Copy the remaining base code into your project. Make sure that the python folder from the basecode_exercise3.zip package is located in the same folder as the obj and the shader folder.

(b) Download and install Python and the SciPy/NumPy libraries `http://www.scipy.org/`, for example using the free Anaconda distribution `http://continuum.io/downloads`. Make sure that python is callable from the system's console by entering "'python"' (be it Linux, Windows or Mac). We will use the sparse least

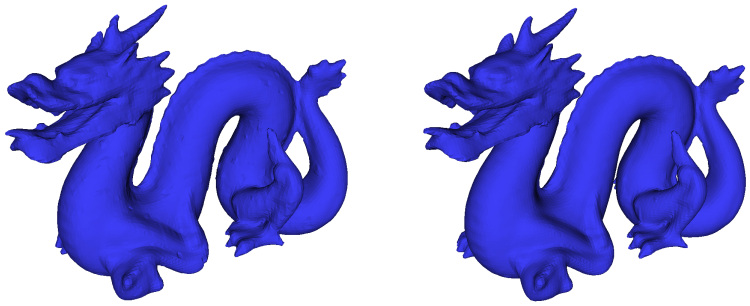

Figure 4: Using the regularization term introduced in the exercise session can introduce artifacts: without refinements (left) the surface can have bumps, with two refinement steps the bumps disappear (right).

squares solver provided in the SciPy package as a black box solver for the arising linear systems.

(c) Test if the Python setup is correct by running `PythonSetupTest.java`. A message "'Congratulations! Python works!"' should appear.

(d) If you want to use your own Hashtree implementation, add the two methods `refineTree(int)` and `refine()` to your implementation, they can be found in the provided octree implementation. Refining the octree in proximity of a point cloud effectively gets rid of artifacts in the SSD variant we implement. See Figure 4

2. Implement the SSD matrices corresponding to the D_0 and the D_1 term as described in the lecture, and the regularization term as described in the exercise session. Use the CSRMatrix class to define your sparse matrices. Construct the appropriate righthand-side of your linear system and solve it using the provided SCIPY.solve method. Use dual domain cube marching to create a polygon mesh from the computed per Vertex values. Hints:

   - Use the tests described in the exercise session to debug the matrices.
   - Play with the parameters λ_i to tune your result. Changing the control parameters of the octree and adding one or two refining ssteps (HashOctree.refine()) also have an impact on the result (See Figure for results on the teapot mesh).

   4 Points

3 Hand-In

Don't forget to:

1. Commit your code via the Ilias exercise page before the deadline.
2. Prepare your code demonstration and reserve a time slot for your demo via the Google Doc shared in the forum.