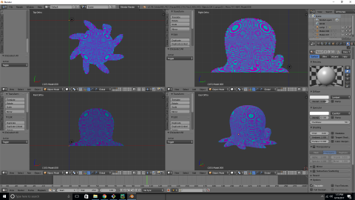
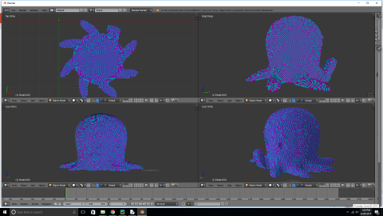
This algorithm first extracts information on vertices and faces from an OBJ file, then reconstructs the 3D model layer by layer and assign material to each voxel, and finally regenerate separate obj files definiting models for each of the materials.

OBJ is a gemoetry definiton file format that represents 3D geometry through the position of each vertex, the UV position of each texture coordinate vertex, vertex normals, and the faces defined as a list of vertices in a counter-clockwise order, texture verrtices, vertex normal to the face. Since both the input model is voxel-based, the input OBJ file contains only six vertex normals, one for each of the face of a cube. In addition, because materials will be assigned later for different parts of the model, the texture coordinate vertices of the input OBJ file could be ignored. Thus, the only information in input OBJ of interest are position of each vertex, and each face as defined by its four corners. An OBJ file usually begins with lines starting with a hash character (#) which are comments on the file that this algorithm ignores. Vertex data begin with "v", faces with "f", texture vertices with "vt", and vertex normals with "vn". Vertex data have the following format: "v x z y," where x, z, and y are coordinates of the vertex. (While OBJ file spec states that coordinates are arranged as x,y,z, after experimenting with Blender, we discovered that the second coordinate in OBJ actually defines the vertex's position on z coordinate in Blender.) Face data is formated as "f v1/vt/vn v2/vt/vn v3/vt/vn v4/vt/vn," where v1, v2, v3, v4 are the four vertices' indicies in the vertex data list (the first vertex in the list begin with index 1), vt the texture vertex but it is usually empty since texture is not definied, and vn is the vertex normal. The vertex and face data extracted from the input OBJ file is then passed to the VoxelModel class to reconstruct the 3D model for assigning different materials later.

Vertices are then organized by their z coordinates so the model can be reconstructed layer by layer. Imagine a VoxelModel object represents a big cuboid made of numerous voxels of the same size as those in the original model that encloses the entire original 3D model. Since each voxel is approximately a cube, voxel size is determined by dividing the difference between the largest and smallest z value by number of layers (which is number of different z coordinates minus one). The width and length of the cuboid is calculated from the difference between the largest and smallest x and y values from the vertex coordinates. Then the algorithm determines where in the cuboid does the originial voxel lies, which voxels are part of the shell of the original model (i.e. it was exposed on the outside), and assign material to each voxel later. To figure out which voxel of the cuboid is empty or solid, the algorithm exames each layer of the model. Each layer can be represented by a 2D grid such that an edge in the grid is a vertical face in the orginal model. From vertex and face data extracted from the original OBJ file, we know where such edges lie (i.e. where the verticle faces exist). For each square in the grid, count how many edges there are to its right (including its own right edge) in the same row. If there are odd edges, it is enclosed in the model, or not otherwise, because whenever we encounter an edge, the content changes from solid to empty or empty to solid. With this method, we can figure out exactly which voxels belong to the original model.

Next, the algorithm assigns materials (or "color") to each voxel in the model based on number of colors specified by the user. The goal is to evenly spread out each color over the entire model so in a later painting process, the user can freely paint with any of the colors s/he specified earlier; because the color voxels are small enough, the color swatch that is made of multiple discrete voxels of the same color will look like a continous stroke. First, the algorithm needs to determine which voxels are on the surface of the model; these visible voxels will be printed with photochromics materials so they can be colored later. The algorithm checks if a voxel has voxels adjacent to all six of its faces; if so, the voxel is inside, or it's outside otherwise. The algorithm then colors the exposed voxels. There are two ways to determine the coloring. The first one is to repeatedly stagger different colors. Imagine the whole cuboid is colored in a checker-like fashion and each voxel in the model will be the same color as the cube in the cuboid that it occupies. For example, if two colors are used, and the first row is 1, 2, 1, 2, then the next row will be 2, 1, 2, 1. The row directly above the first row will be 2, 1, 2, 1. This method works most of the time. However, for objects with a smooth convex surface, sometimes several voxels with the same color will cluster.



The second coloring method is to not only take into account of the color of the adjacent voxels but also that of voxels sharing an edge. To determine the color of a voxel, each color is given a score computed as such: the more a color is used by its neighboring voxels (both voxels sharing a face and those sharing an edge), the higher the score is, with larger weight given to voxel sharing a face. If not all colors receive a score of zero, the voxel is assigned to the color with the lowest score; otherwise, the voxel is assigned to the default color which is the color of the space it occupies as described in method one. If multiple colors have the lowest score, the default color is assigned if it is among those colors; otherwise, the first color with the lowest score is assigned. With this method, clustering happens less often and on a smaller scale. 

Finally, the algorithm generates a separate OBJ file for each of the materials. File material1.obj to materialn.obj where n is the number of colors specified by the user earlier are for the outer shell the model, and file material(n+1).obj is for the inside of the model. These n+1 models combined is the original model. The files are genereated by taking each voxel and writing its eight vertices and six faces to the appropriate file. One downside of this method is that it creates many duplicated vertices and faces. Duplicates could be removed later in Blender.