Technische Universität München
Fakultät für Informatik
Prof. Nassir Navab

Summer Semester 2016
Exercise Sheet 9
PRACTICAL

Medical Augmented Reality

Exercise 1 Load and render a mesh

In this exercise we will load a more complex mesh from a file. The format we are using is the Wavefront OBJ format (see `https://en.wikipedia.org/wiki/Wavefront_.obj_file`). You will write a simple parser for this text based mesh descriptor file. It might be wise to create a `Mesh` class for this. On the course website we will provide a sample (medical) mesh but all popular 3D software can import and export OBJ files.

(a) We will use the `VectorXX` templates from the `Eigen` library. Our meshes will consist of vertices and triangles with each vertex consisting of a 3D (`Vector3f`) location, 2D (`Vector2f`) texture coordinate and a 3D (`Vector3f`) surface normal. Create a `std::vector` for each of the three vertex attributes. Keep in mind that for `Eigen` types in STL containers you have to specify the allocator[1].

```
std::vector<Eigen::Vector3f, Eigen::aligned_allocator<Eigen::Vector3f> >
vertices;
```

(b) The triangles of the mesh are stored as lists of indices corresponding to the three attribute arrays. So create a struct `Index`, give it three `unsigned int vertexIdx`, `texCoordIdx` and `normalIdx` and create another `std::vector` of those.

(c) Now it is time to parse the `*.obj` file. Maybe the most simple way to parse a `txt` file is with `std::ifstream`. Open your `obj` file and read each line separately. Lines starting with `v` describe a vertex position. Add the three following floats to your vertex array. `vt` and `vn` are texture coordinates (2 floats) and normals (3 floats). Add them to their respective arrays as well. You will need to flip ($y := 1 - y$) the y coordinate of the texture coordinates since OpenGL has a flipped texture coordinate system.

(d) Finally parse the faces `f`. Here we need to do a little work since we want to fill our indices array with triangles only, but faces in the `obj` format can have any number of vertices ≥ 3. For this we will re insert the first and the previous `Index` into the `indices` array for every polygon vertex index we read after the third one. See figure 1 for details. The vertices in each face are described by a 3-tuple `A/B/C` where `A` is the index into the vertex array, `B` for the texture coordinates and `C` for the normals. Indices in `obj` files are 1 based so you need to subtract 1 to index your C++ arrays!

(e) after loading the mesh you might want to translate it such that its center of mass is at (0,0,0). For this compute the mean of all vertex locations and subtract it from each of them.

(f) You should have a list of indices that represents triangles where every `Index` consists of a position, texture coordinate and normal index. You also have arrays for the actual values for each of those attributes.

[1]See also `http://eigen.tuxfamily.org/dox-devel/group__TopicStlContainers.html`

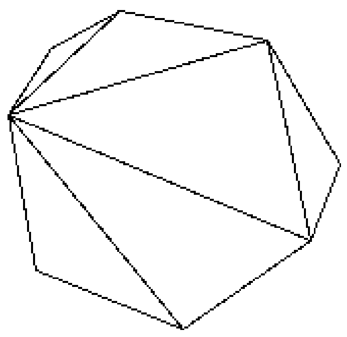
Figure 1: How to triangulate a convex polygon

(g) Now we want to render the mesh. For rendering we will wrap our calls to transfor information to the GPU again in between `glBegin(GL_TRIANGLES)` and `glEnd()`. Drawing is straight forward now: it is basically the same as the cube rendering but we will need the list of indices. Iterate over all indices and index your vertex positions, texture coordinates and normals and call `glVertex3f`, `glTexCoord2f` and `glNormal3f` respectively. Since you already converted the faces to triangles and your indices to start at 0 during loading, this is all you have to do.

(h) Render the mesh onto one of the unused markers.

Exercise 2 Optional: Load and display the texture for the mesh

Optionally you can implement texture loading such that the objects will look nicer!

(a) During `obj` file parsing, when you encounter the line with `mtllib` you can parse that file as well. Basically the only interesting part for the texture is the line with `map_Kd`. It tells you the filename of the (diffuse) texture. Of course you can also parse and set the lighting conditions the sub-parts of the model and everything else the `obj` format provides - if you want.

(b) Load the image using `cv::imread()`. Generate one texture (`GLuint`) using `glGenTextures`. Bind it (`GL_TEXTURE_2D`) and setup the environment (`glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_REPLACE)`). Now we have to set some sampling rules:

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
```

(c) Finally load the data `(GLvoid*)image.data` given as `GL_UNSIGNED_BYTE` in the `GL_BGR_EXT` format into the texture of format `GL_RGBA` using `glTexImage2D`.

(d) It is a good idea to check `glGetError` if something went wrong.

(e) For rendering, you just need to bind your texture to the stage:

```
glEnable(GL_TEXTURE_2D);
glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);
glBindTexture(GL_TEXTURE_2D, texture);
```

and set the color to white.