Tutorial 7 : Model loading

Until now, we hardcoded our cube directly in the source code. I'm sure you will agree that this was cumbersome and not very handy.

In this tutorial we will learn how to load 3D meshes from files. We will do this just like we did for the textures : we will write a tiny, very limited loader, and I'll give you some pointers to actual libraries that can do this better that us.

To keep this tutorial as simple as possible, we'll use the OBJ file format, which is both very simple and very common. And once again, to keep things simple, we will only deal with OBJ files with 1 UV coordinate and 1 normal per vertex (you don't have to know what a normal is right now).

Loading the OBJ

Our function, located in common/objloader.cpp and declared in common/objloader.hpp, will have the following signature :

```
bool loadOBJ(
    const char * path,
    std::vector < glm::vec3 > & out_vertices,
    std::vector < glm::vec2 > & out_uvs,
    std::vector < glm::vec3 > & out_normals
)
```

We want loadOBJ to read the file "path", write the data in out_vertices/out_uvs/out_normals, and return false if something went wrong. std::vector is the C++ way to declare an array of glm::vec3 which size can be modified at will: it has nothing to do with a mathematical vector. Just an array, really. And finally, the & means that function will be able to modify the std::vectors.

Example OBJ file

An OBJ file looks more or less like this :

```
# Blender3D v249 OBJ File: untitled.blend
# www.blender3d.org
mtllib cube.mtl
v 1.000000 -1.000000 -1.000000
v 1.000000 -1.000000 1.000000
v -1.000000 -1.000000 1.000000
v -1.000000 -1.000000 -1.000000
v 1.000000 1.000000 -1.000000
v 0.999999 1.000000 1.000001
v -1.000000 1.000000 1.000000
v -1.000000 1.000000 -1.000000
vt 0.748573 0.750412
vt 0.749279 0.501284
vt 0.999110 0.501077
vt 0.999455 0.750380
vt 0.250471 0.500702
vt 0.249682 0.749677
vt 0.001085 0.750380
vt 0.001517 0.499994
vt 0.499422 0.500239
vt 0.500149 0.750166
vt 0.748355 0.998230
vt 0.500193 0.998728
vt 0.498993 0.250415
vt 0.748953 0.250920
vn 0.000000 0.000000 -1.000000
vn -1.000000 -0.000000 -0.000000
vn -0.000000 -0.000000 1.000000
vn -0.000001 0.000000 1.000000
vn 1.000000 -0.000000 0.000000
vn 1.000000 0.000000 0.000001
vn 0.000000 1.000000 -0.000000
vn -0.000000 -1.000000 0.000000
usemtl Material_ray.png
s off
f 5/1/1 1/2/1 4/3/1
f 5/1/1 4/3/1 8/4/1
f 3/5/2 7/6/2 8/7/2
f 3/5/2 8/7/2 4/8/2
f 2/9/3 6/10/3 3/5/3
f 6/10/4 7/6/4 3/5/4
f 1/2/5 5/1/5 2/9/5
f 5/1/6 6/10/6 2/9/6
f 5/1/7 8/11/7 6/10/7
f 8/11/7 7/12/7 6/10/7
f 1/2/8 2/9/8 3/13/8
f 1/2/8 3/13/8 4/14/8
```

So :

- `#` is a comment, just like // in C++
- usemtl and mtllib describe the look of the model. We won't use this in this tutorial.
- v is a vertex
- vt is the texture coordinate of one vertex
- vn is the normal of one vertex
- f is a face

v, vt and vn are simple to understand. f is more tricky. So, for f 8/11/7 7/12/7 6/10/7 :

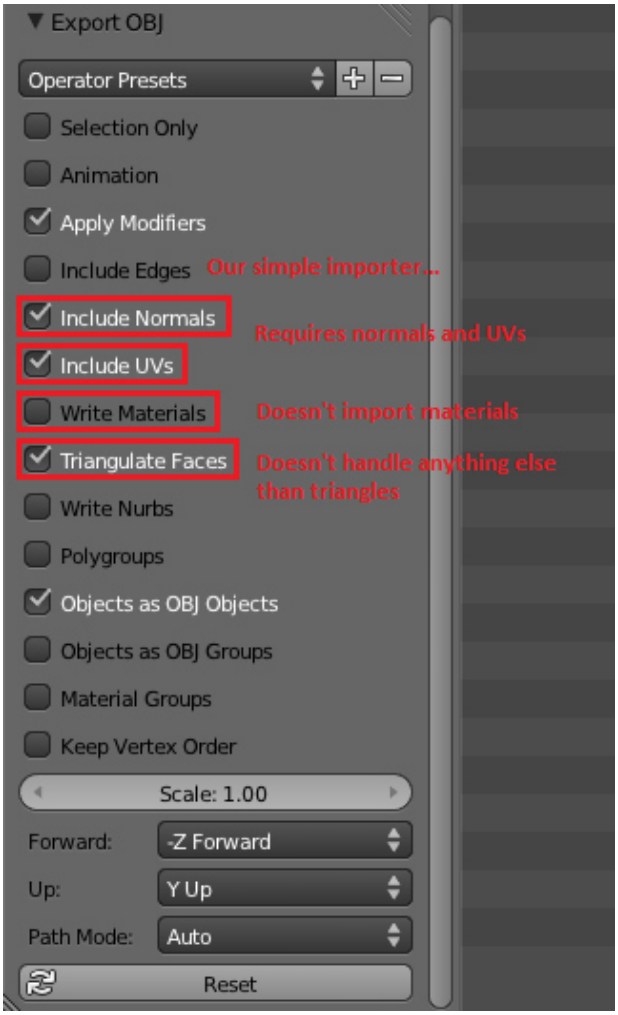
- 8/11/7 describes the first vertex of the triangle
- 7/12/7 describes the second vertex of the triangle
- 6/10/7 describes the third vertex of the triangle (duh)
- For the first vertex, 8 says which vertex to use. So in this case, -1.000000 1.000000 -1.000000 (index start to 1, not to 0 like in C++)
- 11 says which texture coordinate to use. So in this case, 0.748355 0.998230
- 7 says which normal to use. So in this case, 0.000000 1.000000 -0.000000

These numbers are called indices. It's handy because if several vertices share the same position, you just have to write one "v" in the file, and use it several times. This saves memory.

The bad news is that OpenGL can't be told to use one index for the position, another for the texture, and another for the normal. So the approach I took for this tutorial is to make a standard, non-indexed mesh, and deal with indexing later, in Tutorial 9, which will explain how to work around this.

Creating an OBJ file in Blender

Since our toy loader will be severely limited, we have to be extra careful to set the right options when exporting the file. Here's how it should look in Blender :



Reading the file

Ok, down with the actual code. We need some temporary variables in which we will store the contents of the .obj :

```
std::vector< unsigned int > vertexIndices, uvIndices, normalIndices;
std::vector< glm::vec3 > temp_vertices;
std::vector< glm::vec2 > temp_uvs;
std::vector< glm::vec3 > temp_normals;
```

Since Tutorial 5 : A Textured Cube, you know how to open a file :

```
FILE * file = fopen(path, "r");
if( file == NULL ){
    printf("Impossible to open the file !\n");
    return false;
}
```

Let's read this file until the end :

```
while( 1 ){

    char lineHeader[128];
    // read the first word of the line
    int res = fscanf(file, "%s", lineHeader);
    if (res == EOF)
        break; // EOF = End Of File. Quit the loop.

    // else : parse lineHeader
```

(notice that we assume that the first word of a line won't be longer than 128, which is a very silly assumption. But for a toy parser, it's all right)

Let's deal with the vertices first :

```
if ( strcmp( lineHeader, "v" ) == 0 ){
    glm::vec3 vertex;
    fscanf(file, "%f %f %f\n", &vertex.x, &vertex.y, &vertex.z );
    temp_vertices.push_back(vertex);
```

i.e : If the first word of the line is "v", then the rest has to be 3 floats, so create a glm::vec3 out of them, and add it to the vector.

```
}else if ( strcmp( lineHeader, "vt" ) == 0 ){
    glm::vec2 uv;
    fscanf(file, "%f %f\n", &uv.x, &uv.y );
    temp_uvs.push_back(uv);
```

i.e if it's not a "v" but a "vt", then the rest has to be 2 floats, so create a glm::vec2 and add it to the vector.

same thing for the normals :

```
}else if ( strcmp( lineHeader, "vn" ) == 0 ){
    glm::vec3 normal;
    fscanf(file, "%f %f %f\n", &normal.x, &normal.y, &normal.z );
    temp_normals.push_back(normal);
```

And now the "f", which is more difficult :

```
}else if ( strcmp( lineHeader, "f" ) == 0 ){
    std::string vertex1, vertex2, vertex3;
    unsigned int vertexIndex[3], uvIndex[3], normalIndex[3];
    int matches = fscanf(file, "%d/%d/%d %d/%d/%d %d/%d/%d\n",
```

```
&vertexIndex[0], &uvIndex[0], &normalIndex[0], &vertexIndex[1],
&uvIndex[1], &normalIndex[1], &vertexIndex[2], &uvIndex[2],
&normalIndex[2] );
    if (matches != 9){
        printf("File can't be read by our simple parser : ( Try
exporting with other options\n");
        return false;
    }
    vertexIndices.push_back(vertexIndex[0]);
    vertexIndices.push_back(vertexIndex[1]);
    vertexIndices.push_back(vertexIndex[2]);
    uvIndices    .push_back(uvIndex[0]);
    uvIndices    .push_back(uvIndex[1]);
    uvIndices    .push_back(uvIndex[2]);
    normalIndices.push_back(normalIndex[0]);
    normalIndices.push_back(normalIndex[1]);
    normalIndices.push_back(normalIndex[2]);
```

This code is in fact very similar to the previous one, except that there is more data to read.

Processing the data

So what we did there was simply to change the “shape” of the data. We had a string, we now have a set of std::vectors. But it’s not enough, we have to put this into a form that OpenGL likes. Namely, removing the indexes and have plain glm::vec3 instead. This operation is called indexing.

We go through each vertex (each v/vt/vn) of each triangle (each line with a “f”) :

```
    // For each vertex of each triangle
    for( unsigned int i=0; i<vertexIndices.size(); i++ ){
```

the index to the vertex position is vertexIndices[i] :

```
unsigned int vertexIndex = vertexIndices[i];
```

so the position is temp_vertices[vertexIndex-1] (there is a -1 because C++ indexing starts at 0 and OBJ indexing starts at 1, remember ?) :

```
glm::vec3 vertex = temp_vertices[ vertexIndex-1 ];
```

And this makes the position of our new vertex

```
out_vertices.push_back(vertex);
```

The same is applied for UVs and normals, and we're done !

Using the loaded data

Once we've got this, almost nothing changes. Instead of declaring our usual static const GLfloat g_vertex_buffer_data[] = {...}, you declare a std::vector vertices instead (same thing for UVS and normals). You call loadOBJ with the right parameters :

```
// Read our .obj file
std::vector< glm::vec3 > vertices;
std::vector< glm::vec2 > uvs;
std::vector< glm::vec3 > normals; // Won't be used at the moment.
bool res = loadOBJ("cube.obj", vertices, uvs, normals);
```

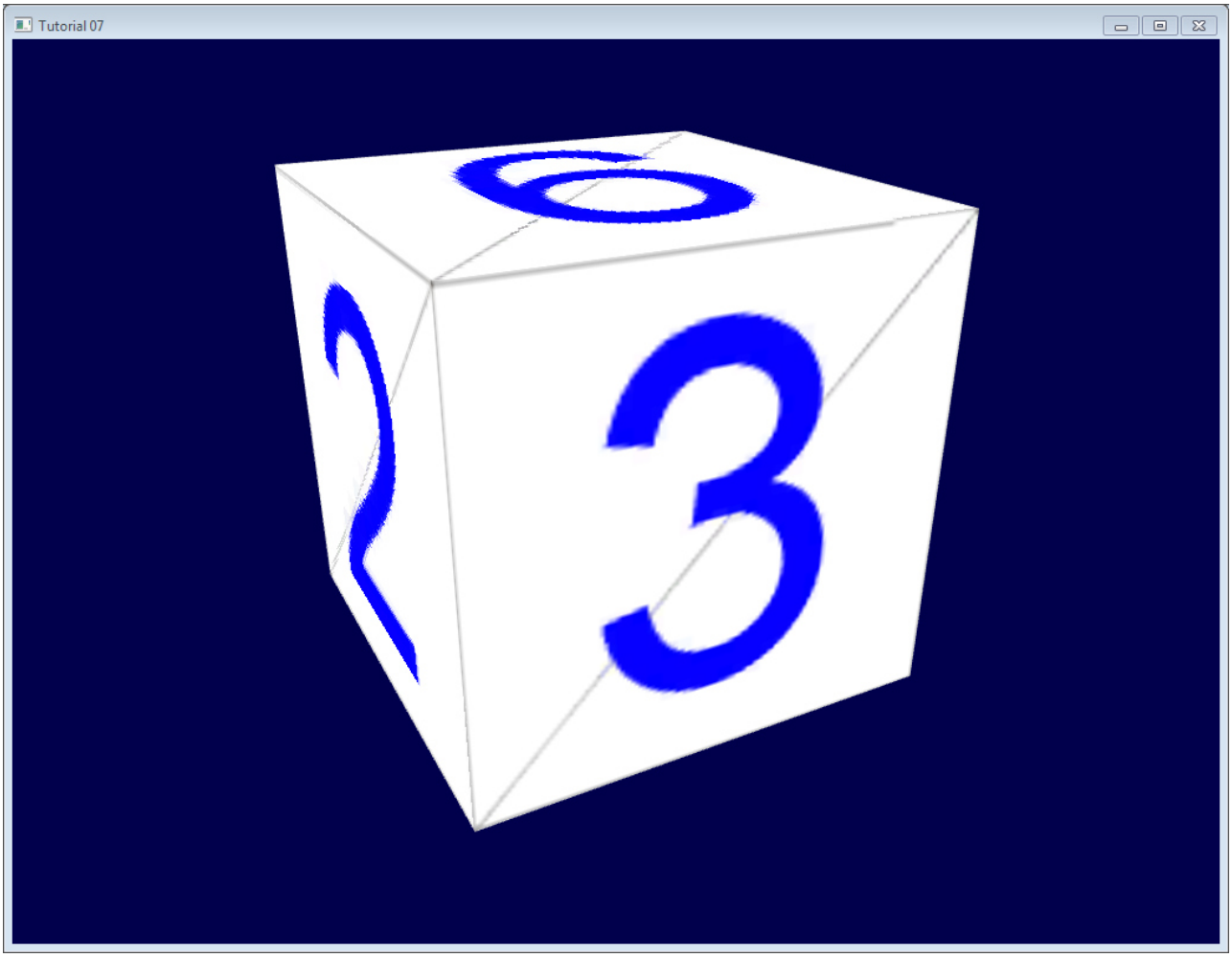
and give your vectors to OpenGL instead of your arrays :

```
glBufferData(GL_ARRAY_BUFFER, vertices.size() * sizeof(glm::vec3),
&vertices[0], GL_STATIC_DRAW);
```

And that's it !

Results

Sorry for the lame texture, I'm NOT a good artist :(Any contribution welcome !



Other formats/loaders

This tiny loader should give you enough to get started, but won't want to use this in real life. Have a look at our Useful Links & Tools page for some tools you can use. Note, however, that you'd better wait for tutorial 9 before *actually *trying to use them.

contact@opengl-tutorial.org

Free tutorials for modern Opengl (3.3 and later) in C/C++