OpenGL Programming/Modern OpenGL Tutorial Load OBJ

Pretty quickly, we'll want to load existing models rather than creating them by hand. Let's import one from Blender.

Apparently there is no easy way to directly load a .blender file, but we can either:

- File > Export in .obj format, and write a OBJ loader
- File > Export in .3ds format, and use lib3ds (http://lib3ds.org/)

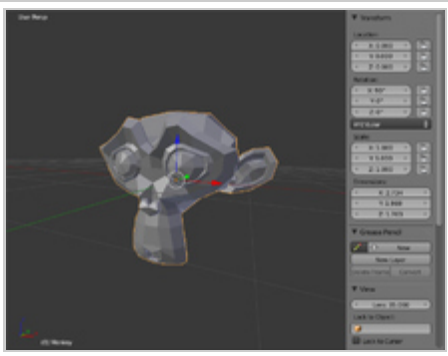
We think that writing the OBJ loader is a good exercise, so we'll start that way.

Contents

- 1 Creating Suzanne
- 2 File format
- 3 Flat-shading - duplicating vertices and normals
  - 3.1 Averaging normals
  - 3.2 Pre-computed normals
- 4 See also

Creating Suzanne

Suzanne is the Blender test model. It has 500 polygons, which makes it a good test for us.



Blender screenshot

To create it, run Blender (we'll use version 2.58), then:

- Remove all elements from the scene (right click on them and press **x**)
- In the top menu, click on Add > Mesh > Monkey
- Type **n** to display the Transform panel and
  - set the location to (0, 0, 0)
  - set the rotation to (90, 0, 0)
- In the top menu, click on File > Export > Wavefront (.obj)
  - to preserve the Blender orientation, carefully set the following options (to switch to "Y-is-up" OpenGL coordinates):
    - Forward: -Z Forward
    - Up: Y Up
  - Tick "Triangulate" so we get triangle faces instead of quad faces

Blender will create two files, suzanne.obj and suzanne.mtl:

- the .obj file contains the mesh : vertices and faces
- the .mtl file contains information on materials (Material Template Library)

For now we'll just load the mesh.

File format

Inspect the .obj file with a text editor. We see that the format is pretty simple:

- structured in lines
- lines starting with # are comments
- o introduces a new object
- v introduces a vertex
- vn introduces a normal
- f introduces a face, using vertex indices, starting at 1

We need to populate several C arrays:

- the vertices
- the elements
- the normals (used for lighting computations)

The format also has other features, but for now we'll leave them aside.

Here's a first, crude implementation that will work for our object.

Our parser is going to be limited (no support for multiple objects, alternative vertex formats, polygons, etc.), but that's enough for our immediate needs.

```
void load_obj(const char* filename, vector<glm::vec4> &vertices, vector<glm::vec3> &normals, vector<GLushort>
{
    ifstream in(filename, ios::in);
    if (!in)
    {
        cerr << "Cannot open " << filename << endl; exit(1);
    }

    string line;
    while (getline(in, line))
    {
        if (line.substr(0,2) == "v ")
        {
            istringstream s(line.substr(2));
            glm::vec4 v; s >> v.x; s >> v.y; s >> v.z; v.w = 1.0f;
            vertices.push_back(v);
        }
        else if (line.substr(0,2) == "f ")
        {
            istringstream s(line.substr(2));
            GLushort a,b,c;
            s >> a; s >> b; s >> c;
            a--; b--; c--;
           elements.push_back(a); elements.push_back(b); elements.push_back(c);
        }
        else if (line[0] == '#')
        {
            /* ignoring this line */
        }
        else
        {
            /* ignoring this line */
        }
    }

    normals.resize(vertices.size(), glm::vec3(0.0, 0.0, 0.0));
    for (int i = 0; i < elements.size(); i+=3)
```

```
    {
        GLushort ia = elements[i];
        GLushort ib = elements[i+1];
        GLushort ic = elements[i+2];
        glm::vec3 normal = glm::normalize(glm::cross(
        glm::vec3(vertices[ib]) - glm::vec3(vertices[ia]),
        glm::vec3(vertices[ic]) - glm::vec3(vertices[ia])));
        normals[ia] = normals[ib] = normals[ic] = normal;
    }
}
```

We used C++ vectors to simplify the memory management. We passed arguments by reference, mostly because the syntax to access pointer to vectors becomes horrid (`(*elements)[i]`)

We can load the .obj file this way:

```
vector<glm::vec4> suzanne_vertices;
vector<glm::vec3> suzanne_normals;
vector<GLushort> suzanne_elements;
[...]
load_obj("suzanne.obj", suzanne_vertices, suzanne_normals, suzanne_elements);
```

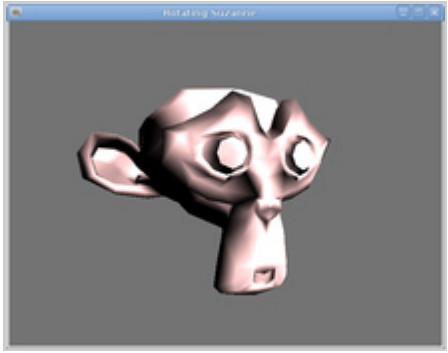
And pass it to OpenGL using:

```
glEnableVertexAttribArray(attribute_v_coord);
// Describe our vertices array to OpenGL (it can't guess its format automatically)
glBindBuffer(GL_ARRAY_BUFFER, vbo_mesh_vertices);
glVertexAttribPointer(
  attribute_v_coord,  // attribute
  4,                  // number of elements per vertex, here (x,y,z,w)
  GL_FLOAT,           // the type of each element
  GL_FALSE,           // take our values as-is
  0,                  // no extra data between each position
  0                   // offset of first element
);

glEnableVertexAttribArray(attribute_v_normal);
glBindBuffer(GL_ARRAY_BUFFER, vbo_mesh_normals);
glVertexAttribPointer(
  attribute_v_normal, // attribute
  3,                  // number of elements per vertex, here (x,y,z)
  GL_FLOAT,           // the type of each element
  GL_FALSE,           // take our values as-is
  0,                  // no extra data between each position
  0                   // offset of first element
);

glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ibo_mesh_elements);
int size;  glGetBufferParameteriv(GL_ELEMENT_ARRAY_BUFFER, GL_BUFFER_SIZE, &size);
glDrawElements(GL_TRIANGLES, size/sizeof(GLushort), GL_UNSIGNED_SHORT, 0);
```

Last, we adjust our view accordingly, with a Y-is-top coordinates system, and the camera facing Suzanne:

```
glm::mat4 view = glm::lookAt(
  glm::vec3(0.0,  2.0, 4.0),   // eye
  glm::vec3(0.0,  0.0, 0.0),   // direction
  glm::vec3(0.0,  1.0, 0.0));  // up
glm::mat4 projection = glm::perspective(45.0f, 1.0f*screen_width/sc
```

I cheated a bit and implemented a Gouraud lighting model. We'll cover this in a bit.

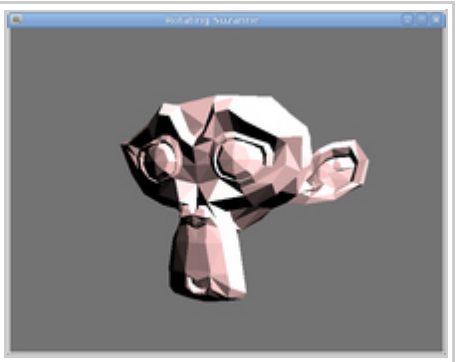


Suzanne, now in our application

Flat-shading - duplicating

vertices and normals

As we discussed in the texturing tutorial, sometimes the same vertex will get different values depending on which face is used. This is the case if we don't want to share normals (and choose an arbitrary face when computing a vertex' normal, as we did above). In that case we need to duplicate the vertex each time it is used with a different normal and then recreate the element array. This will take more time for the loading but it will be faster for OpenGL to process in the long run. The less vertices sent to OpenGL the better. Or, as stated earlier, in this example we will just duplicate the vertices the time they appear; next we can proceed without the element array.



Suzanne with flat-shading

```
for (int i = 0; i < elements.size(); i++) {
  vertices.push_back(shared_vertices[elements[i]]);
  if ((i % 3) == 2) {
    GLushort ia = elements[i-2];
    GLushort ib = elements[i-1];
    GLushort ic = elements[i];
    glm::vec3 normal = glm::normalize(glm::cross(
      shared_vertices[ic] - shared_vertices[ia],
  shared_vertices[ib] - shared_vertices[ia]));
    for (int n = 0; n < 3; n++)
  normals.push_back(normal);
  }
}
```

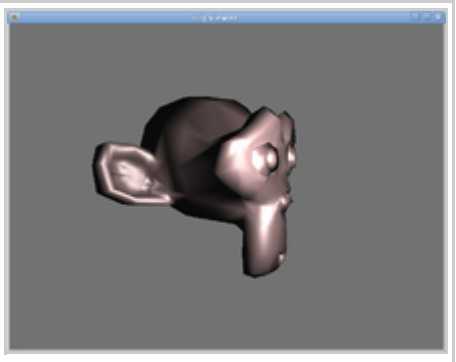
```
glDrawArrays(GL_TRIANGLES, 0, suzanne_vertices.size());
```

With this setup, we can get flat-shading: the varying variable will actually not vary between vertices in the fragment shader, because the normals will be the same for the 3 vertices of every triangle.

Averaging normals

Our algorithm works, but if two faces reference the same vector, then the last face will overwrite the normal for that vertex. This means the object can look very different depending on the face order.

To remedy this, we can average the normal between the two faces. To average two vectors, you take half of the first vector plus half of the second vector. Here we use `nb_seen` to store the vector coefficient, so we can average with a new vector any number of time, without storing the full list of vectors:



Normals averaging

```
mesh->normals.resize(mesh->vertices.size(), glm::vec3(0.0, 0.0, 0.0)
nb_seen.resize(mesh->vertices.size(), 0);
for (int i = 0; i < mesh->elements.size(); i+=3) {
  GLushort ia = mesh->elements[i];
  GLushort ib = mesh->elements[i+1];
  GLushort ic = mesh->elements[i+2];
  glm::vec3 normal = glm::normalize(glm::cross(
    glm::vec3(mesh->vertices[ib]) - glm::vec3(mesh->vertices[ia]),
    glm::vec3(mesh->vertices[ic]) - glm::vec3(mesh->vertices[ia])));

  int v[3];  v[0] = ia;  v[1] = ib;  v[2] = ic;
  for (int j = 0; j < 3; j++) {
    GLushort cur_v = v[j];
    nb_seen[cur_v]++;
    if (nb_seen[cur_v] == 1) {
  mesh->normals[cur_v] = normal;
    } else {
  // average
  mesh->normals[cur_v].x = mesh->normals[cur_v].x * (1.0 - 1.0/nb_s
```

```
    mesh->normals[cur_v].y = mesh->normals[cur_v].y * (1.0 - 1.0/nb_seen[cur_v]) + normal.y * 1.0/nb_seen[cur
    mesh->normals[cur_v].z = mesh->normals[cur_v].z * (1.0 - 1.0/nb_seen[cur_v]) + normal.z * 1.0/nb_seen[cur
    mesh->normals[cur_v] = glm::normalize(mesh->normals[cur_v]);
      }
    }
  }
```

Pre-computed normals

TODO: improve the parser to support .obj normals

The Obj format supports pre-computed normals. Interestingly they are specified in the faces, so if a vertex is present on several faces, it may get different normals, which means we have to use the vertex duplication technique discussed above.

For instance, a basic export of Suzanne references vertex #1 with two different normals #1 and #7:

```
v 0.437500 0.164063 0.765625
...
vn 0.664993 -0.200752 0.719363
...
f 47//1 1//1 3//1
...
f 1//7 11//7 9//7
f 1//7 9//7 3//7
```

By comparison, the MD2/MD3 format (used by Quake among others) also include pre-computed normals, but they are attached to vertices, not to faces.

See also

- TooL (http://sourceforge.net/projects/objloader/): The OpenGL OBJ Loader, released under the GNU GPL (but OpenGL 1.x)

< OpenGL Programming

Browse & download complete code (https://gitlab.com/wikibooks-opengl/modern-tutorials/tree/master)

/Modern_OpenGL_Tutorial_Load_OBJ&oldid=2983992"
