TinyObj Loader Introduction

by Ruen-Rone Lee

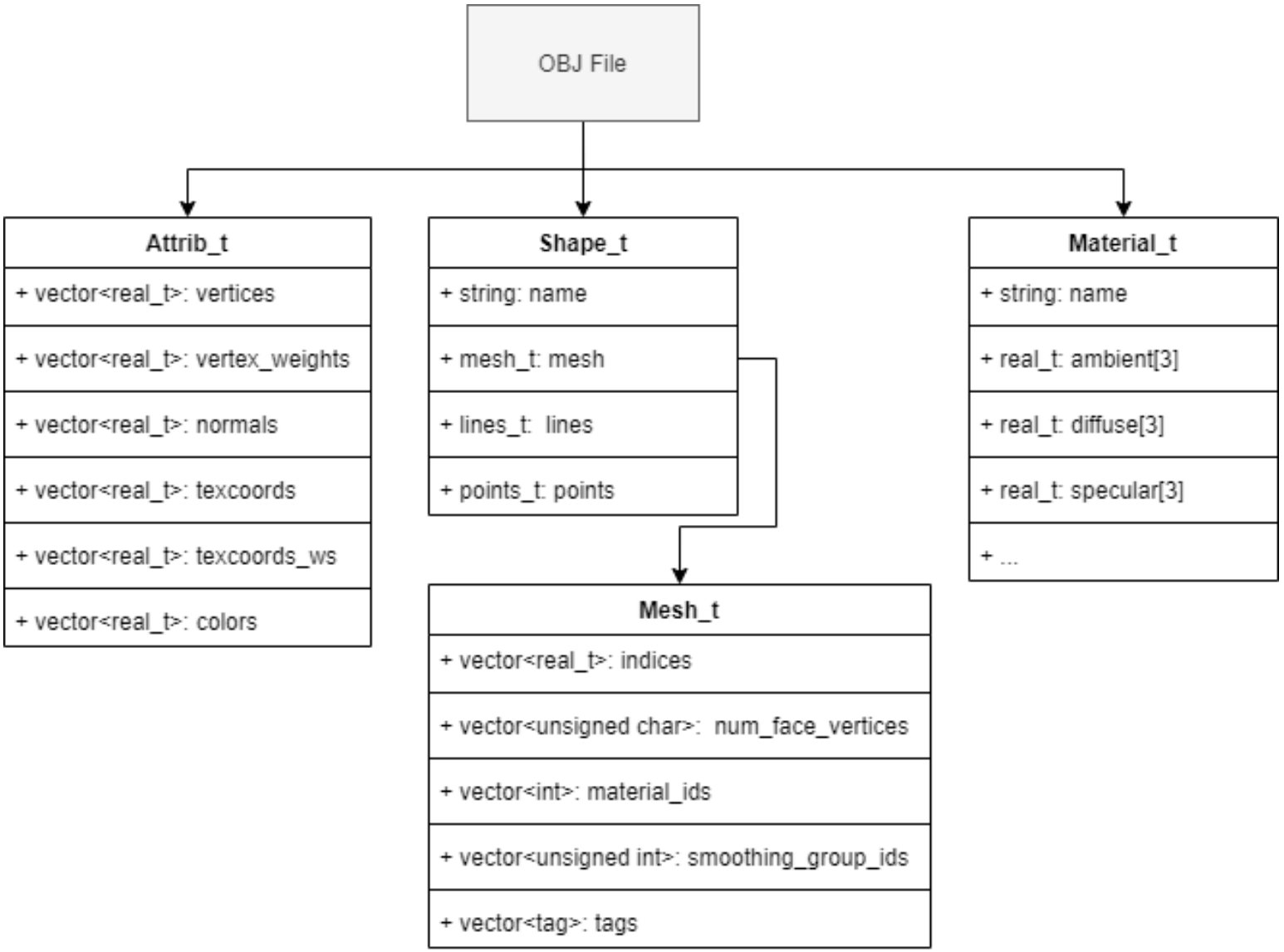
ICL/ITRI

Introduction

- [Wavefront.obj file]
- Positions, normal, texture coordinate…
- Remind Tinyobj(V2.0.0) loader cannot support all data type
- Check tiny_obj_loader.h for detail

```
 32656   v 2.107442 0.835908 2.008115
 32657   v 2.087680 0.816793 2.015349
 32658   v 2.125488 0.809511 2.018409
129743   vn -0.795775 0.237098 -0.557249
129744   vn -0.754805 0.293936 -0.586406
129745   vn -0.330360 0.381367 -0.863378
236154   vt 0.306873 0.233692
236155   vt 0.361487 0.239485
236156   vt 0.356835 0.236813
236157   vt 0.361736 0.238910
262860   f 32656/32656/32656 32657/32657/32657 32658/32658/32658
262861   f 32659/32659/32659 32660/32660/32660 32661/32661/32661
262862   f 32662/32662/32662 32663/32663/32663 32664/32664/32664
```

Data Structure



Load Model

```
// load obj file
void LoadModels(string model_path)
{
    std::vector<tinyobj::shape_t> shapes;
    std::vector<tinyobj::material_t> materials;
    tinyobj::attrib_t attrib;

    std::string err;
    std::string warn;
    bool ret = tinyobj::LoadObj(&attrib, &shapes, &material
s, &warn, &err, model_path);

    printf("Load Models ! Shapes size %d Material size %d\n
", shapes.size(), materials.size());
    // Error handling
```

Parse different property into data structure

Attribute Data Structure

- **Store vertices, normal and tex_coordinate data**

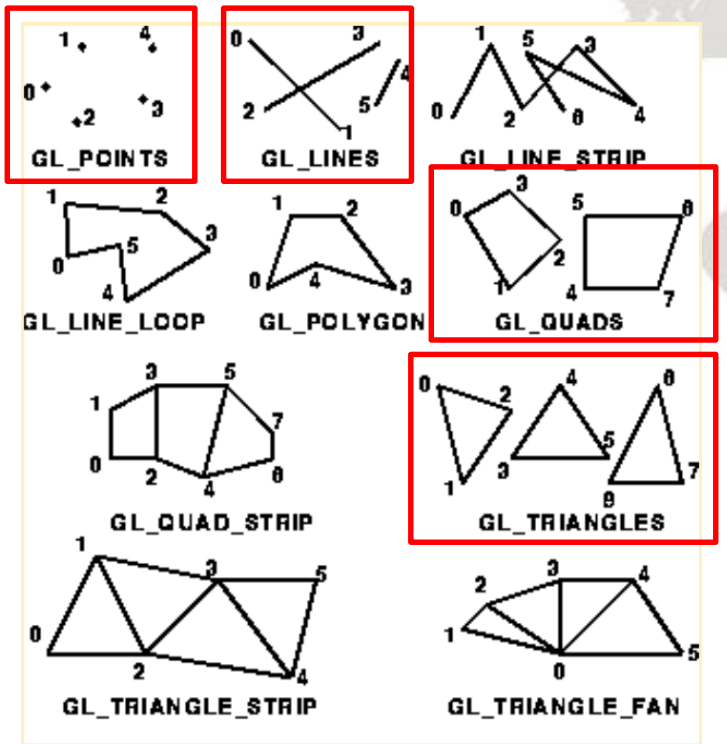
```
struct attrib_t
{
    std::vector<real_t> vertices; // v(xyz)

    // for backward compatibility, we store vertex weight
in separate array
    std::vector<real_t> vertex_weights; // v (w)
    std::vector<real_t> normals;        // vn
    std::vector<real_t> texcoords;      // vt (uv)

// for backward compatibility, we store vertex weight in
separate array
    std::vector<real_t> texcoord_ws;    // vt (w)
    std::vector<real_t> colors;         // vertex color
```

Shape Data Structure

- **Store the mesh information**

```
struct shape_t
{
    std::string name; // v(xyz)
    mesh_t mesh;
    lines_t lines;
    points_t points;
}
```

```
struct mesh_t
{
    std::vector<index_t> indices;
    std::vector<unsigned char>
        num_face_vertices;
    std::vector<int> material_ids;
    std::vector<unsigned int>
        smoothing_group_ids;
    std::vector<tag_t> tags;
}
```



Material Data Structure

- **Store texture name, material property.**

```
struct material_t
{
    std::string name;
    real_t ambient[3];
    real_t diffuse[3];
    real_t specular[3];
    real_t transmittance[3];
    real_t emission[3];
    real_t shininess;
    real_t ior;
    real_t dissolve; // 1 == opaque; 0 == fully transparent

    // illumination model
    int illum;
    int dummy;
    ...
```

- Simple material example

Extract Mesh Data

- **Extract data from data structure**

```
size_t index_offset = 0;
for (size_t f = 0; f < shape->mesh.num_face_vertices.size(); f++) {
    int fv = shape->mesh.num_face_vertices[f];
    // Loop over vertices in the face.
    for (size_t v = 0; v < fv; v++) {
        // access to vertex
        tinyobj::index_t idx = shape->mesh.indices[index_offset + v];
        vertices.push_back(attrib->vertices[3 * idx.vertex_index + 0]);
        vertices.push_back(attrib->vertices[3 * idx.vertex_index + 1]);
        vertices.push_back(attrib->vertices[3 * idx.vertex_index + 2]);
        // Optional: vertex colors
        colors.push_back(attrib->colors[3 * idx.vertex_index + 0]);
        colors.push_back(attrib->colors[3 * idx.vertex_index + 1]);
        colors.push_back(attrib->colors[3 * idx.vertex_index + 2]);
    }
    index_offset += fv;
}
```