**Lab 2 Visualitzacio Avançada**

**Loading ASES and creating meshes**

**Diego Andersen Cabañero (159240), Francisco Armijo Momblan (159212), Davide Fiocchi (190034), Marc Sánchez Quibus (159247)**

1. **ASE loader.**

This method allows us to obtain the position, normal, texture, even material info about the object to load. The following lists are used along the load.

std::vector< Vector3 > vertices; //here we store the vertices

std::vector< Vector3 > normals; //here we store the normals

std::vector< Vector2 > uvs; //here we store the texture coordinates

std::vector<Vector2> fuvs; //here we depoy the texture info

std::vector<Triangle> faces; //Easier way to deal with triangles

std::vector< Vector3 > fvertices; //Here we deploy the vértices info

We created the Triangle class to avoid problems at implementation. It is a simple structure with three integers, which will be used to define the position of the triangles in the ASE File (not the coordinates of triangles).

We will explain the loadASE function in several blocks, because its complexity.

void Mesh::loadASE(const std::string& fileName, float scale) {

TextParser t(addCurrentPath(fileName).c\_str());

t.seek("\*MESH\_NUMVERTEX");

int numVertices = t.getint();

t.seek("\*MESH\_NUMFACES");

int numFaces = t.getint();

//Tenemos en cuenta las caras para la lista de triangulos

faces.resize(numFaces);

We obtain the number of vertices and faces presents in the ASE file using the Parser library given to us, to use them later at the loops. We also set the size of “faces” list of Triangles, to store them later.

for (int i = 0; i < numVertices; i++) {

t.seek("\*MESH\_VERTEX");

t.getword();

float v1 = t.getfloat();

float v2 = t.getfloat();

float v3 = t.getfloat();

Vector3 v = Vector3(v1\*scale, v2\*scale, v3\*scale);

//\*\*\*\*\*\*\*\*\*\*\*\*

vertices.push\_back(Vector3(v.x, v.z ,v.y));

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

}

We search the vertices of the ASE file, saving them as floats, because we are manipulating the actual coordinates, not the indexes of the ASE file, storing them on a Vector 3D, scaled according to the input. We push the vector to vertices according to the coordinate system of the framework.

for (int i = 0; i< numFaces; i++) {

t.seek("\*MESH\_FACE");

t.getword();

t.seek("A:");

faces[i].a = t.getint();

t.seek("B:");

faces[i].b = t.getint();

t.seek("C:");

faces[i].c = t.getint();

}

Similar to the previous loop. In this case, we are storing the indexes of the vertices of each face in our faces list, not the real coordinates.

for (int i = 0; i < numFaces; i++) {

const Triangle & t = faces[i];

fvertices.push\_back(vertices[t.a]);

fvertices.push\_back(vertices[t.b]);

fvertices.push\_back(vertices[t.c]);

}

Finally, we push the actual coordinates on fvertices. We declare the Triangle t, which will have three integers, one per each vertex, according to its position on the original list, in the ASE file. We have stored the faces in terms of indexes, and the current coordinates for each index, so we only have to relation them and, in fvertices, we will have a list with all the coordinates.

t.seek("\*MESH\_NUMTVERTEX");

int numTextures = t.getint();

for(int i=0; i< numTextures; i++){

t.seek("\*MESH\_TVERT");

t.getword();

float f1 = t.getfloat();

float f2 = t.getfloat();

float f3 = t.getfloat();

Vector2 uv = Vector2(f1, f2);

uvs.push\_back(Vector2(uv.x,uv.y));

}

t.seek("\*MESH\_NUMTVFACES");

int numTextureFaces = t.getint();

faces.resize(numTextureFaces);

for (int i = 0; i< numTextureFaces; i++) {

t.seek("\*MESH\_TFACE");

t.getword();

faces[i].a = t.getint();

faces[i].b = t.getint();

faces[i].c = t.getint();

}

for (int i = 0; i < numTextureFaces; i++) {

const Triangle & tv = faces[i];

fuvs.push\_back(uvs[tv.a]);

fuvs.push\_back(uvs[tv.b]);

fuvs.push\_back(uvs[tv.c]);

}

The method to obtain the textures is exactly the same that we used on the faces, since the data structure of the ASE file is also the same, except one detail: the textures are expressed in 3 dimensions in the ASE file but the framework is set up to store it in 2 dimensions. We seek the three components, but only store the 2 first.

.

for (int i = 0; i < numTextureFaces; i++) {

t.seek("\*MESH\_FACENORMAL");

normals.push\_back(Vector3(t.getfloat(), t.getfloat(), t.getfloat()));// Floats pq son coordenadas directas

} }

Obtain the normal is easier, since they are directly written on the ASE file. We simply search for each one and push

**Rendering a model.**

In Mesh::Render() function, we take care that the correct variables are put in each of the OpenGL calls. If we put the loadASE function inside the render, It will run. But the next steps of the Lab (changing the mode and the model) will not work if leave this, so this does not appear in the code delivered.

**Changing the model and the mode**

We add a few variables to Application class.

std::vector<std::string> ASES = { "teapot.ASE", "Girl.ASE", "Box2.ASE" };

float sizes[3] = { 0.1 , 10 , 0.1 };

int primitive = GL\_TRIANGLES;

int aseModel = 3;

Basically, a List with the names of the ASE models, an array of its sizes, primitive will be used to change the GL Mode and aseModel will be used to change the ASE Model.

We changed a bit the flow of the rendering. In Main.cpp, first we call to application->init(), and this renders by default the teapot. In Init, we can load the model using the following line:

mesh->loadASE(ASES[aseModel%3], sizes[aseModel%3]);

that charges the Teapot (can be changed).

We can change the model pressing + or -. For that, we implemented a few options more in the onKeyPressed switch.

case SDLK\_PLUS: {

aseModel++;

init();

}

break;

case SDLK\_MINUS: {

if (aseModel < 3) {

aseModel=aseModel + 3;

}

aseModel--;

init();

}

Break;

}

Basically, we add or subtract 1 to the aseModel variable, that charges the ASE accord to the position of the list and call again to the init method. We use the modulus to avoid crash the computer subtracting to aseModel too many times.

Girl.ASE is a bit slower than the others at the load, since it’s the most complex model of the three, but it completes the rendering in an acceptable amount of time.

To change the GL mode, we declared the primitive variable and added the following cases to the switch.

case SDLK\_p: {

primitive = GL\_POINTS;

}

break;

case SDLK\_l: {

primitive = GL\_LINES;

}

break;

case SDLK\_f: {

primitive = GL\_TRIANGLES;

}

break;

If we press P, L or F, the primitive value will change and primitive value is used in the render function in application

mesh->render(primitive);

that is also called in each iteration of the mainloop, so, the GL Mode change is applied almost instantly.

Was also required to implement the move method using the keys. We made that at the previous lab, simply including the cases on the onKeyPressed switch:

case SDLK\_LEFT: {

camera->move(Vector3(1, 0, 0));

}

break;

case SDLK\_RIGHT: {

camera->move(Vector3(-1, 0, 0));

}

break;

case SDLK\_UP: {

camera->move(Vector3(0, -1, 0));

}

break;

case SDLK\_DOWN: {

camera->move(Vector3(0, 1, 0));

}

break;