OpenGL en la práctica: Modelos 3D

Prof. Ruben Dario Escandon Guzman

Departamento de Ciencias e Ingeniería de la Computación
Pontificia Universidad Javeriana Cali

rubend.escandon@javerianacali.edu.co

March 21, 2019

Contenido

1. Introducción
2. Describiendo Mesh.h
3. Describiendo Model.h

Introducción

En las aplicaciones gráficas prácticas, generalmente hay muchos modelos complicados e interesantes que son mucho más bonitos de ver que un contenedor estático.

Sin embargo, estos son más complicados de programar que un simple contenedor estático, para ello se encarga los artistas 3D en sus herramientas tales como Blender, Maya, 3dMax, zBrush, entre otros.

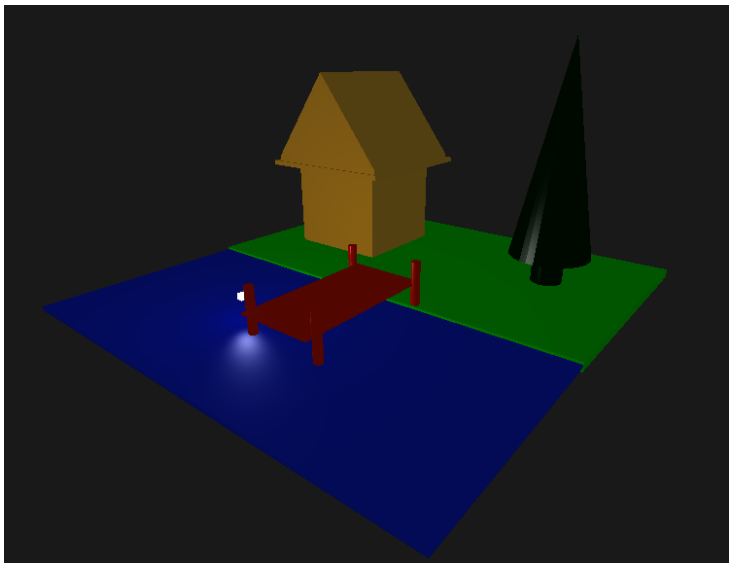

Figure: Un modelo ms complejo

Introducción

Los artistas 3D se apoyan en las herramientas de diseño 3D y crean estos modelos complicados a través de un proceso de mapeo UV. El cual es un proceso que permite proyectar una imagen 2D (textura) en un objeto en 3D

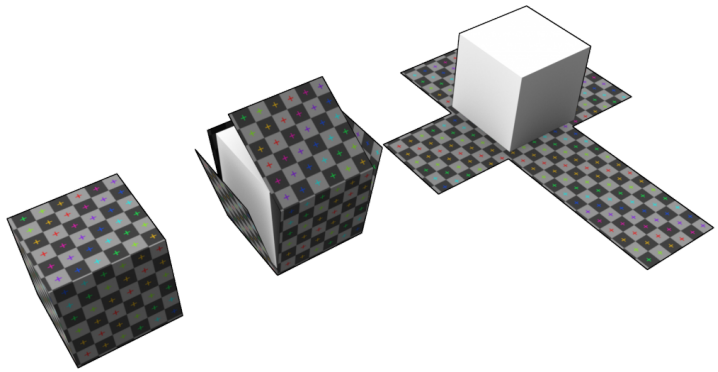

Figure: Analogía de mapeo UV

Porqué .OBJ WaveFront?

Los artistas cuentan con un amplio conjunto de herramientas para crear modelos y exportarlos con alta calidad sin tener que preocuparse demasiado por los detalles técnicos. Además de tenemos una amplia variedad de archivos para exportarlos OBJ y collada.

Por ejemplo, los archivos Collada son extremadamente extensos y contiene modelos, texturas, luces, materiales, animación, cámaras, información de escena, mientras que los Wavefront OBJ solo contienen datos de modelo con informacin de material menor como colores de modelo y mapas difusos/especulares.

Estructura .obj y librería para cargarlo

La estructura de archivos .obj está compuesto básicamente de vértices, coordenadas de las texturas, normales, el uso del material y las caras para la mallas. También, si hay materiales aplicados un archivo .mtl es vinculado al archivo .obj, este contiene las características del color, textura e iluminación implementadas.

Una popular librería para la carga de modelos 3D es llamada Assimp *Open Asset Import Library*. Carga todos los datos del modelo en las estructuras de datos generalizadas de Assimp y no importa que tipo de archivo exportado uses siempre es cargado en la misma estructura de Assimp.

Estructura .obj y librería Assimp

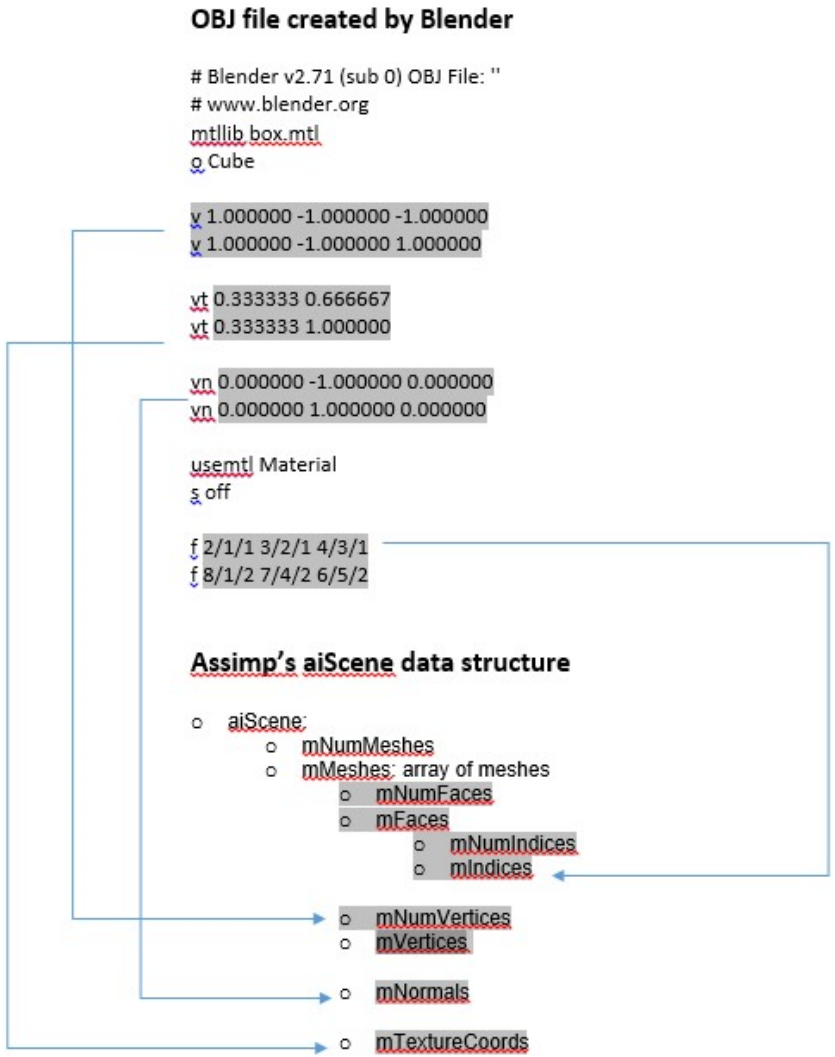

Figure: Estructura de carga de un archivo .obj mediante Assimp

Estructura .obj y librería Assimp

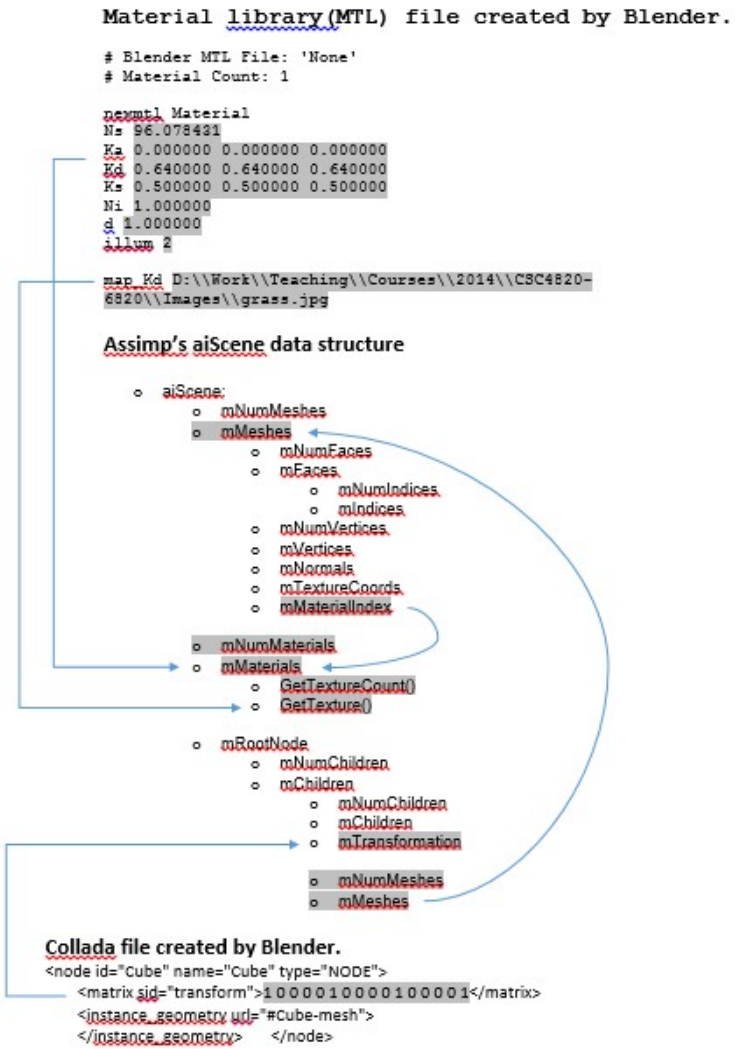

Figure: Estructura de carga del archivo .mtl mediante Assimp

Partes de la rutina Mesh.h

Una rutina llamada "Mesh.h" es la encargada de operar la data de tipo OpenGL obtenido de la estructura de Assimp y luego es enviado a ser renderizado o dibujado

Definir una estructura de vértices del objeto 3D

```
struct Vertex
{
    // Position
    glm::vec3 Position;
    // Normal
    glm::vec3 Normal;
    // TexCoords
    glm::vec2 TexCoords;
};
```

La textura es almacenada por un identificador, un tipo y un lugar donde está guardada.

Definir una estructura para la textura

```
struct Texture
{
    GLuint id;
    string type;
    aiString path;
};
```

Partes de la rutina Mesh.h

Declarar una estructura Material es importante en el caso de modelos que no cuenten con texturas, opten por los materiales y colores del objeto 3D.

Definir una estructura de material

```
struct Material {
    glm::vec3 diffuse;
    glm::vec3 specular;
    glm::vec3 ambient;
    float shininess;
};
```

Partes de la rutina Mesh.h

La estructura base de la archivo Mesh.h es la siguiente.

```
class Mesh
{
public:
    /*  Mesh Data  */
    vector<Vertex> vertices;
    vector<GLuint> indices;
    vector<Texture> textures;

    glm::vec3 ambient;
    glm::vec3 diffuse;
    glm::vec3 specular;
    GLfloat shininess;

    /*  Functions  */
    // Constructor
    Mesh( vector<Vertex> vertices, vector<GLuint> indices, vector<Texture> textures,
            glm::vec3 materialAmbiente, glm::vec3 materialDifuso, glm::vec3 materialSpecular, GLfloat materialShininess){ ... }


    // Render the mesh
    void Draw( Shader shader ){ ... }

private:
    /*  Render data  */
    GLuint VAO, VBO, EBO;

    /*  Functions    */
    // Initializes all the buffer objects/arrays
    void setupMesh( ){ ... }
};
```

Partes de la rutina Mesh.h – Constructor Mesh(...)

Un constructor donde llega la data extraída del modelo mediante las rutinas de Assimp.

```
// Constructor
Mesh( vector<Vertex> vertices, vector<GLuint> indices, vector<Texture> textures,
        glm::vec3 materialAmbiente, glm::vec3 materialDifuso, glm::vec3 materialSpecular, GLfloat materialShininess)
{
    this->vertices = vertices;
    this->indices = indices;
    this->textures = textures;
    this->ambient = materialAmbiente;
    this->diffuse = materialDifuso;
    this->specular = materialSpecular;
    this->shininess = materialShininess;
    // Now that we have all the required data, set the vertex buffers and its attribute pointers.
    this->setupMesh( );
}
```

Partes de la rutina Mesh.h – void setupMesh()

Una vez tenido la data de malla del objeto se necesita un lugar para configurar los buffers apropiados y especificar el diseo del sombreador de vrtices a travs de los punteros de atributo de vrtice.

```
void setupMesh( )
{
    // Create buffers/arrays
    glGenVertexArrays( 1, &this->VAO );
    glGenBuffers( 1, &this->VBO );
    glGenBuffers( 1, &this->EBO );

    glBindVertexArray( this->VAO );
    // Load data into vertex buffers
    glBindBuffer( GL_ARRAY_BUFFER, this->VBO );
    // A great thing about structs is that their memory layout is sequential for all its items.
    // The effect is that we can simply pass a pointer to the struct and it translates perfectly to a glm::vec3/2 array which
    // again translates to 3/2 floats which translates to a byte array.
    glBufferData( GL_ARRAY_BUFFER, this->vertices.size( ) * sizeof( Vertex ), &this->vertices[0], GL_STATIC_DRAW );

    glBindBuffer( GL_ELEMENT_ARRAY_BUFFER, this->EBO );
    glBufferData( GL_ELEMENT_ARRAY_BUFFER, this->indices.size( ) * sizeof( GLuint ), &this->indices[0], GL_STATIC_DRAW );

    // Set the vertex attribute pointers
    // Vertex Positions
    glEnableVertexAttribArray( 0 );
    glVertexAttribPointer( 0, 3, GL_FLOAT, GL_FALSE, sizeof( Vertex ), ( GLvoid * )0 );
    // Vertex Normals
    glEnableVertexAttribArray( 1 );
    glVertexAttribPointer( 1, 3, GL_FLOAT, GL_FALSE, sizeof( Vertex ), ( GLvoid * )offsetof( Vertex, Normal ) );
    // Vertex Texture Coords
    glEnableVertexAttribArray(2);
    glVertexAttribPointer( 2, 2, GL_FLOAT, GL_FALSE, sizeof( Vertex ), ( GLvoid * )offsetof( Vertex, TexCoords ) );

    glBindVertexArray( 0 );
}
```

Partes de la rutina Mesh.h

El definir los vértices mediante un vector de tipo estructura es poderoso, porque alinea en un solo renglón cada item de la estructura, convirtiendose en un insumo para el comando glBufferData()

```
struct Vertex
{
    // Position
    glm::vec3 Position;
    // Normal
    glm::vec3 Normal;
    // TexCoords
    glm::vec2 TexCoords;
};
```

Ejemplo: Vextex = [Vertex.position, Vextex Normal, Vertex.TexCoords]

Partes de la rutina Mesh.h – void Draw(...)

La representación o renderizado del modelo está gobernada de acuerdo a esta rutina y antes de ser enviado los vértices a ser dibujados mediante glDrawElements, deben ser unidas las texturas en el orden que van a ser usadas por el modelo, de lo contrario se dibujarian con las texturas equivocadas.

```
// Render the mesh
void Draw( Shader shader )
{
    // Bind appropriate textures
    GLuint diffuseNr = 1;
    GLuint specularNr = 1;


    if (textures.size() > 0) {
        for (GLuint i = 0; i < this->textures.size(); i++){ ... }
    }
    else{ ... }


    // Also set each mesh's shininess property to a default value
    //(if you want you could extend this to another mesh property and possibly change this value)
    glUniform1f( glGetUniformLocation( shader.ID, "material.shininess" ), 32.0f );

    // Draw mesh
    glBindVertexArray( this->VAO );
    glDrawElements( GL_TRIANGLES, this->indices.size( ), GL_UNSIGNED_INT, 0 );
    glBindVertexArray( 0 );

    // Always good practice to set everything back to defaults once configured.
    for ( GLuint i = 0; i < this->textures.size( ); i++ )
    {
        glActiveTexture( GL_TEXTURE0 + i );
        glBindTexture( GL_TEXTURE_2D, 0 );
    }
}
```

Partes de la rutina Mesh.h – void Draw(...)

Por lo tanto, hay un condicional que define la presencia o no de texturas. En caso de **Si**, se define un "texture_diffuseX" y "texture_specularX", donde ese **X** será el número de un contador respectivo al tipo de la textura e irá aumentado de acuerdo a cantidad que exista.

```
if (textures.size() > 0) {
    for (GLuint i = 0; i < this->textures.size(); i++)
    {
        glActiveTexture(GL_TEXTURE0 + i); // Active proper texture unit before binding
                                          // Retrieve texture number (the N in diffuse_textureN)
        stringstream ss;
        string number;
        string name = this->textures[i].type;


        if (name == "texture_diffuse")
        {
            ss << diffuseNr++; // Transfer GLuint to stream
        }
        else if (name == "texture_specular")
        {
            ss << specularNr++; // Transfer GLuint to stream
        }

        number = ss.str();
        // Now set the sampler to the correct texture unit
        glUniform1i(glGetUniformLocation(shader.ID, (name + number).c_str()), i);
        // And finally bind the texture
        glBindTexture(GL_TEXTURE_2D, this->textures[i].id);
    }
}
else
```

Partes de la rutina Mesh.h – void Draw(...)

Sin embargo, si el modelo no cuenta con textura pero si con materiales, optará por operar su material difuso, especular y su resplandor.

```
else
{
    glm::vec3 lightColor;
    lightColor.r = diffuse.r;
    lightColor.g = diffuse.g;
    lightColor.b = diffuse.b;


    glm::vec3 diffuseColor = lightColor * glm::vec3(0.5f);
    glm::vec3 ambientColor = diffuseColor * glm::vec3(1.0f);
    glUniform3f(glGetUniformLocation(shader.ID, "light.ambient"), ambientColor.r, ambientColor.g, ambientColor.b);
    glUniform3f(glGetUniformLocation(shader.ID, "light.diffuse"), diffuseColor.r, diffuseColor.g, diffuseColor.b);
    glUniform3f(glGetUniformLocation(shader.ID, "light.specular"), 1.0f, 1.0f, 1.0f);

    // ...

    glUniform3f(glGetUniformLocation(shader.ID, "material.ambient"),  ambient.x,  ambient.y,  ambient.z);
    glUniform3f(glGetUniformLocation(shader.ID, "material.diffuse"),  diffuse.x,  diffuse.y,  diffuse.z);
    glUniform3f(glGetUniformLocation(shader.ID, "material.specular"), specular.x, specular.y, specular.z);
}
```

Rutina Model.h

Es una clase que permite representar un modelo 3D completo mediante la carga de múltiples mallas, múltiples objetos, materiales y texturas.

```
class Model
{
public:
    /*  Functions   */
    // Constructor, expects a filepath to a 3D model.
    Model( const GLchar *path ) { ... }

    // Draws the model, and thus all its meshes
    void Draw( Shader shader ) { ... }

private:
    /*  Model Data  */
    vector<Mesh> meshes;
    struct Material { ... };

    aiMaterial mat;
    Material materiales;
    string directory;
    vector<Texture> textures_loaded;    // Stores all the textures loaded so far, optimization to make sure textures aren't loaded mor

    /*  Functions   */
    // Loads a model with supported ASSIMP extensions from file and stores the resulting meshes in the meshes vector.
    void loadModel( string path ) { ... }
    // Processes a node in a recursive fashion. Processes each individual mesh located at the node and repeats this process on its chi
    void processNode( aiNode* node, const aiScene* scene ) { ... }
    Mesh processMesh( aiMesh *mesh, const aiScene *scene ) { ... }

    // Checks all material textures of a given type and loads the textures if they're not loaded yet.
    // The required info is returned as a Texture struct.
    vector<Texture> loadMaterialTextures( aiMaterial *mat, aiTextureType type, string typeName ) { ... }
    Material loadMaterial(aiMaterial* mat) { ... }
};
```

Rutina Model.h

La clase Model contiene un vector de objetos de tipo malla (Mesh), los cuales requieren que le demos una ubicación de archivo en su constructor.

A continuación, carga el archivo .obj a través de loadModel(), Función que se llama en el constructor Model. Todas las funciones privadas están diseñadas para procesar una parte de la rutina de importación de Assimp "loadModel", "processNode", "processMesh".

También almacenamos el directorio de la ruta del archivo que más tarde necesitaremos al cargar texturas o materiales "loadMaterialTextures" y "loadMaterial".