Politecnico di Torino

VR @POLITO

GRAINS

GRAphics and INtelligent Systems group

Laboratorio 09

OpenGL

Part IV - Assimp

Outline

- Assimp
- Mesh class
- Model class

3D Model loading

In all the scenes so far, only triangles, planes or cubes are used. Generally, in common graphics applications, there are more complicated and interesting models to be rendered. However, it is not possible to manually define all the vertices, normals, and texture coordinates of the complicated shapes, as done for the previous example. What it would be really useful, it would be a tool that allows OpenGL developers to import models created by artist with 3D graphic suites such as Blender, 3DS Max or Maya. Such software are able to generate all the vertex coordinates, vertex normals, and texture coordinates while exporting them to a model file format. All the technical aspects are hidden in the exported model file.

3D Model loading

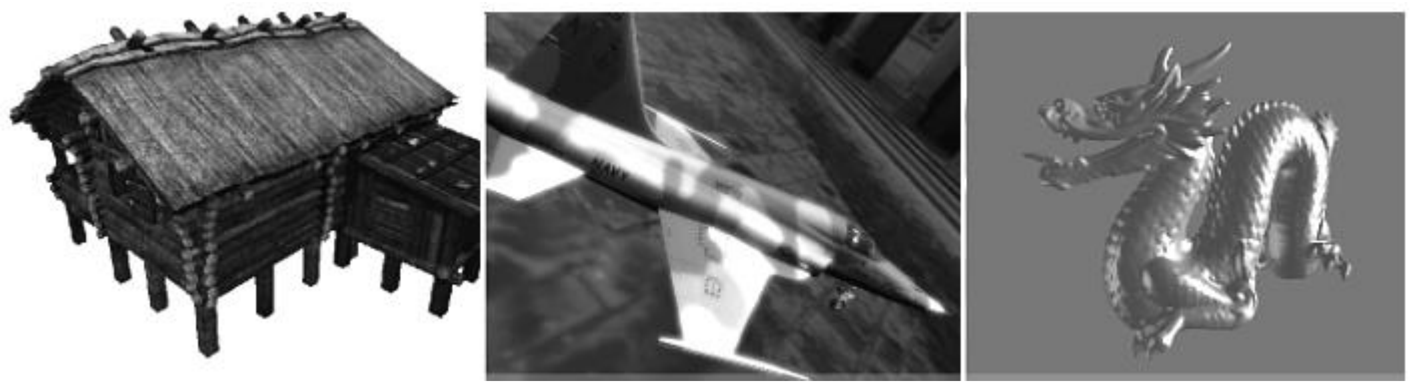
In this way OpenGL developers can parse the exported model files and extract all the relevant information that can be stored in a format that OpenGL understands. There are dozens of different file formats. The Wavefront object format (.obj) is generally considered to be an easy-to-parse model format. It only contains model data with minor material information like model colors and diffuse/specular maps.

Assimp

Assimp (Open Asset Import Library) is a very popular model importing library (link: https://assimp.org/index.html).

Assimp is able to import dozens of different model file formats (and export to some as well) by loading all the model's data into Assimp's generalized data structures. As soon as Assimp has loaded the model, it is possible to retrieve all the necessary data by leveraging Assimp's data structures.

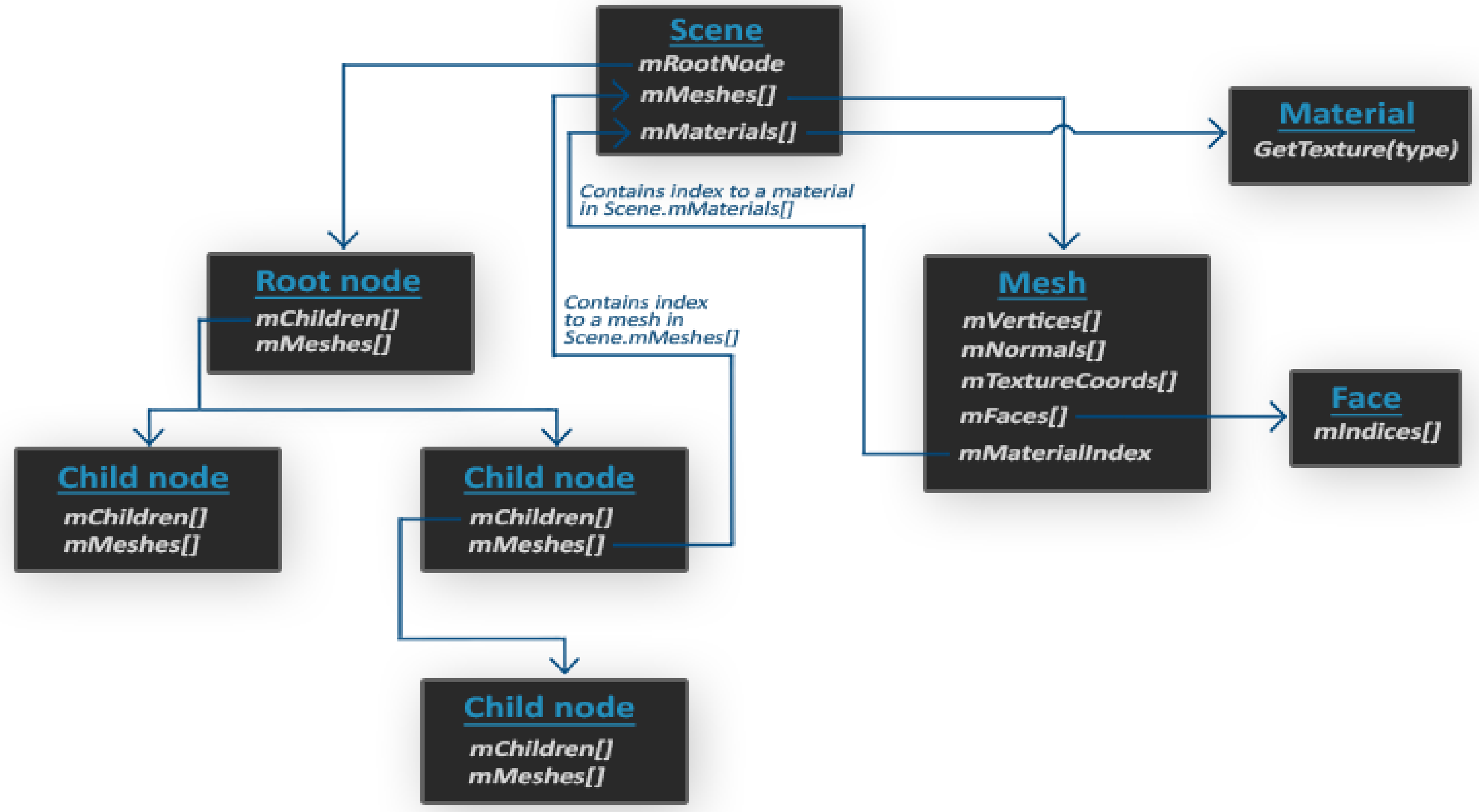
Because the data structure of Assimp stays the same, regardless of the type of file format imported, it abstracts from all the different file formats.



Assimp

When a model is imported via Assimp, it is loaded into a scene object that contains all the data of the imported model/scene. Assimp then has a collection of nodes where each node contains indices to data stored in the scene object where each node can have any number of children.

Assimp data structure



Assimp data structure

All the data of the scene/model is contained in the **Scene** object like all the materials and the meshes. The scene object also contains a reference to the root node of the scene.

The **Root node** may contain children nodes (like all other nodes) and could have a set of indices that point to mesh data in the scene object's `mMeshes` array. The scene's `mMeshes` array contains the actual Mesh objects, the values in the `mMeshes` array of a node are only indices for the scene's meshes array.

Assimp data structure

A **Mesh** object itself contains all the relevant data required for rendering, e.g., vertex positions, normal vectors, texture coordinates, faces, and the material of the object. A mesh contains several faces.

A **Face** represents a render primitive of the object (triangles, squares, points). A face contains the indices of the vertices that form a primitive. Because the vertices and the indices are separated, it will be useful to render via index buffer (see Example 4).

Finally a mesh also links to a **Material object** that hosts several functions to retrieve the material properties of an object. For example, it includes information about colors and/or texture maps (like diffuse and specular maps).

Assimp

The common workflow include the following step:

- Load an object into a Scene object,
- Recursively retrieve the corresponding Mesh objects from each of the nodes
- Process each Mesh object to retrieve the vertex data, indices, and its material properties.

It is worth noticing that the above workflow elaborates a collection of mesh data that can be combined to create a single **Model** object. This is done to allow 3D artists to create models that have several sub-models/shapes that they consist of. Each of those single shapes is called a **Mesh**. A single mesh is the minimal representation of what it is needed to draw an object in OpenGL (vertex data, indices, and material properties). A model (usually) consists of several meshes.

Mesh

With Assimp it is possible to load many different models into the application, but once loaded they're all stored in Assimp's data structures. These data have to be transformed to a format that OpenGL understands in order to render the objects. A mesh represents a single drawable entity, so it is convenient to define a **mesh class**.

A mesh should at least need a set of *vertices*, where each vertex contains a position vector, a normal vector, and a texture coordinate vector. A mesh should also contain indices for indexed drawing, and material data in the form of textures (diffuse/specular maps).

Mesh

Required vertex attributes are stored in a struct called `Vertex`.

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

The texture data are organized in a `Texture` struct the includes a field type to specify if it is a diffuse or specular texture.

```
struct Texture {
    unsigned int id;
    string type;
    string path;
};
```

Mesh

The structure of the mesh class is reported in the following:

```
class Mesh {
public:
    // mesh data
    vector<Vertex> vertices;
    vector<unsigned int> indices;
    vector<Texture> textures;

    Mesh(vector<Vertex> vertices, vector<unsigned int> indices, vector<Texture> textures);

    void Draw(Shader& shader);

private:
    //  render data
    unsigned int VAO, VBO, EBO;

    void setupMesh();
};
```

Mesh

The constructor sets the class's public variables with the constructor's corresponding argument variables

```
Mesh(vector<Vertex> vertices, vector<unsigned int> indices,
vector<Texture> textures)
{
    this->vertices = vertices;
    this->indices = indices;
    this->textures = textures;

    setupMesh();
}
```

Mesh
Initialization

The `setupMesh` function sets the appropriate buffers and specifies the vertex shader layout via vertex attribute pointers.

```
void setupMesh()
{
     glGenVertexArrays(1, &VAO);
     glGenBuffers(1, &VBO);
     glGenBuffers(1, &EBO);

     glBindVertexArray(VAO);
     glBindBuffer(GL_ARRAY_BUFFER, VBO);
     glBufferData(GL_ARRAY_BUFFER, vertices.size() * 
sizeof(Vertex), &vertices[0], GL_STATIC_DRAW);

     glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EBO);
     glBufferData(GL_ELEMENT_ARRAY_BUFFER, indices.size() * 
sizeof(unsigned int), &indices[0], GL_STATIC_DRAW);

    // set the vertex attribute pointers
    // vertex positions
    glEnableVertexAttribArray(0);
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 
sizeof(Vertex), (void*)0);
    // vertex normals
    glEnableVertexAttribArray(1);
    glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 
sizeof(Vertex), (void*)offsetof(Vertex, Normal));
    // vertex texture coords
    glEnableVertexAttribArray(2);
    glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, 
sizeof(Vertex), (void*)offsetof(Vertex, TexCoords));
}
```

Mesh

Initialization

Struct have an important property in C++ that their memory layout is sequential, i.e., the struct's variables are stored in sequential order. This means that struct are easily convertible to a buffer array. For example, for a `Vertex` struct defined as:

```
Vertex vertex;
vertex.Position = glm::vec3(0.2f, 0.4f, 0.6f);
vertex.Normal = glm::vec3(0.0f, 1.0f, 0.0f);
vertex.TexCoords = glm::vec2(1.0f, 0.0f);
```

The memory layout of the struct would be equal to:

```
[0.2f, 0.4f, 0.6f, 0.0f, 1.0f, 0.0f, 1.0f, 0.0f];
```

Thanks to this property, it is possible to directly pass a pointer to a large list of Vertex structs as the buffer's data and they translate perfectly to what `glBufferData` expects as its argument

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

Mesh

Initialization

Another great use of structs is a preprocessor directive called `offsetof(s,m)` that takes as its first argument a struct and as its second argument a variable name of the struct. The macro returns the byte offset of that variable from the start of the struct. This is perfect for defining the offset parameter of the `glVertexAttribPointer` function:

```
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex),
(void*)offsetof(Vertex, Normal));
```

Using a struct like this doesn't only make the code more readable, but also allows developers to easily extend the structure. For example, if an additional vertex attribute is considered, it can be easily added to the struct and, due to its flexible nature, the rendering code won't break.

Mesh

Rendering

Before rendering the mesh, the bind with the appropriate textures has to be activated before calling the `glDrawElements` function. However, this is somewhat difficult since it is not known in advance how many (if any) textures the mesh has and what type they may have. So how do the texture units and samplers in the shaders have to be set? To solve the issue a certain naming convention is assumed: each diffuse texture is named `texture_diffuseN`, and each specular texture should be named `texture_specularN` where N is any number ranging from 1 to the maximum number of texture samplers allowed.

Using this convention, it is possible to process any amount of textures on a single mesh.

Mesh

Rendering

The resulting drawing code then becomes:

```cpp
void Draw(Shader& shader)
{
      unsigned int diffuseNr = 1;
      unsigned int specularNr = 1;
      for (unsigned int i = 0; i < textures.size(); i++)
      {
            glActiveTexture(GL_TEXTURE0 + i); // active proper texture unit before binding
            string number;
            string name = textures[i].type;
            if (name == "texture_diffuse")
                  number = std::to_string(diffuseNr++);
            else if (name == "texture_specular")
                  number = std::to_string(specularNr++);

            shader.setInt(("material." + name + number).c_str(), i);
            glBindTexture(GL_TEXTURE_2D, textures[i].id);
      }
      glActiveTexture(GL_TEXTURE0);
      glBindVertexArray(VAO);
      glDrawElements(GL_TRIANGLES, static_cast<unsigned int>(indices.size()), GL_UNSIGNED_INT, 0);
      glBindVertexArray(0);
  }
```

Model

The `Model` class is a new class that represents a model in its entirety, i.e., a collection of multiple meshes, possibly with multiple textures. A house, that contains a wooden balcony, a tower, and perhaps a swimming pool, could still be loaded as a single model.

Model

The class structure of the `Model` class is:

```
class Model
{
public:
    Model(char* path)
    {
        loadModel(path);
    }

    void Draw(Shader& shader);

private:
    // model data
    vector<Mesh> meshes;
    string directory;

    void loadModel(string path);
    void processNode(aiNode* node, const aiScene* scene);
    Mesh processMesh(aiMesh* mesh, const aiScene* scene);
    vector<Texture> loadMaterialTextures(aiMaterial* mat,
aiTextureType type, string typeName);
};
```

Model

The Model class contains a vector of Mesh objects and requires a file location in its constructor. It then loads the file via the loadModel function that is called in the constructor.

The private functions are all designed to process a part of Assimp's import routine. The directory of the file path is also stored since it is needed for loading textures.

The Draw function basically loops over each of the meshes to call their respective Draw function:

```
// draws the model, and thus all its meshes
void Draw(Shader& shader)
{
    for (unsigned int i = 0; i < meshes.size(); i++)
        meshes[i].Draw(shader);
}
```

Mesh

Importing a 3D model into OpenGL

To import a model and translate it to the presented structure, the appropriate headers of Assimp are needed:

```
#include <assimp/Importer.hpp>
#include <assimp/scene.h>
#include <assimp/postprocess.h>
```

Mesh

Importing a 3D model into OpenGL

The `loadModel` function is the first function directly called from the constructor. Within `loadModel`, Assimp is used to load the model into a data structure of Assimp called scene object, i.e., the root object of Assimp's data interface. The great thing about Assimp is that it neatly abstracts from all the technical details of loading all the different file formats:

```
Assimp::Importer importer;
const aiScene* scene = importer.ReadFile(path, aiProcess_Triangulate |
aiProcess_GenSmoothNormals | aiProcess_FlipUVs |
aiProcess_CalcTangentSpace);
```

Mesh

Importing a 3D model into OpenGL

First an `Importer` object is declared from Assimp's namespace and then its `ReadFile` function is called. The function expects a file path and several post-processing options as its second argument. In this way Assimp is forced to do extra calculations/operations on the imported data:

- `aiProcess_Triangulate`: if the model does not (entirely) consist of triangles, it should transform all the model's primitive shapes to triangles first
- `aiProcess_FlipUVs`: flips the texture coordinates on the y-axis where necessary during processing
- `aiProcess_GenNormals`: creates normal vectors for each vertex if the model doesn't contain normal vectors.
- `aiProcess_SplitLargeMeshes`: splits large meshes into smaller sub-meshes which is useful if your rendering has a maximum number of vertices allowed and can only process smaller meshes.
- `aiProcess_OptimizeMeshes`: does the reverse by trying to join several meshes into one larger mesh, reducing drawing calls for optimization.

postprocess.h File Reference: https://assimp.sourceforge.net/lib_html/postprocess_8h.html

Mesh

Importing a 3D model into OpenGL

The complete `loadModel` function is listed here:

```
void loadModel(string const& path)
{
    Assimp::Importer importer;
    const aiScene* scene = importer.ReadFile(path, aiProcess_Triangulate | 
aiProcess_GenSmoothNormals | aiProcess_FlipUVs | aiProcess_CalcTangentSpace);

    if (!scene || scene->mFlags & AI_SCENE_FLAGS_INCOMPLETE || !scene->mRootNode)
    {
        cout << "ERROR::ASSIMP:: " << importer.GetErrorString() << endl;
        return;
    }

    directory = path.substr(0, path.find_last_of('/'));

    processNode(scene->mRootNode, scene);
}
```

Mesh

Importing a 3D model into OpenGL

If the model loading completes successfully, all of the scene's nodes have to be processed. Only the first node (root node) is passed to the recursive `processNode` function. Because each node (possibly) contains a set of children first the node in question has to be processed, and then the processing can move to all the node's children. This behavior fits a recursive structure, so a recursive function is defined for this aim. A recursive function is a function that executes some processing and recursively calls the same function with different parameters until a certain condition is met. In this case, the exit condition is met when all nodes have been processed.

Mesh

Importing a 3D model into OpenGL

In the Assimp's structure, each node contains a set of mesh indices. Each index points to a specific mesh located in the scene object. Hence, the `processNode` function retrieves these mesh indices, retrieves each mesh, process each mesh, and then does this all again for each of the node's children nodes. The content of the function is shown below:

```
void processNode(aiNode* node, const aiScene* scene)
{
    for (unsigned int i = 0; i < node->mNumMeshes; i++)
    {
        aiMesh* mesh = scene->mMeshes[node->mMeshes[i]];
        meshes.push_back(processMesh(mesh, scene));
    }
    for (unsigned int i = 0; i < node->mNumChildren; i++)
    {
          processNode(node->mChildren[i], scene);
    }
}
```

Mesh

Assimp to Mesh

Translating an `aiMesh` object to a mesh object requires to access each of the mesh's relevant properties and store them in the Mesh object.

The general structure of the `processMesh` function becomes:

```
Mesh processMesh(aiMesh* mesh, const aiScene* scene)
{
    vector<Vertex> vertices;
    vector<unsigned int> indices;
    vector<Texture> textures;

    for (unsigned int i = 0; i < mesh->mNumVertices; i++)
    {
        Vertex vertex;
        // 1. process vertex positions, normals and texture
coordinates
        [...]
        vertices.push_back(vertex);
    }
    // 2. process indices
    [...]

    // 3. process material
    if (mesh->mMaterialIndex >= 0)
   {
        [...]
    }

    return Mesh(vertices, indices, textures);
}
```

Mesh

Assimp to Mesh

Retrieving the vertex data is relatively simple: a Vertex struct is defined that is added to the vertices array after each loop iteration. A loop is executed for as many vertices as there exist within the mesh (retrieved via mesh->mNumVertices). Within the iteration, this struct is filled with all the relevant data. For vertex positions this is done as follows:

```
glm::vec3 vector;
vector.x = mesh->mVertices[i].x;
vector.y = mesh->mVertices[i].y;
vector.z = mesh->mVertices[i].z;
vertex.Position = vector;
```

A temporary `vec3` is defined for transferring Assimp's data to, since Assimp maintains its own data types for vector, matrices and strings that can not be directly converted to glm's data types.

Mesh

Assimp to Mesh

Similarly, the procedure for the normals is:

```
vector.x = mesh->mNormals[i].x;
vector.y = mesh->mNormals[i].y;
vector.z = mesh->mNormals[i].z;
vertex.Normal = vector;
```

Texture coordinates are roughly the same, but Assimp allows a model to have up to 8 different texture coordinates per vertex. In this example only the first set of texture coordinates is used.

```
if (mesh->mTextureCoords[0])
{
    glm::vec2 vec;
    vec.x = mesh->mTextureCoords[0][i].x;
    vec.y = mesh->mTextureCoords[0][i].y;
    vertex.TexCoords = vec;
}
else
    vertex.TexCoords = glm::vec2(0.0f, 0.0f);
```

Mesh

Indices

Assimp's interface defines each mesh as having an array of faces, where each face represents a single primitive. In this example, faces are always triangles, due to the use of the `aiProcess_Triangulate` option. A face contains the indices of the vertices and the order for drawing its primitive. Therefore, it is necessary to iterate over all the faces and store all the face's indices in the indices vector.

```
for (unsigned int i = 0; i < mesh->mNumFaces; i++)
{
    aiFace face = mesh->mFaces[i];
    for (unsigned int j = 0; j < face.mNumIndices; j++)
        indices.push_back(face.mIndices[j]);
}
```

Mesh

Material

Similar to nodes, a mesh only contains an index to a material object. To retrieve the material of a mesh, it is first necessary to index the scene's `mMaterials` array. he mesh's material index is set in its `mMaterialIndex` property, that can be also queried to check if the mesh contains a material or not.

```
if (mesh->mMaterialIndex >= 0)
{
    aiMaterial* material = scene->mMaterials[mesh->mMaterialIndex];
    vector<Texture> diffuseMaps = loadMaterialTextures(material,
aiTextureType_DIFFUSE, "texture_diffuse");

    textures.insert(textures.end(), diffuseMaps.begin(), diffuseMaps.end());
vector<Texture> specularMaps = loadMaterialTextures(material, aiTextureType_SPECULAR,
"texture_specular");

    textures.insert(textures.end(), specularMaps.begin(), specularMaps.end());
}
```

Mesh

Material

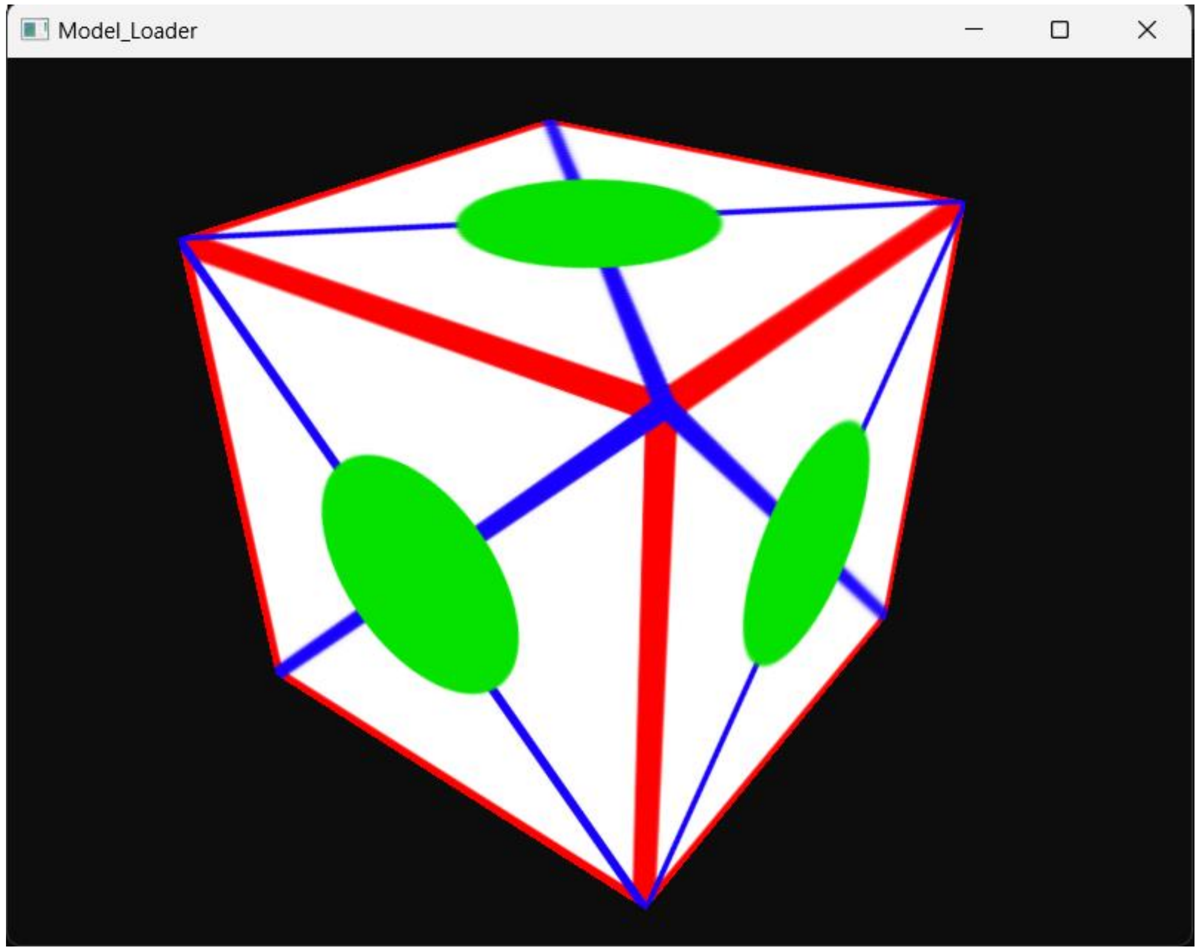
The `loadMaterialTextures` function iterates over all the texture locations of the given texture type, retrieves the texture's file location and then loads and generates the texture and stores the information in a Vertex struct.

```
vector<Texture> loadMaterialTextures(aiMaterial* mat, aiTextureType type, string typeName)
{
    vector<Texture> textures;
    for (unsigned int i = 0; i < mat->GetTextureCount(type); i++)
    {
        aiString str;
        mat->GetTexture(type, i, &str);
        Texture texture;
        texture.id = TextureFromFile(str.C_Str(), directory);
        texture.type = typeName;
        texture.path = str;
        textures.push_back(texture);
    }
    return textures;
}
```

Blender and OpenGL (Assimp)

Model Loder demo

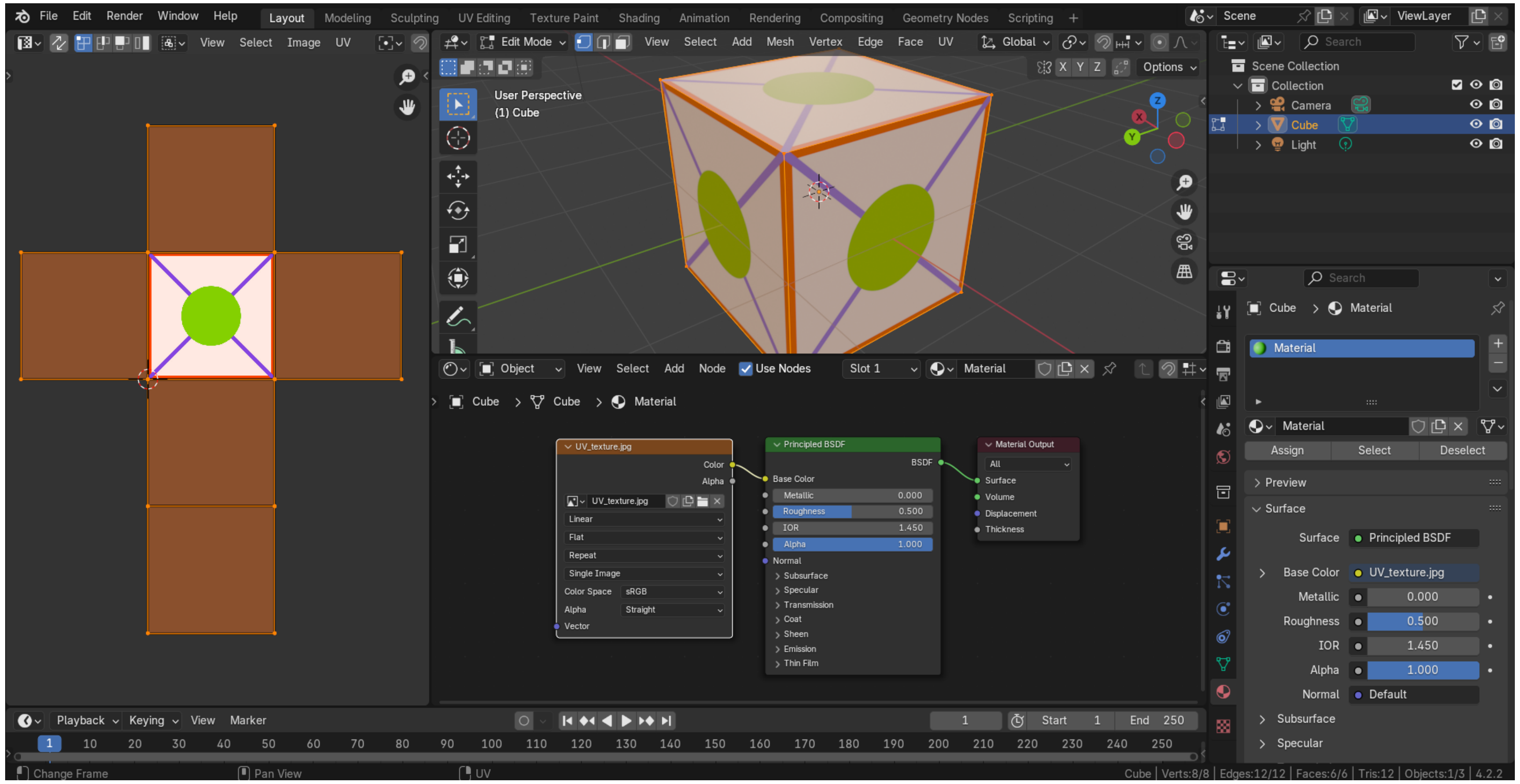
This example shows the overall workflow to create a mesh in Blender, export it as a .obj file, import it into the OpenGL application, and draw it in the scene.



Blender and OpenGL (Assimp)

Creating a textured 3D model using Blender

The first step is to create the 3D textured model in Blender. After modeling the UV mapping technique can be leveraged to texture it.



UV_texture.jpg: Color, Alpha; Linear; Flat; Repeat; Single Image; Color Space: sRGB; Alpha: Straight; Vector

Principled BSDF: BSDF; Base Color; Metallic 0.000; Roughness 0.500; IOR 1.450; Alpha 1.000; Normal; Subsurface; Specular; Transmission; Coat; Sheen; Emission; Thin Film

Material Output: All; Surface; Volume; Displacement; Thickness

Cube > Material

Surface: Principled BSDF
Base Color: UV_texture.jpg
Metallic: 0.000
Roughness: 0.500
IOR: 1.450
Alpha: 1.000
Normal: Default
Subsurface
Specular

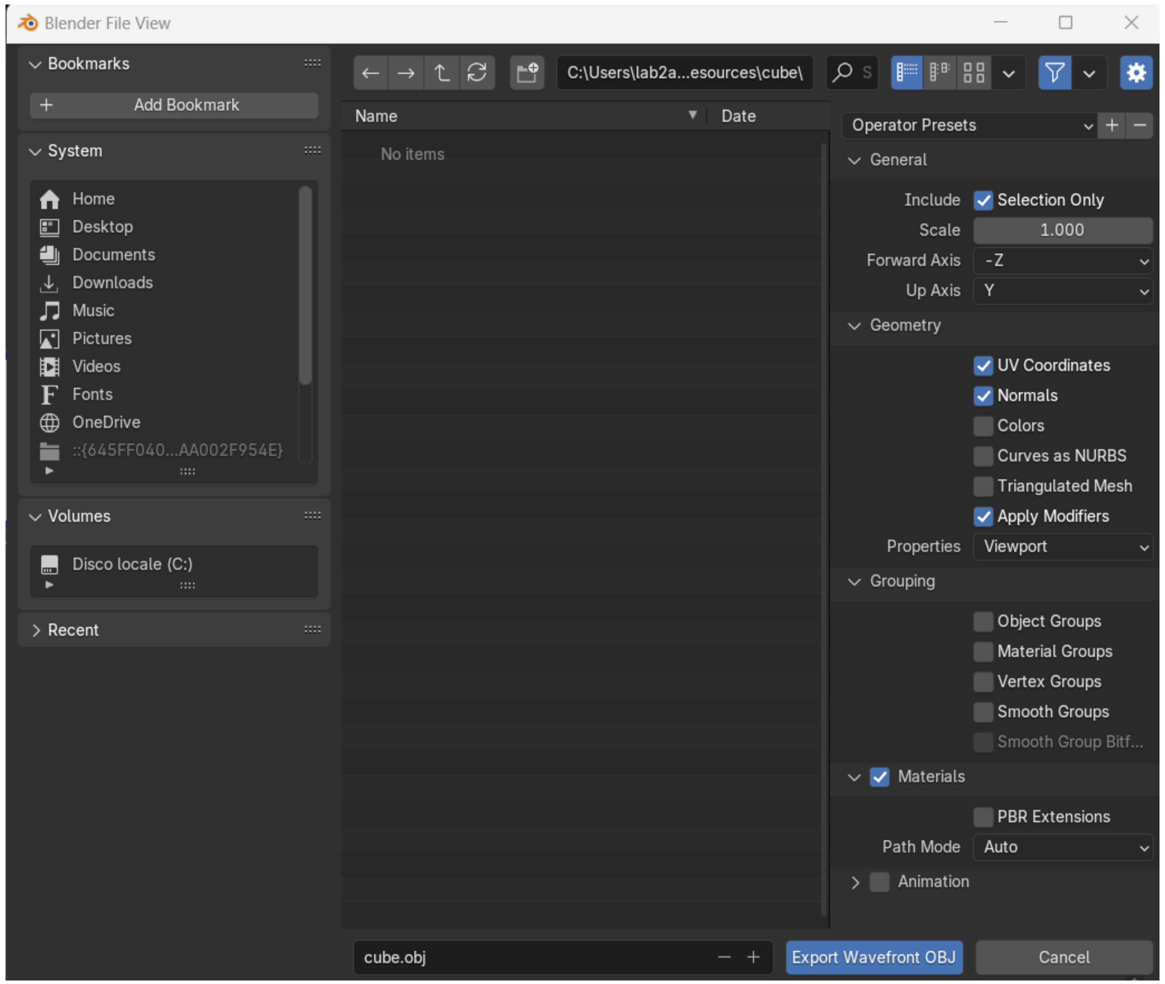
Start 1 End 250

Cube | Verts:8/8 | Edges:12/12 | Faces:6/6 | Tris:12 | Objects:1/3 | 4.2.2

Blender and OpenGL (Assimp)

Creating a textured 3D model using Blender

When creating the model, it is suggested to create a folder containing all the required assets, i.e., .blend and .png file. Move the files within the "resources" folder in the Visual Studio project. Export the 3D model using the Blender built-in functionalities (*File>>Export>>Wavefront.obj*). It is possible to export only the selected object to optimize the model loading procedure.



C:\Users\lab2a...esources\cube\

General
Include: ✓ Selection Only
Scale: 1.000
Forward Axis: -Z
Up Axis: Y

Geometry
✓ UV Coordinates
✓ Normals
☐ Colors
☐ Curves as NURBS
☐ Triangulated Mesh
✓ Apply Modifiers
Properties: Viewport

Grouping
☐ Object Groups
☐ Material Groups
☐ Vertex Groups
☐ Smooth Groups
☐ Smooth Group Bitf...

✓ Materials
☐ PBR Extensions
Path Mode: Auto

☐ Animation

cube.obj

Blender and OpenGL (Assimp)

Load the model

Using the constructor of the `Model` class, it is possible to specify the model to be loaded.

```
Model ourModel("resources/cube/cube.obj");
```

Wireframe visualization can be activated using the `glPolygonMode` function

```
glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
```

Blender and OpenGL (Assimp)

Rendering the model

The model can be rendered using the following code.

```
ourShader.use();
// view/projection transformations
glm::mat4 projection = glm::perspective(glm::radians(camera.Zoom),
(float)SCR_WIDTH / (float)SCR_HEIGHT, 0.1f, 100.0f);
glm::mat4 view = camera.GetViewMatrix();
ourShader.setMat4("projection", projection);
ourShader.setMat4("view", view);
// render the loaded model
glm::mat4 model = glm::mat4(1.0f);
model = glm::translate(model, glm::vec3(0.0f, 0.0f, 0.0f)); // translate the
model at the center of the scene
model = glm::scale(model, glm::vec3(1.0f, 1.0f, 1.0f));// scale the model down
ourShader.setMat4("model", model);
ourModel.Draw(ourShader);
```