

Computer Graphics

– Model Loading

J.-Prof. Dr. habil. Kai Lawonn

Assimp

Introduction

- So far, we mostly worked with a box
- Now, we want to work with more complicated and interesting models
- Unlike the box, we can't manually define all the vertices, normals and texture coordinates of complicated shapes like houses, vehicles or human-like characters
- Goal is to import these models into the application; models that were carefully designed by 3D artists in tools like Blender, 3DS Max or Maya

Introduction

- 3D modeling tools allow artists to create complicated shapes and apply textures (via uv-mapping)
- The tools generate the vertex coordinates, normals and texture coordinates
- This way, artists can create high quality models without having to care too much about the technical details (technical aspects are hidden in the exported model file)

Introduction

- Our job to parse these exported model files and extract all the relevant information so we can store them in a format that OpenGL understands
- A common issue is however that there are dozens of different file formats where each exports the model data in its own unique way
- Model formats like the Wavefront .obj only contains model data with minor material information like model colors and diffuse/specular maps
- Model formats like the XML-based Collada file format are extremely extensive and contain models, lights, many types of materials, animation data, cameras, complete scene information and much more
- The .obj format is generally considered to be an easy-to-parse model format

Introduction

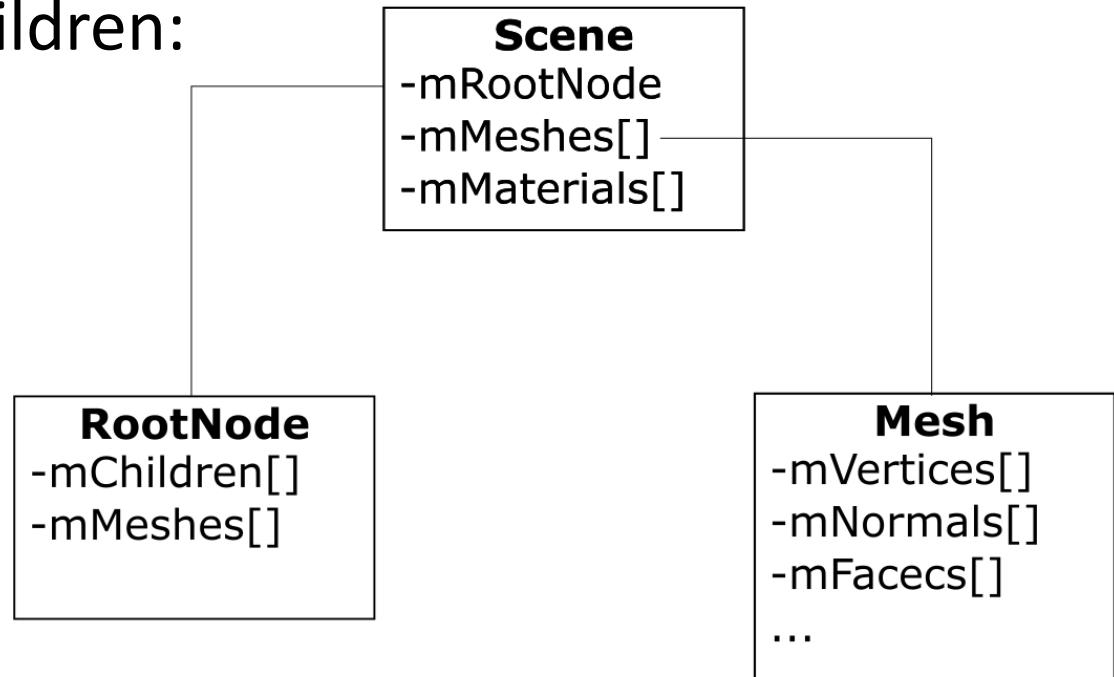
- All by all, there are many file formats where a common general structure between them usually does not exist
- Have to write an own importer if we want to import a model from these file formats
- Luckily, there just happens to be a library for this

Assimp

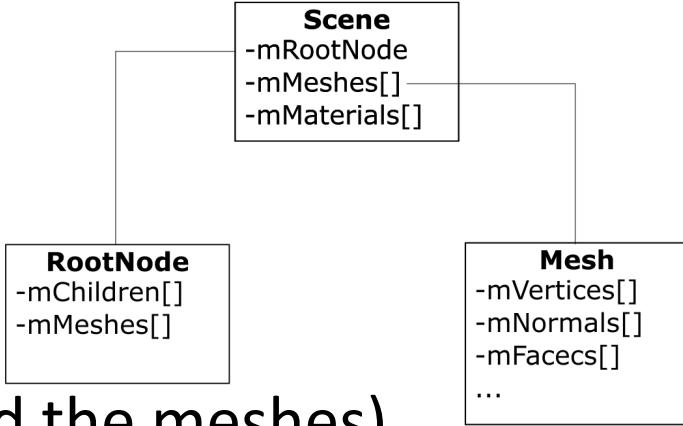
- A very popular model importing library out there is called Assimp that stands for Open Asset Import Library
- Assimp is able to import dozens of different model file formats (and export to some as well)
- As soon as Assimp has loaded the model, we can use it to process the data
- Data structure of Assimp stays the same, regardless of the type of file

Assimp

- Importing a model via Assimp it loads the entire model into a scene
- It has a collection of nodes where each node contains data
- Each node can have any number of children:



Assimp



- All the data is contained in the Scene object (materials and the meshes)
- Root node'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
- A Mesh contains all the relevant data, e.g., vertex positions, normal vectors, texture coordinates, faces and the material
- A mesh contains faces representing a render primitive (triangles, squares, points)
- A face contains the indices of the vertices that form a primitive → easy to render via an index buffer
- A mesh also contains a Material object, e.g., consists of colors and/or texture maps (like diffuse and specular maps)

Assimp

- Goal: load an object into a Scene object
- Get the Mesh object to retrieve the vertex data, indices and its material properties
- The result is then a collection of mesh data that we want to contain in a single Model object

Model

When modelling objects in modelling toolkits, artists generally do not create an entire model out of a single shape (each model has several sub-models/shapes)

Each of those single shapes that a model is composed of is called a mesh.

Think of a human-like character: artists usually model the head, limbs, clothes, weapons all as separate components and the combined result of all these meshes represents the final model.

A single mesh is the minimal representation of what we need to draw an object in OpenGL (vertex data, indices and material properties).

A model (usually) consists of several meshes.

Building Assimp

Name

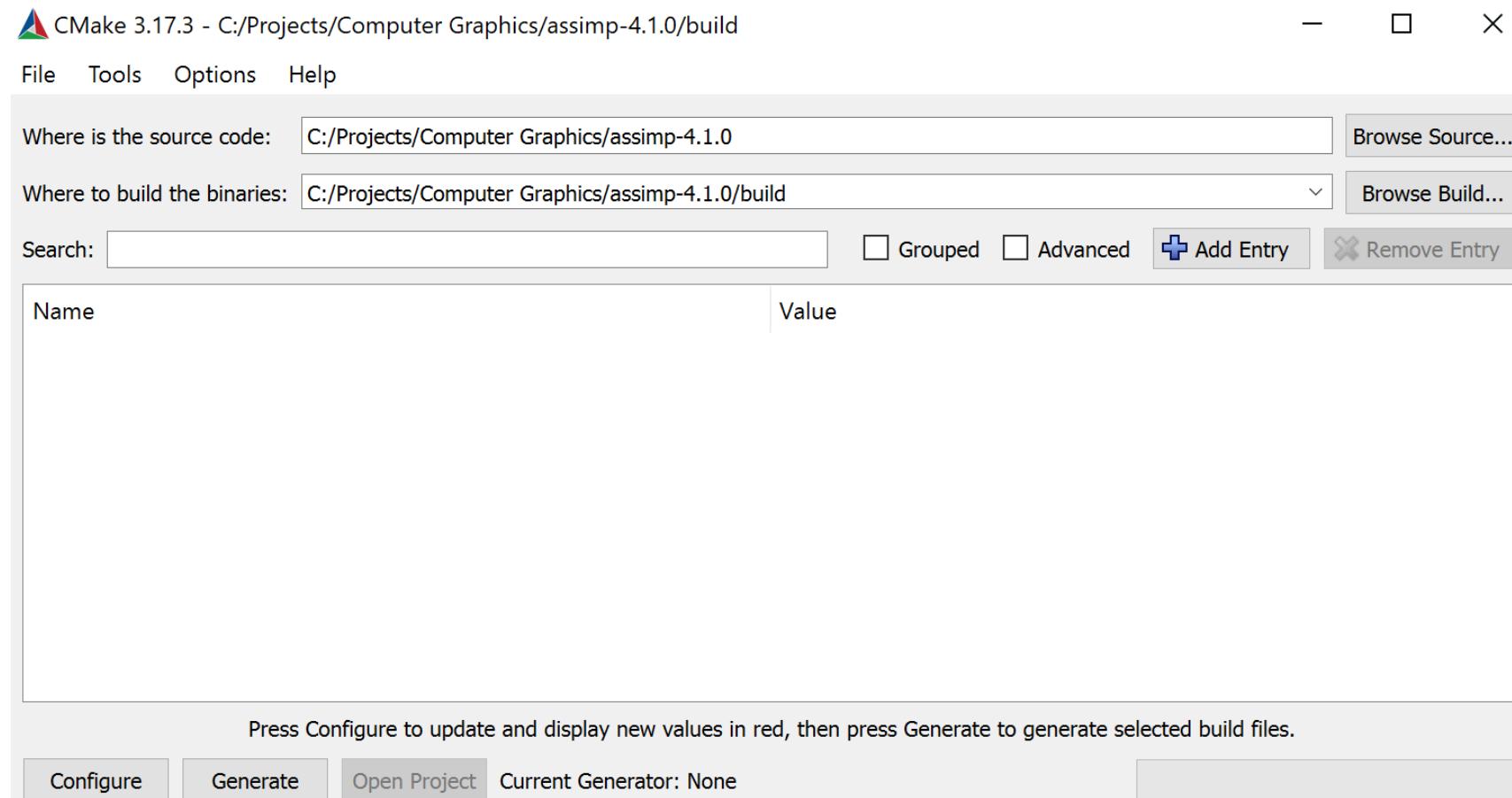
	cmake-modules
	code
	contrib
	doc
	include
	packaging
	port
	samples
	scripts
	test
	tools
	.editorconfig
	assimp.pc.in
	AssimpBuildTreeSettings.cmake.in
	AssimpConfig.cmake.in
	assimp-config.cmake.in
	AssimpConfigVersion.cmake.in
	assimp-config-version.cmake.in
	CHANGES
	CMakeLists
	CodeConventions.md
	CREDITS
	INSTALL
	LICENSE
	README
	Readme.md
	revision.h.in

Building Assimp

- First, go to <http://assimp.org/> and download the newest version
- Afterwards, unzip the folder

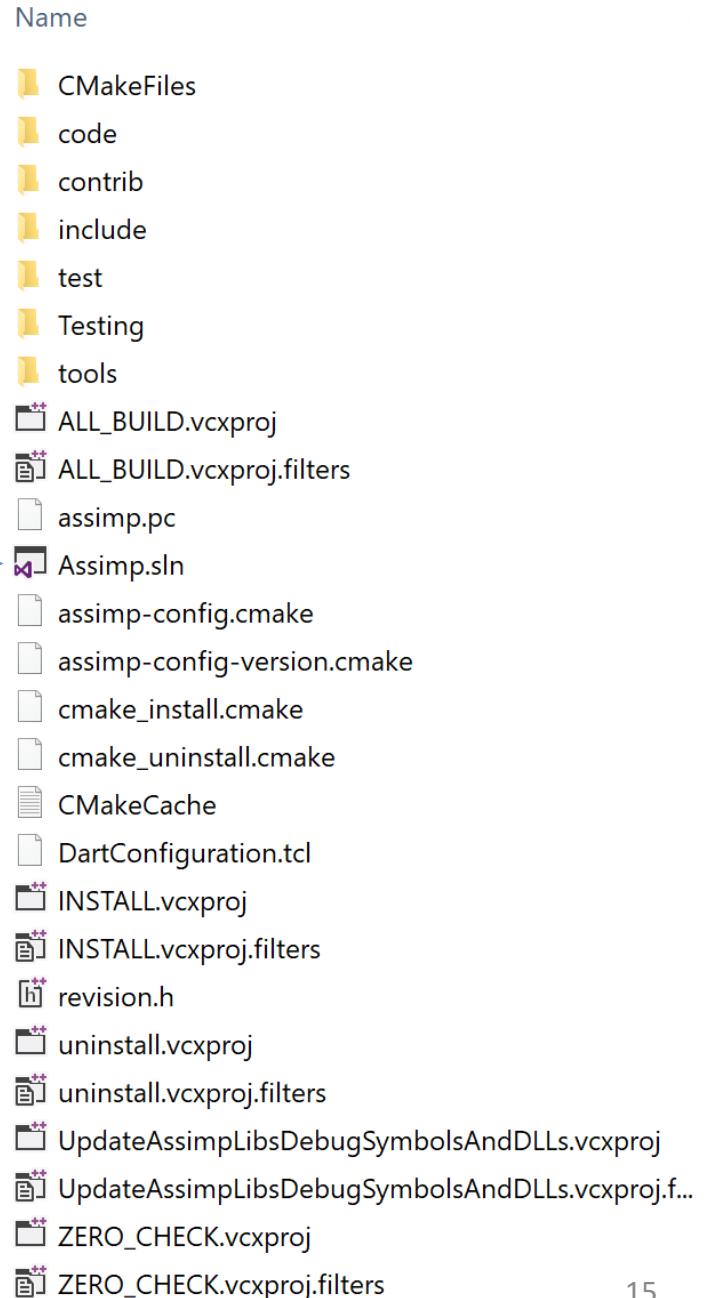
Building Assimp

- Open Cmake
- Enter the correct path
- ,Configure'
- Select the correct VS version



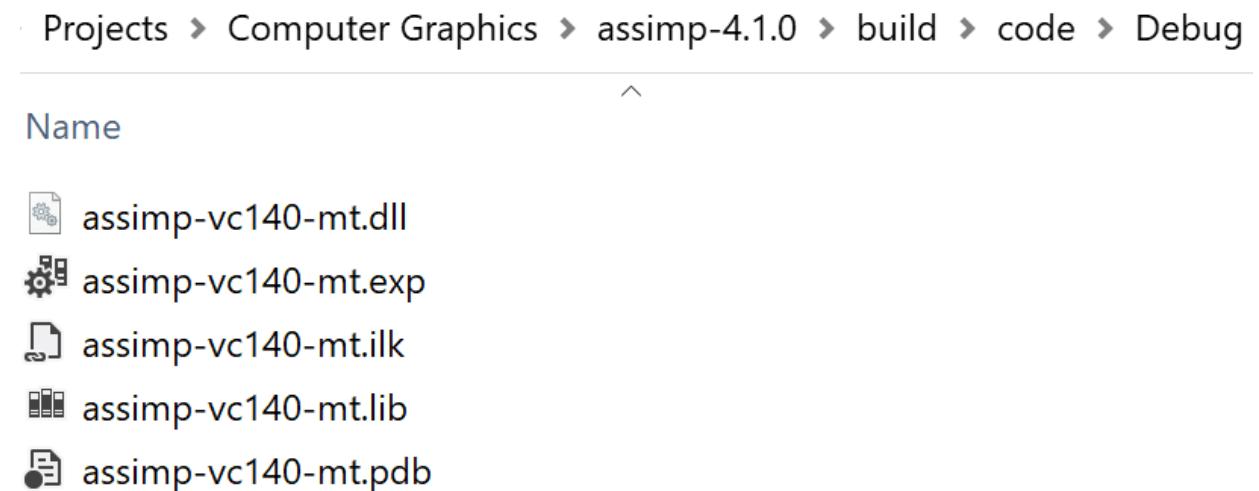
Building Assimp

- After (probably a few warnings) click ,Generate'
- In the newly created 'build' folder open Assimp.sln
- Press 'F5' and pray



Building Assimp

- Depending on the VS settings, the .dll and .lib will be in the code/Debug or code/Release folder



Building Assimp

- Two options:
- 1. The simplest approach is to copy the .dll file to the folder where the compiled .exe is located

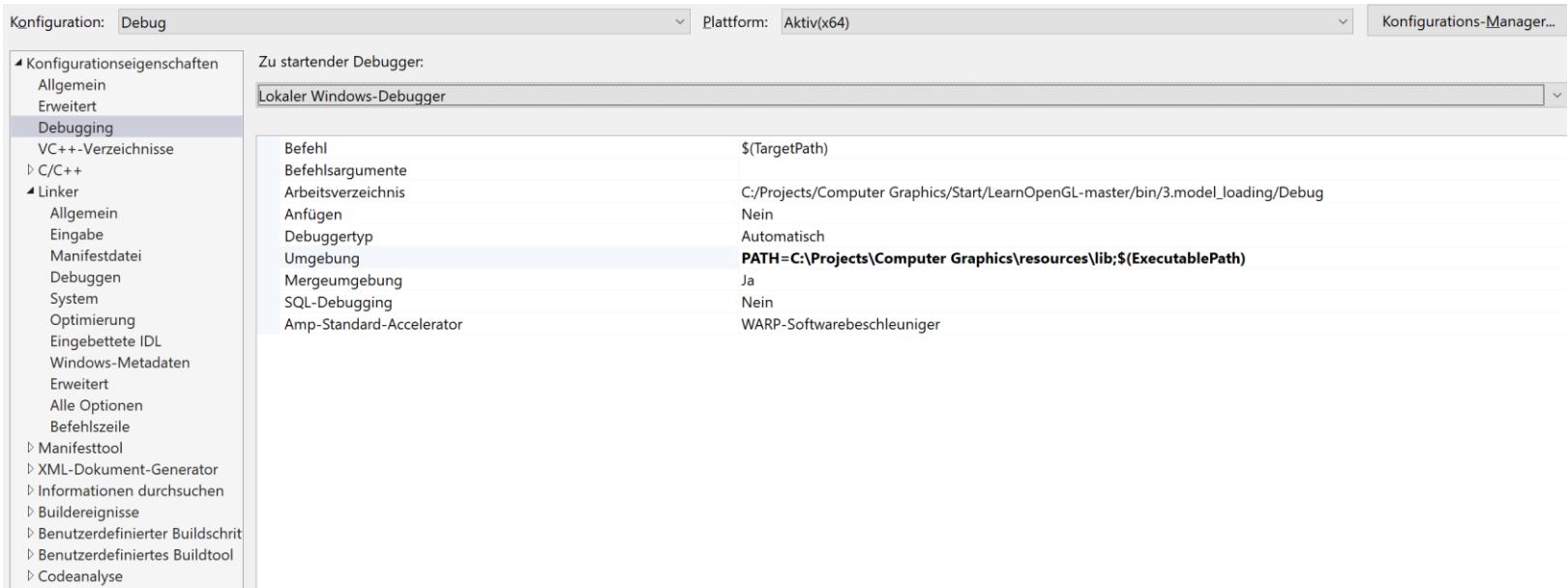
Projects > Computer Graphics > Start > FirstProject > x64 > Debug

Name

 assimp-vc140-mt.dll
 FirstProject
 FirstProject.ilk
 FirstProject.pdb

Building Assimp

- Two options:
- 2. Copy the .dll and the .lib to our lib folder; in VS go to Project>Properties>Configuration Properties>Debugging in the "Environment" property type:
PATH=C:\Projects\Computer Graphics\resources\lib;\$(ExecutablePath)



Name

	assimp-vc140-mt.dll
	assimp-vc140-mt.lib
	glfw3.dll
	glfw3.lib
	glfw3dll.lib

Building Assimp

- It helps to do the previous step in the 3.model_loading__1.model_loading project
- Once it compiles you did everything correct (otherwise you get an error, which states that the .dll cannot be found)

Mesh

Introduction

- Assimp can load many different models into the application in their own data structures
- We need to transform that data to a format that OpenGL understands so that we can render the objects
- Let's start by defining a mesh class of our own
- 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)

Introduction

- For a mesh class we define a vertex in OpenGL:

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

- The required vectors are stored in a struct called Vertex that we can use to index each of the vertex attributes

Introduction

- Aside from a Vertex struct we also want to organize the texture data in a Texture struct:

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

- We store the id of the texture and its type e.g. a diffuse texture or a specular texture

Introduction

- Knowing the actual representation of a vertex and a texture, start defining the structure of the mesh class:

```
class Mesh {  
public:  
    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:  
    unsigned int VAO, VBO, EBO;  
    void setupMesh();  
};
```

Introduction

- Constructor contains all the necessary data
- In the setupMesh function the buffers will be initialized
- The mesh will be drawn in the Draw function
- Note, that a shader is given to the Draw function → can set several uniforms before drawing (like linking samplers to texture units)

Introduction

- Constructor sets the class's public variables with the constructor's corresponding argument variables
- Then call the setupMesh function

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

    setupMesh();
}
```

Initialization

- Now, set up the setupMesh function

```
void setupMesh()
{
    glGenVertexArrays(1, &VA0);
    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);
}
```

Initialization

- Now, set up the setupMesh function

```
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));

glBindVertexArray(0);
}
```

Initialization

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

- Advantage: Structs' memory layout is sequential
- Represent a struct as an array contains the struct's variables in sequential order, which directly translates to a float (actually byte) array that we want for an array buffer
- For example, if we have a filled Vertex struct its memory layout would be equal to:

```
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);  
// = [0.2f, 0.4f, 0.6f, 0.0f, 1.0f, 0.0f, 1.0f, 0.0f];
```

Initialization

- With this, we can 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);
```

- Sizeof operator can also be used on the struct for the appropriate size in bytes (should be 32 bytes (8 floats * 4 bytes each))

Initialization

- Another advantage: a preprocessor directive called `offsetof(s,m)`
- First argument is a struct
- Second argument a variable name of the struct
- It returns the byte offset of that variable from the start:

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

- Note, also set the stride parameter equal to the size of the Vertex struct

Initialization

- Using a struct like this provides more readable code and flexibility to easily extend the structure
- Can simply add another vertex attribute we and the rendering code won't break

Rendering

- The last function to define is the Draw function
- First, want to bind the appropriate textures before calling `glDrawElements`
- Actually, difficult because we don't know how many (if any) textures the mesh has and what type they might have
- So how do we set the texture units and samplers in the shaders?

Rendering

- For this, assume certain naming convention:
 - diffuse texture: `texture_diffuseN`
 - specular texture: `texture_specularN` where N is the maximum number
- Say we have 3 diffuse textures and 2 specular textures for a particular mesh, their texture samplers should then be called:

```
uniform sampler2D texture_diffuse1;  
uniform sampler2D texture_diffuse2;  
uniform sampler2D texture_diffuse3;  
  
uniform sampler2D texture_specular1;  
uniform sampler2D texture_specular2;
```

Rendering

- By this convention, define as many texture samplers as we want in the shaders and if a mesh actually does contain (so many) textures we know what their names
- Can also process any amount of textures on a single mesh and the developer is free to use as many of those as s/he wants by simply defining the proper samplers

Rendering

There are many solutions to problems like this, don't hesitate to come up with your own creative solution.

Rendering

- The resulting drawing code then becomes:

```
void Draw(Shader &shader)
{
    unsigned int diffuseNr = 1;
    unsigned int specularNr = 1;
    for(unsigned int i = 0; i < textures.size(); i++)
    {
        glBindTexture(GL_TEXTURE0 + i);
        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++);
```

Rendering

- The resulting drawing code then becomes:

```
    glUniform1i(glGetUniformLocation(shader.ID, (name +
number).c_str()), i);
    glBindTexture(GL_TEXTURE_2D, textures[i].id);
}
glBindVertexArray(VAO);
glDrawElements(GL_TRIANGLES, indices.size(), GL_UNSIGNED_INT, 0);
glBindVertexArray(0);
glActiveTexture(GL_TEXTURE0);
}
```

Rendering

- First calculate the N-component per texture type and concatenate it to the texture's type string to get the appropriate uniform name
- We then locate the appropriate sampler, give it the location value to correspond with the currently active texture unit and bind the texture
- This is also the reason we need the shader in the Draw function
- Also added "material." to the resulting uniform name because we usually store the textures in a material struct

Rendering

Remember:

variable++ returns the variable and then increments it
++variable increments the variable and then returns it

Model

Introduction

- Now, we create another class that represents a model in its entirety, that is, a model that contains multiple meshes, possibly with multiple objects, e.g., a house, that contains a wooden balcony, a tower and perhaps a swimming pool
- We'll load the model via Assimp and translate it to multiple Mesh objects we've created previously

Introduction

- Class structure of the Model class:

```
class Model
{
public:
    /* Functions */
    Model(char* path)
    {
        loadModel(path);
    }
    void Draw(Shader shader);
private:
    /* Model Data */
    vector<Mesh> meshes;
    string directory;
    /* Functions */
    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);
};
```

Introduction

- The Model class contains a vector of Mesh objects and requires us to give it a file location in its constructor
- It then loads the file right away 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
- Also store the directory of the file path that is later needed when loading textures

Introduction

- The Draw function is nothing special and basically loops over each of the meshes to call their respective Draw function

```
void Draw(Shader &shader)
{
    for(unsigned int i = 0; i < meshes.size(); i++)
        meshes[i].Draw(shader);
}
```

Importing a 3D model into OpenGL

- To import a model and translate it to our own structure, need to include the appropriate headers of Assimp:

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

Importing a 3D model into OpenGL

- First, loadModel is called directly from the constructor
- Within loadModel use Assimp to load the model
- Assimp abstracts from all the technical details of loading all the different file formats and does all this with a single one-liner:

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

Importing a 3D model into OpenGL

- First declare an actual Importer object from Assimp's namespace
`(Assimp::Importer importer;)`
- Then call its ReadFile (`importer.ReadFile`) function (expects a file path and post-processing options)
- Assimp allows to specify several options to do some extra calculations/operations on the imported data
 - `aiProcess_Triangulate`: if the model does not (entirely) consist of triangles it will be afterwards
 - `aiProcess_FlipUVs`: flips the texture coordinates on the y-axis (most images in OpenGL were reversed)

Importing a 3D model into OpenGL

- A few other useful options are:
- `aiProcess_GenNormals` : creates normals for each vertex (if not available)
- `aiProcess_SplitLargeMeshes` : splits large meshes into smaller sub-meshes, useful if rendering has a maximum number of vertices
- `aiProcess_OptimizeMeshes` : actually does the reverse by trying to join several meshes into one larger mesh, reducing drawing calls for optimization

Importing a 3D model into OpenGL

- The hard work lies in using the returned scene object to translate the loaded data to an array of Mesh objects
- The complete loadModel function is listed here:

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

Importing a 3D model into OpenGL

- After loading, check if the scene and the root node of the scene are not null and check one of its flags to see if the returned data is incomplete:
`(if(!scene || scene->mFlags & AI_SCENE_FLAGS_INCOMPLETE || !scene->m rootNode))`
- If this is true, report an error via the importer's GetErrorString function and return
`cout << "ERROR::ASSIMP:: " << importer.GetErrorString() << endl; return;`
- Also retrieve the directory path of the given file path
`directory = path.substr(0, path.find_last_of('/'));`
- Then pass the first node (root node) to the recursive processNode function
- First process the node in question, and then continue processing all the node's children and so on → Recursive structure

Importing a 3D model into OpenGL

- Each node contains a set of mesh indices, each points to a specific mesh located in the scene object
- Want to retrieve mesh indices, retrieve each mesh, process each mesh and then do this all again for each of the node's children nodes:

```
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);
    }
}
```

Importing a 3D model into OpenGL

- First check each of the node's mesh indices and retrieve the corresponding mesh by indexing the scene's mMeshes array

```
aiMesh* mesh = scene->mMeshes[node->mMeshes[i]];
```

- Returned mesh is passed to the processMesh function returning a Mesh

```
meshes.push_back(processMesh(mesh, scene));
```

- Afterwards, iterate all of the node's children and call the same processNode

```
processNode(node->mChildren[i], scene);
```

Importing a 3D model into OpenGL

We could basically forget about processing any of the nodes and simply loop through all of the scene's meshes directly without doing all this complicated stuff with indices

The reason is that it defines a parent-child relation between meshes

By recursively iterating through these relations we can actually define certain meshes to be parents of other meshes

To translate a car mesh, we can make sure that all its children (an engine mesh, a steering wheel mesh and its tire meshes) translate as well

Assimp to Mesh

- Translating an aiMesh object to a mesh object is not difficult
- Access each of the mesh's relevant properties and store them in our own object

Assimp to Mesh

- The general structure of the processMesh function then 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;
        ...
        vertices.push_back(vertex);
    }
    ...
    if (mesh->mMaterialIndex >= 0)
    {
        ...
    }
    return Mesh(vertices, indices, textures);
}
```

Assimp to Mesh

- Processing a mesh basically consists of 3 sections: retrieving all the vertex data, retrieving the mesh's indices and finally retrieving the relevant material data
- The processed data is stored in one of the 3 vectors and from those a Mesh is created and returned to the function's caller

Assimp to Mesh

- Retrieving the vertex data is pretty simple: define a Vertex struct that add to the vertices array after each iteration
- We loop for as much vertices there exist within the mesh (retrieved via `mesh->mNumVertices`)
- Within the iteration fill this struct 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;
```

Assimp to Mesh

Assimp calls their vertex position array `mVertices`

Assimp to Mesh

- Similar for the normals:

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

Assimp to Mesh

- Texture coordinates are roughly the same, but Assimp allows a model to have up to 8 different texture coordinates per vertex
- Only care about the first set of texture coordinates also check if the mesh actually contains texture coordinates:

```
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);
```

Assimp to Mesh

- The vertex struct is now completely filled with the required vertex attributes and we can push it to the back of the vertices vector:

```
vertices.push_back(vertex);
```

Indices

- Assimp defined each mesh having an array of faces representing a single primitive (triangles in our case → aiProcess_Triangulate)
- A face contains the indices that define the vertices needed to draw
- So 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]);
}
```

Material

- A mesh only contains an index to a material object → need to index the scene's mMaterials array
- The mesh's material index is set in its mMaterialIndex property:

```
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());
}
```

Material

- First, retrieve the aiMaterial object from the scene's mMaterials array
- Then load the mesh's diffuse and/or specular textures
- A material object internally stores an array of texture locations for each texture type, different texture types prefixed with aiTextureType_
- Use a helper function called loadMaterialTextures to retrieve the textures from the material → returns a vector of Texture structs that we store at the end of the model's textures vector

Material

- `loadMaterialTextures` function iterates over all the texture locations, 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(), this->directory);
        texture.type = typeName;
        texture.path = str.C_Str();
        textures.push_back(texture);
    }
    return textures;
}
```

Material

- First, check the amount of textures stored in the material with GetTextureCount
- Retrieve each of the texture's file locations with GetTexture (stores the result in an aiString)
- Use TextureFromFile that loads a texture (with SOIL) and returns the texture's ID

Material

Note, we assume that texture files are in the same directory as the location of the model itself (or local to the model, e.g., model/texture) → concatenate the texture location string and the directory string from the loadModel function

Some models use absolute paths for their texture locations → won't work on each machine

In that case you probably want to manually edit the file to use local paths for the textures (if possible)

A large Optimization

- There is still a large (not completely necessary) optimization
- Most scenes re-use some textures, e.g., a house with a granite texture for its walls, could also be applied to the floor, ceilings, staircase, etc.
- Loading textures is not a cheap operation, currently a new texture is loaded and generated for each mesh even if the same texture has been loaded several times before → may be a bottleneck

A large Optimization

- Add one small tweak to the model code by storing all of the loaded textures globally, before load a texture, first check if it hasn't been loaded already
- If so, skip the entire loading routine
- To compare textures, store their path as well:

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

A large Optimization

- Then, store all the loaded textures in another vector declared at the top of the model's class file:

```
vector<Texture> textures_loaded;
```

A large Optimization

- In the loadMaterialTextures function, compare the texture path with all the textures in the `textures_loaded` vector for similarity
- If so, skip the texture loading/generation part and use the located texture struct as the mesh's texture:

```
...
mat->GetTexture(type, i, &str);
bool skip = false;
for(unsigned int j = 0; j < textures_loaded.size(); j++){
    if(std::strcmp(textures_loaded[j].path.data(), str.C_Str()) == 0)
    {
        textures.push_back(textures_loaded[j]);
        skip = true;
        break;
    }
}
if(!skip)
{
    Texture texture; ...
}
```

A large Optimization

Some versions of Assimp tend to load models quite slow when using the debug version and/or the debug mode of your IDE so be sure to test it out with release versions as well if you run into slow loading times.

No more Containers!

- Simply run the
3.model_loading__1.model_l
oading example



Own Model Load Class*

Introduction

- Let's assume you are confronted with a new file format that cannot be handled by Assimp
- This happens a lot in the field of visualization

Introduction

- For simplicity, we want to load an .obj mesh again
- This time, we write a loader of our own!

Wavefront .obj

- .obj file format is a simple that represents 3D geometry
- For our loader we need to learn about the structure of this file format

Wavefront .obj

- Anything after # is a comment

```
# this is a comment
```

- A vertex can have the coordinates (x,y,z,[w])

```
v 1.23 3.42 -2.81
v 4.91 3.37 -3.57
v 7.67 6.91 -1.21
```

- Texture coordinates can have the coordinates (u,[v, w])

```
vt 0.50 1.13
vt 1.45 1.25
vt 0.91 2.47
```

Wavefront .obj

- A vertex normal has the form (x,y,z) (not necessarily unit vector)

```
vn 0.24 0.54 -0.34  
vn 0.11 0.33 -0.33  
vn 0.27 0.11 -0.91
```

- Face elements

```
f 1 4 2  
f 1/2 5/4 2/3  
f 1/2/2 3/4/3 8/5/1  
f 1//1 2//3 3//9
```

Wavefront .obj

- The face elements uses indices used to define the face (e.g., triangle)

```
f v1 v2 v3  
f 1 4 2          # a triangle with the vertice 1, 4, 2
```

- Additionally, with texture coordinates

```
f v1/vt1 v2/vt2 v3/vt3  
f 1/2 5/4 2/3      # a triangle with the vertice 1, 5, 2 and the  
                   # texture coordinates 2, 4, 3
```

- Now, with vector normals

```
f v1/vt1/vn1 v2/vt2/vn2 v3/vt3/vn3  
f 1/2/2 3/4/3 8/5/1  # a triangle with the vertice 1, 3, 8 and the  
                   # texture coordinates 2, 4, 35  
                   # a normals 2, 3, 1
```

Wavefront .obj

- Indices with normal only

```
f v1//vn1 v2 //vn2 v3 //vn3  
f 1//1 2//3 3//9      # a triangle with the vertice 1, 2, 3  
                      # and the normals 1, 3, 9
```

Wavefront .obj

- For our loader, we assume that we have no normals given and the faces are in the format $f\ v1\ v2\ v3$

class Obj_Mesh

- We create a class Obj_Mesh, consisting of a header Obj_Mesh.h and a cpp file Obj_Mesh.cpp

```
#pragma once
#include <iostream>
#include <fstream>
#include <string>
class Obj_Mesh
{
public:
    Obj_Mesh();
    int Load(char* filename);
    void calculateNormal(float* coord1, float* coord2, float* coord3, float
normals[4]);
    float* normal;
    float* vertices;
    unsigned int* indices;...
```

class Obj_Mesh

- We create a class Obj_Mesh, consisting of a header Obj_Mesh.h and a cpp file Obj_Mesh.cpp

```
...
unsigned int TotalPointComponents;
unsigned int TotalTriangles;
}
```

class Obj_Mesh

- Constructor:

```
Obj_Mesh::Obj_Mesh()
{
    this->TotalPointComponents = 0;
    this->TotalTriangles = 0;
}
```

class Obj_Mesh

- We start with the load function:

```
int Obj_Mesh::Load(char* filename)
{
    std::string line;
    std::ifstream objFile(filename);

    if (objFile.is_open())// If obj file is open, continue
    {...}
```

class Obj_Mesh

- We start with the load function:

```
objFile.seekg(0, objFile.end);      // Go to the end of file
long fileSize = objFile.tellg();    // get the size of the file
objFile.seekg(0, objFile.beg);     // Back to the beginning

vertices = (float*)malloc(fileSize); // Allocate memory for vertices, ...
indices = (unsigned int*)malloc(fileSize * sizeof(unsigned int));
normals = (float*)malloc(fileSize * sizeof(float));

while (!objFile.eof())
{
```

class Obj_Mesh

- We start with the load function:

```
std::getline(objFile, line); // Get a line from file

if (line.c_str()[0] == 'v' && line.c_str()[1] == ' ') // vertex?
{
    line[0] = ' '; // Set first character empty to use sscanf

    sscanf_s(line.c_str(), "%f %f %f ", // Read floats and set vertices
    &vertices[TotalPointComponents],
    &vertices[TotalPointComponents + 1],
    &vertices[TotalPointComponents + 2]);

    normals[TotalPointComponents] = 0;           // Set normal to 0 (important later)
    normals[TotalPointComponents + 1] = 0;
    normals[TotalPointComponents + 2] = 0;

    TotalPointComponents += 3; // Add 3
```

class Obj_Mesh

- We start with the load function:

```
if (line.c_str()[0] == 'f' && line.c_str()[1] == ' ') // face?  
{  
    line[0] = ' ';  
  
    int tmpIndices[4] = { 0, 0, 0 };  
    sscanf_s(line.c_str(), "%i %i %i", // f 1 2 3  
             &tmpIndices[0],  
             &tmpIndices[1],  
             &tmpIndices[2]);  
  
    tmpIndices[0] -= 1; // OBJ starts from 1  
    tmpIndices[1] -= 1;  
    tmpIndices[2] -= 1;  
  
    indices[3 * TotalTriangles + 0] = tmpIndices[0];  
    indices[3 * TotalTriangles + 1] = tmpIndices[1];  
    indices[3 * TotalTriangles + 2] = tmpIndices[2];  
    TotalTriangles += 1; }
```

class Obj_Mesh

- We start with the load function:

```
    }
}
else
{
std::cout << "Unable to open file";
return 0;
}
objFile.close();
```

class Obj_Mesh

- Calculate the normal

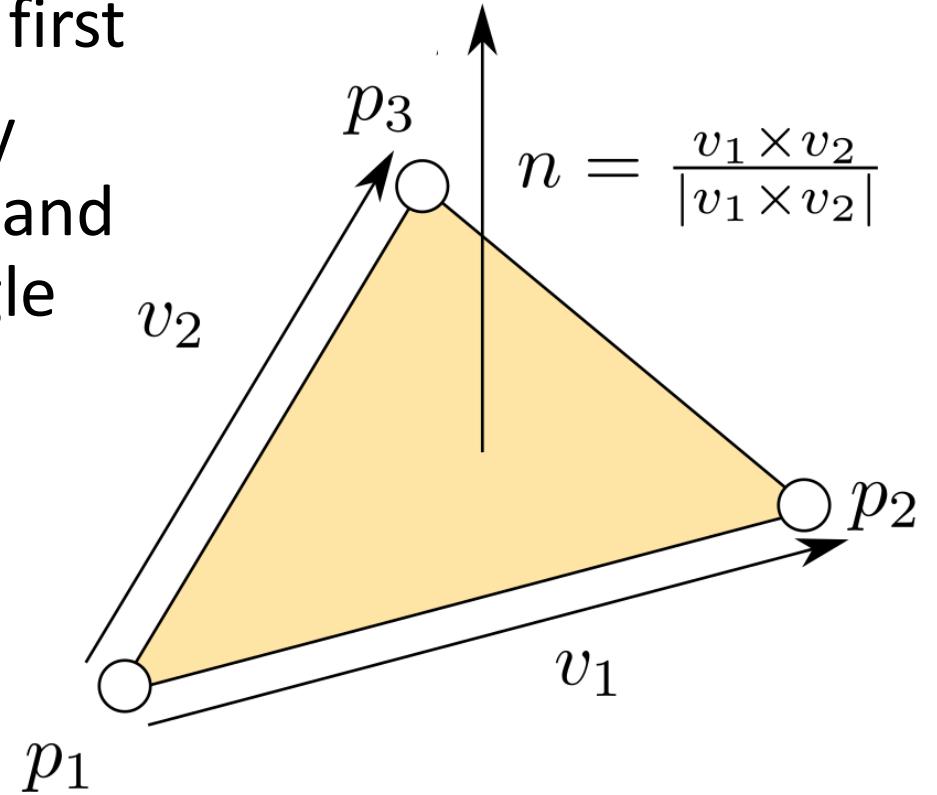
```
for (int i = 0; i < TotalTriangles; i++)
{
    int ind1 = indices[3 * i];
    int ind2 = indices[3 * i + 1];
    int ind3 = indices[3 * i + 2];

    float coord1[3] = { vertices[3 * ind1 + 0], vertices[3 * ind1 + 1], vertices[3 * ind1 + 2] };
    float coord2[3] = { vertices[3 * ind2 + 0], vertices[3 * ind2 + 1], vertices[3 * ind2 + 2] };
    float coord3[3] = { vertices[3 * ind3 + 0], vertices[3 * ind3 + 1], vertices[3 * ind3 + 2] };

    float norm[4];
    this->calculateNormal(coord1, coord2, coord3, norm);
}
```

class Obj_Mesh

- We calculate the normal of the triangle first
- The normal of a vertex is determined by adding the normal of incident triangles and weight them with the area of the triangle



class Obj_Mesh

- Calculate the normal

```
void Obj_Mesh::calculateNormal(float* coord1, float* coord2, float* coord3, float normals[4])
{
    float va[3], vb[3], vr[3], val;
    va[0] = coord1[0] - coord2[0];
    va[1] = coord1[1] - coord2[1];
    va[2] = coord1[2] - coord2[2];
    vb[0] = coord1[0] - coord3[0];
    vb[1] = coord1[1] - coord3[1];
    vb[2] = coord1[2] - coord3[2];
    /* cross product */
    vr[0] = va[1] * vb[2] - vb[1] * va[2];
    vr[1] = vb[0] * va[2] - va[0] * vb[2];
    vr[2] = va[0] * vb[1] - vb[0] * va[1];
    /* normalization factor */
    val = sqrt(vr[0] * vr[0] + vr[1] * vr[1] + vr[2] * vr[2]);

    normals[0] = vr[0] / val;
    normals[1] = vr[1] / val;
    normals[2] = vr[2] / val;
    normals[3] = val/2;}
```

class Obj_Mesh

- Calculate the normal (add and weight them)

```
this->calculateNormal(coord1, coord2, coord3, norm);

normals[3 * ind1 + 0] += norm[0] * norm[3];
normals[3 * ind1 + 1] += norm[1] * norm[3];
normals[3 * ind1 + 2] += norm[2] * norm[3];

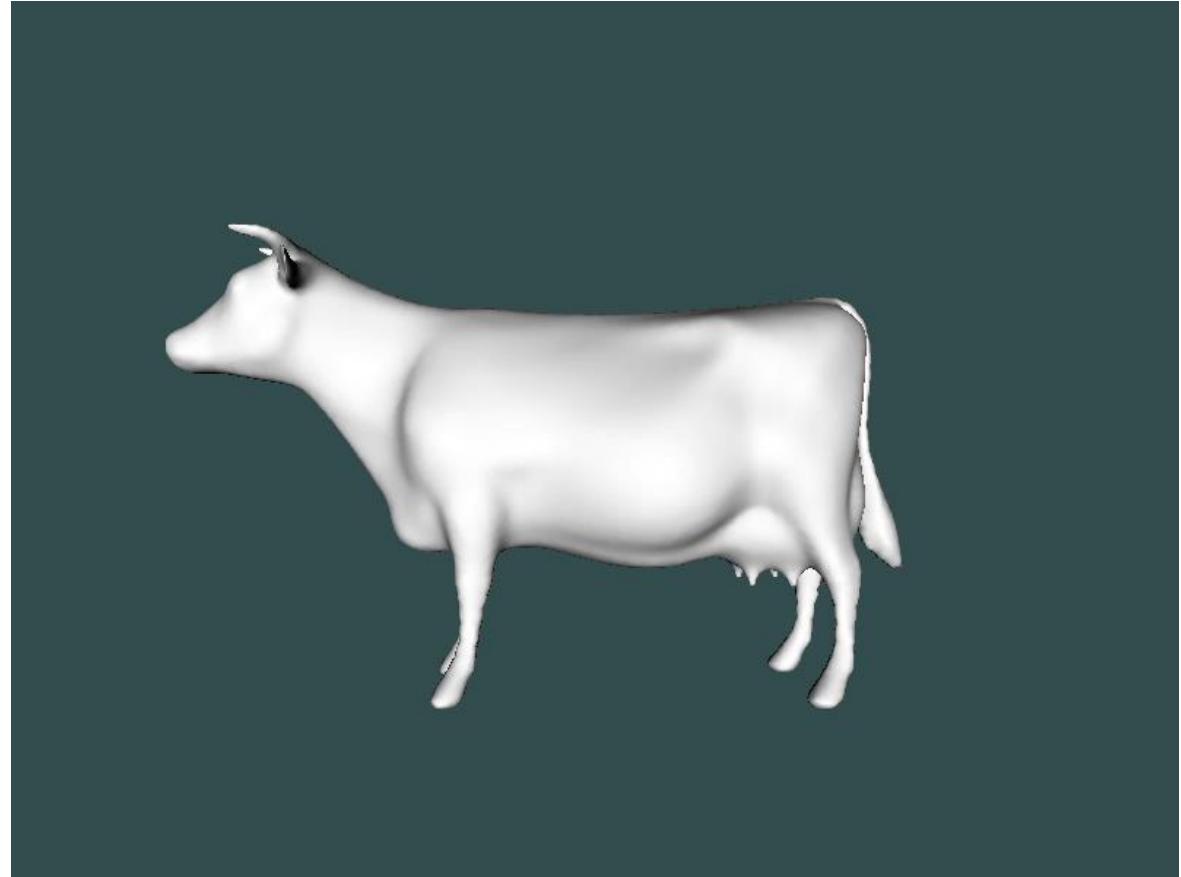
normals[3 * ind2 + 0] += norm[0] * norm[3];
normals[3 * ind2 + 1] += norm[1] * norm[3];
normals[3 * ind2 + 2] += norm[2] * norm[3];

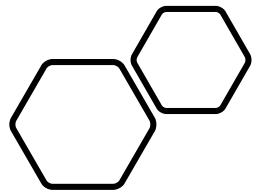
normals[3 * ind3 + 0] += norm[0] * norm[3];
normals[3 * ind3 + 1] += norm[1] * norm[3];
normals[3 * ind3 + 2] += norm[2] * norm[3];
}

return 1;
}
```

class Obj_Mesh

- Homework: Make it run
- Do not forget to add the missing variables in the class
- Bind the buffers for the vertices, normals, indices





Questions???