Computer Graphics Mid-Term - Spooky Model

Ernests Kuznecovs - 17332791

November 21, 2020

Demo: `https://www.youtube.com/watch?v=gt2GXJnMqp8`

Project Structure and Building

As well as learning OpenGL, this assignment was used as an opportunity to get with grips with C++ programming and project build process. Using Visual Studio failed provide an intuitive way to understand how it all works so a Linux environemnt with CMake and a simple text editor was used.

Project Structure

The root directory has the structure as follows:

```
CG
|-build
|-models
|-textures
|-shaders
|-libs
|-src
|-CMakeLists.txt
```

- 'build' contains the compiled binaries as well as the main binary executable.
- 'models' contain the .obj files of meshes, as well as .mtl for the textures and also a .json file that declares a mesh hierarchy.
- 'textures' contain the image files that are used as textures for the models.

- 'shaders' contain the text files for the fragment and vertex shaders.
- 'libs' contain the third party libraries that are used alongside this project.
- 'src' contains the source code of the project.
- 'CMakeLists.txt' is the file that specifies how the project should be built.

CMakeLists.txt

Using CMake to build the project helped with understanding the different types of libraries that a C++ project can use.

The root CMakeLists.txt file looks like:

```
cmake_minimum_required(VERSION 2.8)
project(CG)

add_executable(app src/main.cpp)

add_subdirectory(libs)
find_package(glfw3 3.3 REQUIRED)
find_package(assimp REQUIRED)

target_include_directories(app PRIVATE src ${GLM_INCLUDE_DIR})
target_link_libraries(app glad glfw stb assimp json ${CMAKE_DL_LIBS})
```

C++ building has two phases: compiling, and, linking. During compilation of a file, the included non implemented headers of functions are ignored and replaced by placeholders, the file is compiled into a binary, a .o file.

Once the .o files are created, all the included functions are linked from other compiled binaries, files with the suffix .o/.obj/.so/.dll/.a/. This is linking.

One way of adding a library to CMake project is by using 'find package()' this looks for the libraries and their CMake file on the machine. These libraries are already compiled, and only have to be linked with the main project. These compiled libraries can either be static(.a/.o/./obj) or dynamic(.dll/.so). Static libraries become baked in with the final binary executable, while dynamic libraries, are not and therefore the individual library files have to exist at runtime.

GLM was downloaded on the system, so 'target include directories' is used to specify where the header file is located.

The directory 'libs' also has a CMakeLists.txt file, and it declares the single header file libraries stb, and json. Since these libraries are single header files, all the code is contained in one .h file. Before compilation the c++ preprocessor replaces the included functions in the file with the functions in the header file. Since the single header files also contain the implementation, the file that includes the header files compiles everything all at once, removing the need to link.

src folder

The src folder contains the source code to structure the OpenGL project in an object-oriented fashion.

```
src
|- main.cpp
|- shader.h
|- camera.h
|- model.h
|- mesh.h
|- mesh_hierarchy.h
```

The separated files can be used as a good framework to explain the functionality of the project, this report will take advantage of that and will do so in the next sections.

main.cpp

This explanation of 'main.cpp' will serve as an overview of the project.

'main.cpp' can be thought of as the root of the project. The glfw library is used to create a window on the machine, glfw also is responsible for detecting the users mouse and keyboard events, for each type of event, a function can be defined that will be called whenever an event is triggered. These function callbacks are used to process some of the users inputs, such as adjust the matrices associated with the camera.

```
GLFWwindow* window = glfwCreateWindow(WIDTH, HEIGHT, "Computer Graphics", NULL, NULL);
// .......
glfwSetCursorPosCallback(window, mouse_callback);
```

After the window and events are set up, the two different sets of shaders are set up one for spider, and one for the plane, along with loading the spider.obj model, enabling its model hierarchy, and then loading the cyan coloured plane.

```
// spider setup
Shader spider_shader("../shaders/model.vert", "../shaders/model.frag");
char modelPath[] = "../models/spider.obj";
char modelHierarchyPath[] = "../models/spider.json";
Model spider_model(modelPath, modelHierarchyPath, true);
MeshHierarchy mh = spider_model.hierarchy;

// cyan plane setup
Shader plane_shader("../shaders/shader.vert", "../shaders/shader.frag");
float vertices[] = {
    0.5f, 0.0f, 0.5f,
    0.5f, 0.0f,-0.5f,
    -0.5f, 0.0f,-0.5f,
    -0.5f, 0.0f, 0.5f
};

unsigned int indices[] = {
  0, 1, 3,
  1, 2, 3
};
unsigned int VBO, VAO, EBO;
glGenVertexArrays(1, &VAO);
glGenBuffers(1, &VBO);
glGenBuffers(1, &EBO);
glBindVertexArray(VAO);
glBindBuffer(GL_ARRAY_BUFFER, VBO);
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EBO);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(indices), indices, GL_STATIC_DRAW);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float), (void*)0);
glEnableVertexAttribArray(0);
glBindBuffer(GL_ARRAY_BUFFER, 0);
glBindVertexArray(0);
```

Once everything is initialised, the program enters the drawing loop.
Before drawing, the screen is cleared.

```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

Then, the appropriate shader is enabled for the model that is about to be drawn

```
spider_shader.enable();
```

The projection model matrix is generated using glm::perspective. View matrix is retrieved from the camera object (initialised even before the main function). These two matrices are sent to the vertex shaders uniform matrices. Then the spider models animation cycle is triggered, which selects a hierarchy and applies a transformation to it, for this project sine and cosine functions are used to animate the spiders hierarchies.

```
glm::mat4 legs1_model = glm::mat4(1.0f);
legs1_model = glm::rotate(legs1_model, glm::radians(glm::sin(world_time * 20) * 8), gl
model_hierarchy.setTransform("set1", legs1_model);

glm::mat4 frontleft_model = glm::mat4(1.0f);
frontleft_model = glm::rotate(frontleft_model, glm::radians(glm::sin(world_time * 4) *
model_hierarchy.setTransform("Bein1Li", frontleft_model);

glm::mat4 legs2_model = glm::mat4(1.0f);
legs2_model = glm::rotate(legs2_model, glm::radians(glm::cos(world_time * 20) * 8), gl
model_hierarchy.setTransform("set2", legs2_model);
```

Once the transormation on the model has been set, the draw funtion is called on the spider model. This compiles all the transformaions in the hierarchy, and for each mesh, sends the mesh's corresponding model matrix to the vertex shader, binds the meshes vertex array object buffer and then calls draw for each of the model's meshes.

Once the spider is drawn, the shader is switched to the plane shader and projection, view and model matrices are sent to the shader, associated VAO is binded and drawn.

At the end of drawing, glfw is called to poll for keyboard events, and then the events are proccessed depending which ones are pressed.

mesh hierarchy.h

The mesh hierarchy represents how parts of the models meshes are related to each other.

A tree coressponding to the datastructure below was used to represent the hierarchy.

```
struct HierarchyNode
{
  string hierarchy_name;
  glm::mat4 transformation;
  vector<HierarchyNode*> children;

  HierarchyNode(string name) : hierarchy_name{ name }, transformation{ glm::mat4(1.0f)
};
```

Using the mesh hierarchy

The system to declare a models mesh hiearachy will use the name of each of the mesh's in the model. The declaration of the hierarchy will be done in a json file separate from the code, and will be loaded and parsed into a MeshHierarchy class and interface.

Meshes not specified by the json file will be added as a child of root.

```
{
    "set2": ["Bein4Li", "Bein3Re", "Bein2Li", "Bein1Re"],
    "set1": ["Bein4Re", "Bein3Li", "Bein2Re", "Bein1Li"],
    "legs" : ["set1", "set2"]
}
```

Each node in the hierarchy contains a transformation, that when compiled, will be applied to all of its children. The nodes are the very bottom of the tree are the meshes.

```
void compileTransforms()
{
  compileTransformsRecursive(hierarchy_nodes["root"]);
}
void compileTransformsRecursive(HierarchyNode *node)
{
  for (auto child : node->children)
    {
      child->transformation = child->transformation * node->transformation;
      compileTransformsRecursive(child);
    }
}
```

The MeshHierarchy class provides the interface of allowing the user to target which hierarchy to apply a transofmation to.

```
void setTransform(string hierarchy_name, glm::mat4 &transformation)
{
  hierarchy_nodes[hierarchy_name]->transformation = transformation;
}
```

And also retrieve the compiled transformation from a mesh.

```
glm::mat4 getModelMatrix(string mesh_name)
{
     return hierarchy_nodes[mesh_name]->transformation;
}
```

Retrieving mesh names

Using the assimp command line interface, 'assimp info <$\text{model}_{\text{file}}$>' gives a summary of the model's info including the individual mesh names.

```
Meshes:  (name) [vertices / bones / faces | primitive_types]
    0 (HLeib01): [71 / 0 / 80 | triangle]
    1 (OK): [91 / 0 / 60 | triangle]
    2 (Bein1Li): [111 / 0 / 98 | triangle]
    3 (Bein1Re): [117 / 0 / 98 | triangle]
    4 (Bein2Li): [113 / 0 / 98 | triangle]
    5 (Bein2Re): [115 / 0 / 98 | triangle]
    6 (Bein3Re): [107 / 0 / 98 | triangle]
    7 (Bein3Li): [107 / 0 / 98 | triangle]
    8 (Bein4Re): [113 / 0 / 98 | triangle]
    9 (Bein4Li): [113 / 0 / 98 | triangle]
    10 (Zahn): [9 / 0 / 7 | line]
    11 (Zahn): [33 / 0 / 28 | triangle]
    12 (klZahn): [9 / 0 / 7 | line]
    13 (klZahn): [35 / 0 / 28 | triangle]
    14 (Kopf): [79 / 0 / 90 | triangle]
    15 (Brust): [17 / 0 / 20 | triangle]
    16 (Kopf2): [79 / 0 / 90 | triangle]
    17 (Zahn2): [9 / 0 / 7 | line]
    18 (Zahn2): [33 / 0 / 28 | triangle]
```

```
19 (klZahn2): [9 / 0 / 7 | line]
20 (klZahn2): [35 / 0 / 28 | triangle]
21 (Auge): [36 / 0 / 38 | triangle]
22 (Duplicate05): [36 / 0 / 38 | triangle]
```

In this case the spider.obj is created by german modelers, hence the german names for each of the parts of the spider, e.g "Bein" meaning leg.

shader.h

The shader file contains a class that represents a single shader program. The vertex and fragment shader file paths are taken as input, the files are read, the resulting strings from the files are compiled, shader program is created, the id is retrieved, and attatched the vertex and fragment shader to it.

Each of these steps checked for the errors associated with each.

The shader class also defines functions to interact with the gl state. Functions for sending uniform matrices to the shader and also a function to activate the shader.

```
void enable()
{
    glUseProgram(shader_program_id);
}
```

camera.h

The camera class is in charge of mainting the view matrix, this matrix is a part of the model/view/projection transformations. The view matrix gives the illusion that it is only the camera that is moving, but in reality every vertex that is being drawn is multiplied by the view matrix, therefore everything moves around the camera.

- Camera position

  Just a vector in world space.

- Camera direction that it's looking at

  Camera's position vector - origin of the scene

- Vector pointing to the right

Cross product of the world space up direction and the camera direction. This gives us a vector perpendicular to both which points to the positive x axis of the camera.

- Vector pointing upwards from the camera

  Cross product of camera direction and the camera vector pointing to the right.

The glm::LookAt function can then be used to calculate the view matrix.

```
view = glm::lookAt(camera_pos, camera_pos + camera_front, camera_up);
```

The cameraPos + cameraFront will ensure that the camera looks the same directions as it moves.

Looking

The looking around is achieved through altering the camera front variable. This achieved through using euler angles, that can represent any rotation in 3d.

```
front.x = cos(glm::radians(mouse_x)) * cos(glm::radians(mouse_y));
front.y = sin(glm::radians(mouse_y));
front.z = sin(glm::radians(mouse_x)) * cos(glm::radians(mouse_y));
camera_front = glm::normalize(front);
```

The mouse event handler calls the camera class to update the $\text{camera}_{\text{front}}$ variable with the newly inputed mouse movements.

model.h and mesh.h

The model class holds the meshes, hierarchy, and textures associated with a model.

Assimp

The model class takes the filepath to a model file such as .obj, or other ones suppported by assimp.

Using Assimp the model path is loaded and assimp returns an assimp scene object. It has a data structure that of a tree. Each node in the tree contains a number of meshes, each mesh contains vertices, texture coordiantes, and, indices for each of the meshs faces (ie, the three vertices making up a triangle).

Each mesh also contains a material index, this can be used to retrieve the assimp material object, with this object the textures can be retrieved and loaded.

Meshes

For each mesh extracted from assimp, a mesh object is created, using the vertices, indices, and textures. Each mesh object has its own VAO, with a VBO, and EBO. This VAO is setup upon the initialisation of the Mesh, the three buffers are generated, the vertex data is bound to the VBO ($GL_{ARRAYBUFFER}$), the indices data is bound to the EBO ($GL_{ELEMENTARRAYBUFFER}$).

The attribute pointers are set up with the static locations defined in the vertex shader.i.e

```
layout (location = 0) in vec3 pos;
layout (location = 1) in vec3 normal;
layout (location = 2) in vec2 texture_coord;

glEnableVertexAttribArray(0);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex), (void*)0);
glEnableVertexAttribArray(1);
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex), (void*)offsetof(Vertex
glEnableVertexAttribArray(2);
glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, sizeof(Vertex), (void*)offsetof(Vertex
```

The mesh class has a draw function, this draw function binds the VAO and textures to draw and and calls draw.