John H. Rucker

CSC 4820

Home Work 2

CSC4820/6820 Interactive Computer Graphics

Fall 2014

Homework 2

Due date: 11:59 pm October 28, 2014 (Tuesday)

The aim of this homework is to help improve your understanding the process of loading and displaying a 3D model as well as the structure of an OpenGL/GLSL program.

Answer the following questions. Use your code from project 1 or this [sample program](https://docs.google.com/viewer?a=v&pid=sites&srcid=ZGVmYXVsdGRvbWFpbnxnc3Vjb21wdXRlcmdyYXBoaWNzfGd4Ojg3MDlhM2Y5NDFiMjU2Ng) as examples. If possible, use code segments as examples to answer the questions.

* Briefly describe the process of loading a 3D model and display it using OpenGL/GLSL.

Answer 🡪

* 1) Create a 3d object file in Blender or any other 3d modeling software.
* 2) After file is created load/import the file using Assimp

//===================================================================

// Importing the model with Assimp

//===================================================================

bool ImportFrom3DFile(const std::string& pFile)

{

//check if file exists

std::ifstream fin(pFile.c\_str());

if (!fin.fail())

{

fin.close();

}

else{

printf("Couldn't open file: %s\n", pFile.c\_str());

printf("%s\n", import.GetErrorString());

return false;

}

//The 3D model file is loaded into Assimp's data structure through the following function

sceneOnScreen = import.ReadFile(pFile, aiProcessPreset\_TargetRealtime\_Quality);

// Now we can access the file's contents.

// Everything will be cleaned up by the importer

return true;

* }

3) After the file is loaded into assimp it is then pushed to be displayed in the window

// Function that loads 3d model to window

void generateVAOandUBuffer(const aiScene \*fd)

{

}

* Briefly describe the following data structure stored in Assimp’s

Answer 🡪

aiScene object: - Assimp uses an aiScene object to represent the loaded mesh. The aiScene object contains mesh structures that encapsulate parts of the model model. Here the drawing of the 3d object is guided by the node tree inside of aiScene.

// The global Assimp scene object

const aiScene\* sceneOnScreen = NULL;

* node tree – the node tree describes the hierarchical relationship of the different meshes of the 3D object. Contained in aiScene.

void renderRecur(const aiScene \*fd, const aiNode\* nd) shows how to traverse the node tree in a aiScene object and draw the 3D meshes attached to each node.

* Mesh – this data structure uses Assimp to load the model and then creates vertex buffers, index buffers, and texture objects that contain the data of the model. aMesh.textIndex = texId; glGenBuffers(1, &(aMesh.blockIndex));
* Face – stores model topology data usually consisting of triangles, quadrilaterals

or other simple convex polygons.

// Create array with faces

// have to convert from Assimp format to array

unsigned int \*faceArray;

faceArray = (unsigned int \*)malloc(sizeof(unsigned int)\* mesh->mNumFaces \* 3);

unsigned int faceIndex = 0;

* face indices – Indices or elements are an easy way to refer to vertex that can be used to create primitives or refer to objects
* // Copy face indices from aiMesh to faceArray.
* for (unsigned int t = 0; t < mesh->mNumFaces; ++t) {
* const aiFace\* face = &mesh->mFaces[t]; // Go through the list of aiFace.
* Describe the relationship among node tree, mesh, vertices, faces, and face indices.

aiScene (Data Structure)🡪(contains) 🡪node tree🡪(stores in arrays aiMesh)mesh🡪vertices 🡪 faces🡪face indices

// Copy face indices from aiMesh to faceArray.

for (unsigned int t = 0; t < mesh->mNumFaces; ++t) {

const aiFace\* face = &mesh->mFaces[t]; // Go through the list of aiFace

// For each aiFace, copy its indices to faceArray.

memcpy(&faceArray[faceIndex], face->mIndices, 3 \* sizeof(unsigned int));

faceIndex += 3;

}

* For basic 3D model display, what do we really want to retrieve from an aiScene object? We want to retrieve the mesh. Once the mesh is loaded successfully, we get a pointer to an aiScene object. This object contains the entire model contents, divided into aiMesh structures.
* //===================================================================
* // Importing the model with Assimp
* //===================================================================
* bool ImportFrom3DFile(const std::string& pFile)
* {
* //The 3D model file is loaded into Assimp's data structure through the following function
* sceneOnScreen = import.ReadFile(pFile, aiProcessPreset\_TargetRealtime\_Quality);
* aiVector3D display\_min, display\_max, scene\_center;
* get\_container(&display\_min, &display\_max);
* float temp;
* temp = display\_max.x - display\_min.x;
* temp = display\_max.y - display\_min.y > temp ? display\_max.y - display\_min.y : temp;
* temp = display\_max.z - display\_min.z > temp ? display\_max.z - display\_min.z : temp;
* modelWindowSize = 8.f / temp; // Model zoom percentage
* // Everything will be cleaned up by the importer
* return true;
* }

What are vertex arrays and element arrays?

A vertex array an array used to store vertex data such as vertex positions, normal, texture coordinates and color information. // Generate A Vertex Array Object for mesh

// Bind the VAO so it's the active one.

// This VAO contains four VBOs: index, position, normal, and texture coordinates.

glGenVertexArrays(1, &(aMesh.vao));

glBindVertexArray(aMesh.vao);

A element array (index buffers) an array that enables to reuse the same vertex over and over again. Stores indices.

// Generate and bind an index buffer for face indices (elements)

glGenBuffers(1, &buffer);

glBindBuffer(GL\_ELEMENT\_ARRAY\_BUFFER, buffer);

// Fill the buffer with indices

glBufferData(GL\_ELEMENT\_ARRAY\_BUFFER, sizeof(unsigned int)\* mesh->mNumFaces \* 3, faceArray, GL\_STATIC\_DRAW);

What are the relationship between a vertex array and an element array (index buffers) from the same object? Vertex array is a container for vertex data and the index buffer will contain only the offsets to our vertex buffer object

// Transfer the vertex position array (stored in aiMesh's member variable mNormals) to the VBO.

glBufferData(GL\_ARRAY\_BUFFER, sizeof(float) \* 3 \* mesh->mNumVertices, mesh->mNormals, GL\_STATIC\_DRAW);

// Link this VBO with a variable in the vertex shader.

glEnableVertexAttribArray(normLoc);

glVertexAttribPointer(normLoc, 3, GL\_FLOAT, 0, 0, 0);

* What are Vertex Array Object (VAO) and Vertex Buffer Object (VBO)?

VAO - is a object that encapsulates all the data that is associated with the vertex processor. Instead of containing the actual data, it holds references to the vertex buffers, the index buffer and the layout specification of the vertex itself.

* // This VAO contains four VBOs: index, position, normal, and texture coordinates.

glGenVertexArrays(1, &(aMesh.vao));

glBindVertexArray(aMesh.vao);

VBO – is an OpenGL feature that provides methods for uploading vertex data position, color, texture etc. to the video card for rendering.

// Generate a Vertex Buffer Object (VBO) for vertex texture coordinates

if (mesh->HasTextureCoords(0))

{

float \*texCoords = (float \*)malloc(sizeof(float) \* 2 \* mesh->mNumVertices);

for (unsigned int k = 0; k < mesh->mNumVertices; ++k)

{

texCoords[k \* 2] = mesh->mTextureCoords[0][k].x;

texCoords[k \* 2 + 1] = mesh->mTextureCoords[0][k].y;

}

glGenBuffers(1, &buffer);

glBindBuffer(GL\_ARRAY\_BUFFER, buffer);

glBufferData(GL\_ARRAY\_BUFFER, sizeof(float) \* 2 \* mesh->mNumVertices, texCoords, GL\_STATIC\_DRAW);

glEnableVertexAttribArray(coorLoc);

glVertexAttribPointer(coorLoc, 2, GL\_FLOAT, 0, 0, 0);

}

* What's the relationship between them?

VAO’s allow you to store multiple VBO’s in one VAO. With this ability, we can now store vertex data and colour data in separate VBO’s, but in the same VAO.

So we’ve now filled in our vertices coordinates, we need to go and create a Vertex Array Object which is done with a call to glGenVertexArrays. We then need a call to glBindVertexArray to make the VAO active. Once the VAO is active, we need to call glGenBuffers to create our Vertex Buffer Object, and then we also need to bind it with glBindBuffer.

So it goes in this order:  
1. Generate Vertex Array Object glGenVertexArrays(1, &(aMesh.vao));  
2. Bind Vertex Array Object glBindVertexArray(aMesh.vao);  
3. Generate Vertex Buffer Object glGenBuffers(1, &buffer);  
4. Bind Vertex Buffer Object glBindBuffer(GL\_ELEMENT\_ARRAY\_BUFFER, buffer);

* Describe the relationship between VBOs and vertex arrays and element arrays.

VBO’s are filled with vertex array and element array data

// Generate a Vertex Buffer Object (VBO) for vertex normals

if (mesh->HasNormals())

{

glGenBuffers(1, &buffer);

glBindBuffer(GL\_ARRAY\_BUFFER, buffer);

// Transfer the vertex position array (stored in aiMesh's member variable mNormals) to the VBO.

glBufferData(GL\_ARRAY\_BUFFER, sizeof(float) \* 3 \* mesh->mNumVertices, mesh->mNormals, GL\_STATIC\_DRAW);

// Link this VBO with a variable in the vertex shader.

glEnableVertexAttribArray(normLoc);

glVertexAttribPointer(normLoc, 3, GL\_FLOAT, 0, 0, 0);

}

* How many VAOs and VBOs are created if the dog.obj model is loaded?

As many as needed to be created.

* What are vertex and fragment shaders?

Vertex Shaders

layout (std140) uniform Matrices {

uniform mat4 projMatrix;

uniform mat4 viewMatrix;

uniform mat4 modelMatrix;

};

in vec3 position;

in vec3 normal;

in vec2 texCoord;

out vec4 vertexPos;

out vec2 TexCoord;

out vec3 Normal;

void main()

{

Normal = normalize(vec3(viewMatrix \* modelMatrix \* vec4(normal,0.0)));

TexCoord = vec2(texCoord);

gl\_Position = projMatrix \* viewMatrix \* modelMatrix \* vec4(position,1.0);

}

Fragment Shaders

void main()

{

vec4 color;

vec4 amb;

float intensity;

vec3 lightDir;

vec3 n;

lightDir = normalize(vec3(1.0,1.0,1.0));

n = normalize(Normal);

intensity = max(dot(lightDir,n),0.0);

if (texCount == 0) {

color = diffuse;

amb = ambient;

}

else {

color = texture(texUnit, TexCoord);

amb = color \* 0.33;

}

out\_color = (color \* intensity) + amb;

//out\_color = vec4(texCount,0.0,0.0,1.0);

}

* In this example, what do the vertex and fragment shader do?

Vertex shader - vertex shader can manipulate the attributes of vertices.

Fragment shader is part of the rasterization step, where the image is calculated and the pixels between the vertices are filled in

* How the job of displaying a 3D model is divided between the OpenGL part and the GLSL part? Opengl part – responsible for translating model data to the

GLSL part responsible for executing code on the GPU level.

GLuint shaderConfig()

{

char \*vertshade = NULL, \*fragshade = NULL, \*fragshade2 = NULL;

GLuint p, v, f;

v = glCreateShader(GL\_VERTEX\_SHADER);

f = glCreateShader(GL\_FRAGMENT\_SHADER);

vertshade = textFileRead(vertexFileName);

fragshade = textFileRead(fragmentFileName);

What are the responsibilities of the OpenGL functions? load, compile, and link.

What are the responsibilities of the GLSL shaders? set the shaders ready for execution.

* How is VAO and VBO related to shader? VAO🡪VBO🡪shader

// Link this VBO with a variable in the vertex shader.

glEnableVertexAttribArray(vertLoc);

glVertexAttribPointer(vertLoc, 3, GL\_FLOAT, 0, 0, 0);

* What's the purpose of glBindVertexArray() and glBindBuffer()?

glBindVertexArray — bind a vertex array object

glBindBuffer — bind a named buffer object

* What's the purpose of glUseProgram()?

glUseProgram — install a program object as part of current rendering state

* What happens when you call glDrawElements()?

glDrawElements — render primitives from array data

* What's the difference between glDrawElements() and glDrawArrays()?

glDrawArrays submits the vertices in linear order, as they are stored in the vertex arrays.  
With glDrawElements you have to supply an index buffer. Indices allow you to submit the vertices in any order, and to reuse vertices that are shared between triangles.

Answer the following questions. Use your code from project 1 or this [sample program](https://docs.google.com/viewer?a=v&pid=sites&srcid=ZGVmYXVsdGRvbWFpbnxnc3Vjb21wdXRlcmdyYXBoaWNzfGd4OjNiYmUxZDJmMTg3NjlkMWE) as examples. If possible, use code segments as examples to answer the questions.

* How is the motion data stored in OpenGL and GLSL? In the Uniform Buffer
* Describe the process of transforming an object in OpenGL and GLSL.

Data is held (binded) by the VAO then released to the VBO.

How is the process of transformation divided between the OpenGL part of the program and the GLSL part of the program? Transformation in the Opengl part handles the calculations and parameter setup and the GLSL part pieces them together. Don’t by the Model, View and Projection matrices

Describe the purpose of model matrix, view matrix, and projection matrix. Separate transformations cleanly

What is the relationship between these matrices and vertex positons? They are one in the same .

* What is glm library used for? This library provides classes and functions designed and implemented following as strictly as possible the GLSL conventions and functionalities so that when a programmer knows GLSL, he knows GLM as well, making it really easy to use.
* What’s the purpose of glViewport() function? Sets the viewport.
* What is the purpose of putting a vertex through model transformation, view transformation, projection transformation, and viewport transformation?

After that, OpenGL applies the projection matrix to yield clip coordinates. This transformation defines a viewing volume; objects outside this volume are clipped so that they're not drawn in the final scene. After this point, the perspective division is performed by dividing coordinate values by w, to produce normalized device coordinates. Finally, the transformed coordinates are converted to window coordinates by applying the viewport transformation. You can manipulate the dimensions of the viewport to cause the final image to be enlarged, shrunk, or stretched.

* How do you control the animation of an object via keyboard or mouse?

By assigning keyboard or mouse commands.

void processKeys(unsigned char key, int xx, int yy)

{

double f = .05;

switch (key) {

case 27: glutLeaveMainLoop(); break;

case 'o': r -= 0.1f; break;

case 'h': r += 0.1f; break;

case 'X': xTrans += -f; break;

case 'x': xTrans += f; break;

case 'Y': yTrans += -f; break;

case 'y': yTrans += f; break;

case 'Z': zTrans += -f; break;

case 'z': zTrans += f; break;

}

cameraX = r \* sin(alpha \* 3.14f / 180.0f) \* cos(beta \* 3.14f / 180.0f);

cameraZ = r \* cos(alpha \* 3.14f / 180.0f) \* cos(beta \* 3.14f / 180.0f);

cameraY = r \* sin(beta \* 3.14f / 180.0f);

glutKeyboardFunc(processKeys);

glutSpecialFunc(processSpecialKeys);

Submit the report in text, Word, or PDF files to Desire2Learn under the dropbox “Homework2”. Write your name in the report.