Csc 8820 Advanced Graphics Algorithms

PROJECT-2

Ashish Tiwari

#include <fstream>

#include <iostream>

#include <GL/glew.h>

#include <GL/freeglut.h>

// ASSIMP library for loading 3D files

#include "assimp/Importer.hpp"

#include "assimp/PostProcess.h"

#include "assimp/Scene.h"

// GLM library for mathematic functions

#include <glm/glm.hpp>

#include <glm/gtx/projection.hpp>

#include <glm/gtc/matrix\_transform.hpp>

#include <glm/gtx/transform2.hpp>

#include <glm/gtc/type\_ptr.hpp>

#include <glm/gtc/matrix\_access.hpp>

#include <glm/gtc/matrix\_inverse.hpp>

// SOIL library for loading texture images

#include <SOIL.h>

// This header file contains error checking functions

#include "check\_error.hpp"

// This header file contains utility functions that print out the content of

// aiScene data structure.

#include "assimp\_utilities.hpp"

using namespace std;

using namespace glm;

#define BUFFER\_OFFSET( offset ) ((GLvoid\*) offset)

// Default folders for 3D models, images, and shaders

// Update these for your computer.

const char \* defaultModelFolder = "..\\..\\..\\Models\\";

const char \* defaultImageFolder = "..\\..\\..\\Images\\";

const char \* defaultShaderFolder = "..\\..\\..\\code\\Shaders\\";

const char \*imageFileName = "..\\..\\..\\Images\\cedarstone.jpg";

//-------------------------

// Shader related variables

// Shader file names

const char\* vShaderFilename = "Tiwari\_vShader.glsl";

const char\* fShaderFilename = "Tiwari\_fShader.glsl";

// Index of the shader program

GLuint program;

//---------------------------

// 3D object related variable

// 3D object file name

const char \* objectFileName = "sphere.obj";

// This is the Assimp importer object that loads the 3D file.

Assimp::Importer importer;

// Global Assimp scene object

const aiScene\* scene = NULL;

// This array stores the VAO indices for each corresponding mesh in the aiScene object.

// For example, vaoArray[0] stores the VAO index for the mesh scene->mMeshes[0], and so on.

unsigned int \*vaoArray = NULL;

// This is the 1D array that stores the face indices of a mesh.

unsigned int \*faceArray = NULL;

//---------------------------------

// Vertex related variables

struct VertexAttributeLocations {

GLint vPos; // Index of the in variable vPos in the vertex shader

GLint vNormal; // Index of the in variable vNormal in the vertex shader

GLint vTextureCoord; // Index of the in variable vTextureCoord in the vertex shader

};

VertexAttributeLocations vertexAttributeLocations;

//---------------------------------

// Transformation related variables

struct MatrixLocations {

GLuint mvpMatrixID; // uniform attribute: model-view-projection matrix

GLint modelMatrixID; // uniform variable: model-view matrix

GLint normalMatrixID; // uniform variable: normal matrix for transforming normal vectors

};

MatrixLocations matrixLocations;

mat4 projMatrix; // projection matrix

mat4 viewMatrix; // view matrix

//-----------------

// Camera related variables

vec3 defaultCameraPosition = vec3(0.0f, 0.0f, 3.0f);

vec3 defaultCameraLookAt = vec3(0.0f, 0.0f, 0.0f);

vec3 defaultCameraUp = vec3(0.0f, 1.0f, 0.0f);

float defaultFOV = 60.0f; // in degrees

float defaultNearPlane = 0.1f;

float defaultFarPlane = 1000.0f;

int windowWidth = 600;

int windowHeight = 400;

//---------------------------

// Lighting related variables

// surface material properties

struct SurfaceMaterialProperties {

float ambient[4]; //ambient component

float diffuse[4]; //diffuse component

float specular[4]; // Surface material property: specular component

float emission[4];

float shininess;

};

SurfaceMaterialProperties \*surfaceMaterials = NULL;

struct SurfaceMaterialLocations {

unsigned int ambient;

unsigned int diffuse;

unsigned int specular;

unsigned int emission;

unsigned int shininess;

};

SurfaceMaterialLocations surfaceMaterialLocations;

// Maximum number of lights

// This number must be coordinated with the same variable

// in the fragment shader.

const unsigned int maxNumLightSources = 50;

// I prefer to use individual arrays for each lighting parameter,

// rather than using an array of structure that contains all the lighting

// parameters. The reason is that with a structure, I'll need to use a uniform block

// in the fragment shader. I have had some strange problems with uniform blocks.

// I think it's safer and easier to debug by passing lighting parameters as individual

// uniform variables.

// Because there may be multiple light sources, each lighting parameter is an array.

// On the shader's side, it's an uniform array.

// The lighting parameters are initialized with default lighting parameters.

float lightPosition[maxNumLightSources][4] = { { 1.0f, 1.0f, 1.0f, 1.0f },{ -1.0f, 1.0f, 1.0f, 1.0f } };

float lightDirection[maxNumLightSources][4] = { { 0.0f, 0.0f, -1.0f, 1.0 },{ 0.0f, 0.0f, -1.0f, 1.0 } };

float lightDiffuse[maxNumLightSources][4] = { { 1.0f, 1.0f, 1.0f, 1.0f },{ 1.0f, 1.0f, 1.0f, 1.0f } };

float lightSpecular[maxNumLightSources][4] = { { 1.0f, 1.0f, 1.0f, 1.0f },{ 1.0f, 1.0f, 1.0f, 1.0f } };

float lightAmbient[maxNumLightSources][4] = { { 0.2f, 0.2f, 0.2f, 1.0f },{ 0.2f, 0.2f, 0.2f, 1.0f } };

float lightConstantAttenuation[maxNumLightSources] = { 1.0f,1.0f };

float lightLinearAttenuation[maxNumLightSources] = { 0.5f,0.5f };

float lightQuadraticAttenuation[maxNumLightSources] = { 0.1f,0.1f };

float spotlightInnerCone[maxNumLightSources] = { 0.3f,0.6f }; // inner cone cutoff angle (in radians)

float spotlightOuterCone[maxNumLightSources] = { 2.0f,2.3f }; // spotlight cutoff angle (in radians)

int lightType[maxNumLightSources] = { 1,1 }; // default point light source

// The actual number of lights in the scene.

unsigned int numLights = 2;

// Locations of the lighting parameters in the shader

struct LightSourceLocations {

unsigned int position;

unsigned int direction;

unsigned int ambient;

unsigned int diffuse;

unsigned int specular;

unsigned int constantAttenuation;

unsigned int linearAttenuation;

unsigned int quadraticAttenuation;

unsigned int spotlightInnerCone;

unsigned int spotlightOuterCone;

unsigned int type;

unsigned int eyePosition;

unsigned int hasTexture;

unsigned int numLights;

};

LightSourceLocations lightSourceLocations;

// ------------------------------------

// Texture mapping related variables.

float\* textureCoordArray = 0;

unsigned int\* textureObjectIDArray = 0;

unsigned int textureUnit;

// User interactions related parameters

float rotateX = 0;

float rotateY = 0;

bool useMouse = false;

float scaleFactor = 1.0f;

float xTranslation = 0.0f;

float yTranslation = 0.0f;

float zTranslation = 0.0f;

float transformationStep = 1.0f;

//----------

// Functions

//---------------

// Load a 3D file

const aiScene\* load3DFile(const char \*filename) {

ifstream fileIn(filename);

// Check if the file exists.

if (fileIn.good()) {

fileIn.close(); // The file exists.

}

else {

fileIn.close();

cout << "Unable to open the 3D file." << endl;

return false;

}

cout << "Loading 3D file " << filename << endl;

// Load the 3D file using Assimp. The content of the 3D file is stored in an aiScene object.

const aiScene\* sceneObj = importer.ReadFile(filename, aiProcessPreset\_TargetRealtime\_Quality);

// Check if the file is loaded successfully.

if (!sceneObj)

{

// Fail to load the file

cout << importer.GetErrorString() << endl;

return false;

}

else {

cout << "3D file " << filename << " loaded." << endl;

}

// Print the content of the aiScene object, if needed.

// This function is defined in the check\_error.hpp file.

printAiSceneInfo(sceneObj, PRINT\_AISCENE\_SUMMARY);

return sceneObj;

}

//-------------------

// Read a shader file

const char \*readShaderFile(const char \* filename) {

ifstream shaderFile(filename);

if (!shaderFile.is\_open()) {

cout << "Cannot open the shader file " << filename << endl;

return NULL;

}

string line;

// Must created a new string, otherwise the returned pointer

// will be invalid

string \*shaderSourceStr = new string();

while (getline(shaderFile, line)) {

\*shaderSourceStr += line + '\n';

}

const char \*shaderSource = shaderSourceStr->c\_str();

shaderFile.close();

return shaderSource;

}

// --------------------------------------

// Convert an aiColor3D vector to an array.

void copyAiColor3DToFloat4(float\* vector1, aiColor3D vector2) {

vector1[0] = vector2.r;

vector1[1] = vector2.g;

vector1[2] = vector2.b;

vector1[3] = 1.0f;

}

// --------------------------------------

// Convert an aiVector3D to an array.

void copyAiVector3DToFloat4(float\* vector1, aiVector3D vector2) {

vector1[0] = vector2.x;

vector1[1] = vector2.y;

vector1[2] = vector2.z;

vector1[3] = 1.0f;

}

// ----------------------------------

// Get only the file name out of a path

string getFileName(const string& s) {

char sep = '/';

#ifdef \_WIN32

sep = '\\';

#endif

size\_t i = s.rfind(sep, s.length());

if (i != string::npos) {

return(s.substr(i + 1, s.length() - i));

}

return(s);

}

// ---------------------------------------

// Load and build shaders

bool prepareShaders() {

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// Load and build the shaders.

GLuint vShaderID, fShaderID;

// Create empty shader objects

vShaderID = glCreateShader(GL\_VERTEX\_SHADER);

checkGlCreateXError(vShaderID, "vShaderID");

if (vShaderID == 0) {

return false;

}

fShaderID = glCreateShader(GL\_FRAGMENT\_SHADER);

checkGlCreateXError(fShaderID, "fShaderID");

if (fShaderID == 0) {

return false;

}

const char\* vShader = readShaderFile(

(string(defaultShaderFolder) + getFileName(string(vShaderFilename))).c\_str());

if (!vShader) {

return false;

}

// OpenGL fragment shader source code

const char\* fShader = readShaderFile(

(string(defaultShaderFolder) + getFileName(string(fShaderFilename))).c\_str());

if (!fShader) {

return false;

}

// Attach shader source code the shader objects

glShaderSource(vShaderID, 1, &vShader, NULL);

glShaderSource(fShaderID, 1, &fShader, NULL);

// Compile the vertex shader object

glCompileShader(vShaderID);

printShaderInfoLog(vShaderID); // Print error messages, if any.

// Compile the fragment shader object

glCompileShader(fShaderID);

printShaderInfoLog(fShaderID); // Print error messages, if any.

// Create an empty shader program object

program = glCreateProgram();

checkGlCreateXError(program, "program");

if (program == 0) {

return false;

}

// Attach vertex and fragment shaders to the shader program

glAttachShader(program, vShaderID);

glAttachShader(program, fShaderID);

// Link the shader program

glLinkProgram(program);

// Check if the shader program can run in the current OpenGL state, just for testing purposes.

glValidateProgram(program);

printShaderProgramInfoLog(program); // Print error messages, if any.

return true;

}

// ------------------------------------------

// Get shader variable locations

void getShaderVariableLocations() {

glUseProgram(program);

vertexAttributeLocations.vPos = glGetAttribLocation(program, "vPos");

checkGlGetXLocationError(vertexAttributeLocations.vPos, "vPos");

vertexAttributeLocations.vNormal = glGetAttribLocation(program, "vNormal");

checkGlGetXLocationError(vertexAttributeLocations.vNormal, "vNormal");

vertexAttributeLocations.vTextureCoord = glGetAttribLocation(program, "vTextureCoord");

checkGlGetXLocationError(vertexAttributeLocations.vTextureCoord, "vTextureCoord");

// Get the ID of the uniform matrix variable in the vertex shader.

matrixLocations.mvpMatrixID = glGetUniformLocation(program, "mvpMatrix");

if (matrixLocations.mvpMatrixID == -1) {

cout << "There is an error getting the handle of GLSL uniform variable mvp\_matrix." << endl;

}

matrixLocations.modelMatrixID = glGetUniformLocation(program, "modelMatrix");

if (matrixLocations.modelMatrixID == -1) {

cout << "There is an error getting the handle of GLSL uniform variable mvMatrix." << endl;

}

matrixLocations.normalMatrixID = glGetUniformLocation(program, "normalMatrix");

if (matrixLocations.mvpMatrixID == -1) {

cout << "There is an error getting the handle of GLSL uniform variable normalMatrix." << endl;

}

surfaceMaterialLocations.ambient = glGetUniformLocation(program, "Kambient");

surfaceMaterialLocations.diffuse = glGetUniformLocation(program, "Kdiffuse");

surfaceMaterialLocations.specular = glGetUniformLocation(program, "Kspecular");

surfaceMaterialLocations.emission = glGetUniformLocation(program, "emission");

surfaceMaterialLocations.shininess = glGetUniformLocation(program, "shininess");

lightSourceLocations.position = glGetUniformLocation(program, "lightSourcePosition");

lightSourceLocations.direction = glGetUniformLocation(program, "lightDirection");

lightSourceLocations.diffuse = glGetUniformLocation(program, "diffuseLightIntensity");

lightSourceLocations.specular = glGetUniformLocation(program, "specularLightIntensity");

lightSourceLocations.ambient = glGetUniformLocation(program, "ambientLightIntensity");

lightSourceLocations.constantAttenuation = glGetUniformLocation(program, "constantAttenuation");

lightSourceLocations.linearAttenuation = glGetUniformLocation(program, "linearAttenuation");

lightSourceLocations.quadraticAttenuation = glGetUniformLocation(program, "quadraticAttenuation");

lightSourceLocations.spotlightInnerCone = glGetUniformLocation(program, "spotlightInnerCone");

lightSourceLocations.spotlightOuterCone = glGetUniformLocation(program, "spotlightOuterCone");

lightSourceLocations.type = glGetUniformLocation(program, "lightType");

lightSourceLocations.eyePosition = glGetUniformLocation(program, "eyePosition");

lightSourceLocations.hasTexture = glGetUniformLocation(program, "hasTexture");

lightSourceLocations.numLights = glGetUniformLocation(program, "numLights");

textureUnit = glGetUniformLocation(program, "texUnit");

checkGlGetXLocationError(textureUnit, "textureUnit");

}

//--------------------------------------

// Load 3D data from 3D file with Assimp

// The Assimp data structure consists of a scene graph and multiple arrays: meshes, materials,

// lights, cameras, embedded textures, and animations.

// In this program, we don't process embedded texture and animation.

// Each mesh contains multiple arrays: vertices, normals, texture coordinates, and faces.

// - This function associates each array with a VBO and create a VAO for each mesh.

// - This function also copies the data from Assimp's materials array to the surfaceMaterials

// array so it can be transferred to the shader.

// - This function creates an array of texture objects, one for each material.

// - This function copies the data from Assimp's light array to the arrays of light parameters so

// they can be transferred to the shader.

bool load3DData() {

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// Load the 3D file

// This variable temporarily stores the VBO index.

GLuint buffer;

// Load the 3D file using Assimp.

// Assume that the 3D file is stored in the default model folder.

scene = load3DFile(

(string(defaultModelFolder) + string(getFileName(objectFileName))).c\_str());

if (!scene) {

return false;

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// Retrieve vertex arrays from the aiScene object and bind them with VBOs and VAOs.

// Create an array to store the VAO indices for each mesh.

vaoArray = (unsigned int\*)malloc(sizeof(unsigned int) \* scene->mNumMeshes);

// Go through each mesh stored in the aiScene object, bind it with a VAO,

// and save the VAO index in the vaoArray.

for (unsigned int i = 0; i < scene->mNumMeshes; i++) {

const aiMesh\* currentMesh = scene->mMeshes[i];

// Create an empty Vertex Array Object (VAO). VAO is only available from OpenGL 3.0 or higher.

// Note that the vaoArray[] index is in sync with the mMeshes[] array index.

// That is, for mesh #0, the corresponding VAO index is stored in vaoArray[0], and so on.

glGenVertexArrays(1, &vaoArray[i]);

glBindVertexArray(vaoArray[i]);

if (currentMesh->HasPositions()) {

// Create an empty Vertex Buffer Object (VBO)

glGenBuffers(1, &buffer);

glBindBuffer(GL\_ARRAY\_BUFFER, buffer);

// Bind (transfer) the vertex position array (stored in aiMesh's member variable mVertices)

// to the VBO.

// Note that the vertex positions are stored in a continuous 1D array (i.e. mVertices) in the aiScene object.

glBufferData(GL\_ARRAY\_BUFFER, sizeof(float) \* 3 \* currentMesh->mNumVertices,

currentMesh->mVertices, GL\_STATIC\_DRAW);

// Associate this VBO with an the vPos variable in the vertex shader.

// The vertex data and the vertex shader must be connected.

glEnableVertexAttribArray(vertexAttributeLocations.vPos);

glVertexAttribPointer(vertexAttributeLocations.vPos, 3, GL\_FLOAT, GL\_FALSE, 0, BUFFER\_OFFSET(0));

}

if (currentMesh->HasFaces()) {

// Create an array to store the face indices (elements) of this mesh.

// This is necessary becaue face indices are NOT stored in a continuous 1D array inside aiScene.

// Instead, there is an array of aiFace objects. Each aiFace object stores a number of (usually 3) face indices.

// We need to copy the face indices into a continuous 1D array.

faceArray = (unsigned int\*)malloc(sizeof(unsigned int) \* currentMesh->mNumFaces \* currentMesh->mFaces[0].mNumIndices);

// copy the face indices from aiScene into a 1D array faceArray.

int faceArrayIndex = 0;

for (unsigned int j = 0; j < currentMesh->mNumFaces; j++) {

for (unsigned int k = 0; k < currentMesh->mFaces[j].mNumIndices; k++) {

faceArray[faceArrayIndex] = currentMesh->mFaces[j].mIndices[k];

faceArrayIndex++;

}

}

// Create an empty VBO

glGenBuffers(1, &buffer);

// This VBO is an GL\_ELEMENT\_ARRAY\_BUFFER, not a GL\_ARRAY\_BUFFER.

// GL\_ELEMENT\_ARRAY\_BUFFER stores the face indices (elements), while

// GL\_ARRAY\_BUFFER stores vertex positions.

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

glBufferData(GL\_ELEMENT\_ARRAY\_BUFFER,

sizeof(unsigned int) \* currentMesh->mNumFaces \* currentMesh->mFaces[0].mNumIndices,

faceArray, GL\_STATIC\_DRAW);

}

if (currentMesh->HasNormals()) {

// Create an empty Vertex Buffer Object (VBO)

glGenBuffers(1, &buffer);

glBindBuffer(GL\_ARRAY\_BUFFER, buffer);

// Bind (transfer) the vertex normal array (stored in aiMesh's member variable mNormals)

// to the VBO.

// Note that the vertex normals are stored in a 1D array (i.e. mVertices)

// in the aiScene object.

glBufferData(GL\_ARRAY\_BUFFER, sizeof(float) \* 3 \* currentMesh->mNumVertices,

currentMesh->mNormals, GL\_STATIC\_DRAW);

// Associate this VBO with an the vPos variable in the vertex shader.

// The vertex data and the vertex shader must be connected.

glEnableVertexAttribArray(vertexAttributeLocations.vNormal);

glVertexAttribPointer(vertexAttributeLocations.vNormal, 3, GL\_FLOAT, GL\_FALSE, 0, BUFFER\_OFFSET(0));

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// Set up texture mapping data

// Each mesh may have multiple UV(texture) channels (multi-texture). Here we only use

// the first channel. Call currentMesh->GetNumUVChannels() to get the number of UV channels

// for this mesh.

if (currentMesh->HasTextureCoords(0)) {

// Create an empty Vertex Buffer Object (VBO)

glGenBuffers(1, &buffer);

glBindBuffer(GL\_ARRAY\_BUFFER, buffer);

// mTextureCoords is different from mVertices or mNormals. It is a 2D array, not a 1D array.

// So we need to copy it to a 1D texture coordinate array.

// The first dimension of this array is the texture channel for this mesh.

// The second dimension is the vertex index number.

// The number of texture coordinates is always the same as the number of vertices.

textureCoordArray = (float \*)malloc(sizeof(float) \* 2 \* currentMesh->mNumVertices);

unsigned int k = 0;

for (unsigned int j = 0; j < currentMesh->mNumVertices; j++) {

textureCoordArray[k] = currentMesh->mTextureCoords[0][j].x;

k++;

textureCoordArray[k] = currentMesh->mTextureCoords[0][j].y;

k++;

}

// Bind (transfer) the texture coordinate array to the VBO.

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

// Associate this VBO with the vTextureCoord variable in the vertex shader.

// The vertex data and the vertex shader must be connected.

glVertexAttribPointer(vertexAttributeLocations.vTextureCoord, 2, GL\_FLOAT, GL\_FALSE, 0, BUFFER\_OFFSET(0));

glEnableVertexAttribArray(vertexAttributeLocations.vTextureCoord);

}

//Close the VAOs and VBOs for later use.

glBindVertexArray(0);

glBindBuffer(GL\_ARRAY\_BUFFER, 0);

glBindBuffer(GL\_ELEMENT\_ARRAY\_BUFFER, 0);

} // end for

//-----------------------------------------------

// Create an array to store surface material data.

// The surface material data in the Assimp data structure cannot be directly transferred to the shader, so

// we need to copy them to our own data structure first.

surfaceMaterials = (SurfaceMaterialProperties \*)malloc(sizeof(SurfaceMaterialProperties) \* scene->mNumMaterials);

// Create an array to store texture object IDs, one texture object per material. Not all materials will have

// texture objects.

// Note that we only have one texture object per material. This means we only load one texture per material.

textureObjectIDArray = (unsigned int\*)malloc(sizeof(unsigned int) \* scene->mNumMaterials);

// Copy all the Assimp material data to our own C data structure.

for (unsigned int i = 0; i < scene->mNumMaterials; i++) {

aiMaterial\* currentMaterial = scene->mMaterials[i];

aiColor3D color(0.0f, 0.0f, 0.0f);

currentMaterial->Get(AI\_MATKEY\_COLOR\_AMBIENT, color);

copyAiColor3DToFloat4(surfaceMaterials[i].ambient, color);

currentMaterial->Get(AI\_MATKEY\_COLOR\_DIFFUSE, color);

copyAiColor3DToFloat4(surfaceMaterials[i].diffuse, color);

currentMaterial->Get(AI\_MATKEY\_COLOR\_SPECULAR, color);

copyAiColor3DToFloat4(surfaceMaterials[i].specular, color);

currentMaterial->Get(AI\_MATKEY\_COLOR\_EMISSIVE, color);

copyAiColor3DToFloat4(surfaceMaterials[i].emission, color);

float shininess = 0.0f;

currentMaterial->Get(AI\_MATKEY\_SHININESS, shininess);

surfaceMaterials[i].shininess = shininess;

// To keep it simple, we only retrieve the diffuse type texture.

if (currentMaterial->GetTextureCount(aiTextureType\_DIFFUSE) > 0) {

int texIndex = 0; // To keep it simple, we only retrieve the first texture for each material.

aiString path; // filename

// Get the diffuse texture file path for this material.

aiReturn texFound = currentMaterial->GetTexture(aiTextureType\_DIFFUSE, texIndex, &path);

if (texFound == AI\_SUCCESS) {

string filename = getFileName(path.data); // get only the filename

// Use SOIL to load texture image. SOIL will create a texture object for this texture

// image and return the texture object ID.

// We assume that the image is stored in the default texture image folder, not necessarily the

// texture file path stored in the 3D file.

textureObjectIDArray[i] = SOIL\_load\_OGL\_texture((string(defaultImageFolder) + filename).c\_str(),SOIL\_LOAD\_AUTO, SOIL\_CREATE\_NEW\_ID, SOIL\_FLAG\_MIPMAPS);

// If the returned texture ID > 0, it means the imaged is loaded successfully.

if (textureObjectIDArray[i] <= 0) {

cout << "Couldn't create a texture object for the texture image: " << filename.c\_str() << endl;

} // end if

}

else {

textureObjectIDArray[i] = 0;

cout << "Couldn't find the texture file for mesh #" << i << endl;

} // end if (texture is found)

}

else {

textureObjectIDArray[i] = 0;

cout << "There is no texture for mesh #" << i << endl;

}

} // end for

// Copy data from Assimp's light parameters to our own C data struction, which makes it easier to transfer

// it to the shader.

if (scene->HasLights()) {

// Because the lighting parameters need to passed to the shader and GLSL doesn't support

// dynamic memory allocation, we have to use a static array to store lighting parameters.

// We also need to set a maximum number of lights.

// If the actual number of lights are smaller than the maximum number of lights, use the

// actual number. Otherwise, use the maximum number of lights.

numLights = std::min(scene->mNumLights, maxNumLightSources);

for (unsigned int i = 0; i < numLights; i++) {

aiLight\* currentLight = scene->mLights[i];

copyAiColor3DToFloat4(lightAmbient[i], currentLight->mColorAmbient);

copyAiColor3DToFloat4(lightDiffuse[i], currentLight->mColorDiffuse);

copyAiColor3DToFloat4(lightSpecular[i], currentLight->mColorSpecular);

copyAiVector3DToFloat4(lightPosition[i], currentLight->mPosition);

copyAiVector3DToFloat4(lightDirection[i], currentLight->mDirection);

lightConstantAttenuation[i] = currentLight->mAttenuationConstant;

lightLinearAttenuation[i] = currentLight->mAttenuationLinear;

lightQuadraticAttenuation[i] = currentLight->mAttenuationQuadratic;

spotlightInnerCone[i] = currentLight->mAngleInnerCone;

spotlightOuterCone[i] = currentLight->mAngleOuterCone;

switch (currentLight->mType) {

case aiLightSource\_POINT:

lightType[i] = 1;

break;

case aiLightSource\_DIRECTIONAL:

lightType[i] = 2;

break;

case aiLightSource\_SPOT:

lightType[i] = 3;

break;

default:

lightType[i] = 0;

break;

}

}

}

return true;

}

//-------------------------------

//Prepare the shaders and 3D data

bool init()

{

// Load shaders

if (prepareShaders() == false) {

return false;

}

getShaderVariableLocations();

if (load3DData() == false) {

return false;

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// Set up other OpenGL states.

// Turn on visibility test.

glEnable(GL\_DEPTH\_TEST);

// Draw the object in wire frame mode.

// You can comment out this line to draw the object in

// shaded mode. But without lighting, the object will look very dark.

// glPolygonMode(GL\_FRONT\_AND\_BACK, GL\_LINE);

// This is the background color.

glClearColor(0.0, 0.0, 0.0, 0.0);

// Uncomment this line for debugging purposes.

// Check error

checkOpenGLError("init()");

return true;

}

//------------------------------------------------

// Traverse the scene graph to find a camera, update its location and direction,

// and then create the view and projection matrices.

void nodeTreeTraversalCamera(const aiNode\* node, aiMatrix4x4 matrix) {

if (!node) {

cout << "nodeTreeTraversalCamera(): Null node" << endl;

return;

}

// Camera and lights reference a specific node by name, if any.

string name = node->mName.C\_Str();

// Calculate this (camera) node's transformation matrix.

aiMatrix4x4 currentTransformMatrix = matrix \* node->mTransformation;

// Check every camera on the camera list

for (unsigned int i = 0; i < scene->mNumCameras; i++) {

aiCamera\* currentCamera = scene->mCameras[i];

string currentCameraName = currentCamera->mName.C\_Str();

// If the current camera is the same as the camera node ...

if (currentCameraName.compare(name) == 0) {

// It's not clear whether we also need to multiply the camera's local matrix.

// Maybe it's necessary for some 3D file format.

aiMatrix4x4 cameraMatrix;

currentCamera->GetCameraMatrix(cameraMatrix);

//currentTransformMatrix = currentTransformMatrix \*cameraMatrix;

// Get the camera position, look-at, and up vector.

// Don't modify aiCamera's member variables mPosition, mLookAt, mUp directly.

aiVector3D cameraPosition = currentCamera->mPosition;

aiVector3D cameraLookAtPosition = currentCamera->mLookAt;

aiVector3D cameraUpVector = currentCamera->mUp;

// Transform the camera position, lookAt, and up vector

cameraPosition = currentTransformMatrix \* cameraPosition;

cameraLookAtPosition = currentTransformMatrix \* cameraLookAtPosition;

cameraUpVector = currentTransformMatrix \* cameraUpVector;

cameraUpVector.Normalize(); // Remember to normalize the UP vector.

// Pass the eye position to the shader. We'll need it for calculating

// the specular color.

float eyePosition[3] = { cameraPosition.x, cameraPosition.y, cameraPosition.z };

glUniform3fv(lightSourceLocations.eyePosition, 1, eyePosition);

// Build the projection and view matrices

// It's better to use the window's aspect than using the aspect ratio from the 3D file.

projMatrix = perspective(currentCamera->mHorizontalFOV,

(float)windowWidth / (float)windowHeight,

currentCamera->mClipPlaneNear,

currentCamera->mClipPlaneFar);

// Create a view matrix

// You specify where the camera location and orientation and GLM will create a view matrix.

viewMatrix = lookAt(vec3(cameraPosition.x, cameraPosition.y, cameraPosition.z),

vec3(cameraLookAtPosition.x, cameraLookAtPosition.y, cameraLookAtPosition.z),

vec3(cameraUpVector.x, cameraUpVector.y, cameraUpVector.z));

} // end if camera name is the same

} // end for

// Recursively visit and find a camera in child nodes. This is a depth-first traversal.

for (unsigned int j = 0; j < node->mNumChildren; j++) {

nodeTreeTraversalCamera(node->mChildren[j], currentTransformMatrix);

}

}

//----------------------------------------------

// Traverse the scene graph to update the locations and directions of light sources.

void nodeTreeTraversalLight(const aiNode\* node, aiMatrix4x4 matrix) {

if (!node) {

cout << "nodeTreeTraversal(): Null node" << endl;

return;

}

string nodeName = node->mName.C\_Str();

// Calculate this (light) node's transformation matrix.

aiMatrix4x4 currentTransformMatrix = matrix \* node->mTransformation;

// Check every light in the light array to see if there is a match.

for (unsigned int i = 0; i < numLights; i++) {

aiLight\* currentLight = scene->mLights[i];

string currentLightName = currentLight->mName.C\_Str();

// If the current light is the same as the light node ...

if (currentLightName.compare(nodeName) == 0) {

aiVector3D transformedLightPosition =

currentTransformMatrix \* currentLight->mPosition;

aiVector3D transformedLightDirection =

currentTransformMatrix \* currentLight->mDirection;

// Update the light position and direction in the lightSources Array.

copyAiVector3DToFloat4(lightPosition[i], transformedLightPosition);

copyAiVector3DToFloat4(lightDirection[i], transformedLightDirection);

} // end if

} // end for

// Recursively visit and find a light in child nodes. This is a depth-first traversal.

for (unsigned int j = 0; j < node->mNumChildren; j++) {

nodeTreeTraversalLight(node->mChildren[j], currentTransformMatrix);

}

}

//--------------------------------------------------------------------------------------------

// Traverse the node tree in the aiScene object and draw the meshes associated with each node.

// This function is called recursively to perform a depth-first tree traversal.

// The second argument is the parent node's accumulated transformation matrix.

void nodeTreeTraversalMesh(const aiNode\* node, aiMatrix4x4 matrix) {

if (!node) {

cout << "nodeTreeTraversal(): Null node" << endl;

return;

}

// Multiply the parent's node's transformation matrix with this node's transformation matrix.

aiMatrix4x4 currentTransformMatrix = matrix \* node->mTransformation;

if (node->mNumMeshes > 0) {

// If this node contains meshes, we'll caculate the model-view-projection matrix for it.

// Conver the transformation matrix from aiMatrix4x4 to glm:mat4 format.

// aiMatrix4x4 is row major.

mat4 modelMatrix = mat4(1.0);

modelMatrix = row(modelMatrix, 0, vec4(currentTransformMatrix.a1,

currentTransformMatrix.a2, currentTransformMatrix.a3, currentTransformMatrix.a4));

modelMatrix = row(modelMatrix, 1, vec4(currentTransformMatrix.b1,

currentTransformMatrix.b2, currentTransformMatrix.b3, currentTransformMatrix.b4));

modelMatrix = row(modelMatrix, 2, vec4(currentTransformMatrix.c1,

currentTransformMatrix.c2, currentTransformMatrix.c3, currentTransformMatrix.c4));

modelMatrix = row(modelMatrix, 3, vec4(currentTransformMatrix.d1,

currentTransformMatrix.d2, currentTransformMatrix.d3, currentTransformMatrix.d4));

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// Combine the model, view, and project matrix into one model-view-projection matrix.

// Model matrix is then multiplied with view matrix and projection matrix to create a combined

// model\_view\_projection matrix.

// The view and projection matrices are created in the previous traversal of the scene that processes

// the camera data.

// The sequence of multiplication is important here. Model matrix, view matrix, and projection matrix

// must be multiplied from right to left, because the vertex position is on the right hand side.

mat4 mvpMatrix = projMatrix \* viewMatrix \* modelMatrix;

// Create a normal matrix to transform normals.

// We don't need to include the view matrix here because the lighting is done

// in world space.

mat3 normalMatrix = mat3(1.0);

normalMatrix = column(normalMatrix, 0, vec3(modelMatrix[0][0], modelMatrix[0][1], modelMatrix[0][2]));

normalMatrix = column(normalMatrix, 1, vec3(modelMatrix[1][0], modelMatrix[1][1], modelMatrix[1][2]));

normalMatrix = column(normalMatrix, 2, vec3(modelMatrix[2][0], modelMatrix[2][1], modelMatrix[2][2]));

normalMatrix = inverseTranspose(normalMatrix);

// Draw all the meshes associated with the current node.

// Certain node may have multiple meshes associated with it.

for (unsigned int i = 0; i < node->mNumMeshes; i++) {

// This is the index of the mesh associated with this node.

int meshIndex = node->mMeshes[i];

const aiMesh\* currentMesh = scene->mMeshes[meshIndex];

// The model\_view\_projection matrix is transferred to the shader to be used in the vertex shader.

// Here we send the combined model\_view\_projection matrix to the shader so that we don't have to do the same

// multiplication in the vertex shader repeatedly. Note that the vertex shader is executed for each vertex.

glUniformMatrix4fv(matrixLocations.mvpMatrixID, 1, GL\_FALSE, glm::value\_ptr(mvpMatrix));

glUniformMatrix4fv(matrixLocations.modelMatrixID, 1, GL\_FALSE, glm::value\_ptr(modelMatrix));

glUniformMatrix3fv(matrixLocations.normalMatrixID, 1, GL\_FALSE, glm::value\_ptr(normalMatrix));

// This is the material for this mesh

unsigned int materialIndex = currentMesh->mMaterialIndex;

// Pass the material data to the shader. The material data is copied from Assimp's data structure

// to our own data structure in load3DData().

glUniform4fv(surfaceMaterialLocations.ambient, 1, surfaceMaterials[currentMesh->mMaterialIndex].ambient);

glUniform4fv(surfaceMaterialLocations.diffuse, 1, surfaceMaterials[currentMesh->mMaterialIndex].diffuse);

glUniform4fv(surfaceMaterialLocations.specular, 1, surfaceMaterials[currentMesh->mMaterialIndex].specular);

glUniform4fv(surfaceMaterialLocations.emission, 1, surfaceMaterials[currentMesh->mMaterialIndex].emission);

glUniform1f(surfaceMaterialLocations.shininess, surfaceMaterials[currentMesh->mMaterialIndex].shininess);

// Transfer texture image to the shader.

if (textureObjectIDArray[currentMesh->mMaterialIndex] > 0) {

glActiveTexture(GL\_TEXTURE1);

glBindTexture(GL\_TEXTURE\_2D, textureObjectIDArray[currentMesh->mMaterialIndex]);

// We only use texture unit 1. Here 1 means Texture Unit 1.

// This tells fragment shader to retrieve texture from Texture Unit 1.

glUniform1i(textureUnit, 1);

// Tell the shader there is no texture so don't do texture mapping.

glUniform1i(lightSourceLocations.hasTexture, 1);

}

else {

glUniform1i(lightSourceLocations.hasTexture, 0); // No texture

}

// This mesh should have already been associated with a VAO in a previous function.

// Note that mMeshes[] array and the vaoArray[] array are in sync.

// That is, for mesh #0, the corresponding VAO index is stored in vaoArray[0], and so on.

// Bind the corresponding VAO for this mesh.

glBindVertexArray(vaoArray[meshIndex]);

// How many faces are in this mesh?

unsigned int numFaces = currentMesh->mNumFaces;

// How many indices are for each face?

unsigned int numIndicesPerFace = currentMesh->mFaces[0].mNumIndices;

// The second parameter is crucial. This is the number of face indices, not the number of faces.

// (numFaces \* numIndicesPerFace) is the total number of elements(face indices) of this mesh.

// Now draw all the faces. We know these faces are triangle because in

// importer.ReadFile(filename, aiProcessPreset\_TargetRealtime\_Quality);

// "aiProcessPreset\_TargetRealtime\_Quality" indicates that the 3D object will be triangulated.

glDrawElements(GL\_TRIANGLES, (numFaces \* numIndicesPerFace), GL\_UNSIGNED\_INT, 0);

// We are done with the current VAO. Move on to the next VAO, if any.

glBindVertexArray(0);

}

} // end if (node->mNumMeshes > 0)

// Uncomment this line for debugging purposes.

// checkOpenGLError();

// Recursively visit and draw all the child nodes. This is a depth-first traversal.

// Even if this node does not contain mesh, we still need to pass down the transformation matrix.

for (unsigned int j = 0; j < node->mNumChildren; j++) {

nodeTreeTraversalMesh(node->mChildren[j], currentTransformMatrix);

}

}

//--------------------------

// Display callback function

void display() {

// Clear the background color and the depth buffer.

glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT);

// Activate the shader program.

glUseProgram(program);

// Traverse the scene graph to update the location and direction of the camera.

if (scene->HasCameras()) {

nodeTreeTraversalCamera(scene->mRootNode, aiMatrix4x4());

}

else {

// If there is no camera data in the file, create the default projection and view matrices.

projMatrix = perspective(radians(defaultFOV), (float)windowWidth / (float)windowHeight, defaultNearPlane, defaultFarPlane);

// Create a view matrix

// You specify where the camera location and orientation and GLM will create a view matrix.

// The first parameter is the location of the camera;

// the second is where the camera is pointing at; the third is the up vector for camera.

// If you need to move or animate your camera during run time, then you need to construct the

// view matrix in display() function.

viewMatrix = lookAt(defaultCameraPosition, defaultCameraLookAt, defaultCameraUp);

}

// Traverse the scene graph to upate the location and direction of the light sources.

if (scene->HasLights()) {

nodeTreeTraversalLight(scene->mRootNode, aiMatrix4x4());

}

// After the lighting parameters are updated, pass them to the shader program.

glUniform4fv(lightSourceLocations.position, numLights, (const float\*)lightPosition);

glUniform4fv(lightSourceLocations.direction, numLights, (const float\*)lightDirection);

glUniform4fv(lightSourceLocations.ambient, numLights, (const float\*)lightAmbient);

glUniform4fv(lightSourceLocations.diffuse, numLights, (const float\*)lightDiffuse);

glUniform4fv(lightSourceLocations.specular, numLights, (const float\*)lightSpecular);

glUniform1fv(lightSourceLocations.constantAttenuation, numLights, lightConstantAttenuation);

glUniform1fv(lightSourceLocations.linearAttenuation, numLights, lightLinearAttenuation);

glUniform1fv(lightSourceLocations.quadraticAttenuation, numLights, lightQuadraticAttenuation);

glUniform1fv(lightSourceLocations.spotlightInnerCone, numLights, spotlightInnerCone);

glUniform1fv(lightSourceLocations.spotlightOuterCone, numLights, spotlightOuterCone);

glUniform1iv(lightSourceLocations.type, numLights, lightType);

glUniform1i(lightSourceLocations.numLights, numLights);

//\*\*\*\*\*\*\*\*\*\*\*\*\*

// Render scene

// First create the transformation matrix that will transform the 3D objects.

// Note that this transformation only applies to the meshes, not the lights and camera.

// Because the transformation matrix is pass down the scene graph through the root, all the meshes are

// transformed. If you want to transform a specific mesh, then you need to attach the transformation matrix to

// that mesh's node on the scene graph.

// These rotation, translation, and scaling parameters are controlled by the mouse and keyboard.

aiMatrix4x4 rotationXMatrix = aiMatrix4x4::RotationX(radians(rotateX), rotationXMatrix);

aiMatrix4x4 rotationYMatrix = aiMatrix4x4::RotationY(radians(rotateY), rotationYMatrix);

aiMatrix4x4 scaleMatrix = aiMatrix4x4::Scaling(aiVector3D(scaleFactor, scaleFactor, scaleFactor), scaleMatrix);

aiMatrix4x4 translationXMatrix = aiMatrix4x4::Translation(aiVector3D(xTranslation, 0.0f, 0.0f),

translationXMatrix);

aiMatrix4x4 translationYMatrix = aiMatrix4x4::Translation(aiVector3D(0.0f, yTranslation, 0.0f),

translationYMatrix);

aiMatrix4x4 translationZMatrix = aiMatrix4x4::Translation(aiVector3D(0.0f, 0.0f, zTranslation),

translationZMatrix);

aiMatrix4x4 overallTransformationMatrix =

translationZMatrix \* translationXMatrix \* translationYMatrix \* rotationXMatrix \* rotationYMatrix \* scaleMatrix;

// Start the node tree traversal and process each node. The overallTransformationMatrix is passed down

// the scene through the root node.

if (scene->HasMeshes()) {

nodeTreeTraversalMesh(scene->mRootNode, overallTransformationMatrix);

}

// Swap front and back buffers. The rendered image is now displayed.

glutSwapBuffers();

}

//----------------------------------------------------------------

// This function is called when the size of the window is changed.

void reshape(int width, int height)

{

// Specify the width and height of the picture within the window

// Creates a viewport matrix and insert it into the graphics pipeline.

// This operation is not done in shader, but taken care of by the hardware.

glViewport(0, 0, width, height);

windowWidth = width;

windowHeight = height;

}

// -------------------------------------------

// Use + or - keys to scale the object.

// Use w, s, a, d keys to move the object along the X and Y axes.

void keyboard(unsigned char key, int x, int y)

{

switch (key) {

case '+':

scaleFactor += 0.1f;

break;

case'-':

scaleFactor -= 0.1f;

break;

case 'w':

case 'W':

zTranslation -= transformationStep;

break;

case 's':

case 'S':

zTranslation += transformationStep;

break;

case 'a':

case 'A':

xTranslation -= transformationStep;

break;

case 'd':

case 'D':

xTranslation += transformationStep;

break;

case 033: // Escape Key

exit(EXIT\_SUCCESS);

break;

}

// Generate a display event, which forces Freeglut to call display().

glutPostRedisplay();

}

//---------------------------------------

// Use up and down keys to move the object along the Y axis.

void specialKeys(int key, int x, int y) {

switch (key) {

case GLUT\_KEY\_UP:

yTranslation += transformationStep;

break;

case GLUT\_KEY\_DOWN:

yTranslation -= transformationStep;

break;

default:

break;

}

// Generate a dislay event to force refreshing the window.

glutPostRedisplay();

}

//-----------------------------------------------

// Click left mouse button to enable and disable rotation

void mouse(int button, int state, int x, int y) {

if (button == GLUT\_LEFT\_BUTTON) {

if (state == GLUT\_DOWN) {

useMouse = !useMouse;

}

}

}

//---------------------------------------------------------------

// Read mouse motion data and convert them to rotation angles.

void passiveMotion(int x, int y) {

int centerX = windowWidth / 2;

int centerY = windowHeight / 2;

if (useMouse) {

rotateY = (float)(x - centerX) \* 0.5f;

rotateX = (float)(y - centerY) \* 0.5f;

// Generate a dislay event to force refreshing the window.

glutPostRedisplay();

}

}

int main(int argc, char\* argv[])

{

glutInit(&argc, argv);

// Initialize double buffer and depth buffer.

glutInitDisplayMode(GLUT\_RGBA | GLUT\_DOUBLE | GLUT\_DEPTH);

glutCreateWindow(argv[0]);

glutReshapeWindow(windowWidth, windowHeight);

// Initialize Glew. This must be called before any OpenGL function call.

glewInit();

// Initialize OpenGL debug context, which is available from OpenGL 4.3+.

// Must call this after glewInit().

initOpenGLDebugContext(true);

// This cannot be called before glewInit().

cout << "OpenGL version " << glGetString(GL\_VERSION) << endl;

if (init()) {

// Register callback functions

glutDisplayFunc(display);

glutReshapeFunc(reshape);

glutKeyboardFunc(keyboard);

glutSpecialFunc(specialKeys);

glutMouseFunc(mouse);

// Register the passive mouse motion call back function

// This function is called when the mouse moves within the window

// while no mouse buttons are pressed.

glutPassiveMotionFunc(passiveMotion);

glutMainLoop();

//Release the dynamically allocated memory blocks.

free(vaoArray);

free(faceArray);

free(textureCoordArray);

free(textureObjectIDArray);

}

}

Vertex Shader

#version 330

in vec3 vPos;

in vec3 vNormal;

in vec2 vTextureCoord;

uniform mat4 mvpMatrix; // model\_view\_project matrix

uniform mat4 modelMatrix; // model view matrix

uniform mat3 normalMatrix; // model matrix

out vec3 N; // The normal vector is passed over to the fragment shader

out vec3 v; // Vertex position is passed over to the fragment shader

out vec2 textureCoord; // The texture coordinates are passed over to the fragment shader

// Note that there is no out color, because the pixel color is calculated

// in the fragment shader.

void main()

{

vec4 position = vec4(vPos.xyz, 1.0);

gl\_Position = mvpMatrix \* position;

vec4 transformedPosition = modelMatrix \* position;

v = transformedPosition.xyz;

N = normalize(normalMatrix \* vNormal);

textureCoord = vTextureCoord;

}

Fragment Shader

#version 330

in vec3 N; // interpolated normal for the pixel

in vec3 v; // interpolated position for the pixel

in vec2 textureCoord; // interpolated texture coordinate for the pixel

const int maxNumLights = 50;

uniform vec4 lightSourcePosition[maxNumLights];

// light direction

uniform vec4 lightDirection[maxNumLights];

uniform vec4 diffuseLightIntensity[maxNumLights];

uniform vec4 specularLightIntensity[maxNumLights];

uniform vec4 ambientLightIntensity[maxNumLights];

// for calculating the light attenuation

uniform float constantAttenuation[maxNumLights];

uniform float linearAttenuation[maxNumLights];

uniform float quadraticAttenuation[maxNumLights];

// Spotlight cutoff angle

uniform float spotlightOuterCone[maxNumLights];

uniform float spotlightInnerCone[maxNumLights];

uniform int lightType[maxNumLights];

uniform int numLights;

uniform vec4 Kambient;

uniform vec4 Kdiffuse;

uniform vec4 Kspecular;

uniform vec4 emission;

uniform float shininess;

uniform vec3 eyePosition;

uniform int hasTexture;

uniform sampler2D texUnit;

out vec4 color;

// This fragment shader is an example of per-pixel lighting.

void main() {

// Now calculate the parameters for the lighting equation:

// color = Ka \* Lag + (Ka \* La) + attenuation \* ((Kd \* (N dot L) \* Ld) + (Ks \* ((N dot HV) ^ shininess) \* Ls))

// Ka, Kd, Ks: surface material properties

// Lag: global ambient light (not used in this example)

// La, Ld, Ls: ambient, diffuse, and specular components of the light source

// N: normal

// L: light vector

// HV: half vector

// shininess

// attenuation: light intensity attenuation over distance and spotlight angle

color = vec4(0.0, 0.0, 0.0, 1.0);

for (int i = 0; i < numLights; i++) {

vec3 lightVector;

float attenuation = 1.0;

if (lightType[i] == 1) {

// point light source

lightVector = normalize(lightSourcePosition[i].xyz - v);

// calculate light attenuation

float distance = distance(lightSourcePosition[i].xyz, v);

attenuation = 1.0 / (constantAttenuation[i] + (linearAttenuation[i] \* distance)

+ (quadraticAttenuation[i] \* distance \* distance));

}

else if (lightType[i] == 2) {

// directional light source. The light position is actually the light vector.

lightVector = lightSourcePosition[i].xyz;

// For directional lights, there is no light attenuation.

attenuation = 1.0;

}

else if (lightType[i] == 3) {

// spotlight source

lightVector = normalize(lightSourcePosition[i].xyz - v);

float distance = distance(lightSourcePosition[i].xyz, v);

float spotEffect = dot(normalize(lightDirection[i].xyz), normalize(lightVector));

// spotlightInnerCone is in radians, not degrees.

if (spotEffect > cos(spotlightInnerCone[i])) {

// If the vertex is in the spotlight cone

attenuation = spotEffect / (constantAttenuation[i] + linearAttenuation[i] \* distance +

quadraticAttenuation[i] \* distance \* distance);

}

else if (spotEffect > cos(spotlightOuterCone[i])) {

// Between inner and outer spotlight cone, make the light attenuate sharply.

attenuation = (pow(spotEffect, 12)) / (constantAttenuation[i] + linearAttenuation[i] \* distance +

quadraticAttenuation[i] \* distance \* distance);

}

else {

// If the fragment is outside of the spotlight cone, then there is no light.

attenuation = 0.0;

}

}

else {

attenuation = 0.0;

}

//calculate Diffuse Color

float NdotL = max(dot(N, lightVector), 0.0);

vec4 diffuseColor = Kdiffuse \* diffuseLightIntensity[i] \* NdotL;

// calculate Specular color. Here we use the original Phong illumination model.

vec3 E = normalize(eyePosition - v);

vec3 R = normalize(-reflect(lightVector, N)); // light reflection vector

float RdotE = max(dot(R, E), 0.0);

vec4 specularColor = Kspecular \* specularLightIntensity[i] \* pow(RdotE, shininess);

// ambient color

vec4 ambientColor = Kambient \* ambientLightIntensity[i];

color += ambientColor + emission + attenuation \* (diffuseColor + specularColor);

}

if (hasTexture == 1) {

// Perform the texture mapping.

// Retrieve the texture color from the texture image using the texture

// coordinates.

vec4 textureColor = texture(texUnit, textureCoord);

// Combine the lighting color with the texture color.

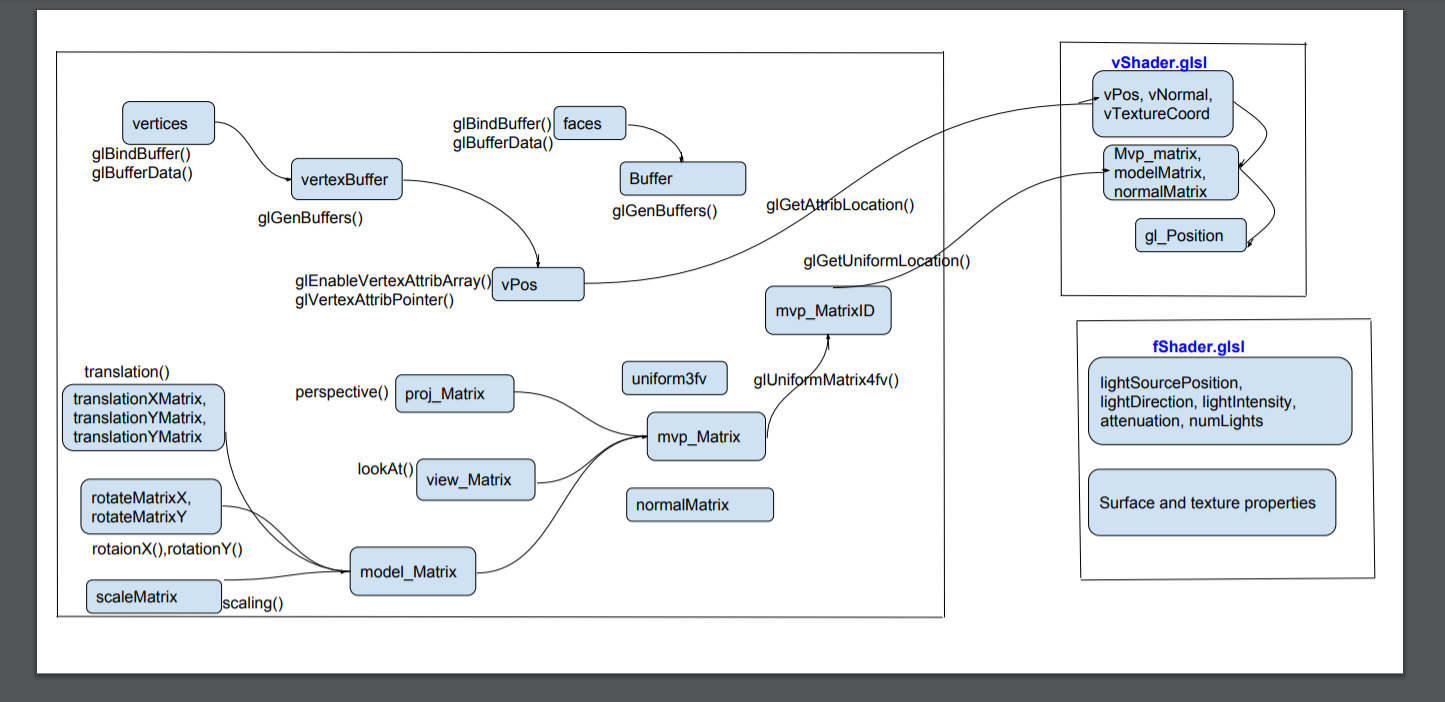
// You can use different methods to combine the two colors.

color = mix(color, textureColor, 0.5f);

}

}

DATA FLOW DIAGRAM



Figure

A close up of a piece of paper

Description generated with very high confidence

Figure