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

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*TODO\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// literally everything

// Find new way to convert string -> float NVM it works

// In addLight...will there be brackets around the falloff??

// or just the int?

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

#define \_CRT\_SECURE\_NO\_DEPRECATE

#include <vector>

#include <iostream>

#include <fstream>

#include <cmath>

#include "include/Eigen/Eigen"

#include "include/glew/include/GL/glew.h"

#include "include/glfw-3.2.1/include/GLFW/glfw3.h"

//#include "include/lpngTWO/png.h"

//#include "include/lpngTWO/zlib-1.2.8/zlib.h"

#define ZLIB\_WINAPI

#include <sstream>

#include <string>

extern "C" {

#include "include/libpngTWO/zlib-1.2.8/zlib.h"

#include "include/libpngTWO/lpng/png.h"

}

#include <fstream>

#include <stdlib.h>

#include <time.h>

#include <math.h>

#define PI 3.14159265 // Should be used from mathlib

using namespace std;

using namespace Eigen;

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

// Global Variables

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

int Width\_global = 700;

int Height\_global = 500;

//add the argument vars here

string amString = "lta"; //ambient arg

string difString = "-kd"; //diffuse arg

string specString = "-ks"; //specular arg

string uPowString = "-spu"; //power u arg

string vPowString = "-spv"; //power v arg

string sPowString = "-sp"; //specular power arg

string pLight = "ltp"; //point light arg

int pLightFalloff = 0;

string dLight = "ltd"; //dir light arg

int numTransforms = 0;

Vector3f RGB;

Vector3f normal;

Vector3f diffuse;

Vector3f specular;

Vector3f ambient;

Vector3f tempXYZ;

Vector3f tempRGB;

Vector3f xyz;

Vector3f tempK(0.0f, 0.0f, 0.0f);

//GL STUFF

GLfloat translation[3] = { 0.0f, 0.0f, 0.0f };

//Point Lights//

ArrayXXf pointLights = ArrayXXf::Zero(5, 6);

int numPointLights = 0;

//directional lights//

ArrayXXf dirLights = ArrayXXf::Zero(5, 6);

int numDirLights = 0;

inline float sqr(float x) { return x\*x; }

int color\_type;

int interlace\_type;

string objFile;

string inFile;

ofstream myfile;

png\_bytepp row\_pointers;

png\_color\_8\_struct pixels[500][700];

Matrix4f TransformMatrix = Matrix4f::Identity();

struct camera {

Vector3f e;// = Vector3f(0.0f, 0.0f, 0.0f);

Vector3f ll;// = Vector3f(0.0f, 0.0f, 0.0f);

Vector3f lr;// = Vector3f(0.0f, 0.0f, 0.0f);

Vector3f ul;// = Vector3f(0.0f, 0.0f, 0.0f);

Vector3f ur;// = Vector3f(0.0f, 0.0f, 0.0f);

camera() :

e(Vector3f(0.0f, 0.0f, 0.0f)),

ll(Vector3f(0.0f, 0.0f, 0.0f)),

lr(Vector3f(0.0f, 0.0f, 0.0f)),

ul(Vector3f(0.0f, 0.0f, 0.0f)),

ur(Vector3f(0.0f, 0.0f, 0.0f))

{}

camera(Vector3f eye, Vector3f lowL, Vector3f lowR, Vector3f upL, Vector3f upR) :

e(eye), ll(lowL), lr(lowR), ul(upL), ur(upR)

{

}

} cam;

struct material {

Vector3f ka = Vector3f(0.0f, 0.0f, 0.0f);

Vector3f kd = Vector3f(0.0f, 0.0f, 0.0f);

Vector3f ks = Vector3f(0.0f, 0.0f, 0.0f);

float ksp;

Vector3f kr = Vector3f(0.0f, 0.0f, 0.0f);

material() :

ka(Vector3f(0.0f, 0.0f, 0.0f)),

kd(Vector3f(0.0f, 0.0f, 0.0f)),

ks(Vector3f(0.0f, 0.0f, 0.0f)),

ksp(0.0f),

kr(Vector3f(0.0f, 0.0f, 0.0f))

{}

material(Vector3f a, Vector3f d, Vector3f s, float p, Vector3f r) :

ka(a), kd(d), ks(s), ksp(p), kr(r)

{

}

} mat;

struct sphere {

Vector3f center;// = Vector3f(0.0f, 0.0f, 0.0f);

float radius;

Matrix4f transform; //= Matrix4f();

Matrix4f invTransform;// = Matrix4f();

material sphMat;

sphere() :

center(Vector3f(0.0f, 0.0f, 0.0f)),

radius(0.0f),

transform(Matrix4f()),

invTransform(Matrix4f()),

sphMat(material())

{}

sphere(Vector3f c, float r, const Matrix4f& t, const Matrix4f& inv, material m) : //{

center(c), radius(r), transform(t), invTransform(inv), sphMat(m)

{

}

} sph;

struct triangle {

Vector3f a = Vector3f(0.0f, 0.0f, 0.0f);

Vector3f b = Vector3f(0.0f, 0.0f, 0.0f);

Vector3f c = Vector3f(0.0f, 0.0f, 0.0f);

} tri;

void addCam(std::vector<std::string> argArray) {

int index = 0;

std::string f(argArray[index + 1]);

cam.e(0) = std::stof(f);

f = argArray[index + 2];

cam.e(1) = std::stof(f);

f = argArray[index + 3];

cam.e(2) = std::stof(f);

f = argArray[index + 4];

cam.ll(0) = std::stof(f);

f = argArray[index + 5];

cam.ll(1) = std::stof(f);

f = argArray[index + 6];

cam.ll(2) = std::stof(f);

f = argArray[index + 7];

cam.lr(0) = std::stof(f);

f = argArray[index + 8];

cam.lr(1) = std::stof(f);

f = argArray[index + 9];

cam.lr(2) = std::stof(f);

f = argArray[index + 10];

cam.ul(0) = std::stof(f);

f = argArray[index + 11];

cam.ul(1) = std::stof(f);

f = argArray[index + 12];

cam.ul(2) = std::stof(f);

f = argArray[index + 13];

cam.ur(0) = std::stof(f);

f = argArray[index + 14];

cam.ur(1) = std::stof(f);

f = argArray[index + 15];

cam.ur(2) = std::stof(f);

//return index + 15;

}

void addSphere(std::vector<std::string> argArray) {

int index = 0;

std::string f(argArray[index + 1]);

sph.center(0) = std::stof(f);

f = argArray[index + 2];

sph.center(1) = std::stof(f);

f = argArray[index + 3];

sph.center(2) = std::stof(f);

f = argArray[index + 4];

sph.radius = std::stof(f);

sph.transform = TransformMatrix;

sph.invTransform = TransformMatrix.inverse();

//return index + 4;

}

void addTri(std::vector<std::string> argArray) {

int index = 0;

std::string f(argArray[index + 1]);

tri.a(0) = std::stof(f);

f = argArray[index + 2];

tri.a(1) = std::stof(f);

f = argArray[index + 3];

tri.a(2) = std::stof(f);

f = argArray[index + 4];

tri.b(0) = std::stof(f);

f = argArray[index + 5];

tri.b(1) = std::stof(f);

f = argArray[index + 6];

tri.b(2) = std::stof(f);

f = argArray[index + 7];

tri.c(0) = std::stof(f);

f = argArray[index + 8];

tri.c(1) = std::stof(f);

f = argArray[index + 9];

tri.c(2) = std::stof(f);

Vector4f tria = Vector4f(tri.a(0), tri.a(1), tri.a(2), 1.0f);

tria = TransformMatrix \* tria;

tri.a(0) = tria(0);

tri.a(1) = tria(1);

tri.a(2) = tria(2);

Vector4f trib = Vector4f(tri.b(0), tri.b(1), tri.b(2), 1.0f);

trib = TransformMatrix \* trib;

tri.b(0) = trib(0);

tri.b(1) = trib(1);

tri.b(2) = trib(2);

Vector4f tric = Vector4f(tri.c(0), tri.c(1), tri.c(2), 1.0f);

tric = TransformMatrix \* tric;

tri.c(0) = tric(0);

tri.c(1) = tric(1);

tri.c(2) = tric(2);

//return index + 9;

}

void addMat(std::vector<std::string> argArray) {

int index = 0;

std::string f(argArray[index + 1]);

mat.ka(0) = std::stof(f);

f = argArray[index + 2];

mat.ka(1) = std::stof(f);

f = argArray[index + 3];

mat.ka(2) = std::stof(f);

f = argArray[index + 4];

mat.kd(0) = std::stof(f);

f = argArray[index + 5];

mat.kd(1) = std::stof(f);

f = argArray[index + 6];

mat.kd(2) = std::stof(f);

f = argArray[index + 7];

mat.ks(0) = std::stof(f);

f = argArray[index + 8];

mat.ks(1) = std::stof(f);

f = argArray[index + 9];

mat.ks(2) = std::stof(f);

f = argArray[index + 10];

mat.ksp = std::stof(f);

f = argArray[index + 11];

mat.kr(0) = std::stof(f);

f = argArray[index + 12];

mat.kr(1) = std::stof(f);

f = argArray[index + 13];

mat.kr(2) = std::stof(f);

//return index + 12;

}

void addTransform(std::vector<std::string> words) {

string arg = words[0];

if (arg.compare("xfr") == 0) {

if (words.size() == 4) {

std::string srx(words[1]);//

std::string sry(words[2]);//

std::string srz(words[3]);//

float rx = std::stof(srx);

float ry = std::stof(sry);

float rz = std::stof(srz);

float theta = Vector3f(rx, ry, rz).norm(); //magnitude of rotation vector

Vector3f rhat = Vector3f(rx, ry, rz);

rhat.normalize();

Matrix3f Rx;

Rx << 0.0f, -rhat(2), rhat(1),

rhat(2), 0.0f, -rhat(0),

-rhat(1), rhat(0), 0.0f;

Matrix3f R = Matrix3f::Identity() + sin(theta)\*Rx + (1 - cos(theta))\*(Rx\*Rx); //Resulting matrix

Matrix4f Rotation;

Rotation << R(0), R(1), R(2), 0.0f,

R(3), R(4), R(5), 0.0f,

R(6), R(7), R(8), 0.0f,

0.0f, 0.0f, 0.0f, 1.0f;

TransformMatrix = TransformMatrix \* Rotation;

numTransforms++;

}

else if (arg.compare("xfs") == 0) {

if (words.size() == 4) {

std::string ssx(words[1]);

std::string ssy(words[2]);

std::string ssz(words[3]);

float sx = std::stof(ssx);

float sy = std::stof(ssy);

float sz = std::stof(ssz);

Matrix4f S;

S << sx, 0.0f, 0.0f, 0.0f,

0.0f, sy, 0.0f, 0.0f,

0.0f, 0.0f, sz, 0.0f,

0.0f, 0.0f, 0.0f, 1.0f;

TransformMatrix = TransformMatrix \* S;

numTransforms++;

}

}

else if (arg.compare("xft") == 0) {

if (words.size() == 4) {

std::string stx(words[1]);

std::string sty(words[2]);

std::string stz(words[3]);

float tx = std::stof(stx);

float ty = std::stof(sty);

float tz = std::stof(stz);

numTransforms++;

Matrix4f T;

T << 1.0f, 0.0f, 0.0f, tx,

0.0f, 1.0f, 0.0f, ty,

0.0f, 0.0f, 1.0f, tz,

0.0f, 0.0f, 0.0f, 1.0f;

TransformMatrix = TransformMatrix \* T;

}

}

}

}

//add Args to correct matricies//

//PLEASE DOUBLE CHECK THE LIVING HELL OUT OF THIS I AM SO RUSTY

//IT HAS BEEN A YEAR SINCE I USED C

void addLight(std::vector<std::string> argArray) {

int index = 0;

int lenArgArray = argArray.size();

string typeLight = argArray[index];

//this is really ugly ... let me know if you know a better way to convert from char\*\* to float

std::string fx(argArray[index + 1]);

float x = std::stof(fx);

std::string fy(argArray[index + 2]);

float y = std::stof(fy);

std::string fz(argArray[index + 3]);

float z = std::stof(fz);

std::string fr(argArray[index + 4]);

float r = std::stof(fr);

std::string fg(argArray[index + 5]);

float g = std::stof(fg);

std::string fb(argArray[index + 6]);

float b = std::stof(fb);

ArrayXXf\* lightMatrix;

int i = 0;

if (typeLight.compare("ltp") == 0) {

lightMatrix = &pointLights;

i = numPointLights;

numPointLights++;

if ((index + 6) < lenArgArray) {

if (argArray[index + 6].compare("1") == 0) {

pLightFalloff = 1;

}

else if (argArray[index + 6].compare("2") == 0) {

pLightFalloff = 2;

}

}

}

else {

lightMatrix = &dirLights;

i = numDirLights;

numDirLights++;

}

// #IMissPython

//wait.. I think I fixed it...

(\*lightMatrix)(i, 0) = x;

(\*lightMatrix)(i, 1) = y;

(\*lightMatrix)(i, 2) = z;

(\*lightMatrix)(i, 3) = r;

(\*lightMatrix)(i, 4) = g;

(\*lightMatrix)(i, 5) = b;

//return index + 6;

}

float get3dDistance(Vector3f c1, Vector3f c2)

{

float dx = c2(0) - c1(0);

float dy = c2(1) - c1(1);

float dz = c2(2) - c1(2);

return sqrt((float)(dx \* dx + dy \* dy + dz \* dz));

}

Vector3f computeAnisotropicSpecular(Vector3f n, Vector3f l, Vector3f lRGB, Vector3f view, Vector3f ks, float sPower, float uPower, float vPower) {

float nu = uPower;

float nv = vPower;

Vector3f y = Vector3f(0.0f, 1.0f, 0.0f);

Vector3f v = y - (n \* n.dot(y));

v.normalize();

Vector3f u = v.cross(n);

u.normalize();

Vector3f h = (view + l);

h.normalize();

float exponent = ((nu \* pow(h.dot(u), 2.0f)) + (nv \* (pow(h.dot(v), 2.0f))))

/ (1.0f - pow(h.dot(n), 2.0f));

Vector3f rgb = Vector3f(0.0f, 0.0f, 0.0f);

Vector3f r = -l + 2.0f \* (normal.dot(l)) \* normal;

r.normalize();

rgb(0) = ks(0) \* pow(max(r.dot(view), 0.0f), exponent);

rgb(1) = ks(1) \* pow(max(r.dot(view), 0.0f), exponent);

rgb(2) = ks(2) \* pow(max(r.dot(view), 0.0f), exponent);

return rgb;

}

Vector3f computeDiffuse(Vector3f normal, Vector3f lightXYZ, Vector3f lRGB, Vector3f kd) {

Vector3f rgb = Vector3f(0.0f, 0.0f, 0.0f);

rgb(0) = lRGB(0) \* kd(0);

rgb(1) = lRGB(1) \* kd(1);

rgb(2) = lRGB(2) \* kd(2);

rgb = rgb \* max(normal.dot(lightXYZ), 0.0f);

return rgb;

}

Vector3f computeSpecular(Vector3f normal, Vector3f lightXYZ, Vector3f lRGB, Vector3f view, Vector3f ks, float sPower) {

Vector3f rgb = Vector3f(0.0f, 0.0f, 0.0f);

Vector3f r = -lightXYZ + 2.0f \* (normal.dot(lightXYZ)) \* normal;

r.normalize();

rgb(0) = lRGB(0) \* ks(0) \* pow(max(r.dot(view), 0.0f), sPower);

rgb(1) = lRGB(1) \* ks(1) \* pow(max(r.dot(view), 0.0f), sPower);

rgb(2) = lRGB(2) \* ks(2) \* pow(max(r.dot(view), 0.0f), sPower);

//rgb = computeAnisotropicSpecular(normal, lightXYZ, view);

rgb(0) = lRGB(0) \* rgb(0);

rgb(1) = lRGB(1) \* rgb(1);

rgb(2) = lRGB(2) \* rgb(2);

return rgb;

}

Vector3f clamp(Vector3f vector) {

Vector3f newVect = Vector3f(0.0f, 0.0f, 0.0f);

newVect(0) = max(min(1.0f, vector(0)), 0.0f);

newVect(1) = max(min(1.0f, vector(1)), 0.0f);

newVect(2) = max(min(1.0f, vector(2)), 0.0f);

return newVect;

}

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

// PHONG EQUATION FOR A VOXEL (i,j, z)

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

Vector3f phongShading(Vector3f surfaceNormal, Vector3f Intensity, Vector3f Ka, Vector3f Kd, Vector3f Ks,

Vector3f lightDir, float sPower) {

diffuse = Vector3f(0.0f, 0.0f, 0.0f);

specular = Vector3f(0.0f, 0.0f, 0.0f);

ambient = Ka;

tempK = Vector3f(0.0f, 0.0f, 0.0f);

float distance;

surfaceNormal.normalize();

//xyz.normalize();

Vector3f tempViewer = surfaceNormal - cam.e;

tempViewer.normalize();

//adding lights

int k = 0;

while (k < numDirLights) {

tempXYZ(0) = dirLights(k, 0);

tempXYZ(1) = dirLights(k, 1);

tempXYZ(2) = dirLights(k, 2);

tempRGB(0) = dirLights(k, 3);

tempRGB(1) = dirLights(k, 4);

tempRGB(2) = dirLights(k, 5);

//tempXYZ = (tempXYZ - Vector3f(centerX, centerY, 0.0f)) / radius;

//tempXYZ.normalize();

tempRGB = clamp(tempRGB);

diffuse += computeDiffuse(normal, -tempXYZ, tempRGB, Kd);

specular += computeSpecular(normal, -tempXYZ, tempRGB, tempViewer, Ks, sPower);

k++;

}

k = 0;

while (k < numPointLights) {

tempXYZ(0) = pointLights(k, 0);

tempXYZ(1) = pointLights(k, 1);

tempXYZ(2) = pointLights(k, 2);

tempRGB(0) = pointLights(k, 3);

tempRGB(1) = pointLights(k, 4);

tempRGB(2) = pointLights(k, 5);

tempRGB = clamp(tempRGB);

//tempXYZ = (tempXYZ - Vector3f(centerX, centerY, 0.0f)) / radius;

//fix this

tempXYZ.normalize();

distance = get3dDistance(tempXYZ, xyz);

diffuse += computeDiffuse(normal, tempXYZ, tempRGB / pow(distance, pLightFalloff), Kd);

specular += computeSpecular(normal, tempXYZ, tempRGB / pow(distance, pLightFalloff), tempViewer, Ks, sPower);

k++;

}

RGB = ambient + diffuse + specular;

RGB = clamp(RGB);

return RGB;

}

void parseLine(string line) {

std::istringstream iss(line);

std::vector<std::string> words;

while (std::getline(iss, line, ' ')) {

words.push\_back(line);

}

myfile << words[0] << std::endl;

myfile << line << std::endl;

string arg = words[0];

if (arg.compare("cam") == 0) {

addCam(words);

}

else if (arg.compare("sph") == 0) {

addSphere(words);

}

else if (arg.compare("tri") == 0) {

addTri(words);

}

else if (arg.compare("obj") == 0) {

//addObjFile(words);

//this may actually be the command line input

//not sure

}

else if (arg.compare("ltd") == 0 | arg.compare("lta") == 0

| arg.compare("ltp") == 0) {

addLight(words);

}

else if (arg.compare("mat") == 0) {

addMat(words);

}

else if (arg.compare("xft") == 0 | arg.compare("xfr") == 0 |

arg.compare("xfs") == 0) {

addTransform(words);

}

else if (arg.compare("xfz") == 0) {

//transformMatrix = Matrix4f::Identity();

//WTF WHY IS THIS SAYING IT DOESN"T EXIST?

}

else if (arg.compare("xfz") == 0) {

//reset transformation?

//I guess just Identity-out the transformation matrix?

//???????????

addTransform(words);

}

//std::cout << line.length();

}

void parseLines() {

//std::ifstream input(objFile);

//for(std::string line; getline(input, line); )

// parseLine(line);

std::ifstream file(inFile);

std::string temp;

while (std::getline(file, temp)) {

myfile << temp << std::endl;

//parseLine(temp);

}

myfile.close();

}

void RayTracerMain() {

for (int i = 0; i < Width\_global; i++) {

for (int j = 0; j < Height\_global; j++) {

//generate ray

//do something useful.

//setColorRGBA(i, j, 0, 0, 0);

}

}

}

const char\* generateFileName() {

//inFile last two chars are the num

//int num = generateFileNum();

string beginning = "image-";

string end = ".png";

string mid = std::to\_string(000);

string filename = beginning + mid + end;

char const \*name = filename.c\_str();

return name;

}

void initPixelMatricies() {

color\_type = PNG\_COLOR\_TYPE\_RGB\_ALPHA;

interlace\_type = PNG\_INTERLACE\_NONE;

for (int i = 0; i < 500; i++) {

//row\_pointers[i] = (png\_color\_8\_struct)pixels[i];

png\_bytep row = new png\_byte;

for (int j = 0; j < 700; j++) {

png\_color\_8\_struct pixel = pixels[i][j];

\*row++ = pixel.red;

\*row++ = pixel.green;

\*row++ = pixel.blue;

}

row\_pointers[i] = row;

}

}

void setColorRGBA(int i, int j, png\_byte r, png\_byte g, png\_byte b) {

pixels[i][j].red = (png\_byte)r;

pixels[i][j].green = (png\_byte)g;

pixels[i][j].blue = (png\_byte)b;

pixels[i][j].alpha = (png\_byte)255; //full opacity

//myfile << "pixels[i][j]:" << i << ";" << j << pixels[i][j].red<< std::endl;

//myfile << rowAddr << std::endl;

}

void writeFinalPNG(char \*filename) {

//int y;

FILE \*fp = fopen(filename, "wb");

if (!fp) abort();

png\_structp png = png\_create\_write\_struct(PNG\_LIBPNG\_VER\_STRING, NULL, NULL, NULL);

if (!png) abort();

png\_infop info = png\_create\_info\_struct(png);

if (!info) abort();

if (setjmp(png\_jmpbuf(png))) abort();

png\_init\_io(png, fp);

png\_set\_IHDR(

png,

info,

500, 700,

8,

PNG\_COLOR\_TYPE\_RGBA,

PNG\_INTERLACE\_NONE,

PNG\_COMPRESSION\_TYPE\_DEFAULT,

PNG\_FILTER\_TYPE\_DEFAULT

);

png\_write\_info(png, info);

png\_write\_image(png, row\_pointers);

png\_write\_end(png, NULL);

for (int y = 0; y < 700; y++) {

//free(row\_pointers[y]);

}

//free(row\_pointers);

fclose(fp);

}

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

// myfile.open("redVals.txt");

//

// initPixelMatricies();

// for (int i = 0; i < 500; i++) {

// for (int j = 0; j < 700; j++) {

// setColorRGBA(i, j, (png\_byte)255, (png\_byte)0, (png\_byte)0);

// }

//

// }

// char \*fileNameNew = "../../test.png";

//

// writeFinalPNG(fileNameNew);

// myfile.close();

// return 0;

//}

void initializeRendering()

{

glfwInit();

}

void setPixel(float x, float y, GLfloat r, GLfloat g, GLfloat b) {

glColor3f(r, g, b);

glVertex2f(x + 0.5, y + 0.5); // The 0.5 is to target pixel centers

// Note: Need to check for gap bug on inst machines.

}

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

// Keyboard inputs

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

//static void key\_callback(GLFWwindow\* window, int key, int scancode, int action, int mods)

//{

// switch (key) {

//

// case GLFW\_KEY\_ESCAPE: glfwSetWindowShouldClose(window, GLFW\_TRUE); break;

// case GLFW\_KEY\_Q: glfwSetWindowShouldClose(window, GLFW\_TRUE); break;

// case GLFW\_KEY\_LEFT:

// if (action) translation[0] -= 0.01f \* Width\_global; break;

// case GLFW\_KEY\_RIGHT:

// if (action) translation[0] += 0.01f \* Width\_global; break;

// case GLFW\_KEY\_UP:

// if (action) translation[1] += 0.01f \* Height\_global; break;

// case GLFW\_KEY\_DOWN:

// if (action) translation[1] -= 0.01f \* Height\_global; break;

// case GLFW\_KEY\_F:

// if (action) auto\_strech = !auto\_strech; break;

// case GLFW\_KEY\_SPACE: break;

//

// default: break;

// }

//

//}

//

//int max(int a, int b) {

// if (a > b) {

// return a;

// }

// return b;

//}

//int min(int a, int b) {

// if (a > b) {

// return b;

// }

// return b;

//}

void display(GLFWwindow\* window)

{

glClearColor(0.0f, 0.0f, 0.0f, 0.0f); //clear background screen to black

glClear(GL\_COLOR\_BUFFER\_BIT); // clear the color buffer (sets everything to black)

glMatrixMode(GL\_MODELVIEW); // indicate we are specifying camera transformations

glLoadIdentity(); // make sure transformation is "zero'd"

//----------------------- code to draw objects --------------------------

glPushMatrix();

glTranslatef(translation[0], translation[1], translation[2]);

//drawCircle(Width\_global / 2.0, Height\_global / 2.0, min(Width\_global, Height\_global) \* 0.8 / 2.0);

for (int i = 0; i < 500; i++) {

for (int j = 0; j < 700; j++) {

png\_color\_8\_struct pix = pixels[i][j];

float r = (float)pix.red;

float g = (float)pix.green;

float b = (float)pix.blue;

setPixel(i, j, r, g, b);

}

}

glPopMatrix();

glfwSwapBuffers(window);

}

void size\_callback(GLFWwindow\* window, int width, int height)

{

// Get the pixel coordinate of the window

// it returns the size, in pixels, of the framebuffer of the specified window

glfwGetFramebufferSize(window, &Width\_global, &Height\_global);

glViewport(0, 0, Width\_global, Height\_global);

glMatrixMode(GL\_PROJECTION);

glLoadIdentity();

glOrtho(0, Width\_global, 0, Height\_global, 1, -1);

display(window);

}

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

//This initializes glfw

bool arg = false;

string printOut = "there was an arg!";

//viewer.normalize();

initializeRendering();

//if (argc >= 2) {

// arg = true;

// string argName;

// int next;

// for (int i = 0; i < argc; i++) {

// argName = argv[i];

// printf("%s\n", argName.c\_str());

// next = i;

// if (argName.compare(amString) == 0 | argName.compare(difString) == 0 | argName.compare(specString) == 0) {

// //call function for these rgb vals

// //check that next three values are floats

// //0, 1, 2 = x, y, z

// //3, 4, 5 = r,g,b, must be between 0 - 1

// //check that argc >= i + 3

// //next = func return;

// next = addCoeffsRGBS(argv, i);

// }

// else if (argName.compare(uPowString) == 0 | argName.compare(vPowString) == 0 | argName.compare(sPowString) == 0) {

// //check that the next value is a float

// //next val is > 0 & val < max\_float

// next = addPowCoeff(argv, i);

// }

// else if (argName.compare("-asm") == 0) {

// isPhong = false;

// }

// else

// if (argName.compare(pLight) == 0 | argName.compare(dLight) == 0) {

// //check that the next 6 values are floats

// next = addLight(argv, i);

// }

// i = next;

// }

//}

GLFWwindow\* window = glfwCreateWindow(Width\_global, Height\_global, "CS184", NULL, NULL);

if (!window)

{

cerr << "Error on window creating" << endl;

glfwTerminate();

return -1;

}

const GLFWvidmode \* mode = glfwGetVideoMode(glfwGetPrimaryMonitor());

if (!mode)

{

cerr << "Error on getting monitor" << endl;

glfwTerminate();

return -1;

}

glfwMakeContextCurrent(window);

// Get the pixel coordinate of the window

// it returns the size, in pixels, of the framebuffer of the specified window

glfwGetFramebufferSize(window, &Width\_global, &Height\_global);

glMatrixMode(GL\_PROJECTION);

glLoadIdentity();

glOrtho(0, Width\_global, 0, Height\_global, 1, -1);

glfwSetWindowTitle(window, "CS184");

glfwSetWindowSizeCallback(window, size\_callback);

//glfwSetKeyCallback(window, key\_callback);

while (!glfwWindowShouldClose(window)) // infinite loop to draw object again and again

{ // because once object is draw then window is terminated

display(window);

/\*if (auto\_strech) {

glfwSetWindowSize(window, mode->width, mode->height);

glfwSetWindowPos(window, 0, 0);

}\*/

glfwPollEvents();

}

return 0;

}