#include <iostream>

#include <GL/glew.h>

#include <GLFW/glfw3.h>

#include <glm.hpp>

#include <gtc/matrix\_transform.hpp>

#include <gtc/type\_ptr.hpp>

#include <cstdlib>

#include <math.h>

#include <stdlib.h>

#include <vector>

#include <iostream>

#include <fstream>

#include <thread>

#include <FastNoise.h>

using namespace std;

float positions[] = {

-0.5f, -0.5f, -0.5f, 1.0f, 0.0f, 0.0f,

0.5f, -0.5f, -0.5f,1.0f, 0.0f, 0.0f,

0.5f, 0.5f, -0.5f, 1.0f, 0.0f, 0.0f, // BACK

0.5f, 0.5f, -0.5f,1.0f, 0.0f, 0.0f,

-0.5f, 0.5f, -0.5f,1.0f, 0.0f, 0.0f,

-0.5f, -0.5f, -0.5f,1.0f, 0.0f, 0.0f,

-0.5f, -0.5f, 0.5f, 0.0f,1.0f, 0.0f,

0.5f, -0.5f, 0.5f, 0.0f,1.0f, 0.0f,

0.5f, 0.5f, 0.5f, 0.0f,1.0f, 0.0f, // FRONT

0.5f, 0.5f, 0.5f, 0.0f,1.0f, 0.0f,

-0.5f, 0.5f, 0.5f, 0.0f,1.0f, 0.0f,

-0.5f, -0.5f, 0.5f, 0.0f,1.0f, 0.0f,

-0.5f, 0.5f, 0.5f, 0.0f, 0.0f,1.0f,

-0.5f, 0.5f, -0.5f,0.0f, 0.0f,1.0f,

-0.5f, -0.5f, -0.5f,0.0f, 0.0f,1.0f, //LEFT

-0.5f, -0.5f, -0.5f,0.0f, 0.0f,1.0f,

-0.5f, -0.5f, 0.5f,0.0f, 0.0f,1.0f,

-0.5f, 0.5f, 0.5f,0.0f, 0.0f,1.0f,

0.5f, 0.5f, 0.5f,0.5f, 0.5f,0.0f,

0.5f, 0.5f, -0.5f,0.5f, 0.5f,0.0f,

0.5f, -0.5f, -0.5f,0.5f, 0.5f,0.0f, // RIGHT

0.5f, -0.5f, -0.5f,0.5f, 0.5f,0.0f,

0.5f, -0.5f, 0.5f,0.5f, 0.5f,0.0f,

0.5f, 0.5f, 0.5f,0.5f, 0.5f,0.0f,

-0.5f, -0.5f, -0.5f, 0.5f,0.0f, 0.5f,

0.5f, -0.5f, -0.5f, 0.5f,0.0f, 0.5f,

0.5f, -0.5f, 0.5f, 0.5f,0.0f, 0.5f, // BOTTOM

0.5f, -0.5f, 0.5f, 0.5f,0.0f, 0.5f,

-0.5f, -0.5f, 0.5f, 0.5f,0.0f, 0.5f,

-0.5f, -0.5f, -0.5f, 0.5f,0.0f, 0.5f,

-0.5f, 0.5f, -0.5f,0.0f, 0.5f, 0.5f,

0.5f, 0.5f, -0.5f, 0.0f, 0.5f, 0.5f,

0.5f, 0.5f, 0.5f, 0.0f, 0.5f, 0.5f, // TOP

0.5f, 0.5f, 0.5f, 0.0f, 0.5f, 0.5f,

-0.5f, 0.5f, 0.5f,0.0f, 0.5f, 0.5f,

-0.5f, 0.5f, -0.5f,0.0f, 0.5f, 0.5f

};

glm::mat4 projectionTransform(1.0f);

GLFWwindow\* window;

FastNoise noiseGenerator;

int vertexCt = 0;

int verticesInCube = 6 \* 6;

unsigned int shader;

bool shouldGenerateChunks = true;

bool regenerateChunks = true;

struct {

float width = 800;

float height = 600;

float fov = 45.0;

} screen;

struct {

struct {

double x;

double y;

} mouse;

struct {

bool isKeyPressed[GLFW\_KEY\_MENU];

} keyboard;

} input;

struct {

struct {

float x = 0.0f;

float y = 0.0f;

float z = 0.0f;

} position;

struct {

float x = 0;

float y = 0;

float z = 0;

} looking;

struct {

int health = 5;

int hunger = 0;

} vitals;

float height = 1.2f;

float forwardSpeed = 0.15f;

float strafeSpeed = 0.1f;

float flySpeed = 1.0f;

bool isJumping = false;

} player;

struct {

struct {

struct {

int x;

int z;

} current;

int size = 17;

int totalCount = 0;

static const int toGenerate = 6;

static const int toRender = 5;

int rendered = 0;

} chunks;

} world;

bool flying = true;

float jumpCounter = 0;

float jumpHeight = player.height \* 2;

float jumpSpeed = 6;

int jumpBlockX = 0;

int jumpBlockZ = 0;

const int chunksToRenderAcross = world.chunks.toRender \* 2 + 1;

const int chunksToGenerateAcross = world.chunks.toGenerate \* 2 + 1;

float \*\*\*chunks = new float\*\*[1000000];

float cameraRotX = 0, cameraRotY = 0;

float cameraSpeed = 0.5f;

int \*chunkX = new int[10000];

int \*chunkZ = new int[10000];

intptr\_t \*chunkAddresses = new intptr\_t[10000];

ofstream myFile;

int dirX;

int dirZ;

#define BIOME\_PLAINS 0

int getHeight(int biome, int x, int z){

int result;

switch (biome) {

case BIOME\_PLAINS:

result = noiseGenerator.GetNoise(x, z) \* 15;

break;

}

result = result;

return result;

}

float\*\* addChunk(float chunkX, float chunkZ) {

//vector<float> blocks((world.chunks.size+2) \* 255 \* (world.chunks.size + 2));

int \*blocks = new int[(world.chunks.size + 2) \* 255 \* (world.chunks.size + 2)];

int maxBlocks = 0;

int totalBlocks = 0;

//vector<float> chunkMesh(10000000);

float \*chunkMesh = new float[2000000];

float \*xPos = new float[1000000];

float \*yPos = new float[1000000];

float \*zPos = new float[1000000];

int a = 0;

int iter = 0;

int chunkLim = (world.chunks.size - 1) / 2;

int chunkLimY = 90;

for (int x = -chunkLim - 1; x <= chunkLim + 1; x++) {

for (int y = -chunkLimY - 1; y <= chunkLimY + 1; y++) {

for (int z = -chunkLim - 1; z <= chunkLim + 1; z++) {

//createBlock(glm::vec3(x + world.chunks.size \* (chunkX[i]), 0.0f, z + world.chunks.size \* (chunkZ[i])), glm::vec3(1.0f), glm::vec3(1.0f), glm::vec4(0.0f, (x % 2 == 0 && z % 2 == 0) ? (0.8f) : (0.5f), 0.0f, 1.0f));

if (x >= -chunkLim &&

x <= chunkLim &&

y >= -chunkLimY &&

y <= chunkLimY &&

z >= -chunkLim &&

z <= chunkLim) {

int heightVal = getHeight(BIOME\_PLAINS, world.chunks.size\*chunkX + x, world.chunks.size\*chunkZ + z);

if (y < heightVal) {

blocks[totalBlocks] = 9;

xPos[totalBlocks] = x;

yPos[totalBlocks] = y;

zPos[totalBlocks] = z;

}

}

totalBlocks++;

//cout << totalBlocks << endl;

}

}

}

for (int x = 0; x < totalBlocks; x++) {

//myFile << "type: " << blocks[x] << " ... " << xPos[x] << " " << yPos[x] << " " << zPos[x] << endl;

if (!(blocks[x] < 0)) {

bool renderFront = false;

bool renderBack = false;

bool renderTop = false;

bool renderBottom = false;

bool renderLeft = false;

bool renderRight = false;

if (blocks[x - 1] < 0) {

renderBack = true;

// BACK

}

if (blocks[x + 1] < 0) {

// FRONT

renderFront = true;

}

if (blocks[x - (world.chunks.size + 2)] < 0) { // 2 is offset to account for "air" blocks surrounding chunk

renderBottom = true;

// BOTTOM

}

if (blocks[x + (world.chunks.size + 2)] < 0) { // 2 is offset to account for "air" blocks surrounding chunk

renderTop = true;

// TOP

}

if (blocks[x - (world.chunks.size + 2) \* (chunkLimY \* 2 + 1 + 2)] <0) { // 2 is offset to account for "air" blocks surrounding chunk. 1 accounts for 0

renderLeft = true;

// LEFT

}

if (blocks[x + (world.chunks.size + 2) \* (chunkLimY \* 2 + 1 + 2)] <0) { // 2 is offset to account for "air" blocks surrounding chunk. 1 accounts for 0

renderRight = true;

// RIGHT

}

// BACK FRONT LEFT RIGHT BOTTOM TOP

for (int i = 0; i < 6; i++) {

if (i == 0 && !renderBack ||

i == 1 && !renderFront ||

i == 2 && !renderLeft ||

i == 3 && !renderRight ||

i == 4 && !renderBottom ||

i == 5 && !renderTop) {

continue;

}

for (int j = 0; j < 36; j++) {

float value;

if (j % 6 == 0) {

value = positions[36 \* i + j] + xPos[x];

}

if (j % 6 == 1) {

value = positions[36 \* i + j] + yPos[x];

}

if (j % 6 == 2) {

value = positions[36 \* i + j] + zPos[x];

}

if (j % 6 == 3) {

value = positions[36 \* i + j];

}

if (j % 6 == 4) {

value = positions[36 \* i + j];

}

if (j % 6 == 5) {

value = positions[36 \* i + j];

}

chunkMesh[iter] = value;

//cout << iter << endl;

iter++;

}

}

}

}

delete[] blocks;

blocks = NULL;

delete[] xPos;

xPos = NULL;

delete[] yPos;

yPos = NULL;

delete[] zPos;

zPos = NULL;

float \*\*chunk = new float\*[3];

float \*chunX = new float;

\*chunX = chunkX;

float \*chunZ = new float;

\*chunZ = chunkZ;

chunk[0] = chunX;

chunk[1] = chunZ;

chunk[2] = chunkMesh;

return chunk;

}

void jump(float height, float speed) {

if (!player.isJumping) {

jumpHeight = height;

jumpSpeed = speed;

player.isJumping = true;

jumpBlockX = player.position.x;

jumpBlockZ = player.position.z;

jumpCounter = 0;

}

}

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

{

input.keyboard.isKeyPressed[key] = action;

if (key == GLFW\_KEY\_F && action == GLFW\_RELEASE) {

cout << "here" << endl;

flying = !flying;

}

}

void handleInput() {

if (input.keyboard.isKeyPressed[GLFW\_KEY\_W]) {

player.position.z -= (flying?player.flySpeed:player.forwardSpeed) \* cos(glm::radians(cameraRotY));

player.position.x += (flying ? player.flySpeed : player.forwardSpeed) \* sin(glm::radians(cameraRotY));

}

if (input.keyboard.isKeyPressed[GLFW\_KEY\_S]) {

player.position.z += (flying ? player.flySpeed : player.forwardSpeed) \* cos(glm::radians(cameraRotY));

player.position.x -= (flying ? player.flySpeed : player.forwardSpeed) \* sin(glm::radians(cameraRotY));

}

if (input.keyboard.isKeyPressed[GLFW\_KEY\_A]) {

player.position.x -= (flying ? player.flySpeed : player.forwardSpeed) \* cos(glm::radians(cameraRotY));

player.position.z -= (flying ? player.flySpeed : player.forwardSpeed) \* sin(glm::radians(cameraRotY));

}

if (input.keyboard.isKeyPressed[GLFW\_KEY\_D]) {

player.position.x += (flying ? player.flySpeed : player.forwardSpeed) \* cos(glm::radians(cameraRotY));

player.position.z += (flying ? player.flySpeed : player.forwardSpeed) \* sin(glm::radians(cameraRotY));

}

if (input.keyboard.isKeyPressed[GLFW\_KEY\_LEFT\_SHIFT]) {

if (flying) {

player.position.y -= player.forwardSpeed;

}

else {

// CROUCH

}

}

if (input.keyboard.isKeyPressed[GLFW\_KEY\_SPACE]) {

if (flying) {

player.position.y += player.forwardSpeed;

}

else {

jump(player.height \* 2, 6);

}

}

int newCurrChunkX = floor((player.position.x + world.chunks.size / 2) / world.chunks.size);

int newCurrChunkZ = floor((player.position.z + world.chunks.size / 2) / world.chunks.size);

if (!(world.chunks.current.x == newCurrChunkX) ||

!(world.chunks.current.z == newCurrChunkZ)) {

dirX = newCurrChunkX - world.chunks.current.x;

dirZ = newCurrChunkZ - world.chunks.current.z;

world.chunks.current.x = newCurrChunkX;

world.chunks.current.z = newCurrChunkZ;

}

}

static int CompileShader(unsigned int type, const string& source) {

int id = glCreateShader(type);

const char\* src = source.c\_str();

glShaderSource(id, 1, &src, nullptr);

glCompileShader(id);

return id;

}

static unsigned int CreateShader(const string& vertexShader, const string& fragmentShader) {

unsigned int program = glCreateProgram();

unsigned int vs = CompileShader(GL\_VERTEX\_SHADER, vertexShader);

unsigned int fs = CompileShader(GL\_FRAGMENT\_SHADER, fragmentShader);

glAttachShader(program, vs);

glAttachShader(program, fs);

glLinkProgram(program);

glValidateProgram(program);

glDeleteShader(vs);

glDeleteShader(fs);

return program;

}

int cube[36];

int blocks = 0;

unsigned int buffer;

void renderChunk(float\*\*& chunk) {

glBufferData(GL\_ARRAY\_BUFFER, 1000000, chunk[2], GL\_DYNAMIC\_DRAW);

glm::mat4 modelTranslate(1.0f);

glm::mat4 modelRotate(1.0f);

glm::mat4 modelScale(1.0f);

modelTranslate = glm::translate(modelTranslate, glm::vec3(world.chunks.size \* \*chunk[0] - player.position.x, 0.0f - player.position.y, world.chunks.size \* \*chunk[1] - player.position.z));

glm::mat4 modelTransform = modelRotate \* modelTranslate \* modelScale;

int modelUniform = glGetUniformLocation(shader, "model");

glUniformMatrix4fv(modelUniform, 1, GL\_FALSE, glm::value\_ptr(modelTransform));

glDrawArrays(GL\_TRIANGLES, 0, 1000000);

}

void generateChunks() {

for (int x = 0; x < chunksToGenerateAcross; x++) {

for (int z = 0; z < chunksToGenerateAcross; z++) {

bool alreadyExists = false;

int a = 0;

for (int i = 0; i < world.chunks.totalCount; i++) {

if (\*chunks[i] != 0) {

if (\*chunks[i][0] == world.chunks.current.x + x - world.chunks.toGenerate &&

\*chunks[i][1] == world.chunks.current.z + z - world.chunks.toGenerate) {

alreadyExists = true;

}

if (\*chunks[i][0] < world.chunks.current.x - world.chunks.toGenerate ||

\*chunks[i][0] > world.chunks.current.x + world.chunks.toGenerate ||

\*chunks[i][1] < world.chunks.current.z - world.chunks.toGenerate ||

\*chunks[i][1] > world.chunks.current.z + world.chunks.toGenerate) {

delete[] chunks[i][0];

delete[] chunks[i][1];

delete[] chunks[i][2];

\*chunks[i] = 0;

}

}

}

if (!alreadyExists) {

//chunks[world.chunks.totalCount - chunksAcross \* chunksAcross] = 0;

chunks[world.chunks.totalCount] = addChunk(world.chunks.current.x + x - world.chunks.toGenerate, world.chunks.current.z + z - world.chunks.toGenerate);

world.chunks.totalCount++;

}

//cout << world.chunks.totalCount << " ... " << x << ", " << z << endl;

}

regenerateChunks = false;

}

while (!regenerateChunks) {

}

generateChunks();

}

int main(void)

{

if (!glfwInit())

return -1;

window = glfwCreateWindow(screen.width, screen.height, "Hello World", NULL, NULL);

if (!window)

{

glfwTerminate();

return -1;

}

glfwMakeContextCurrent(window);

if (glewInit() != GLEW\_OK) {

//cout << "glew Error" << endl;

}

glEnable(GL\_DEPTH\_TEST);

glfwSetInputMode(window, GLFW\_CURSOR, GLFW\_CURSOR\_HIDDEN);

glfwSetKeyCallback(window, key\_callback);

string vertexShader =

"#version 330 core\n"

"\n"

"layout(location = 0) in vec4 position;"

"layout(location = 1) in vec3 color;"

"out vec4 fragColor;"

"uniform mat4 view;"

"uniform mat4 model;"

"uniform mat4 projection;"

"\n"

"void main(){\n"

"fragColor = vec4(color,1.0);"

"gl\_Position = projection \* view \* model \* position;"

"}\n";

string fragmentShader =

"#version 330 core\n"

"\n"

"in vec4 fragColor;"

"out vec4 outColor;"

"uniform vec4 color;"

"\n"

"void main(){\n"

"outColor = fragColor;"

"}\n";

shader = CreateShader(vertexShader, fragmentShader);

glUseProgram(shader);

int projectionUniform = glGetUniformLocation(shader, "projection");

int viewUniform = glGetUniformLocation(shader, "view");

myFile.open("blocks.txt", ios::trunc);

noiseGenerator.SetNoiseType(FastNoise::PerlinFractal);

player.position.y = getHeight(BIOME\_PLAINS, player.position.x, player.position.z) + player.height;

glGenBuffers(1, &buffer);

glBindBuffer(GL\_ARRAY\_BUFFER, buffer);

glVertexAttribPointer(0, 3, GL\_FLOAT, GL\_FALSE, sizeof(float) \* 6, 0);

glVertexAttribPointer(1, 3, GL\_FLOAT, GL\_FALSE, sizeof(float) \* 6, (void\*)(3 \* sizeof(float)));

glEnableVertexAttribArray(0);

glEnableVertexAttribArray(1);

/\*

if (world.chunks.size % 2 == 0) {

world.chunks.size++;

}

\*/

thread t1(generateChunks);

/\* Loop until the user closes the window \*/

while (!glfwWindowShouldClose(window))

{

/\* Render here \*/

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

cout << "playerPosition: " << player.position.x << ", " <<player.position.y << ", " << player.position.z << endl;

if (glfwGetWindowAttrib(window, GLFW\_FOCUSED))

{

glfwGetCursorPos(window, &input.mouse.x, &input.mouse.y);

if (input.mouse.x > 0 && input.mouse.y > 0 && input.mouse.x < screen.width && input.mouse.y < screen.height) {

handleInput();

//cout << "mouse: " << input.mouse.x <<", " << input.mouse.y << endl;

if (cameraRotX < 90 && cameraRotX > -90 ||

(cameraRotX >= 90 && (input.mouse.y - screen.height / 2) \* cameraSpeed < 0) ||

(cameraRotX <= -90 && (input.mouse.y - screen.height / 2) \* cameraSpeed > 0)) {

cameraRotX += (input.mouse.y - screen.height / 2) \* cameraSpeed;

}

cameraRotY += (input.mouse.x - screen.width / 2) \* cameraSpeed;

//cout << "cameraRot: " << cameraRotX << ", " << cameraRotY << endl;

player.looking.x = sin(glm::radians(cameraRotY));

player.looking.y = -sin(glm::radians(cameraRotX));

player.looking.z = -cos(glm::radians(cameraRotY));

//cout << "looking: " << player.looking.x << ", " << player.looking.y << ", " << player.looking.z << endl;

}

glfwSetCursorPos(window, screen.width / 2, screen.height / 2);

}

if (!flying) {

if (!player.isJumping) {

if (player.position.y > getHeight(BIOME\_PLAINS, player.position.x, player.position.z) + player.height) {

player.position.y -= 0.25f;

}

else if (player.position.y < getHeight(BIOME\_PLAINS, player.position.x, player.position.z) + player.height - 0.25f) {

jump(player.height \* .7, 9);

}

else {

player.position.y = getHeight(BIOME\_PLAINS, player.position.x, player.position.z) + player.height;

}

}

else {

jumpCounter++;

player.position.y = getHeight(BIOME\_PLAINS, jumpBlockX, jumpBlockZ) + player.height + sin(glm::radians(jumpCounter \* jumpSpeed)) \* jumpHeight;

if (player.position.y < getHeight(BIOME\_PLAINS, player.position.x, player.position.z) + player.height) {

player.position.y = getHeight(BIOME\_PLAINS, player.position.x, player.position.z) + player.height;

player.isJumping = false;

jumpCounter = 0;

}

}

}

/\*

if (player.position.y > getHeight(BIOME\_PLAINS, player.position.x, player.position.z)) {

if (!player.isJumping) {

}

else{

player.position.y = getHeight(BIOME\_PLAINS, player.position.x, player.position.z) + 1.5 + sin(counter);

}

}

\*/

if (world.chunks.totalCount > 0) {

for (int i = 0; i < world.chunks.totalCount; i++) {

if (\*chunks[i] != 0) {

int shouldRender = true;

//cout << world.chunks.current.x << ", " << world.chunks.current.z << endl;

if ((player.looking.z < -.5 && world.chunks.current.z < \*chunks[i][1]) || \*chunks[i][1] < world.chunks.current.z - world.chunks.toRender) {

shouldRender = false;

}

if ((player.looking.z > .5 && world.chunks.current.z > \*chunks[i][1]) || \*chunks[i][1] > world.chunks.current.z + world.chunks.toRender) {

shouldRender = false;

}

if ((player.looking.x < -.5 && world.chunks.current.x <\*chunks[i][0]) || \*chunks[i][0] < world.chunks.current.x - world.chunks.toRender) {

shouldRender = false;

}

if ((player.looking.x > .5 && world.chunks.current.x > \*chunks[i][0]) || \*chunks[i][0] > world.chunks.current.x + world.chunks.toRender) {

shouldRender = false;

}

if (shouldRender) {

renderChunk(chunks[i]);

}

}

}

}

regenerateChunks = true;

//renderChunk();

/\*

if (world.chunks.totalCount > 0) {

blocks = 0;

for (int i = 0; i < world.chunks.totalCount; i++) {

cout << sizeof(blocks) << endl;

if ((player.looking.z < -.5 && chunkZ[i] <= world.chunks.current.z && chunkZ[i] > world.chunks.current.z - world.chunks.toRender && chunkX[i]> world.chunks.current.x - world.chunks.toRender && chunkX[i]<world.chunks.current.x + world.chunks.toRender) ||

(player.looking.z > .5 && chunkZ[i] >= world.chunks.current.z && chunkZ[i] < world.chunks.current.z + world.chunks.toRender && chunkX[i]> world.chunks.current.x - world.chunks.toRender && chunkX[i]<world.chunks.current.x + world.chunks.toRender) ||

(player.looking.x < -.5 && chunkX[i] <= world.chunks.current.x && chunkX[i] > world.chunks.current.x - world.chunks.toRender && chunkZ[i]> world.chunks.current.z - world.chunks.toRender && chunkZ[i]<world.chunks.current.z + world.chunks.toRender) ||

(player.looking.x > .5 && chunkX[i] >= world.chunks.current.x && chunkX[i] < world.chunks.current.x + world.chunks.toRender && chunkZ[i]> world.chunks.current.z - world.chunks.toRender && chunkZ[i]<world.chunks.current.z + world.chunks.toRender)) {

for (int x = -world.chunks.size / 2; x < world.chunks.size / 2; x++) {

for (int z = -world.chunks.size / 2; z < world.chunks.size / 2; z++) {

createBlock(glm::vec3(x + world.chunks.size \* (chunkX[i]), 0.0f, z + world.chunks.size \* (chunkZ[i])), glm::vec3(1.0f), glm::vec3(1.0f), glm::vec4(0.0f, (x % 2 == 0 && z % 2 == 0) ? (0.8f) : (0.5f), 0.0f, 1.0f));

}

}

world.chunks.rendered++;

}

}

}

else {

createBlock(glm::vec3(0.0f,0.0f,0.0f), glm::vec3(1.0f), glm::vec3(1.0f), glm::vec4(0.0f, 1.0f, 0.0f, 1.0f));

}

\*/

//cout << blocks << endl;

//cout << "world.chunks.rendered: " << world.chunks.rendered << "/" << world.chunks.totalCount << endl;

projectionTransform = glm::perspective(glm::radians(screen.fov), screen.width / screen.height, 0.1f, 255.0f);

glUniformMatrix4fv(projectionUniform, 1, GL\_FALSE, glm::value\_ptr(projectionTransform));

glm::mat4 viewTransform(1.0f);

glm::mat4 viewRotateX(1.0f);

glm::mat4 viewRotateY(1.0f);

viewRotateX = glm::rotate(viewRotateX, glm::radians(cameraRotX), glm::vec3(1.0f, 0.0f, 0.0f));

viewRotateY = glm::rotate(viewRotateY, glm::radians(cameraRotY), glm::vec3(0.0f, 1.0f, 0.0f));

viewTransform = viewRotateX \* viewRotateY;// \*viewRotate \* viewScale;

glUniformMatrix4fv(viewUniform, 1, GL\_FALSE, glm::value\_ptr(viewTransform));

world.chunks.rendered = 0;

//system("cls");

glfwSwapBuffers(window);

glfwPollEvents();

}

t1.detach();

for (int i = 0; i < world.chunks.totalCount; i++) {

for (int z = 0; z < 3; z++) {

delete[] chunks[i][z];

}

delete[] chunks[i];

}

delete[] chunks;

glDeleteProgram(shader);

glfwTerminate();

return 0;

}