#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>

#include <algorithm>

using namespace std;

float positions[] = {

-0.5f, -0.5f, -0.5f, 0.341f, 0.231f, 0.047f,

0.5f, -0.5f, -0.5f,0.341f, 0.231f, 0.047f,

0.5f, 0.5f, -0.5f, 0.341f, 0.231f, 0.047f, // BACK

0.5f, 0.5f, -0.5f,0.341f, 0.231f, 0.047f,

-0.5f, 0.5f, -0.5f,0.341f, 0.231f, 0.047f,

-0.5f, -0.5f, -0.5f,0.341f, 0.231f, 0.047f,

-0.5f, -0.5f, 0.5f, 0.341f,0.231f, 0.047f,

0.5f, -0.5f, 0.5f, 0.341f,0.231f, 0.047f,

0.5f, 0.5f, 0.5f, 0.341f,0.231f, 0.047f, // FRONT

0.5f, 0.5f, 0.5f, 0.341f,0.231f, 0.047f,

-0.5f, 0.5f, 0.5f, 0.341f,0.231f, 0.047f,

-0.5f, -0.5f, 0.5f, 0.341f,0.231f, 0.047f,

-0.5f, 0.5f, 0.5f, 0.341f, 0.231f,0.047f,

-0.5f, 0.5f, -0.5f,0.341f, 0.231f,0.047f,

-0.5f, -0.5f, -0.5f,0.341f, 0.231f,0.047f, //LEFT

-0.5f, -0.5f, -0.5f,0.341f, 0.231f,0.047f,

-0.5f, -0.5f, 0.5f,0.341f, 0.231f,0.047f,

-0.5f, 0.5f, 0.5f,0.341f, 0.231f,0.047f,

0.5f, 0.5f, 0.5f,0.341f, 0.231f,0.047f,

0.5f, 0.5f, -0.5f,0.341f, 0.231f,0.047f,

0.5f, -0.5f, -0.5f,0.341f, 0.231f,0.047f, // RIGHT

0.5f, -0.5f, -0.5f,0.341f, 0.231f,0.047f,

0.5f, -0.5f, 0.5f,0.341f, 0.231f,0.047f,

0.5f, 0.5f, 0.5f,0.341f, 0.231f,0.047f,

-0.5f, -0.5f, -0.5f, 0.341f,0.231f, 0.047f,

0.5f, -0.5f, -0.5f, 0.341f,0.231f, 0.047f,

0.5f, -0.5f, 0.5f, 0.341f,0.231f, 0.047f, // BOTTOM

0.5f, -0.5f, 0.5f, 0.341f,0.231f, 0.047f,

-0.5f, -0.5f, 0.5f, 0.341f,0.231f, 0.047f,

-0.5f, -0.5f, -0.5f, 0.341f,0.231f, 0.047f,

-0.5f, 0.5f, -0.5f,0.192f, 0.388f, 0.0f,

0.5f, 0.5f, -0.5f, 0.192f, 0.388f, 0.0f,

0.5f, 0.5f, 0.5f, 0.192f, 0.388f, 0.0f, // TOP

0.5f, 0.5f, 0.5f, 0.192f, 0.388f, 0.0f,

-0.5f, 0.5f, 0.5f,0.192f, 0.388f, 0.0f,

-0.5f, 0.5f, -0.5f,0.192f, 0.388f, 0.0f

};

glm::mat4 projectionTransform(1.0f);

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;

bool hasFocus = false;

GLFWwindow\* instance;

} window;

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;

float fallSpeed = -0.25f;

} player;

struct {

struct {

struct {

int x;

int z;

} current;

int size = 17;

int totalCount = 0;

static const int toGenerate = 7;

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

#define BIOME\_HILLS 1

#define BIOME\_MOUNTAINS 2

int currentBiome = BIOME\_PLAINS;

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

int result;

switch (biome) {

case BIOME\_PLAINS:

noiseGenerator.SetNoiseType(FastNoise::PerlinFractal);

noiseGenerator.SetFrequency(0.01);

result = noiseGenerator.GetNoise(x, z) \* 10;

break;

case BIOME\_HILLS:

noiseGenerator.SetNoiseType(FastNoise::PerlinFractal);

noiseGenerator.SetFrequency(0.01);

result = noiseGenerator.GetNoise(x, z) \* 25;

break;

case BIOME\_MOUNTAINS:

noiseGenerator.SetNoiseType(FastNoise::SimplexFractal);

noiseGenerator.SetFrequency(0.005);

result = noiseGenerator.GetNoise(x, z) \* 75;

break;

}

result = result;

return result;

}

float\* drawChunk(float \*\*chunk) {

float \*chunkMesh = new float[2000000];

//cout << \*chunk[2] << endl;

int vertexCt = 0;

for (int x = 0; x < \*chunk[2]; x++) {

//myFile << "type: " << \*chunk[3][x] << " ... " << \*chunk[4][x] << " " << \*chunk[5][x] << " " << \*chunk[6][x] << endl;

if (!(chunk[3][x] < 0)) {

bool drawFront = false;

bool drawBack = false;

bool drawTop = false;

bool drawBottom = false;

bool drawLeft = false;

bool drawRight = false;

if (chunk[5][x] + 1 > getHeight(currentBiome, \*chunk[0] \* world.chunks.size + chunk[4][x], \*chunk[1] \* world.chunks.size + chunk[6][x] - 1)) {

drawTop = true;

// BACK

}

if (chunk[5][x] + 1 > getHeight(currentBiome, \*chunk[0] \* world.chunks.size + chunk[4][x], \*chunk[1] \* world.chunks.size + chunk[6][x] + 1)) {

// FRONT

drawBottom = true;

}

if (chunk[3][x - (world.chunks.size + 2)] < 0 && player.position.y < chunk[5][x]) { // 2 is offset to account for "air" \*chunk[3] surrounding \*chunk

drawBottom = true;

// BOTTOM

}

if (chunk[3][x + (world.chunks.size + 2)] < 0) { //&& player.position.y > chunk[5][x]) { // 2 is offset to account for "air" \*chunk[3] surrounding \*chunk

drawTop = true;

// TOP

}

if (chunk[5][x] + 1 > getHeight(currentBiome, \*chunk[0] \* world.chunks.size + chunk[4][x] - 1, \*chunk[1] \* world.chunks.size + chunk[6][x])) { // 2 is offset to account for "air" \*chunk[3] surrounding \*chunk. 1 accounts for 0

drawFront = true;

// LEFT

}

if (chunk[5][x] + 1 > getHeight(currentBiome, \*chunk[0] \* world.chunks.size + chunk[4][x] + 1, \*chunk[1] \* world.chunks.size + chunk[6][x])) { // 2 is offset to account for "air" \*chunk[3] surrounding \*chunk. 1 accounts for 0

drawRight = true;

// RIGHT

}

// BACK FRONT LEFT RIGHT BOTTOM TOP

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

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

i == 1 && !drawFront ||

i == 2 && !drawFront ||

i == 3 && !drawRight ||

i == 4 && !drawBottom ||

i == 5 && !drawTop) {

continue;

}

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

float value;

if (j % 6 == 0) {

value = positions[36 \* i + j] + chunk[4][x];

}

if (j % 6 == 1) {

value = positions[36 \* i + j] + chunk[5][x];

}

if (j % 6 == 2) {

value = positions[36 \* i + j] + chunk[6][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[vertexCt] = value;

vertexCt++;

}

}

}

}

return chunkMesh;

}

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

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

//vector<float> chunkMesh(10000000);

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

int chunkLimY = 90;

int maxBlocks = (world.chunks.size + 2) \* (chunkLimY \* 2 + 2 + 1) \* (world.chunks.size + 2);

float \*blocks = new float[maxBlocks];

float \*xPos = new float[maxBlocks];

float \*yPos = new float[maxBlocks];

float \*zPos = new float[maxBlocks];

int blocksCt = 0;

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(currentBiome, world.chunks.size\*chunkX + x, world.chunks.size\*chunkZ + z);

if (y < heightVal) {

blocks[blocksCt] = 9;

xPos[blocksCt] = x;

yPos[blocksCt] = y;

zPos[blocksCt] = z;

}

}

blocksCt++;

//cout << totalBlocks << endl;

}

}

}

float \*\*chunkData = new float\*[8];

float \*chunX = new float;

\*chunX = chunkX;

float \*chunZ = new float;

\*chunZ = chunkZ;

float \*blocCt = new float;

\*blocCt = blocksCt;

chunkData[0] = chunX;

chunkData[1] = chunZ;

chunkData[2] = blocCt;

chunkData[3] = blocks;

chunkData[4] = xPos;

chunkData[5] = yPos;

chunkData[6] = zPos;

return chunkData;

}

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;

//jumpCounter = 0;

}

}

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.flySpeed;

}

else {

// CROUCH

}

}

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

if (flying) {

player.position.y += player.flySpeed;

}

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) {

//cout << chunk << endl;

glBufferData(GL\_ARRAY\_BUFFER, 1000000, chunk[7], 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);

}

int refMiddleChunkX;

int refMiddleChunkZ;

void generateChunks() {

if(window.hasFocus){

refMiddleChunkX = world.chunks.current.x;

refMiddleChunkZ = world.chunks.current.z;

for (int a = 0; a <= world.chunks.toGenerate; a++) {

for (int b = -1; b <= 1; b++) {

if (b != 0) {

int x = a \* b;

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

bool alreadyExists = false;

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

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

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

\*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];

delete[] chunks[i][3];

delete[] chunks[i][4];

delete[] chunks[i][5];

delete[] chunks[i][6];

delete[] chunks[i][7];

\*chunks[i] = 0;

}

}

}

if (!alreadyExists) {

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

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

chunks[world.chunks.totalCount][7] = drawChunk(chunks[world.chunks.totalCount]);

world.chunks.totalCount++;

}

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

}

}

}

}

}

regenerateChunks = false;

while (!regenerateChunks) {}

generateChunks();

}

bool rerenderChunks = true;

int main(void)

{

if (!glfwInit())

return -1;

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

if (!window.instance)

{

glfwTerminate();

return -1;

}

glfwMakeContextCurrent(window.instance);

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

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

}

glEnable(GL\_DEPTH\_TEST);

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

glfwSetKeyCallback(window.instance, 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);

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);

/\*

refMiddleChunkX = world.chunks.current.x;

refMiddleChunkZ = world.chunks.current.z;

chunks[0] = addChunk(refMiddleChunkX, refMiddleChunkZ);

world.chunks.totalCount++;

chunks[0][7] = drawChunk(chunks[0]);

\*/

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

while (!glfwWindowShouldClose(window.instance))

{

/\* Render here \*/

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

projectionTransform = glm::perspective(glm::radians(window.fov), window.width / window.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));

window.hasFocus = glfwGetWindowAttrib(window.instance, GLFW\_FOCUSED);

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

if (window.hasFocus){

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

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

handleInput();

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

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

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

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

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

}

cameraRotY += (input.mouse.x - window.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.instance, window.width / 2, window.height / 2);

}

regenerateChunks = true;

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]);

}

}

}

}

if (!flying) {

if (!player.isJumping) {

if (player.position.y > getHeight(currentBiome, player.position.x, player.position.z) + player.height) {

player.position.y += player.fallSpeed;

player.fallSpeed -= 0.02f;

}

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

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

}

else {

player.position.y = getHeight(currentBiome, player.position.x, player.position.z) + player.height;

player.fallSpeed = -0.25f;

}

}

else {

jumpCounter++;

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

if (player.position.y < getHeight(currentBiome, player.position.x, player.position.z) + player.height) {

player.position.y = getHeight(currentBiome, player.position.x, player.position.z) + player.height;

player.isJumping = false;

jumpCounter = 0;

}

}

}

glfwSwapBuffers(window.instance);

glfwPollEvents();

}

t1.detach();

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

for (int z = 0; z < 8; z++) {

delete[] chunks[i][z];

}

delete[] chunks[i];

}

delete[] chunks;

glDeleteProgram(shader);

glfwTerminate();

return 0;

}