#ifndef CONTOURBUILDER\_H

#define CONTOURBUILDER\_H

#include <vector>

#include <cmath>

#include "STLReader.h"

struct Point {

float x, y, z;

bool operator==(const Point& other) const {

const float EPSILON = 1e-5f;

return std::abs(x - other.x) < EPSILON &&

std::abs(y - other.y) < EPSILON &&

std::abs(z - other.z) < EPSILON;

}

};

struct Line {

Point start, end;

};

class Polyline {

public:

std::vector<Point> points;

void addPoint(const Point& p) { points.push\_back(p); }

void addPointFront(const Point& p) { points.insert(points.begin(), p); }

bool isClosed() const { return !points.empty() && points.front() == points.back(); }

void merge(const Polyline& other);

};

class ContourBuilder {

public:

std::vector<Polyline> buildContours(const std::vector<IntersectionLine>& lines);

void simplifyContours(std::vector<Polyline>& contours, float epsilon = 0.01f);

private:

std::vector<IntersectionLine> remainingLines;

bool findAndAddNextLine(Polyline& contour);

void mergeContours(std::vector<Polyline>& contours);

void simplifyPolyline(Polyline& polyline, float epsilon);

bool isPointOnLine(const Point& start, const Point& end, const Point& point, float epsilon);

float distancePointToLine(const Point& lineStart, const Point& lineEnd, const Point& point);

};

#endif // CONTOURBUILDER\_H

#ifndef CONTOURFILLER\_H

#define CONTOURFILLER\_H

#include <vector>

#include <string>

#include "ContourBuilder.h"

class ContourFiller {

public:

ContourFiller(float initialFillAngle, float angleIncrement, float fillSpacing, int contourCount,

float controlBoard, float contourPower, float contourSpeed);

void fillLayer(const std::vector<Polyline>& contours, int layerNumber);

void writeCommands(const std::string& filename) const;

void writeLayerSeparator(const std::string& filename, int layerNumber) const;

private:

float initialFillAngle;

float angleIncrement;

float fillSpacing;

int contourCount;

float controlBoard;

float contourPower;

float contourSpeed;

std::vector<std::string> commands;

void generateFillLines(const std::vector<Polyline>& contours, float currentFillAngle);

// 其余方法保持不变

void addCommand(const std::string& command);

std::string formatPoint(const Point& point);

std::vector<Point> findIntersections(const Point& start, const Point& end, const std::vector<Polyline>& contours);

void drawContours(const std::vector<Polyline>& contours);

bool lineIntersection(const Point& p1, const Point& p2, const Point& p3, const Point& p4, Point& intersection);

Point rotatePoint(const Point& p, float angle) const;

Point unrotatePoint(const Point& p, float angle) const;

std::tuple<float, float, float, float> getBoundingBox(const std::vector<Polyline>& contours) const;

bool isPointInsideContours(const Point& p, const std::vector<Polyline>& contours) const;

};

#endif // CONTOURFILLER\_H

#pragma once

#ifndef MODEL\_PROCESSOR\_H

#define MODEL\_PROCESSOR\_H

#include <string>

#include <vector>

#include "MultiSTLManager.h"

class ModelProcessor {

public:

static void processModel(const MultiSTLManager& manager, size\_t modelIndex, float layerHeight, float directionVector[3]);

static void generateCombinedCommands(const std::vector<std::string>& modelCommandFiles,

const std::string& outputFile,

const std::vector<MultiSTLManager::ModelInfo>& modelInfos);

private:

static void writeLayerCommands(std::ofstream& outFile, std::ofstream& machineTestFile, std::ofstream& layerFile,

const std::vector<std::string>& commands, const MultiSTLManager::ModelInfo& modelInfo, size\_t modelIndex);

};

#endif // MODEL\_PROCESSOR\_H

#pragma once

#include <string>

#include <vector>

#include "STLReader.h"

class MultiSTLManager {

public:

struct ModelInfo {

STLReader model;

std::string material;

float position[3];

float write\_da\_x\_a;

float write\_da\_x\_b;

float set\_mark\_speed;

float fillAngle;

float fillSpacing;

float angleIncrement;

int contourCount;

float contourPower;

float contourSpeed;

};

void addModel(const std::string& filename, const std::string& material,

float x, float y, float z,

float write\_da\_x\_a, float write\_da\_x\_b, float set\_mark\_speed,

float fillSpacing, float fillAngle, float angleIncrement, // 新增 angleIncrement 参数

int contourCount, float contourPower, float contourSpeed) {

ModelInfo info;

info.model.readSTL(filename);

info.material = material;

info.position[0] = x;

info.position[1] = y;

info.position[2] = z;

info.write\_da\_x\_a = write\_da\_x\_a; // 控制板

info.write\_da\_x\_b = write\_da\_x\_b; // 激光功率

info.set\_mark\_speed = set\_mark\_speed; // 打印速度

info.fillSpacing = fillSpacing; // 打印间距

info.fillAngle = fillAngle; // 初始填充角度

info.angleIncrement = angleIncrement; // 每层角度增量

info.contourCount = contourCount;

info.contourPower = contourPower;

info.contourSpeed = contourSpeed;

models.push\_back(info);

}

void adjustModelPosition(size\_t modelIndex, float dx, float dy, float dz);

std::vector<std::vector<IntersectionLine>> sliceAllModels(float layerHeight, float directionVector[3]) const;

const ModelInfo& getModelInfo(size\_t modelIndex) const {

return models[modelIndex];

}

size\_t getModelCount() const {

return models.size();

}

std::vector<std::vector<IntersectionLine>> sliceSingleModel(size\_t modelIndex, float layerHeight, float directionVector[3]) const {

if (modelIndex >= models.size()) {

return {};

}

const auto& modelInfo = models[modelIndex];

auto modelSlices = modelInfo.model.sliceAlongDirection(layerHeight, directionVector);

// 调整每个切片中的点的位置

for (auto& slice : modelSlices) {

for (auto& line : slice) {

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

line.point1[i] += modelInfo.position[i];

line.point2[i] += modelInfo.position[i];

}

}

}

return modelSlices;

}

private:

std::vector<ModelInfo> models;

};

#ifndef STLREADER\_H

#define STLREADER\_H

#include <string>

#include <vector>

struct Triangle {

float normal[3];

float vertices[3][3];

};

struct IntersectionLine {

float point1[3];

float point2[3];

};

class STLReader {

public:

void readSTL(const std::string& filename);

void writeTriangles(const std::string& outputFilename) const;

std::vector<std::vector<IntersectionLine>> sliceAlongDirection(float layerHeight, float directionVector[3]) const;

float getMaxProjection(float directionVector[3]) const;

private:

bool isBinarySTL(const std::string& filename) const;

void readBinarySTL(const std::string& filename);

void readASCIISTL(const std::string& filename);

std::vector<Triangle> triangles;

std::vector<IntersectionLine> getIntersectionLines(const Triangle& triangle, float planeDistance, float directionVector[3]) const;

bool isIntersecting(float planeDistance, const float vertex[3], float directionVector[3]) const;

void getIntersectionPoint(float planeDistance, const float v1[3], const float v2[3], float result[3], float directionVector[3]) const;

float dotProduct(const float v1[3], const float v2[3]) const;

void createOrthogonalVectors(const float v[3], float out1[3], float out2[3]) const;

void projectPointsTo2D(float point[3], const float orthogonal1[3], const float orthogonal2[3]) const;

bool arePointsEqual(const float p1[3], const float p2[3]) const;

};

#endif // STLREADER\_H

#include "ContourBuilder.h"

#include <algorithm>

void Polyline::merge(const Polyline& other) {

if (points.back() == other.points.front()) {

points.insert(points.end(), other.points.begin() + 1, other.points.end());

}

else if (points.back() == other.points.back()) {

points.insert(points.end(), other.points.rbegin() + 1, other.points.rend());

}

else if (points.front() == other.points.back()) {

points.insert(points.begin(), other.points.begin(), other.points.end() - 1);

}

else if (points.front() == other.points.front()) {

points.insert(points.begin(), other.points.rbegin(), other.points.rend() - 1);

}

}

std::vector<Polyline> ContourBuilder::buildContours(const std::vector<IntersectionLine>& lines) {

remainingLines = lines;

std::vector<Polyline> contours;

while (!remainingLines.empty()) {

Polyline contour;

IntersectionLine firstLine = remainingLines.front();

remainingLines.erase(remainingLines.begin());

contour.addPoint({ firstLine.point1[0], firstLine.point1[1], firstLine.point1[2] });

contour.addPoint({ firstLine.point2[0], firstLine.point2[1], firstLine.point2[2] });

while (findAndAddNextLine(contour)) {

if (contour.isClosed()) {

break;

}

}

contours.push\_back(contour);

}

mergeContours(contours);

return contours;

}

bool ContourBuilder::findAndAddNextLine(Polyline& contour) {

for (auto it = remainingLines.begin(); it != remainingLines.end(); ++it) {

Point start = { it->point1[0], it->point1[1], it->point1[2] };

Point end = { it->point2[0], it->point2[1], it->point2[2] };

if (start == contour.points.back()) {

contour.addPoint(end);

remainingLines.erase(it);

return true;

}

else if (end == contour.points.back()) {

contour.addPoint(start);

remainingLines.erase(it);

return true;

}

else if (start == contour.points.front()) {

contour.addPointFront(end);

remainingLines.erase(it);

return true;

}

else if (end == contour.points.front()) {

contour.addPointFront(start);

remainingLines.erase(it);

return true;

}

}

return false;

}

void ContourBuilder::mergeContours(std::vector<Polyline>& contours) {

bool merged;

do {

merged = false;

for (size\_t i = 0; i < contours.size(); ++i) {

for (size\_t j = i + 1; j < contours.size(); ) {

if (contours[i].points.front() == contours[j].points.front() ||

contours[i].points.front() == contours[j].points.back() ||

contours[i].points.back() == contours[j].points.front() ||

contours[i].points.back() == contours[j].points.back()) {

contours[i].merge(contours[j]);

contours.erase(contours.begin() + j);

merged = true;

}

else {

++j;

}

}

}

} while (merged);

}

void ContourBuilder::simplifyContours(std::vector<Polyline>& contours, float epsilon) {

for (auto& contour : contours) {

simplifyPolyline(contour, epsilon);

}

}

void ContourBuilder::simplifyPolyline(Polyline& polyline, float epsilon) {

if (polyline.points.size() <= 2) {

return;

}

std::vector<Point> simplified;

simplified.push\_back(polyline.points.front());

for (size\_t i = 1; i < polyline.points.size() - 1; ++i) {

if (!isPointOnLine(polyline.points[i - 1], polyline.points[i + 1], polyline.points[i], epsilon)) {

simplified.push\_back(polyline.points[i]);

}

}

simplified.push\_back(polyline.points.back());

polyline.points = simplified;

}

bool ContourBuilder::isPointOnLine(const Point& start, const Point& end, const Point& point, float epsilon) {

float d = distancePointToLine(start, end, point);

return d <= epsilon;

}

float ContourBuilder::distancePointToLine(const Point& lineStart, const Point& lineEnd, const Point& point) {

float lineLength = std::sqrt(std::pow(lineEnd.x - lineStart.x, 2) + std::pow(lineEnd.y - lineStart.y, 2));

if (lineLength == 0) {

return std::sqrt(std::pow(point.x - lineStart.x, 2) + std::pow(point.y - lineStart.y, 2));

}

float t = ((point.x - lineStart.x) \* (lineEnd.x - lineStart.x) +

(point.y - lineStart.y) \* (lineEnd.y - lineStart.y)) / (lineLength \* lineLength);

t = std::max(0.0f, std::min(1.0f, t));

float projectionX = lineStart.x + t \* (lineEnd.x - lineStart.x);

float projectionY = lineStart.y + t \* (lineEnd.y - lineStart.y);

return std::sqrt(std::pow(point.x - projectionX, 2) + std::pow(point.y - projectionY, 2));

}

#include "ContourFiller.h"

#include <cmath>

#include <algorithm>

#include <fstream>

#include <sstream>

#include <iomanip>

#include <limits>

#ifndef M\_PI

#define M\_PI 3.14159265358979323846

#endif

ContourFiller::ContourFiller(float initialFillAngle, float angleIncrement, float fillSpacing, int contourCount,

float controlBoard, float contourPower, float contourSpeed)

: initialFillAngle(initialFillAngle\* static\_cast<float>(M\_PI) / 180.0f),

angleIncrement(angleIncrement\* static\_cast<float>(M\_PI) / 180.0f),

fillSpacing(fillSpacing),

contourCount(contourCount),

controlBoard(controlBoard),

contourPower(contourPower),

contourSpeed(contourSpeed) {

}

void ContourFiller::fillLayer(const std::vector<Polyline>& contours, int layerNumber) {

commands.clear();

float currentFillAngle = initialFillAngle + layerNumber \* angleIncrement;

generateFillLines(contours, currentFillAngle);

drawContours(contours);

}

void ContourFiller::drawContours(const std::vector<Polyline>& contours) {

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

addCommand("write\_da\_x\_list(" + std::to\_string(controlBoard) + "," + std::to\_string(contourPower) + ")");

addCommand("set\_mark\_speed(" + std::to\_string(contourSpeed \* 0.1862156) + ")");

for (const auto& contour : contours) {

if (!contour.points.empty()) {

addCommand("jump\_abs(" + formatPoint(contour.points[0]) + ")");

for (size\_t j = 1; j < contour.points.size(); ++j) {

addCommand("mark\_abs(" + formatPoint(contour.points[j]) + ")");

}

if (contour.isClosed()) {

addCommand("mark\_abs(" + formatPoint(contour.points[0]) + ")");

}

}

}

}

}

void ContourFiller::generateFillLines(const std::vector<Polyline>& contours, float currentFillAngle) {

if (contours.empty()) return;

auto [minX, minY, maxX, maxY] = getBoundingBox(contours);

float width = maxX - minX;

float height = maxY - minY;

// 计算旋转后的边界框

float cos\_a = std::cos(currentFillAngle);

float sin\_a = std::sin(currentFillAngle);

float rotatedWidth = std::abs(width \* cos\_a) + std::abs(height \* sin\_a);

float rotatedHeight = std::abs(width \* sin\_a) + std::abs(height \* cos\_a);

// 生成填充线

std::vector<std::pair<Point, Point>> fillLines;

float startX = -rotatedWidth / 2;

float endX = rotatedWidth / 2;

bool isLeftToRight = true;

for (float y = -rotatedHeight / 2; y <= rotatedHeight / 2; y += fillSpacing) {

Point start = unrotatePoint({ startX, y, 0 }, currentFillAngle);

Point end = unrotatePoint({ endX, y, 0 }, currentFillAngle);

if (!isLeftToRight) {

std::swap(start, end);

}

start.x += (minX + maxX) / 2;

start.y += (minY + maxY) / 2;

end.x += (minX + maxX) / 2;

end.y += (minY + maxY) / 2;

fillLines.push\_back({ start, end });

isLeftToRight = !isLeftToRight;

}

// 处理每条填充线

for (const auto& [start, end] : fillLines) {

std::vector<Point> intersections = findIntersections(start, end, contours);

if (intersections.size() >= 2) {

for (size\_t i = 0; i < intersections.size() - 1; i += 2) {

addCommand("jump\_abs(" + formatPoint(intersections[i]) + ")");

addCommand("mark\_abs(" + formatPoint(intersections[i + 1]) + ")");

}

}

}

}

std::vector<Point> ContourFiller::findIntersections(const Point& start, const Point& end, const std::vector<Polyline>& contours) {

std::vector<Point> intersections;

for (const auto& contour : contours) {

for (size\_t i = 0; i < contour.points.size(); ++i) {

const Point& p1 = contour.points[i];

const Point& p2 = contour.points[(i + 1) % contour.points.size()];

Point intersection;

if (lineIntersection(start, end, p1, p2, intersection)) {

intersections.push\_back(intersection);

}

}

}

// 排序交点

std::sort(intersections.begin(), intersections.end(),

[&start](const Point& a, const Point& b) {

return (a.x - start.x) \* (a.x - start.x) + (a.y - start.y) \* (a.y - start.y) <

(b.x - start.x) \* (b.x - start.x) + (b.y - start.y) \* (b.y - start.y);

});

return intersections;

}

bool ContourFiller::lineIntersection(const Point& p1, const Point& p2, const Point& p3, const Point& p4, Point& intersection) {

float x1 = p1.x, y1 = p1.y;

float x2 = p2.x, y2 = p2.y;

float x3 = p3.x, y3 = p3.y;

float x4 = p4.x, y4 = p4.y;

float denom = (y4 - y3) \* (x2 - x1) - (x4 - x3) \* (y2 - y1);

if (std::abs(denom) < 1e-6) return false; // 平行或重合

float ua = ((x4 - x3) \* (y1 - y3) - (y4 - y3) \* (x1 - x3)) / denom;

float ub = ((x2 - x1) \* (y1 - y3) - (y2 - y1) \* (x1 - x3)) / denom;

if (ua < 0 || ua > 1 || ub < 0 || ub > 1) return false; // 交点不在线段上

intersection.x = x1 + ua \* (x2 - x1);

intersection.y = y1 + ua \* (y2 - y1);

intersection.z = 0;

return true;

}

void ContourFiller::addCommand(const std::string& command) {

commands.push\_back(command);

}

std::string ContourFiller::formatPoint(const Point& point) {

std::ostringstream oss;

oss << std::fixed << std::setprecision(6) << point.x << "," << point.y;

return oss.str();

}

void ContourFiller::writeCommands(const std::string& filename) const {

std::ofstream outFile(filename, std::ios::app);

if (!outFile) {

throw std::runtime\_error("Unable to open file: " + filename);

}

for (const auto& command : commands) {

outFile << command << ";" << std::endl;

}

}

void ContourFiller::writeLayerSeparator(const std::string& filename, int layerNumber) const {

std::ofstream outFile(filename, std::ios::app);

if (!outFile) {

throw std::runtime\_error("Unable to open file: " + filename);

}

outFile << "\n//;Layer " << layerNumber << " Fill Commands\n";

}

Point ContourFiller::rotatePoint(const Point& p, float angle) const {

float cos\_a = std::cos(angle);

float sin\_a = std::sin(angle);

return {

p.x \* cos\_a - p.y \* sin\_a,

p.x \* sin\_a + p.y \* cos\_a,

p.z

};

}

Point ContourFiller::unrotatePoint(const Point& p, float angle) const {

float cos\_a = std::cos(-angle);

float sin\_a = std::sin(-angle);

return {

p.x \* cos\_a - p.y \* sin\_a,

p.x \* sin\_a + p.y \* cos\_a,

p.z

};

}

std::tuple<float, float, float, float> ContourFiller::getBoundingBox(const std::vector<Polyline>& contours) const {

float minX = std::numeric\_limits<float>::max();

float minY = std::numeric\_limits<float>::max();

float maxX = std::numeric\_limits<float>::lowest();

float maxY = std::numeric\_limits<float>::lowest();

for (const auto& contour : contours) {

for (const auto& point : contour.points) {

minX = std::min(minX, point.x);

minY = std::min(minY, point.y);

maxX = std::max(maxX, point.x);

maxY = std::max(maxY, point.y);

}

}

return std::make\_tuple(minX, minY, maxX, maxY);

}

bool ContourFiller::isPointInsideContours(const Point& p, const std::vector<Polyline>& contours) const {

int intersectionCount = 0;

for (const auto& contour : contours) {

for (size\_t i = 0; i < contour.points.size(); ++i) {

const Point& p1 = contour.points[i];

const Point& p2 = contour.points[(i + 1) % contour.points.size()];

if (((p1.y > p.y) != (p2.y > p.y)) &&

(p.x < (p2.x - p1.x) \* (p.y - p1.y) / (p2.y - p1.y) + p1.x)) {

intersectionCount++;

}

}

}

return intersectionCount % 2 == 1;

}

#include "ModelProcessor.h"

#include <iostream>

#include <sstream>

#define LAYER\_HEIGHT 1.0f

#define y\_direction { 0.0f, 0.0f,1.0f }

int main() {

MultiSTLManager manager;

manager.addModel("C:\\Users\\zzy\\Desktop\\8.stl", "1", 0, 0, 0, 1.0f, 700.0f, 400.0f, 0.1f, 0.0f, 67.0f, 0 ,240.0f, 600.0f);

std::vector<std::string> modelCommandFiles;

std::vector<MultiSTLManager::ModelInfo> modelInfos;

// 调整模型位置

manager.adjustModelPosition(0, 0, 0, 0);

float layerHeight = LAYER\_HEIGHT;

float directionVector[3] = y\_direction;

// 处理每个模型

for (size\_t i = 0; i < manager.getModelCount(); ++i) {

std::ostringstream commandFileName;

commandFileName << "C:\\output\\model\_" << i << "\_commands.txt";

modelCommandFiles.push\_back(commandFileName.str());

modelInfos.push\_back(manager.getModelInfo(i));

ModelProcessor::processModel(manager, i, layerHeight, directionVector);

}

// 生成组合的命令文件，传递模型信息

ModelProcessor::generateCombinedCommands(modelCommandFiles, "C:\\output\\combined\_commands.txt", modelInfos);

std::cout << "所有模型处理完成。请检查输出文件。" << std::endl;

return 0;

}

#include "ModelProcessor.h"

#include "ContourBuilder.h"

#include "ContourFiller.h"

#include <iostream>

#include <fstream>

#include <sstream>

#include <iomanip>

#include <filesystem>

namespace fs = std::filesystem;

void ModelProcessor::processModel(const MultiSTLManager& manager, size\_t modelIndex, float layerHeight, float directionVector[3]) {

const auto& modelInfo = manager.getModelInfo(modelIndex);

std::ostringstream outFileName;

outFileName << "C:\\output\\model\_" << modelIndex << "\_output.txt";

std::ofstream outFile(outFileName.str());

if (!outFile) {

std::cerr << "无法打开输出文件: " << outFileName.str() << std::endl;

return;

}

outFile << "模型 " << modelIndex << " (材料: " << modelInfo.material << ")" << std::endl;

outFile << "位置: (" << modelInfo.position[0] << ", " << modelInfo.position[1] << ", " << modelInfo.position[2] << ")" << std::endl << std::endl;

auto slices = manager.sliceSingleModel(modelIndex, layerHeight, directionVector);

ContourBuilder contourBuilder;

ContourFiller contourFiller(modelInfo.fillAngle, 67.0f, modelInfo.fillSpacing, modelInfo.contourCount,

1, modelInfo.contourPower, modelInfo.contourSpeed); // 添加 67 度增量

std::ostringstream commandFileName;

commandFileName << "C:\\output\\model\_" << modelIndex << "\_commands.txt";

std::remove(commandFileName.str().c\_str());

for (size\_t i = 0; i < slices.size(); ++i) {

outFile << "层 " << i << " 在距离 = " << i \* layerHeight << std::endl;

// 输出交线（2D 坐标）

outFile << "交线 (2D 坐标):" << std::endl;

for (const auto& line : slices[i]) {

outFile << std::fixed << std::setprecision(6)

<< "从 (" << line.point1[0] << ", " << line.point1[1] << ") 到 ("

<< line.point2[0] << ", " << line.point2[1] << ")" << std::endl;

}

std::vector<Polyline> contours = contourBuilder.buildContours(slices[i]);

contourBuilder.simplifyContours(contours, 1e-7f);

outFile << "轮廓 (2D 坐标):" << std::endl;

for (size\_t j = 0; j < contours.size(); ++j) {

outFile << "轮廓 " << j << ":" << std::endl;

for (const auto& point : contours[j].points) {

outFile << std::fixed << std::setprecision(6)

<< "(" << point.x << ", " << point.y << ")" << std::endl;

}

outFile << (contours[j].isClosed() ? "闭合" : "开放") << std::endl;

}

outFile << std::endl;

contourFiller.writeLayerSeparator(commandFileName.str(), i);

contourFiller.fillLayer(contours, i); // 传递当前层编号 i

contourFiller.writeCommands(commandFileName.str());

}

std::cout << "模型 " << modelIndex << " 处理完成。" << std::endl;

}

void ModelProcessor::generateCombinedCommands(const std::vector<std::string>& modelCommandFiles,

const std::string& outputFile,

const std::vector<MultiSTLManager::ModelInfo>& modelInfos) {

std::vector<std::ifstream> inputFiles;

for (const auto& file : modelCommandFiles) {

inputFiles.emplace\_back(file);

if (!inputFiles.back().is\_open()) {

std::cerr << "Error opening file: " << file << std::endl;

return;

}

}

std::ofstream outFile(outputFile);

std::ofstream machineTestFile(outputFile.substr(0, outputFile.find\_last\_of('.')) + "\_machine\_test.txt");

if (!outFile.is\_open() || !machineTestFile.is\_open()) {

std::cerr << "Error opening output files" << std::endl;

return;

}

// 创建layer\_test文件夹

fs::path layerTestDir = "C:\\output\\layer\_test";

if (!fs::exists(layerTestDir)) {

fs::create\_directories(layerTestDir);

}

std::string line;

std::vector<std::vector<std::string>> layerCommands(modelCommandFiles.size());

int layerNumber = 0;

while (true) {

bool allFilesProcessed = true;

for (size\_t i = 0; i < inputFiles.size(); ++i) {

layerCommands[i].clear();

while (std::getline(inputFiles[i], line)) {

if (line.find(";Layer") != std::string::npos) {

if (!layerCommands[i].empty()) {

allFilesProcessed = false;

break;

}

layerCommands[i].push\_back(line);

}

else if (!line.empty()) {

layerCommands[i].push\_back(line);

}

}

if (!inputFiles[i].eof()) {

allFilesProcessed = false;

}

}

if (allFilesProcessed) break;

// 创建当前层的文件

std::ostringstream layerFileName;

layerFileName << layerTestDir.string() << "\\layer\_" << layerNumber << ".txt";

std::ofstream layerFile(layerFileName.str());

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

std::cerr << "Error opening layer file: " << layerFileName.str() << std::endl;

return;

}

// Write layer start marker

std::string layerStartMarker = "\n//;Begin Layer " + std::to\_string(layerNumber) + "\n";

outFile << layerStartMarker << std::endl;

machineTestFile << layerStartMarker << std::endl;

layerFile << layerStartMarker << std::endl;

for (size\_t i = 0; i < layerCommands.size(); ++i) {

if (!layerCommands[i].empty()) {

writeLayerCommands(outFile, machineTestFile, layerFile, layerCommands[i], modelInfos[i], i);

}

}

// Write layer end marker

std::string layerEndMarker = "//;End Layer " + std::to\_string(layerNumber) + "\n";

outFile << layerEndMarker << std::endl;

machineTestFile << layerEndMarker << std::endl;

layerFile << layerEndMarker << std::endl;

layerFile.close();

layerNumber++;

}

std::cout << "Combined commands have been written to " << outputFile << std::endl;

std::cout << "Machine test commands have been written to " << outputFile.substr(0, outputFile.find\_last\_of('.')) + "\_machine\_test.txt" << std::endl;

std::cout << "Layer files have been written to " << layerTestDir.string() << std::endl;

}

void ModelProcessor::writeLayerCommands(std::ofstream& outFile, std::ofstream& machineTestFile, std::ofstream& layerFile,

const std::vector<std::string>& commands, const MultiSTLManager::ModelInfo& modelInfo, size\_t modelIndex) {

std::string modelHeader = "//;Model " + std::to\_string(modelIndex) + " Commands";

outFile << modelHeader << std::endl;

machineTestFile << modelHeader << std::endl;

layerFile << modelHeader << std::endl;

std::string daCommand = "write\_da\_x\_list(" + std::to\_string(modelInfo.write\_da\_x\_a) + "," +

std::to\_string(modelInfo.write\_da\_x\_b) + ");";

machineTestFile << daCommand << std::endl;

layerFile << daCommand << std::endl;

std::string markSpeedCommand = "set\_mark\_speed(" + std::to\_string(modelInfo.set\_mark\_speed \* 0.1862156) + ");";

machineTestFile << markSpeedCommand << std::endl;

layerFile << markSpeedCommand << std::endl;

for (const auto& cmd : commands) {

outFile << cmd << std::endl;

// Process command for machine\_test.txt and layer file

if (cmd.find("jump\_abs") != std::string::npos || cmd.find("mark\_abs") != std::string::npos) {

size\_t start = cmd.find('(') + 1;

size\_t end = cmd.find(')');

std::string coords = cmd.substr(start, end - start);

std::istringstream iss(coords);

float x, y;

char comma;

if (iss >> x >> comma >> y) {

x \*= 186.2156;

y \*= 186.2156;

std::string processedCmd = cmd.substr(0, start) + std::to\_string(x) + "," + std::to\_string(y) + cmd.substr(end);

machineTestFile << processedCmd << std::endl;

layerFile << processedCmd << std::endl;

}

else {

machineTestFile << cmd << std::endl;

layerFile << cmd << std::endl;

}

}

else {

machineTestFile << cmd << std::endl;

layerFile << cmd << std::endl;

}

}

outFile << std::endl; // Add a blank line between models

machineTestFile << std::endl; // Add a blank line between models

layerFile << std::endl; // Add a blank line between models

}

void MultiSTLManager::adjustModelPosition(size\_t modelIndex, float dx, float dy, float dz) {

if (modelIndex < models.size()) {

models[modelIndex].position[0] += dx;

models[modelIndex].position[1] += dy;

models[modelIndex].position[2] += dz;

}

}

std::vector<std::vector<IntersectionLine>> MultiSTLManager::sliceAllModels(float layerHeight, float directionVector[3]) const {

std::vector<std::vector<IntersectionLine>> allSlices;

for (const auto& modelInfo : models) {

auto modelSlices = modelInfo.model.sliceAlongDirection(layerHeight, directionVector);

// 调整每个切片中的点的位置

for (auto& slice : modelSlices) {

for (auto& line : slice) {

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

line.point1[i] += modelInfo.position[i];

line.point2[i] += modelInfo.position[i];

}

}

}

// 合并切片

if (allSlices.empty()) {

allSlices = modelSlices;

}

else {

for (size\_t i = 0; i < allSlices.size(); ++i) {

allSlices[i].insert(allSlices[i].end(), modelSlices[i].begin(), modelSlices[i].end());

}

}

}

return allSlices;

}

#include "STLReader.h"

#include <fstream>

#include <iostream>

#include <sstream>

#include <cstring>

#include <cmath>

#include <limits>

#include <algorithm>

bool STLReader::isBinarySTL(const std::string& filename) const {

std::ifstream file(filename, std::ios::binary);

if (!file) {

std::cerr << "Cannot open file: " << filename << std::endl;

return false;

}

char header[80];

file.read(header, 80);

file.seekg(0, std::ios::end);

std::streampos fileSize = file.tellg();

file.seekg(80, std::ios::beg);//前80个字节是文件头

uint32\_t numTriangles;

file.read(reinterpret\_cast<char\*>(&numTriangles), sizeof(uint32\_t));

return (fileSize == 80 + 4 + numTriangles \* 50);//法向量：12字节，顶点：3\*12字节，属性：2字节

}

void STLReader::readBinarySTL(const std::string& filename) {

std::ifstream file(filename, std::ios::binary);

if (!file) {

std::cerr << "Cannot open file: " << filename << std::endl;

return;

}

char header[80];

file.read(header, 80);

uint32\_t numTriangles;

file.read(reinterpret\_cast<char\*>(&numTriangles), sizeof(uint32\_t));

triangles.clear();

triangles.reserve(numTriangles);//预分配内存空间

for (uint32\_t i = 0; i < numTriangles; ++i) {

Triangle triangle;

file.read(reinterpret\_cast<char\*>(triangle.normal), 3 \* sizeof(float));

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

file.read(reinterpret\_cast<char\*>(triangle.vertices[j]), 3 \* sizeof(float));

}

uint16\_t attributeByteCount;//读取属性字节

file.read(reinterpret\_cast<char\*>(&attributeByteCount), sizeof(uint16\_t));

triangles.push\_back(triangle);

}

std::cout << "Read " << triangles.size() << " triangles from binary STL file." << std::endl;

}

void STLReader::readSTL(const std::string& filename) {

if (isBinarySTL(filename)) {

std::cout << "Detected binary STL file." << std::endl;

readBinarySTL(filename);

}

else {

std::cout << "Detected ASCII STL file." << std::endl;

readASCIISTL(filename);

}

}

void STLReader::readASCIISTL(const std::string& filename) {

std::ifstream file(filename);

if (!file) {

std::cerr << "Cannot open file: " << filename << std::endl;

return;

}

std::string line;

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

if (line.find("facet normal") != std::string::npos) {

Triangle triangle;

std::istringstream ss(line);

std::string dummy;

ss >> dummy >> dummy >> triangle.normal[0] >> triangle.normal[1] >> triangle.normal[2];

std::getline(file, line); // 跳过 "outer loop"

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

std::getline(file, line);

std::istringstream vertexSS(line);

vertexSS >> dummy >> triangle.vertices[i][0] >> triangle.vertices[i][1] >> triangle.vertices[i][2];

}

std::getline(file, line); // 跳过 "endloop"

std::getline(file, line); // 跳过 "endfacet"

triangles.push\_back(triangle);

}

}

}

void STLReader::writeTriangles(const std::string& outputFilename) const {

std::ofstream outFile(outputFilename);

if (!outFile) {

std::cerr << "Cannot open output file: " << outputFilename << std::endl;

return;

}

outFile << "Number of triangles: " << triangles.size() << std::endl;

for (const auto& triangle : triangles) {

outFile << "facet normal " << triangle.normal[0] << " " << triangle.normal[1] << " " << triangle.normal[2] << std::endl;

outFile << " outer loop" << std::endl;

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

outFile << " vertex " << triangle.vertices[i][0] << " " << triangle.vertices[i][1] << " " << triangle.vertices[i][2] << std::endl;

}

outFile << " endloop" << std::endl;

outFile << "endfacet" << std::endl;

}

}

std::vector<std::vector<IntersectionLine>> STLReader::sliceAlongDirection(float layerHeight, float directionVector[3]) const {

std::vector<std::vector<IntersectionLine>> slices;

float maxProjection = getMaxProjection(directionVector);

// Normalize the direction vector

float magnitude = std::sqrt(directionVector[0] \* directionVector[0] +

directionVector[1] \* directionVector[1] +

directionVector[2] \* directionVector[2]);

float normalized[3] = { directionVector[0] / magnitude,

directionVector[1] / magnitude,

directionVector[2] / magnitude };

float orthogonal1[3], orthogonal2[3];

createOrthogonalVectors(normalized, orthogonal1, orthogonal2);

for (float distance = 0.0f; distance <= maxProjection; distance += layerHeight) {

std::vector<IntersectionLine> slice;

for (const auto& triangle : triangles) {

auto intersectionLines = getIntersectionLines(triangle, distance, normalized);

for (auto& line : intersectionLines) {

// 把线段投影到二维平面上

projectPointsTo2D(line.point1, orthogonal1, orthogonal2);

projectPointsTo2D(line.point2, orthogonal1, orthogonal2);

// 只有当线段的两端点不同时，才将其添加到切片中

if (!arePointsEqual(line.point1, line.point2)) {

slice.push\_back(line);

}

}

}

slices.push\_back(slice);

}

return slices;

}

float STLReader::getMaxProjection(float directionVector[3]) const {

float maxProjection = std::numeric\_limits<float>::lowest();

for (const auto& triangle : triangles) {

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

float projection = dotProduct(triangle.vertices[i], directionVector);

if (projection > maxProjection) {

maxProjection = projection;

}

}

}

return maxProjection;

}

std::vector<IntersectionLine> STLReader::getIntersectionLines(const Triangle& triangle, float planeDistance, float directionVector[3]) const {

std::vector<float\*> intersectingPoints;

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

const float\* v1 = triangle.vertices[i];

const float\* v2 = triangle.vertices[(i + 1) % 3];

if ((dotProduct(v1, directionVector) - planeDistance) \* (dotProduct(v2, directionVector) - planeDistance) < 0) {

float point[3];

getIntersectionPoint(planeDistance, v1, v2, point, directionVector);

intersectingPoints.push\_back(new float[3] { point[0], point[1], point[2] });

}

else if (std::abs(dotProduct(v1, directionVector) - planeDistance) < 1e-5) {

intersectingPoints.push\_back(new float[3] { v1[0], v1[1], v1[2] });

}

}

std::vector<IntersectionLine> lines;

if (intersectingPoints.size() == 2) {

lines.push\_back({ { intersectingPoints[0][0], intersectingPoints[0][1], intersectingPoints[0][2] },

{ intersectingPoints[1][0], intersectingPoints[1][1], intersectingPoints[1][2] } });

}

for (auto& point : intersectingPoints) {

delete[] point;

}

return lines;

}

bool STLReader::isIntersecting(float planeDistance, const float vertex[3], float directionVector[3]) const {

return std::abs(dotProduct(vertex, directionVector) - planeDistance) < 1e-5;

}

void STLReader::getIntersectionPoint(float planeDistance, const float v1[3], const float v2[3], float result[3], float directionVector[3]) const {

float t = (planeDistance - dotProduct(v1, directionVector)) / (dotProduct(v2, directionVector) - dotProduct(v1, directionVector));

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

result[i] = v1[i] + t \* (v2[i] - v1[i]);

}

}

float STLReader::dotProduct(const float v1[3], const float v2[3]) const {

return v1[0] \* v2[0] + v1[1] \* v2[1] + v1[2] \* v2[2];

}

void STLReader::createOrthogonalVectors(const float v[3], float out1[3], float out2[3]) const {

// Find the index of the largest absolute component

int maxIndex = 0;

for (int i = 1; i < 3; ++i) {

if (std::abs(v[i]) > std::abs(v[maxIndex])) {

maxIndex = i;

}

}

// 创建一个正交向量out1

out1[0] = out1[1] = out1[2] = 0.0f;

out1[(maxIndex + 1) % 3] = v[(maxIndex + 2) % 3];

out1[(maxIndex + 2) % 3] = -v[(maxIndex + 1) % 3];

// 标准化out1

float magnitude = std::sqrt(out1[0] \* out1[0] + out1[1] \* out1[1] + out1[2] \* out1[2]);

if (magnitude > 1e-6) { // 避免除以零

out1[0] /= magnitude;

out1[1] /= magnitude;

out1[2] /= magnitude;

}

else {

//如果out1的模长接近于零，则选择另一个方向

out1[(maxIndex + 1) % 3] = 1.0f;

}

// 创建第二个正交向量

out2[0] = v[1] \* out1[2] - v[2] \* out1[1];

out2[1] = v[2] \* out1[0] - v[0] \* out1[2];

out2[2] = v[0] \* out1[1] - v[1] \* out1[0];

// 标准化out2

magnitude = std::sqrt(out2[0] \* out2[0] + out2[1] \* out2[1] + out2[2] \* out2[2]);

if (magnitude > 1e-6) { // 避免除以零

out2[0] /= magnitude;

out2[1] /= magnitude;

out2[2] /= magnitude;

}

}

void STLReader::projectPointsTo2D(float point[3], const float orthogonal1[3], const float orthogonal2[3]) const {

float x = dotProduct(point, orthogonal1);

float y = dotProduct(point, orthogonal2);

point[0] = x;

point[1] = y;

point[2] = 0.0f; //设置z为0

}

bool STLReader::arePointsEqual(const float p1[3], const float p2[3]) const {

const float EPSILON = 1e-5f;

return std::abs(p1[0] - p2[0]) < EPSILON &&

std::abs(p1[1] - p2[1]) < EPSILON &&

std::abs(p1[2] - p2[2]) < EPSILON;

}