#include <CL/sycl.hpp>

#include <cmath>

#include <iostream>

#include <unordered\_map>

#include <vector>

namespace sycl = cl::sycl;

struct Point {

float x, y, z;

};

struct Vector {

float x, y, z;

Vector() : x(0), y(0), z(0) {}

Vector(float x\_, float y\_, float z\_) : x(x\_), y(y\_), z(z\_) {}

Vector operator+(const Vector& rhs) const {

return Vector(x + rhs.x, y + rhs.y, z + rhs.z);

}

Vector operator-(const Vector& rhs) const {

return Vector(x - rhs.x, y - rhs.y, z - rhs.z);

}

Vector operator\*(float s) const { return Vector(x \* s, y \* s, z \* s); }

float dot(const Vector& rhs) const { return x \* rhs.x + y \* rhs.y + z \* rhs.z; }

Vector cross(const Vector& rhs) const {

return Vector(y \* rhs.z - z \* rhs.y, z \* rhs.x - x \* rhs.z, x \* rhs.y - y \* rhs.x);

}

float length() const { return std::sqrt(x \* x + y \* y + z \* z); }

Vector normalize() const {

float len = length();

if (len == 0) {

return Vector(0, 0, 0);

}

return Vector(x / len, y / len, z / len);

}

};

class HashTable {

public:

HashTable(size\_t table\_size) : table\_size\_(table\_size) {

table\_.resize(table\_size\_);

}

void insert(const Point& p) {

auto hash = hash\_point(p);

auto bucket = hash % table\_size\_;

auto& chain = table\_[bucket];

auto it = std::find\_if(chain.begin(), chain.end(),

[&](const Point& q) { return equals(p, q); });

if (it == chain.end()) {

chain.push\_back(p);

}

}

std::vector<Point>& operator[](size\_t bucket) { return table\_[bucket]; }

private:

size\_t hash\_point(const Point& p) const {

std::hash<float> float\_hash;

return float\_hash(p.x) ^ float\_hash(p.y) ^ float\_hash(p.z);

}

bool equals(const Point& p, const Point& q) const {

return p.x == q.x && p.y == q.y && p.z == q.z;

}

size\_t table\_size\_;

std::vector<std::vector<Point>> table\_;

};

class NormalEstimator {

public:

NormalEstimator(const std::vector<Point>& points, float radius)

: points\_(points), radius\_(radius), hash\_table\_(10000) {}

Vector estimate\_normal(const Point& p) {

auto neighbors = get\_neighbors(p);

return compute\_normal(p, neighbors);

}

private:

std::vector<Point> get\_neighbors(const Point& p) {

std::vector<Point> neighbors;

auto hash = hash\_point(p);

auto bucket = hash % hash\_table\_.size();

auto& chain = hash\_table\_[bucket];

for (auto q : chain) {

if (distance(p, q) <= radius\_) {

neighbors.push\_back(q);

}

}

return neighbors;

}

Vector compute\_normal(const Point& p, const std::vector<Point>& neighbors) {

if (neighbors.size() < 3) {

return Vector(0, 0, 0);

}

Vector sum(0, 0, 0);

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

auto& q = neighbors[i];

auto& r = neighbors[(i + 1) % neighbors.size()];

auto v1 = Vector(q.x, q.y, q.z) - Vector(p.x, p.y, p.z);

auto v2 = Vector(r.x, r.y, r.z) - Vector(p.x, p.y, p.z);

sum = sum + v1.cross(v2);

}

return sum.normalize();

}

size\_t hash\_point(const Point& p) const {

std::hash<float> float\_hash;

return float\_hash(p.x) ^ float\_hash(p.y) ^ float\_hash(p.z);

}

float distance(const Point& p, const Point& q) const {

auto dx = p.x - q.x;

auto dy = p.y - q.y;

auto dz = p.z - q.z;

return std::sqrt(dx \* dx + dy \* dy + dz \* dz);

}

const std::vector<Point>& points\_;

float radius\_;

HashTable hash\_table\_;

};

int main() {

std::vector<Point> points = {{0.0f, 0.0f, 0.0f},

{1.0f, 0.0f, 0.0f},

{0.0f, 1.0f, 0.0f},

{-1.0f, 0.0f, 0.0f},

{0.0f, -1.0f, 0.0f},

{0.0f, 0.0f, 1.0f},

{0.0f, 0.0f, -1.0f}};

size\_t table\_size = 10000;

float radius = 1.0f;

sycl::queue q;

sycl::buffer<Point, 1> points\_buf(points.data(), sycl::range<1>(points.size()));

sycl::buffer<HashTable> hash\_table\_buf(&table\_size, sycl::range<1>(1));

q.submit([&](sycl::handler& cgh) {

auto points\_acc = points\_buf.get\_access<sycl::access::mode::read>(cgh);

auto hash\_table\_acc = hash\_table\_buf.get\_access<sycl::access::mode::read\_write>(cgh);

cgh.parallel\_for(sycl::range<1>(points.size()), [=](sycl::id<1> idx) {

hash\_table\_acc[0].insert(points\_acc[idx]);

});

});

auto hash\_table = hash\_table\_buf.get\_host\_access()[0];

NormalEstimator normal\_estimator(points, radius);

for (auto p : points) {

auto normal = normal\_estimator.estimate\_normal(p);

std::cout << "Point (" << p.x << ", " << p.y << ", " << p.z << ") has normal ("

<< normal.x << ", " << normal.y << ", " << normal.z << ")" << std::endl;

}

return 0;

}

```

在这个例子中，我们首先定义