# ДОДАТОК А

## Програмна реалізація на мові програмування C++

```
#include <vector>
namespace SPHAlgorithms{
template<typename _Tp> class Point3{
public:
    typedef _Tp value_type;
    Point3();
    Point3(double _x, double _y);
    Point3(double _x, double _y, double _z);
    Point3(const Point3& pt);
    auto calcNormSqr() const;
    auto calcNorm() const;
    Point3& operator = (const Point3& pt);
    _Tp x, y, z; //< the point coordinates
};
using Point3D = Point3<double>;
using Point3DVector = std::vector<Point3D>;
} //SPHAlgorithms

#include "Point.hpp"
#include "Point.h"
#include <cmath>
namespace SPHAlgorithms{
template<typename _Tp> inline Point3<_Tp>::Point3()
    : x(0),
      y(0),
      z(0) {}
template<typename _Tp> inline Point3<_Tp>::Point3(double _x, double _y)
    : x(_x),
      y(_y),
      z(0) {}
template<typename _Tp> inline Point3<_Tp>::Point3(double _x, double _y, double _z)
    : x(_x),
      y(_y),
      z(_z) {}
template<typename _Tp> inline Point3<_Tp>::Point3(const Point3& pt)
    : x(pt.x),
      y(pt.y),
      z(pt.z) {}
template<typename _Tp> inline auto Point3<_Tp>::calcNormSqr() const{
    return x * x + y * y + z * z;
}
template<typename _Tp> inline auto Point3<_Tp>::calcNorm() const{
    return std::sqrt(calcNormSqr());
}
template<typename _Tp> inline Point3<_Tp>& Point3<_Tp>::operator = (const Point3& pt){
    x = pt.x;
    y = pt.y;
    z = pt.z;
```

```
    return *this;
}
template<typename _Tp> inline bool operator == (const Point3<_Tp>& a, const Point3<_Tp>& b){
    return a.x == b.x && a.y == b.y && a.z == b.z;
}
template<typename _Tp> inline bool operator != (const Point3<_Tp>& a, const Point3<_Tp>& b){
    return a.x != b.x || a.y != b.y || a.z != b.z;
}
template<typename _Tp> inline Point3<_Tp> operator *= (Point3<_Tp>& a, const double b){
    a.x *= b;
    a.y *= b;
    a.z *= b;
    return a;
}
template<typename _Tp> inline Point3<_Tp> operator += (Point3<_Tp>& a, const Point3<_Tp>& b){
    a.x += b.x;
    a.y += b.y;
    a.z += b.z;
    return a;
}
template<typename _Tp> inline Point3<_Tp> operator + (const Point3<_Tp>& a, const Point3<_Tp>& b){
    return Point3<_Tp>(a.x + b.x, a.y + b.y, a.z + b.z);
}
template<typename _Tp> inline Point3<_Tp> operator - (const Point3<_Tp>& a, const Point3<_Tp>& b){
    return Point3<_Tp>(a.x - b.x, a.y - b.y, a.z - b.z);
}
template<typename _Tp> inline Point3<_Tp> operator - (const Point3<_Tp>& a){
    return Point3<_Tp>(-a.x, -a.y, -a.z);
}
template<typename _Tp> inline Point3<_Tp> operator / (const Point3<_Tp> a, const double b){
    return Point3<_Tp>(a.x / b, a.y / b, a.z / b);
}
template<typename _Tp> inline Point3<_Tp> operator / (const double b, const Point3<_Tp> a){
    return Point3<_Tp>(b / a.x, b / a.y, b / a.z);
}
template<typename _Tp> inline Point3<_Tp> operator * (const Point3<_Tp>& a, const double b){
    return Point3<_Tp>(a.x * b, a.y * b, a.z * b);
}
template<typename _Tp> inline Point3<_Tp> operator * (const double b, const Point3<_Tp>& a){
    return Point3<_Tp>(b * a.x, b * a.y, b * a.z);
}
} //SPHAlgorithms

#include "Point.h"
namespace SPHAlgorithms{
struct Cuboid{
    Point3D startingPoint;
    double width, length, height; // x, y, z axis
    Cuboid() :
        startingPoint(Point3D()),
        width(0.),
        length(0.),
        height(0.) {}
```

```
    Cuboid(const Point3D& _startingPoint, const double _width, const double _length, const double
_height) :
        startingPoint(_startingPoint),
        width(_width),
        length(_length),
        height(_height) {}
};
class Volume {
public:
    Volume();
    Volume(const Cuboid& cube);
    ~Volume();
    Cuboid getBoundingCuboid() const;
private:
    Cuboid m_boundingCuboid;
};
} //SPHAlgorithms

#include <vector>
#include <stddef.h>
namespace SPHAlgorithms{
    using SizetVector = std::vector<size_t>;
    using VectorOfSizetVectors = std::vector<SizetVector>;
} //SPHAlgorithms

#include "NeighboursSearch.h"

#include <cfloat>
#include <cmath>

namespace SPHAlgorithms{
static size_t normalizedCuboidWidth;
static size_t normalizedCuboidLength;
static size_t normalizedCuboidHeight;
template <class T>
NeighboursSearch3D<T>::NeighboursSearch3D(const Volume& volume, double radius, double eps)
    : m_volume(volume)
    , m_radius(radius)
    , m_eps(eps)
    , m_boxes(VectorOfSizetVectors()){
        const Cuboid cuboid = m_volume.getBoundingCuboid();
        m_cuboid = cuboid;
        normalizedCuboidWidth = static_cast<size_t>(m_cuboid.width / m_radius);
        normalizedCuboidLength = static_cast<size_t>(m_cuboid.length / m_radius);
        normalizedCuboidHeight = static_cast<size_t>(m_cuboid.height / m_radius);
        m_boxesNumber = static_cast<size_t>(normalizedCuboidWidth *
                                            normalizedCuboidLength *
                                            normalizedCuboidHeight);
        m_boxes.resize(m_boxesNumber);
        m_nearbyBoxes.resize(m_boxesNumber);
        findNearbyBoxes();
    }
template <class T> NeighboursSearch3D<T>::~NeighboursSearch3D() = default;
```

```
template <class T> void NeighboursSearch3D<T>::search(T& points){
    for (size_t i = 0; i < points.size(); i++)
        points[i].neighbours.clear();
    insertPointsIntoBoxes(points);
    for (size_t boxIndex = 0; boxIndex < m_boxes.size(); boxIndex++)
        for (size_t pointIndex = 0; pointIndex < m_boxes[boxIndex].size(); pointIndex++)
            for (size_t nearbyPointIndex = 0; nearbyPointIndex < m_boxes[boxIndex].size(); near-
byPointIndex++)
                if (pointIndex != nearbyPointIndex){
                    Point3D difference = points[m_boxes[boxIndex][pointIndex]].position -
                                         points[m_boxes[boxIndex][nearbyPointIndex]].position;
                    if (difference.calcNormSqr() <= pow(m_radius, 2))
points[m_boxes[boxIndex][pointIndex]].neighbours.push_back(m_boxes[boxIndex][nearbyPointIndex]);}
    for (size_t boxIndex = 0; boxIndex < m_boxes.size(); boxIndex++)
        for (size_t pointIndex = 0; pointIndex < m_boxes[boxIndex].size(); pointIndex++)
            for (size_t nearbyBoxIndex = 0; nearbyBoxIndex < m_nearbyBoxes[boxIndex].size(); near-
byBoxIndex++)
                for (size_t nearbyPointIndex = 0; nearbyPointIndex <
m_boxes[m_nearbyBoxes[boxIndex][nearbyBoxIndex]].size(); nearbyPointIndex++){
                    Point3D difference = points[m_boxes[boxIndex][pointIndex]].position -
points[m_boxes[m_nearbyBoxes[boxIndex][nearbyBoxIndex]][nearbyPointIndex]].position;
                    if (difference.calcNormSqr() - pow(m_radius, 2) <= DBL_EPSILON)
                        points[m_boxes[boxIndex][pointIndex]].neighbours
.push_back(m_boxes[m_nearbyBoxes[boxIndex][nearbyBoxIndex]][nearbyPointIndex]);}
}
template <class T> void NeighboursSearch3D<T>::insertPointsIntoBoxes(const T& points){
    for (size_t i = 0; i < m_boxesNumber; i++)
        m_boxes[i].clear();
    m_pointsSize = points.size();
    for (size_t i = 0; i < m_pointsSize; i++){
        // The Formula is created manually using height layers approach
        size_t widthOffset = static_cast<size_t>(points[i].position.x / m_radius);
        size_t lengthOffset = static_cast<size_t>(points[i].position.y / m_radius) *
                              normalizedCuboidWidth;
        size_t heightOffset = static_cast<size_t>(points[i].position.z / m_radius) *
                              normalizedCuboidLength * normalizedCuboidWidth;
        if (std::abs(points[i].position.x - m_cuboid.width) < m_eps)
            widthOffset -= 1;
        if (std::abs(points[i].position.y - m_cuboid.length) < m_eps)
            lengthOffset -= normalizedCuboidLength;
        if (std::abs(points[i].position.z - m_cuboid.height) < m_eps)
            heightOffset -= normalizedCuboidLength * normalizedCuboidWidth;
        size_t boxIndex = widthOffset + lengthOffset + heightOffset;
        m_boxes[boxIndex].push_back(i);
    }
}
template <class T> void NeighboursSearch3D<T>::findNearbyBoxes(){
    for (size_t boxIndex = 0; boxIndex < m_boxes.size(); boxIndex++){
        const SizetVector boxComponents = getComponentsOfBoxIndex(boxIndex);
        BoxType boxType = getBoxType(boxComponents);
        defineNearbyBoxes(boxType, boxComponents, boxIndex);
    }
}
```

```
template <class T> SizetVector NeighboursSearch3D<T>::getComponentsOfBoxIndex(const size_t boxIn-
dex){
    size_t boxWidth = boxIndex % normalizedCuboidWidth;
    size_t boxHeight = (boxIndex - boxWidth) / (normalizedCuboidWidth * normalizedCuboidLength);
    size_t boxLength = (boxIndex - boxWidth -
                        boxHeight * normalizedCuboidWidth * normalizedCuboidLength) /
                        normalizedCuboidWidth;
    SizetVector components = {boxWidth, boxLength, boxHeight};
    return components;
}
template <class T> typename NeighboursSearch3D<T>::BoxType NeighboursSearch3D<T>::getBoxType(const
SizetVector& components){
    if ((components[0] == 0 || components[0] == normalizedCuboidWidth - 1) &&
        (components[1] == 0 || components[1] == normalizedCuboidLength - 1) &&
        (components[2] == 0 || components[2] == normalizedCuboidHeight - 1))
    {return outerCorner;}
    if ((components[0] != 0 && components[0] != normalizedCuboidWidth - 1) &&
        (components[1] == 0 || components[1] == normalizedCuboidLength - 1) &&
        (components[2] != 0 && components[2] != normalizedCuboidHeight - 1))
    {return outerCenter;}
    if ((components[1] == 0 || components[1] == normalizedCuboidLength - 1) && (
         ((components[0] != 0 && components[0] != normalizedCuboidWidth - 1) &&
          (components[2] == 0 || components[2] == normalizedCuboidHeight - 1)) ||
         ((components[0] == 0 || components[0] == normalizedCuboidWidth - 1) &&
          (components[2] != 0 && components[2] != normalizedCuboidHeight - 1))))
    {return outerLongitual;}
    if ((components[0] == 0 || components[0] == normalizedCuboidWidth - 1) &&
        (components[1] != 0 && components[1] != normalizedCuboidLength - 1) &&
        (components[2] == 0 || components[2] == normalizedCuboidHeight - 1))
    {return innerCorner;}
    if ((components[0] != 0 && components[0] != normalizedCuboidWidth - 1) &&
        (components[1] != 0 && components[1] != normalizedCuboidLength - 1) &&
        (components[2] != 0 && components[2] != normalizedCuboidHeight - 1))
    {return innerCenter;}
    if ((components[1] != 0 && components[1] != normalizedCuboidLength - 1) && (
         ((components[0] != 0 && components[0] != normalizedCuboidWidth - 1) &&
          (components[2] == 0 || components[2] == normalizedCuboidHeight - 1)) ||
         ((components[0] == 0 || components[0] == normalizedCuboidWidth - 1) &&
          (components[2] != 0 && components[2] != normalizedCuboidHeight - 1))))
    { return innerLongitual;}
}
template    <class    T>    void    NeighboursSearch3D<T>::defineNearbyBoxes(const    Neighbours-
Search3D<T>::BoxType boxType,
                                                                const SizetVector& components,
                                                                const size_t boxIndex){
    bool isLeft     = components[0] == 0;
    bool isRight    = components[0] == normalizedCuboidWidth - 1;
    bool isBack     = components[1] == 0;
    bool isFront    = components[1] == normalizedCuboidLength - 1;
    bool isBottom   = components[2] == 0;
    bool isTop      = components[2] == normalizedCuboidHeight - 1;
    switch (boxType) {
        case outerCorner:
```

```
            addNearbyBoxesFor(isLeft, isRight, isTop, isBottom, components, boxIndex);
            isBack ? addForBack(boxIndex) : addForFront(boxIndex);
            break;
        case outerCenter:
            addForCenter(components, boxIndex);
            isBack ? addForBack(boxIndex) : addForFront(boxIndex);
            break;
        case outerLongitual:
            addNearbyBoxesFor(isLeft, isRight, isTop, isBottom, components, boxIndex);
            isBack ? addForBack(boxIndex) : addForFront(boxIndex);
            break;
        case innerCorner:
            addNearbyBoxesFor(isLeft, isRight, isTop, isBottom, components, boxIndex);
            addForMiddle(boxIndex);
            break;
        case innerCenter:
            addForCenter(components, boxIndex);
            addForMiddle(boxIndex);
            break;
        case innerLongitual:
            addNearbyBoxesFor(isLeft, isRight, isTop, isBottom, components, boxIndex);
            addForMiddle(boxIndex);
            break;
    }
}
template <class T> void NeighboursSearch3D<T>:: addNearbyBoxesFor(const bool isLeft,
                                                                  const bool isRight,
                                                                  const bool isTop,
                                                                  const bool isBottom,
                                                                  const SizetVector& components,
                                                                  const size_t boxIndex){
    if (isLeft && isBottom) {
        addForBottomLeft(components, boxIndex);
        return;
    }
    if (isRight && isBottom) {
        addForBottomRight(components, boxIndex);
        return;
    }
    if (isLeft && isTop) {
        addForTopLeft(components, boxIndex);
        return;
    }
    if (isRight && isTop) {
        addForTopRight(components, boxIndex);
        return;
    }
    if (isLeft) {
        addForLeft(components, boxIndex);
        return;
    }
    if (isRight) {
        addForRight(components, boxIndex);
```

```
        return;
    }
    if (isBottom) {
        addForBottom(components, boxIndex);
        return;
    }
    if (isTop) {
        addForTop(components, boxIndex);
        return;
    }
}
template <class T> void NeighboursSearch3D<T>:: addForTopLeft(const SizetVector& components,
                                                              const size_t boxIndex){
    // addRight
    m_nearbyBoxes[boxIndex].push_back(boxIndex + 1);
    // addBotom
    m_nearbyBoxes[boxIndex].push_back(boxIndex - normalizedCuboidWidth * normalizedCuboidLength);
    // addBottomRight
   m_nearbyBoxes[boxIndex].push_back(boxIndex - normalizedCuboidWidth * normalizedCuboidLength + 1);
}
template <class T> void NeighboursSearch3D<T>:: addForTopRight(const SizetVector& components,
                                                               const size_t boxIndex){
    // addLeft
    m_nearbyBoxes[boxIndex].push_back(boxIndex - 1);
    // addBotom
    m_nearbyBoxes[boxIndex].push_back(boxIndex - normalizedCuboidWidth * normalizedCuboidLength);
    // addBottomLeft
   m_nearbyBoxes[boxIndex].push_back(boxIndex - normalizedCuboidWidth * normalizedCuboidLength - 1);
}
template <class T> void NeighboursSearch3D<T>:: addForBottomLeft(const SizetVector& components,
                                                                 const size_t boxIndex){
    // addRight
    m_nearbyBoxes[boxIndex].push_back(boxIndex + 1);
    // addTop
    m_nearbyBoxes[boxIndex].push_back(boxIndex + normalizedCuboidWidth * normalizedCuboidLength);
    // addTopRight
   m_nearbyBoxes[boxIndex].push_back(boxIndex + normalizedCuboidWidth * normalizedCuboidLength + 1);
}
template <class T> void NeighboursSearch3D<T>:: addForBottomRight(const SizetVector& components,
                                                                  const size_t boxIndex){
    // addLeft
    m_nearbyBoxes[boxIndex].push_back(boxIndex - 1);
    // addTop
    m_nearbyBoxes[boxIndex].push_back(boxIndex + normalizedCuboidWidth * normalizedCuboidLength);
    // addTopLeft
   m_nearbyBoxes[boxIndex].push_back(boxIndex + normalizedCuboidWidth * normalizedCuboidLength - 1);
}
template <class T> void NeighboursSearch3D<T>:: addForCenter(const SizetVector& components,
                                                             const size_t boxIndex){
    // addRight
    m_nearbyBoxes[boxIndex].push_back(boxIndex + 1);
    // addLeft
    m_nearbyBoxes[boxIndex].push_back(boxIndex - 1);
```

```
    // addTop
    m_nearbyBoxes[boxIndex].push_back(boxIndex + normalizedCuboidWidth * normalizedCuboidLength);
    // addBottom
    m_nearbyBoxes[boxIndex].push_back(boxIndex - normalizedCuboidWidth * normalizedCuboidLength);
    // addTopRight
   m_nearbyBoxes[boxIndex].push_back(boxIndex + normalizedCuboidWidth * normalizedCuboidLength + 1);
    // addTopLeft
   m_nearbyBoxes[boxIndex].push_back(boxIndex + normalizedCuboidWidth * normalizedCuboidLength - 1);
    // addBottomRight
   m_nearbyBoxes[boxIndex].push_back(boxIndex - normalizedCuboidWidth * normalizedCuboidLength + 1);
    // addBottomLeft
   m_nearbyBoxes[boxIndex].push_back(boxIndex - normalizedCuboidWidth * normalizedCuboidLength - 1);
}
template <class T> void NeighboursSearch3D<T>:: addForLeft(const SizetVector& components,
                                                           const size_t boxIndex){
    // addRight
    m_nearbyBoxes[boxIndex].push_back(boxIndex + 1);
    // addTop
    m_nearbyBoxes[boxIndex].push_back(boxIndex + normalizedCuboidWidth * normalizedCuboidLength);
    // addBottom
    m_nearbyBoxes[boxIndex].push_back(boxIndex - normalizedCuboidWidth * normalizedCuboidLength);
    // addTopRight
   m_nearbyBoxes[boxIndex].push_back(boxIndex + normalizedCuboidWidth * normalizedCuboidLength + 1);
    // addBottomRight
   m_nearbyBoxes[boxIndex].push_back(boxIndex - normalizedCuboidWidth * normalizedCuboidLength + 1);
}
template <class T> void NeighboursSearch3D<T>:: addForRight(const SizetVector& components,
                                                            const size_t boxIndex){
    // addLeft
    m_nearbyBoxes[boxIndex].push_back(boxIndex - 1);
    // addTop
    m_nearbyBoxes[boxIndex].push_back(boxIndex + normalizedCuboidWidth * normalizedCuboidLength);
    // addBottom
    m_nearbyBoxes[boxIndex].push_back(boxIndex - normalizedCuboidWidth * normalizedCuboidLength);
    // addTopLeft
   m_nearbyBoxes[boxIndex].push_back(boxIndex + normalizedCuboidWidth * normalizedCuboidLength - 1);
    // addBottomLeft
   m_nearbyBoxes[boxIndex].push_back(boxIndex - normalizedCuboidWidth * normalizedCuboidLength - 1);
}
template <class T> void NeighboursSearch3D<T>:: addForTop(const SizetVector& components,
                                                          const size_t boxIndex){
    // addRight
    m_nearbyBoxes[boxIndex].push_back(boxIndex + 1);
    // addLeft
    m_nearbyBoxes[boxIndex].push_back(boxIndex - 1);
    // addBottom
    m_nearbyBoxes[boxIndex].push_back(boxIndex - normalizedCuboidWidth * normalizedCuboidLength);
    // addBottomRight
   m_nearbyBoxes[boxIndex].push_back(boxIndex - normalizedCuboidWidth * normalizedCuboidLength + 1);
    // addBottomLeft
   m_nearbyBoxes[boxIndex].push_back(boxIndex - normalizedCuboidWidth * normalizedCuboidLength - 1);
}
template <class T> void NeighboursSearch3D<T>:: addForBottom(const SizetVector& components,
```

```
                                                        const size_t boxIndex){
    // addRight
    m_nearbyBoxes[boxIndex].push_back(boxIndex + 1);
    // addLeft
    m_nearbyBoxes[boxIndex].push_back(boxIndex - 1);
    // addTop
    m_nearbyBoxes[boxIndex].push_back(boxIndex + normalizedCuboidWidth * normalizedCuboidLength);
    // addTopRight
    m_nearbyBoxes[boxIndex].push_back(boxIndex + normalizedCuboidWidth * normalizedCuboidLength +
1);
    // addTopLeft
   m_nearbyBoxes[boxIndex].push_back(boxIndex + normalizedCuboidWidth * normalizedCuboidLength - 1);
}
template <class T> void NeighboursSearch3D<T>:: addForBack(const size_t boxIndex){
    SizetVector nearbyBoxes = m_nearbyBoxes[boxIndex];
    nearbyBoxes.push_back(boxIndex);
    for (int i = 0; i < nearbyBoxes.size(); i++) {
        nearbyBoxes[i] += normalizedCuboidWidth;
        m_nearbyBoxes[boxIndex].push_back(nearbyBoxes[i]);
    }
}
template <class T> void NeighboursSearch3D<T>:: addForFront(const size_t boxIndex){
    SizetVector nearbyBoxes = m_nearbyBoxes[boxIndex];
    nearbyBoxes.push_back(boxIndex);
    for (int i = 0; i < nearbyBoxes.size(); i++) {
        nearbyBoxes[i] -= normalizedCuboidWidth;
        m_nearbyBoxes[boxIndex].push_back(nearbyBoxes[i]);
    }
}
template <class T> void NeighboursSearch3D<T>:: addForMiddle(const size_t boxIndex){
    SizetVector nearbyBoxes = m_nearbyBoxes[boxIndex];
    nearbyBoxes.push_back(boxIndex);
    addForBack(boxIndex);
    for (int i = 0; i < nearbyBoxes.size(); i++) {
        nearbyBoxes[i] -= normalizedCuboidWidth;
        m_nearbyBoxes[boxIndex].push_back(nearbyBoxes[i]);
    }
}
} // namespace SPHAlgorithms

#include "sph/include/Collisions.h"

#include "Config.h"
#include "algorithms/include/Area.h"
#include "algorithms/include/NeighboursSearch.h"
namespace SPHSDK
{
static double calculateF(SPHAlgorithms::Point3D differenceParticleNeighbour){
 return differenceParticleNeighbour.calcNormSqr() - Config::ParticleRadius * Config::ParticleRadius;
}
static SPHAlgorithms::Point3D calculateContactPoint(SPHAlgorithms::Point3D particlePosition,
                                                    SPHAlgorithms::Point3D
                                                                differenceParticleNeighbour){
```

```
    double particleDistance = differenceParticleNeighbour.calcNorm();
return particlePosition + (differenceParticleNeighbour / particleDistance) *Config::ParticleRadius;}
static SPHAlgorithms::Point3D calculateSurfaceNormal(SPHAlgorithms::Point3D
                                                        differenceParticleNeighbour){
    double particleDistance = differenceParticleNeighbour.calcNorm();
    return - differenceParticleNeighbour / particleDistance;
}
static SPHAlgorithms::Point3D calculateVelocity(SPHAlgorithms::Point3D particleVelocity,
                                                SPHAlgorithms::Point3D differenceParticleNeighbour){
        double scalarProduct = particleVelocity.x * differenceParticleNeighbour.x
                             + particleVelocity.y * differenceParticleNeighbour.y
                             + particleVelocity.z * differenceParticleNeighbour.z;
        return particleVelocity - differenceParticleNeighbour * 2 * scalarProduct;
}
void Collision::detectCollisions(ParticleVect& particleVect, const SPHAlgorithms::Volume& volume){
    for (size_t i = 0; i < particleVect.size(); i++){
        for (size_t j = 0; j < particleVect[i].neighbours.size(); j++){
            SPHAlgorithms::Point3D differenceParticleNeighbour =
                    particleVect[i].position - particleVect[particleVect[i].neighbours[j]].position;
            if (calculateF(differenceParticleNeighbour) < 0){
            const SPHAlgorithms::Point3D surfaceNormal =
                      calculateSurfaceNormal(differenceParticleNeighbour);
            particleVect[i].position = calculateContactPoint(particleVect[i].position,
                                                             differenceParticleNeighbour);
            particleVect[i].velocity = calculateVelocity(particleVect[i].velocity, surfaceNormal);
            }
        }
        const SPHAlgorithms::Cuboid cuboid = volume.getBoundingCuboid();
        if (particleVect[i].position.x > cuboid.width - particleVect[i].radius){
            particleVect[i].position.x = cuboid.width - particleVect[i].radius;
            particleVect[i].velocity.x *= -0.5;
        }
        if (particleVect[i].position.x < particleVect[i].radius){
            particleVect[i].position.x = particleVect[i].radius;
            particleVect[i].velocity.x *= -0.5;
        }
        if (particleVect[i].position.y > cuboid.length - particleVect[i].radius){
            particleVect[i].position.y = cuboid.length - particleVect[i].radius;
            particleVect[i].velocity.y *= -0.5;
        }
        if (particleVect[i].position.y < particleVect[i].radius){
            particleVect[i].position.y = particleVect[i].radius;
            particleVect[i].velocity.y *= -0.5;
        }
        if (particleVect[i].position.z > cuboid.height - particleVect[i].radius){
            particleVect[i].position.z = cuboid.height - particleVect[i].radius;
            particleVect[i].velocity.z *= -0.5;
        }
        if (particleVect[i].position.z < particleVect[i].radius){
            particleVect[i].position.z = particleVect[i].radius;
            particleVect[i].velocity.z *= -0.5;
        }
    }
```

```
}

}// namespace SPHSDK

#include "Point.h"

#include <functional>

namespace SPHAlgorithms
{

/**
 * @brief MarchingCubes class implements Marching Cubes algorithm.
 */
class MarchingCubes
{

public:
    /**
     * @brief Generates triangles mesh from function
     * @param f    The function that represents the domain equation
     */
    static Point3FVector generateMesh(std::function<float(float, float, float)> f);

private:
    static Point3FVector MarchingCube(std::function<float(float, float, float)> f, float fX, float
fY, float fZ);

    static void
    fillFoundTriangles(Point3FVector& resultEdgeVertex, const Point3FVector& EdgeVertex, const int
iFlagIndex);

    static Point3FVector findPointIntersection(
        const int iEdgeFlags, const float CubeValue[], const float fX, const float fY, const float
fZ);

    static int determineFlag(const float CubeValue[]);
};

} // namespace SPHAlgorithms

class ROperations

{

public:

    /**

     * @brief Returns conjunction of x and y

     * @param x    The x Cartesian coordinate

     * @param y    The y Cartesian coordinate

     * @return the result of conjuction R-operation in R0 system

     */

    template <class T> static T conjunction(T x, T y);


    /**

     * @brief Returns disjunction of x and y

     * @param x    The x Cartesian coordinate

     * @param y    The y Cartesian coordinate

     * @return the result of disjunction R-operation in R0 system

     */

    template <class T> static T disjunction(T x, T y);

};


} // namespace SPHAlgorithms


#include "ROperations.hpp"

#include <cassert>
#include <cmath>

namespace SPHAlgorithms
{
```

```
template <class T> T ROperations::conjunction(T x, T y)
{
    return x + y - std::sqrt(x * x + y * y);
}

template <class T> T ROperations::disjunction(T x, T y)
{
    return x + y + std::sqrt(x * x + y * y);
}

} // namespace SPHAlgorithms

#include "ROperations.h"

namespace SPHAlgorithms
{

/**
 * @brief The Shapes class contanis various shapes equations constructed using the R-functions meth-
od.
 */
class Shapes
{
private:
    static constexpr auto dis = ROperations::disjunction<float>;
    static constexpr auto con = ROperations::conjunction<float>;

public:
    /**
     * @brief Represents a pawn in 3D.
     * @param x    The x-coordinate.
     * @param y    The y-coordinate.
     * @param z    The z-coordinate.
     * @return a value that > 0 inside the object, = 0 on the border and < 0 outside.
     */
    static float Pawn(float x, float y, float z)
    {
        const float x_sqr = (x - 1.5f) * (x - 1.5f);
        const float y_sqr = (y - 1.5f) * (y - 1.5f);
        const float z1_sqr = (z - 0.75f) * (z - 0.75f);
        const float z2_sqr = (1.f - z) * (1.f - z);
        const float z3_sqr = (1.25f - z) * (1.25f - z);

        return dis(con(con(0.25f - x_sqr - y_sqr, -20.f * (x_sqr + y_sqr) + 1.f + 10.f * z1_sqr), z
* (1.f - z)),
                   dis(0.125f - x_sqr - y_sqr - 20.f * z2_sqr, 0.05f - x_sqr - y_sqr - z3_sqr));
    }

    /**
     * @brief Represents a bishop in 3D.
     * @param x    The x-coordinate.
     * @param y    The y-coordinate.
     * @param z    The z-coordinate.
     * @return a value that > 0 inside the object, = 0 on the border and < 0 outside.
     */
    static float Bishop(float x, float y, float z)
    {
        const float x_sqr = (x - 1.5f) * (x - 1.5f);
        const float y_sqr = (y - 1.5f) * (y - 1.5f);
        const float z1_sqr = (z - 0.85f) * (z - 0.85f);
        const float z2_sqr = (1.25f - z) * (1.25f - z);
        const float z3_sqr = (1.4f - z) * (1.4f - z);

        return dis(con(con(0.25f - x_sqr - y_sqr, -20.f * (x_sqr + y_sqr) + 1.f + 10.f * z1_sqr), z
* (1.25f - z)),
                   dis(0.2f - x_sqr - y_sqr - 20.f * z2_sqr, 0.2f - 5.f * x_sqr - 4.f * y_sqr -
z3_sqr));
    }
};

} // namespace SPHAlgorithms

#include "MarchingCubes.h"
#include "MarchingCubesConfig.h"

namespace SPHAlgorithms
{

Point3FVector MarchingCubes::generateMesh(std::function<float(float, float, float)> f)
{
```

```
    Point3FVector mesh;

    for (float x = X_MIN; x < X_MAX; x += GRID_CUBE_SIZE)
    {
        for (float y = Y_MIN; y < Y_MAX; y += GRID_CUBE_SIZE)
        {
            for (float z = Z_MIN; z < Z_MAX; z += GRID_CUBE_SIZE)
            {
                // iX, iY, iZ are coordinates of the current vertex in grid_cube
                const auto vertices = MarchingCube(f, x, y, z);
                std::copy(vertices.begin(), vertices.end(), std::back_inserter(mesh));
            }
        }
    }
    return mesh;
}

static float adapt(float a, float b)
{
    const float delta = b - a;

    if (std::abs(delta) < PRECIZION)
    {
        return 0.5f;
    }

    return -a / delta;
}

Point3FVector MarchingCubes::MarchingCube(std::function<float(float, float, float)> f, float fX,
float fY, float fZ)
{
    // vector with all triangles found
    Point3FVector trianglesMesh;

    float CubeValue[CUBE_VERTICES_NUMBER];

    // Evaluate value of the cube vertex at each point
    for (int iVertex = 0; iVertex < CUBE_VERTICES_NUMBER; iVertex++)
    {
        CubeValue[iVertex] =
            f(fX + VertexOffset[iVertex][0], fY + VertexOffset[iVertex][1], fZ + VertexOff-
set[iVertex][2]);
    }

    // Find which vertices are inside of the surface and which are outside
    int iFlagIndex = determineFlag(CubeValue);

    // Find which edges are intersected by the surface
    int iEdgeFlags = CubeEdgeFlags[iFlagIndex];

    // If the cube is entirely inside or outside of the surface, then there will be no intersections
    if (iEdgeFlags == 0)
    {
        return trianglesMesh;
    }

    // Fill the triangles that were found.  There can be up to five per cube
    fillFoundTriangles(trianglesMesh, findPointIntersection(iEdgeFlags, CubeValue, fX, fY, fZ),
iFlagIndex);

    return trianglesMesh;
}

void MarchingCubes::fillFoundTriangles(Point3FVector&       resultEdgeVertex,
                                       const Point3FVector& EdgeVertex,
                                       const int            iFlagIndex)
{
    for (int iTriangle = 0; iTriangle < TRIANGLES_MAX_NUMBER_FOR_ONE_CUBE; iTriangle++)
    {
        if (TriangleConnectionTable[iFlagIndex][TRIANGLES_CORNERS_NUMBER * iTriangle] < 0)
        {
            break;
        }

        for (int iCorner = 0; iCorner < TRIANGLES_CORNERS_NUMBER; iCorner++)
        {
            const int iVertex = TriangleConnectionTable[iFlagIndex][TRIANGLES_CORNERS_NUMBER * iTri-
angle + iCorner];
```

```
            resultEdgeVertex.push_back(Point3F(EdgeVertex[iVertex].x, EdgeVertex[iVertex].y,
EdgeVertex[iVertex].z));
        }
    }
}

// Find points of intersection on each edge
Point3FVector MarchingCubes::findPointIntersection(
    const int iEdgeFlags, const float CubeValue[], const float fX, const float fY, const float fZ)
{
    Point3FVector EdgeVertex(12);

    for (int iEdge = 0; iEdge < CUBE_EDGES_NUMBER; iEdge++)
    {
        // if there is an intersection on this edge
        if (iEdgeFlags & (1 << iEdge))
        {
            // Find the two vertices specified by this edge, and interpolate them according to adapt
            const int v0 = EdgeConnection[iEdge][0];
            const int v1 = EdgeConnection[iEdge][1];

            const float f0 = CubeValue[v0];
            const float f1 = CubeValue[v1];

            const float t0 = 1.f - adapt(f0, f1);
            const float t1 = 1.f - t0;

            EdgeVertex[iEdge].x = fX + VertexOffset[v0][0] * t0 + VertexOffset[v1][0] * t1;
            EdgeVertex[iEdge].y = fY + VertexOffset[v0][1] * t0 + VertexOffset[v1][1] * t1;
            EdgeVertex[iEdge].z = fZ + VertexOffset[v0][2] * t0 + VertexOffset[v1][2] * t1;
        }
    }
    return EdgeVertex;
}
int MarchingCubes::determineFlag(const float CubeValue[])
{
    int flagIndex = 0;

    for (int iVertexTest = 0; iVertexTest < CUBE_VERTICES_NUMBER; iVertexTest++)
    {
        if (CubeValue[iVertexTest] > 0)
        {
            flagIndex |= 1 << iVertexTest;
        }
    }
    return flagIndex;
}
} // namespace SPHAlgorithms
```

# ВІДОМІСТЬ АТЕСТАЦІЙНОЇ РОБОТИ

| Позначення | Найменування | Дод. відомості |
|---|---|---|
| | Текстові документи | |
| 1 | Пояснювальна записка | 83 с. |
| 2 | Презентаційний матеріал | 25 с. |
| | Інші документи | |
| 3 | Роздруківки програм | 14 с. |
| 4 | Рецензія | 2 с. |
| 5 | Відгук керівника | 1 с. |

| | | | | | **Оптимізація обчислень при моделюванні руху об’єктів з перешкодами за допомогою графічного процессору** | | | |
|---|---|---|---|---|---|---|---|---|
| | | | | | | | | |
| **Змін** | **Арк.** | **Номер докум.** | **Підп.** | **Дата** | | | | |
| **Розроб.** | | **Литвин І.Р.** | | | **(Тема роботи) Відомість атестаційної роботи** | | **Аркуш** | **Аркушів** |
| **Перевір.** | | **Гибкіна Н.В.** | | | | | | |
| **Н. контр.** | | **Сидоров М.В.** | | | | **ХНУРЕ кафедра ПМ** | | |
| **Затв.** | | **Тевяшев А.Д.** | | | | | | |