Swinburne University of Technology

School of Science, Computing and Engineering Technologies

ASSIGNMENT COVER SHEET

Subject Code:	COS30008
Subject Title:	Data Structures and Patterns
Assignment number and title:	1, Solution Design in C++
Due date:	Wednesday, March 27, 2024, 23:59
Lecturer:	Dr. Markus Lumpe

Your name: Vu Duc Tran **Your student ID:** 104175614

Marker's comments:

Problem	Marks	Obtained
1	26	
2	98	
3	32	
Total	156	

Extension certification:

This assignment has been given an extension and is now due on ____________

Signature of Convener:____________

```
//
//  Vector3D_PS1.cpp
//  problem asm1
//
//  Created by Vu Duc Tran on 17/3/2024.
//
#define _USE_MATH_DEFINES    // must be defined before any #include
#include "Vector3D.h"
#include <cassert>
#include <cmath>
#include <sstream>
#include <iomanip>

using namespace std;


std::string Vector3D::toString() const noexcept
{
    std::stringstream ss;
        //Write
        ss << "[" << std::round(x() * 10000.0f) / 10000.0f << ","
                  << std::round(y() * 10000.0f) / 10000.0f << ","
                  << std::round(w() * 10000.0f) / 10000.0f << "]";
        //Return the resulting string
        return ss.str();
}
```

Matrix3x3_PS1

```
//
//  Matrix3x3_PS1.cpp
//  problem asm1
//
//  Created by Vu Duc Tran on 17/3/2024.
//
#define _USE_MATH_DEFINES     // must be defined before any #include
#include "Matrix3x3.h"
#include "Vector3D.h"
#include <cassert>
#include <cmath>
#include <sstream>
#include <iomanip>

using namespace std;

Matrix3x3 Matrix3x3::operator*( const Matrix3x3& aOther ) const noexcept
{
    Vector3D rows[3];
    for (int i = 0; i < 3; ++i)
    {
        float val[3];
        val[0] = fRows[i][0] * aOther.row(0).x() + fRows[i][1] *
aOther.row(1).x() + fRows[i][2] * aOther.row(2).x();
        val[1] = fRows[i][0] * aOther.row(0).y() + fRows[i][1] *
aOther.row(1).y() + fRows[i][2] * aOther.row(2).y();
        val[2] = fRows[i][0] * aOther.row(0).w() + fRows[i][1] *
aOther.row(1).w() + fRows[i][2] * aOther.row(2).w();
        rows[i] = Vector3D(val[0], val[1], val[2]);
    }

    return Matrix3x3(rows[0], rows[1], rows[2]);
}

float Matrix3x3::det() const noexcept
{
    float res = 0;
    int pos_index[3][3] = {{0, 1, 2}, {1, 2, 0}, {2, 0, 1}};
    int neg_index[3][3] = {{0, 2, 1}, {1, 0, 2}, {2, 1, 0}};
    for (int i = 0; i < 3; ++i)
    {
        res += fRows[0][pos_index[i][0]] * fRows[1][pos_index[i][1]] *
fRows[2][pos_index[i][2]]
               - fRows[0][neg_index[i][0]] * fRows[1][neg_index[i][1]] *
fRows[2][neg_index[i][2]];
    }
    return res;
}


bool Matrix3x3::hasInverse() const noexcept
{
    return (det() != 0);
}
```

Matrix3x3_PS1

```
Matrix3x3 Matrix3x3::transpose() const noexcept
{
    if (!hasInverse())
    {
        return Matrix3x3();
    }
    Vector3D result[3];
    for (int i = 0; i < 3; ++i)
    {
        result[i] = Vector3D(fRows[0][i], row(1)[i], row(2)[i]);
    }
    return Matrix3x3(result[0], result[1], result[2]);
}

Matrix3x3 Matrix3x3::inverse() const noexcept
{
    Vector3D rows[3];
    int template_index[3][2] = {{1, 2}, {0, 2}, {0, 1}};
    for (int i = 0; i < 3; ++i)
    {
        float tmp_value[3];
        for (int j = 0; j < 3; ++j)
        {
            //template_index[i] = row index chosen to calculate the current 
position
            //template_index[j] = column index chosen to calculate the 
current position
            tmp_value[j] = fRows[template_index[i][0]][template_index[j][0]] 
* fRows[template_index[i][1]][template_index[j][1]] - 
fRows[template_index[i][0]][template_index[j][1]] * 
fRows[template_index[i][1]][template_index[j][0]];

            if ((i + j) % 2 != 0) {
                tmp_value[j] *= -1;
            }
        }
        rows[i] = Vector3D(tmp_value[0], tmp_value[1], tmp_value[2]);
    }
    return Matrix3x3(rows[0], rows[1], rows[2]).transpose() * (1 / det());
}

//Write
std::ostream& operator<<( std::ostream& aOStream, const Matrix3x3& aMatrix )
{
    for (int i = 0; i < 3; ++i)
    {
        if (i != 2)
        {
            aOStream << "[" << std::round( aMatrix.row(i).x() * 10000.0f) / 
10000.0f << "," << std::round( aMatrix.row(i).y() * 10000.0f) / 10000.0f << 
"," << std::round( aMatrix.row(i).w() * 10000.0f) / 10000.0f << "],";
        }
        else
        {
            aOStream << "[" << std::round( aMatrix.row(i).x() * 10000.0f) /
10000.0f << "," << std::round( aMatrix.row(i).y() * 10000.0f) / 10000.0f <<
"," << std::round( aMatrix.row(i).w() * 10000.0f) / 10000.0f << "]";
        }
    }
    return aOStream;
}
```

PolygonPS1

```
//
//  PolygonPS1.cpp
//  problem asm1
//
//  Created by Vu Duc Tran on 19/3/2024.
//
#define _USE_MATH_DEFINES     // must be defined before any #include
#include "Matrix3x3.h"
#include "Vector3D.h"
#include <cassert>
#include <cmath>
#include <sstream>
#include <iomanip>
#include "Polygon.h"

using namespace std;

Polygon::Polygon() noexcept :
    fNumberOfVertices()
{}

void Polygon::readData( std::istream& aIStream )
{
    // read input file containing 2D vector data
    // if no data can be read, then exit loop
    while ( aIStream >> fVertices[fNumberOfVertices] )
    {
        fNumberOfVertices++;
    }
}

size_t Polygon::getNumberOfVertices() const noexcept
{
    return fNumberOfVertices;
}

const Vector2D& Polygon::getVertex( size_t aIndex ) const
{
    assert( aIndex < fNumberOfVertices );

    return fVertices[aIndex];
}

float Polygon::getPerimeter() const noexcept
{
    float Result = 0.0f;

    // There have to be at least three vertices
    if ( fNumberOfVertices > 2 )
    {
        // solution without modulus and explicit temporary variables
        for ( size_t i = 1; i < fNumberOfVertices; i++ )
        {
            Result += (fVertices[i] - fVertices[i - 1]).length();
        }
        Result += (fVertices[0] - fVertices[fNumberOfVertices - 1]).length();
    }

    return Result;
}

Polygon Polygon::scale( float aScalar ) const noexcept
{
    Polygon Result = *this;

    for ( size_t i = 0; i < fNumberOfVertices; i++ )
    {
        Result.fVertices[i] = fVertices[i] * aScalar;
    }

    return Result;
}

float Polygon::getSignedArea() const noexcept {
    float Result = (fVertices[fNumberOfVertices - 1].x() - fVertices[0].x()) *
(fVertices[fNumberOfVertices - 1].y() + fVertices[0].y());

    for (int i = 0; i < fNumberOfVertices - 1; ++i) {
        Result += (fVertices[i].x() - fVertices[i + 1].x()) *
(fVertices[i].y() + fVertices[i + 1].y());
    }

    Result /= 2;

    return Result;
}

Polygon Polygon::transform(const Matrix3x3& aMatrix) const noexcept {
    Polygon Result = *this;
    for (int i = 0; i < fNumberOfVertices; ++i) {
        Vector3D temp = Vector3D(fVertices[i]);
        Vector3D dot_product = aMatrix * temp;
        Result.fVertices[i] = Vector2D(dot_product);
    }
    return Result;
}
```