Simulation Lab 1

Date: 10 February 2025

Q1 Sphere Sphere Intersection Test (Summative)

To check if two Spheres intersect you first need to calculate the vector between those two spheres. If the length of that vector is less that the sum of the radii of the two spheres those spheres are intersecting one another. Remember as getting the square root of a number is an expensive operation. Most vector classes include a function to calculate the length (or magnitude) squared so it is better to compare the length of the vector squared with the sum of the two radii squared.

```diff
@@ -0,0 +1,15 @@
+ #pragma once
+ class Vector3 {
+     public:
+         float x, y, z;
+
+         // Constructor
+         Vector3(float x = 0, float y = 0, float z = 0);
+
+         // Vector subtraction
+         Vector3 operator-(const Vector3& other) const;
+
+         // Calculate length squared
+         float LengthSquared() const;
+ };
+
```

```diff
@@ -0,0 +1,16 @@
+ #include "pch.h"
+ #include "Vector3.h"
+
+ // Constructor
+ Vector3::Vector3(float x, float y, float z) : x(x), y(y), z(z) {}
+
+ // Vector subtraction
+ Vector3 Vector3::operator-(const Vector3& other) const {
+     return Vector3(x - other.x, y - other.y, z - other.z);
+ }
+
+ // Calculate the squared length
+ float Vector3::LengthSquared() const {
+     return x * x + y * y + z * z;
+ }
+
```

Simulation Lab 1/Sphere.h

```diff
@@ -0,0 +1,17 @@
+ #pragma once
+ #include "Vector3.h"
+
+ class Sphere {
+ public:
+     Vector3 center;
+     float radius;
+
+     // Constructor
+     Sphere(const Vector3& center, float radius);
+
+     // Method to check collision with another sphere
+     bool CollideWith(const Sphere& other) const;
+ };
+
+
+
```

```diff
@@ -0,0 +1,19 @@
+ #include "pch.h"
+ #include "Sphere.h"
+
+ // Constructor
+ Sphere::Sphere(const Vector3& center, float radius) : center(center), radius(radius)
+
+ // Check collision with another sphere
+ bool Sphere::CollideWith(const Sphere& other) const {
+     // Calculate the squared distance between centers
+     float distanceSquared = (center - other.center).LengthSquared();
+
+     // Calculate the squared sum of radii
+     float radiiSum = radius + other.radius;
+     float radiiSumSquared = radiiSum * radiiSum;
+
+     // Check if the squared distance is less than or equal to squared radii sum
+     return distanceSquared <= radiiSumSquared;
+ }
+
```

Sample-Test1/test.cpp

```
+ TEST(SphereSphereCollision, NoIntersectionCentreAtOrigin) {
+     Sphere sphereA(Vector3(0, 0, 0), 1);
+     Sphere sphereB(Vector3(5, 0, 0), 1);
+     EXPECT_FALSE(sphereA.CollideWith(sphereB)) << "Spheres at (0,0,0) and (5,0,0) with radius 1 should not collide.";
+ }
+
+ TEST(SphereSphereCollision, NoIntersectionOffsetCentre) {
+     Sphere sphereA(Vector3(3, 3, 3), 2);
+     Sphere sphereB(Vector3(10, 10, 10), 2);
+     EXPECT_FALSE(sphereA.CollideWith(sphereB)) << "Spheres at (3,3,3) and (10,10,10) with radius 2 should not collide.";
+ }
+
+ TEST(SphereSphereCollision, OverlappingCentreAtOrigin) {
+     Sphere sphereA(Vector3(0, 0, 0), 2);
+     Sphere sphereB(Vector3(2, 0, 0), 2);
+     EXPECT_TRUE(sphereA.CollideWith(sphereB)) << "Spheres at (0,0,0) and (2,0,0) with radius 2 should overlap.";
+ }
+
+ TEST(SphereSphereCollision, OverlappingOffsetCentre) {
+     Sphere sphereA(Vector3(5, 5, 5), 3);
+     Sphere sphereB(Vector3(8, 5, 5), 3);
+     EXPECT_TRUE(sphereA.CollideWith(sphereB)) << "Spheres at (5,5,5) and (8,5,5) with radius 3 should overlap.";
+ }
+
+ TEST(SphereSphereCollision, FullyContainedCentreAtOrigin) {
+     Sphere sphereA(Vector3(0, 0, 0), 5);
+     Sphere sphereB(Vector3(1, 0, 0), 1);
+     EXPECT_TRUE(sphereA.CollideWith(sphereB)) << "A larger sphere fully containing a smaller one should collide.";
+ }
```

Microsoft Visual Studio Debu

```
Running main() from D:\a\_work\1\s\googletest\googletest\src\gtest_main.cc
[==========] Running 6 tests from 1 test case.
[----------] Global test environment set-up.
[----------] 6 tests from SphereSphereCollision
[ RUN      ] SphereSphereCollision.NoIntersectionCentreAtOrigin
[       OK ] SphereSphereCollision.NoIntersectionCentreAtOrigin (0 ms)
[ RUN      ] SphereSphereCollision.NoIntersectionOffsetCentre
[       OK ] SphereSphereCollision.NoIntersectionOffsetCentre (0 ms)
[ RUN      ] SphereSphereCollision.OverlappingCentreAtOrigin
[       OK ] SphereSphereCollision.OverlappingCentreAtOrigin (0 ms)
[ RUN      ] SphereSphereCollision.OverlappingOffsetCentre
[       OK ] SphereSphereCollision.OverlappingOffsetCentre (0 ms)
[ RUN      ] SphereSphereCollision.FullyContainedCentreAtOrigin
[       OK ] SphereSphereCollision.FullyContainedCentreAtOrigin (0 ms)
[ RUN      ] SphereSphereCollision.FullyContainedOffsetCentre
[       OK ] SphereSphereCollision.FullyContainedOffsetCentre (0 ms)
[----------] 6 tests from SphereSphereCollision (1 ms total)

[----------] Global test environment tear-down
[==========] 6 tests from 1 test case ran. (3 ms total)
[  PASSED  ] 6 tests.

C:\Users\shruti\source\repos\Simulation Lab 1\x64\Debug\Sample-Test1.exe (process 16436) exited with code 0 (0x0).
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the conso
le when debugging stops.
Press any key to close this window . . .
```

Refraction: Spheres are only intersecting when distance ^2 <= (Radius + Radius)^2

- Squared distance b/w sphere center
- And then compare it with sum of (radius)^2

Q2 Closest Distance from a Point to a Line (Summative)

An infinite line can be expressed as a set of points using the parametric equation $\mathbf{P}_L$ + m$\mathbf{D}_L$ for all values of m, where $\mathbf{P}_L$ is a point on the line, and $\mathbf{D}_L$ is the direction of the line. This is typically a unit vector - although in the demo below $\mathbf{P}_L$ is not unit. The diagram below shows an infinite red line, which is defined by the green point on the line ($\mathbf{P}_L$) and the purple direction of the line ($\mathbf{D}_L$). The orange dot represents a point on the line for a specific value of m.

Solution:

Vector.h

```diff
@@ -0,0 +1,15 @@
+ #pragma once
+ class Vector3 {
+     public:
+         float x, y, z;
+
+         // Constructor
+         Vector3(float x = 0, float y = 0, float z = 0);
+
+         // Vector subtraction
+         Vector3 operator-(const Vector3& other) const;
+
+         // Calculate length squared
+         float LengthSquared() const;
+ };
+
```

Vector.cpp

```diff
@@ -0,0 +1,16 @@
+ #include "pch.h"
+ #include "Vector3.h"
+
+ // Constructor
+ Vector3::Vector3(float x, float y, float z) : x(x), y(y), z(z) {}
+
+ // Vector subtraction
+ Vector3 Vector3::operator-(const Vector3& other) const {
+     return Vector3(x - other.x, y - other.y, z - other.z);
+ }
+
+ // Calculate the squared length
+ float Vector3::LengthSquared() const {
+     return x * x + y * y + z * z;
+ }
+
```

Line.h

```cpp
#pragma once
#include "Vector3.h"

class Line {
public:
    Vector3 point;
    Vector3 direction;

    Line(const Vector3& point, const Vector3& direction); // Constructor


    float closestDistance(const Vector3& P_G) const; // Method
};


```

Line.cpp

```cpp
#include "pch.h"
#include "Line.h"

// Constructor
Line::Line(const Vector3& point, const Vector3& direction)
    : point(point), direction(direction.normalize()) {}

// Compute shortest distance
float Line::closestDistance(const Vector3& P_G) const {
    Vector3 V = P_G - point; // Vector from P_L to P_G
    Vector3 projection = direction * (V.dot(direction)); // Projection of V onto D_L
    Vector3 P_A = V - projection; // Perpendicular vector
    return P_A.length(); // Distance is magnitude of perpendicular vector
}


```

Test.cpp:

```cpp
TEST(PointLineDistance, ClosestPoint) {
    Line line(Vector3(0, 0, 0), Vector3(1, 1, 1));
    Vector3 point(2, 3, 4);

    float expectedDistance = std::sqrt(2);
    float actualDistance = line.closestDistance(point);

    std::cout << "Test: ClosestPoint\n";
    std::cout << "Expected: " << expectedDistance << ", Actual: " << actualDistance << "\n";

    EXPECT_NEAR(line.closestDistance(point), expectedDistance, 0.01);
}

TEST(PointLineDistance, PointOnLine) {
    Line line(Vector3(0, 0, 0), Vector3(1, 2, 3));
    Vector3 point(3, 6, 9);

    float expectedDistance = 0.0f;
    float actualDistance = line.closestDistance(point);

    std::cout << "Test: PointOnLine\n";
    std::cout << "Expected: " << expectedDistance << ", Actual: " << actualDistance << "\n";

    EXPECT_NEAR(line.closestDistance(point), 0.0f, 0.01);
}

TEST(PointLineDistance, VerticalCase) {
    Line line(Vector3(2, 0, 0), Vector3(0, 1, 0));
    Vector3 point(4, 5, 3);

    float expectedDistance = 3.61;
    float actualDistance = line.closestDistance(point);
    std::cout << "Test: VerticalCase\n";
    std::cout << "Expected: " << expectedDistance << ", Actual: " << actualDistance << "\n";
    EXPECT_NEAR(line.closestDistance(point), expectedDistance, 0.01);
}
```

Output:

```
Microsoft Visual Studio Debu
[==========] Running 3 tests from 1 test case.
[----------] Global test environment set-up.
[----------] 3 tests from PointPlaneDistance
[ RUN      ] PointPlaneDistance.PointAbovePlane
Test: PointAbovePlane
Expected: 5, Actual: 5
[       OK ] PointPlaneDistance.PointAbovePlane (0 ms)
[ RUN      ] PointPlaneDistance.PointBelowPlane
Test: PointBelowPlane
Expected: 4, Actual: 4
[       OK ] PointPlaneDistance.PointBelowPlane (1 ms)
[ RUN      ] PointPlaneDistance.PointOnPlane
Test: PointOnPlane
Expected: 0, Actual: 0
[       OK ] PointPlaneDistance.PointOnPlane (0 ms)
[----------] 3 tests from PointPlaneDistance (2 ms total)

[----------] Global test environment tear-down
[==========] 3 tests from 1 test case ran. (4 ms total)
[  PASSED  ] 3 tests.

C:\Users\shruti\source\repos\Simulation Lab 1\x64\Debug\Sample-Test1.exe (process 24968) exited with code 0 (0x0).
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the conso
le when debugging stops.
Press any key to close this window . . .
```

Refraction:

- Shortest distance from point p to a line is defined by two points A & B.
- Then convert line into a directional vector D = B – A
- AP = P – A to line direction D to closest point Q
- Now Euclidean distance b/w P & Q for shortest distance

Q3 Sphere Line Intersection Test (Summative)

Now that you have the closest distance to a point on a line you can create a Sphere Line intersection test. All you need to do is to find the shortest distance from the point at the centre of the sphere to the line, and then compare that distance to the radius of the sphere.

Sphere.h:

Simulation Lab 1/Sphere.h

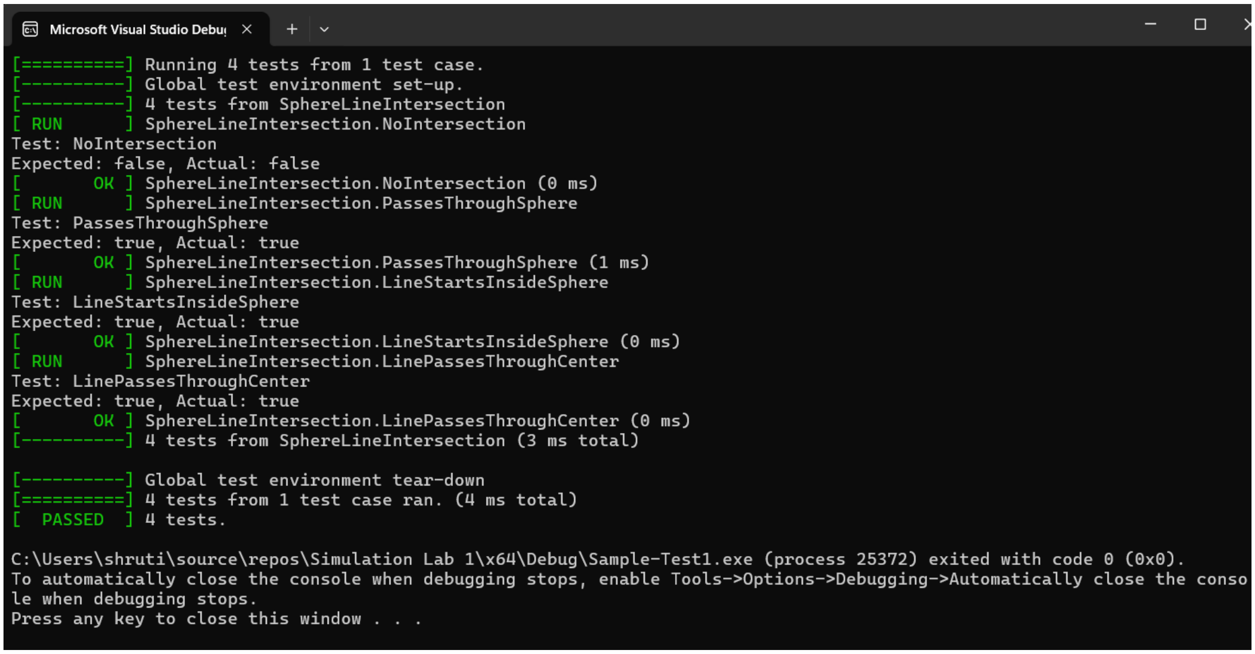
```diff
@@ -1,5 +1,6 @@
 #pragma once
 #include "Vector3.h"
+#include "Line.h"

 class Sphere {
 public:
@@ -10,7 +11,10 @@ class Sphere {
     Sphere(const Vector3& center, float radius);

     // Method to check collision with another sphere
-    bool CollideWith(const Sphere& other) const;
+    //bool CollideWith(const Sphere& other) const;
+
+        //to check if a line intersects with the sphere
+    bool intersects(const Line& line) const;
 };

```

Sphere.cpp:

Simulation Lab 1/Sphere.cpp

```diff
- // Check collision with another sphere
- bool Sphere::CollideWith(const Sphere& other) const{
-     // Calculate the squared distance between centers
-     //float distanceSquared = (center - other.center).LengthSquared();
-
-     // Calculate the squared sum of radii
-     float distanceSquared = (other.center - center).lengthSquared();
-     float radiusSum = radius + other.radius;
-
-     // Check if the squared distance is less than or equal to squared ra
-     //return distanceSquared <= radiiSumSquared;
-     return distanceSquared <= (radiusSum * radiusSum);
+ bool Sphere::intersects(const Line& line) const
+ {
+     float distance = line.closestDistance(center);
+     return distance <= radius;;
```

```cpp
TEST(SphereLineIntersection, NoIntersection) {
    Line line(Vector3(5, 5, 5), Vector3(1, 0, 0));
    Sphere sphere(Vector3(0, 0, 0), 3);
    bool expected = false;
    bool actual = sphere.intersects(line);

    std::cout << "Test: NoIntersection\n";
    std::cout << "Expected: " << (expected ? "true" : "false")
        << ", Actual: " << (actual ? "true" : "false") << "\n";
    EXPECT_FALSE(sphere.intersects(line));
}

TEST(SphereLineIntersection, PassesThroughSphere) {
    Line line(Vector3(10, 0, 0), Vector3(-1, 0, 0));
    Sphere sphere(Vector3(10, 0, 0), 5);
    bool expected = true;
    bool actual = sphere.intersects(line);

    std::cout << "Test: PassesThroughSphere\n";
    std::cout << "Expected: " << (expected ? "true" : "false")
        << ", Actual: " << (actual ? "true" : "false") << "\n";
    EXPECT_TRUE(sphere.intersects(line));
}

TEST(SphereLineIntersection, LineStartsInsideSphere) {
    Line line(Vector3(3, 2, 2), Vector3(1, 0, 0));
    Sphere sphere(Vector3(2, 2, 2), 5);
    bool expected = true;
    bool actual = sphere.intersects(line);

    std::cout << "Test: LineStartsInsideSphere\n";
    std::cout << "Expected: " << (expected ? "true" : "false")
        << ", Actual: " << (actual ? "true" : "false") << "\n";
    EXPECT_TRUE(sphere.intersects(line));
}

TEST(SphereLineIntersection, LinePassesThroughCenter) {
    Line line(Vector3(-5, 0, 0), Vector3(1, 0, 0));
    Sphere sphere(Vector3(0, 0, 0), 3);
    bool expected = true;
    bool actual = sphere.intersects(line);

    std::cout << "Test: LinePassesThroughCenter\n";
```



```
Microsoft Visual Studio Debu
[==========] Running 4 tests from 1 test case.
[----------] Global test environment set-up.
[----------] 4 tests from SphereLineIntersection
[ RUN      ] SphereLineIntersection.NoIntersection
Test: NoIntersection
Expected: false, Actual: false
[       OK ] SphereLineIntersection.NoIntersection (0 ms)
[ RUN      ] SphereLineIntersection.PassesThroughSphere
Test: PassesThroughSphere
Expected: true, Actual: true
[       OK ] SphereLineIntersection.PassesThroughSphere (1 ms)
[ RUN      ] SphereLineIntersection.LineStartsInsideSphere
Test: LineStartsInsideSphere
Expected: true, Actual: true
[       OK ] SphereLineIntersection.LineStartsInsideSphere (0 ms)
[ RUN      ] SphereLineIntersection.LinePassesThroughCenter
Test: LinePassesThroughCenter
Expected: true, Actual: true
[       OK ] SphereLineIntersection.LinePassesThroughCenter (0 ms)
[----------] 4 tests from SphereLineIntersection (3 ms total)

[----------] Global test environment tear-down
[==========] 4 tests from 1 test case ran. (4 ms total)
[  PASSED  ] 4 tests.

C:\Users\shruti\source\repos\Simulation Lab 1\x64\Debug\Sample-Test1.exe (process 25372) exited with code 0 (0x0).
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the conso
le when debugging stops.
Press any key to close this window . . .
```

Refraction: the line intersect if shortest distance from sphere to line <= sphere's radius

- So in previous question we already calculate the shortest distance.
- Now we need to make a bool function which checks is the sphere intersects with line or not. By distance <= radius

Q4. Closest Distance from a Point to a Plane (Summative)

You can define a plane as a point on the plane and the normal vector; the unit vector that is perpendicular to the plane. It can also be useful to store two other vectors that are on the plane.

In the diagram (and interactive demo) below the Plane is being defined by three points in the Plane. The green sphere, and the points at the end of the blue and red vectors. The green vector is the cross product of the red and blue vectors, and so is perpendicular to the red and blue vectors, and the plane. Note that the green vector is ***not*** a unit length vector.

In the interactive demo you can move the orange sphere around on the plane by using the sliders to adjust the values of S and T. This represents the number of blue vectors (S) and red vectors (T) to get to that point from the green point on the plane - so the points on the plane could be defined as $\mathbf{P_P} + S\mathbf{P_B} + T\mathbf{P_R}$

```cpp
class Vector3 {
    public:
        float x, y, z;

        // Constructor
        Vector3();
        Vector3(float x, float y, float z);
        float lengthSquared() const;

        // Vector subtraction
        Vector3 operator-(const Vector3& other) const;
        Vector3 operator+(const Vector3& other) const;
        Vector3 operator*(float scalar) const;
        float dot(const Vector3& other) const;
        float length() const;
        Vector3 normalize() const;

        // Print
        void print() const;
};
```

```cpp
}

// Subtraction
Vector3 Vector3::operator-(const Vector3& other) const {
    return Vector3(x - other.x, y - other.y, z - other.z);
}

// Addition
Vector3 Vector3::operator+(const Vector3& other) const {
    return Vector3(x + other.x, y + other.y, z + other.z);
}

// Scalar multiplication
Vector3 Vector3::operator*(float scalar) const {
    return Vector3(x * scalar, y * scalar, z * scalar);
}

// Dot product
float Vector3::dot(const Vector3& other) const {
    return x * other.x + y * other.y + z * other.z;
}


// Length (magnitude)
float Vector3::length() const {
    return std::sqrt(x * x + y * y + z * z);
}

// Normalize
Vector3 Vector3::normalize() const {
    float len = length();
    return (len > 0) ? *this * (1.0f / len) : Vector3(0, 0, 0);
}

// Print
```

Simulation Lab 1/Plane.h

```diff
@@ -0,0 +1,14 @@
+ #pragma once
+ #include "pch.h"
+ #include "Vector3.h"
+
+ class Plane {
+ public:
+     Vector3 point;   // A point on the plane
+     Vector3 normal;  // Normal vector of the plane
+
+     Plane(const Vector3& point, const Vector3& normal);
+     float closestDistance(const Vector3& P_G) const;
+ };
+
+
```

Simulation Lab 1/Plane.cpp

```diff
@@ -0,0 +1,13 @@
+ #include "pch.h"
+ #include "Plane.h"
+ #include <cmath>
+ Plane::Plane(const Vector3& point, const Vector3& normal)
+     : point(point), normal(normal.normalize()) {}
+
+ // Compute shortest distance from a point to the plane
+ float Plane::closestDistance(const Vector3& P_G) const {
+     // Vector from plane point to general point
+     Vector3 V = P_G - point;
+     // Perpendicular distance using dot product
+     return std::fabs(V.dot(normal));
+ }
```

Sample-Test1 (Global Scope)

```cpp
//.......................................................................
TEST(PointPlaneDistance, PointAbovePlane) {
    Plane plane(Vector3(0, 0, 0), Vector3(0, 0, 1));
    Vector3 point(2, 3, 5);

    float expected = 5.0;
    float actual = plane.closestDistance(point);

    std::cout << "Test: PointAbovePlane\n";
    std::cout << "Expected: " << expected << ", Actual: " << actual << "\n";

    EXPECT_NEAR(actual, expected, 0.01);
}

TEST(PointPlaneDistance, PointBelowPlane) {
    Plane plane(Vector3(0, 0, 0), Vector3(0, 0, 1));
    Vector3 point(2, 3, -4);

    float expected = 4.0;
    float actual = plane.closestDistance(point);

    std::cout << "Test: PointBelowPlane\n";
    std::cout << "Expected: " << expected << ", Actual: " << actual << "\n";

    EXPECT_NEAR(actual, expected, 0.01);
}

TEST(PointPlaneDistance, PointOnPlane) {
    Plane plane(Vector3(1, 1, 1), Vector3(1, 1, 1).normalize());
    Vector3 point(0, 2, 1);

    float expected = 0.0;
    float actual = plane.closestDistance(point);

    std::cout << "Test: PointOnPlane\n";
    std::cout << "Expected: " << expected << ", Actual: " << actual << "\n";

    EXPECT_NEAR(actual, expected, 0.01);
}

int main(int argc, char** argv) {
    ::testing::InitGoogleTest(&argc, argv);
    return RUN_ALL_TESTS();
}
```

```
[==========] Running 3 tests from 1 test case.
[----------] Global test environment set-up.
[----------] 3 tests from PointPlaneDistance
[ RUN      ] PointPlaneDistance.PointAbovePlane
Test: PointAbovePlane
Expected: 5, Actual: 5
[       OK ] PointPlaneDistance.PointAbovePlane (1 ms)
[ RUN      ] PointPlaneDistance.PointBelowPlane
Test: PointBelowPlane
Expected: 4, Actual: 4
[       OK ] PointPlaneDistance.PointBelowPlane (0 ms)
[ RUN      ] PointPlaneDistance.PointOnPlane
Test: PointOnPlane
Expected: 0, Actual: 0
[       OK ] PointPlaneDistance.PointOnPlane (1 ms)
[----------] 3 tests from PointPlaneDistance (2 ms total)

[----------] Global test environment tear-down
[==========] 3 tests from 1 test case ran. (2 ms total)
[  PASSED  ] 3 tests.

C:\Users\shruti\source\repos\Simulation Lab 1\x64\Debug\Sample-Test1.exe (process 22232) exited with code 0 (0x0).
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the conso
le when debugging stops.
Press any key to close this window . . .
```

Refraction:

So the plain is define by

- Point P on the plan
- A normal vector N (which is perpendicular to the plan)
- Distance = (Pg - Pp).N/|N|
- Where, Pg = Genral point
- . = dot product
- |N| = magnitude (length)

Also i used std::fabs it computes the absolute value of a floating point number

Q5. Sphere to Plane Collision (Summative)

Again once you can find the closest point to a plane finding out if a sphere and a plane are intersecting is simple. You just need to find out if the closest distance from the centre of the sphere to the plane is less that the radius.

You are encouraged to do your own research to find the distance from a point to a plane. One way is to project the Vector from the point on the plane to the general point onto the normal vector of the plane.

Sphere.h:

```
//to check if a line intersects with the sphere
//bool intersects(const Line& line) const;
bool planeIntersects(const Plane& plane) const;
```

```
bool Sphere::planeIntersects(const Plane& plane) const
{
    float distance = plane.closestDistance(center);
    return distance <= radius;
}
```

```
TEST(SpherePlaneCollision, SphereAbovePlane_NoCollision) {
    Plane plane(Vector3(0, 0, 0), Vector3(0, 0, 1));
    Sphere sphere(Vector3(0, 0, 5), 3);

    bool expected = false;
    bool actual = sphere.planeIntersects(plane);

    std::cout << "Test: SphereAbovePlane_NoCollision\n";
    std::cout << "Expected: " << (expected ? "true" : "false")
        << ", Actual: " << (actual ? "true" : "false") << "\n";

    EXPECT_EQ(actual, expected);
}

TEST(SpherePlaneCollision, SphereTouchesPlane) {
    Plane plane(Vector3(0, 0, 0), Vector3(0, 0, 1));
    Sphere sphere(Vector3(0, 0, 3), 3);

    bool expected = true;
    bool actual = sphere.planeIntersects(plane);

    std::cout << "Test: SphereTouchesPlane\n";
    std::cout << "Expected: " << (expected ? "true" : "false")
        << ", Actual: " << (actual ? "true" : "false") << "\n";

    EXPECT_EQ(actual, expected);
}

TEST(SpherePlaneCollision, SphereIntersectsPlane) {
    Plane plane(Vector3(0, 0, 0), Vector3(0, 0, 1));
    Sphere sphere(Vector3(0, 0, 2), 3);

    bool expected = true;
    bool actual = sphere.planeIntersects(plane);

    std::cout << "Test: SphereIntersectsPlane\n";
    std::cout << "Expected: " << (expected ? "true" : "false")
        << ", Actual: " << (actual ? "true" : "false") << "\n";

    EXPECT_EQ(actual, expected);
}
```

Refraction:

To define sphere – center, radius

For plane – point, normal vector

For the sphere to intersect with plane -

Shortest distance from the sphere center to plan is less than radius