

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 than 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.

```
... @@ -0,0 +1,15 @@
```

```
1 + #pragma once
2 + class Vector3 {
3 +     public:
4 +         float x, y, z;
5 +
6 +         // Constructor
7 +         Vector3(float x = 0, float y = 0, float z = 0);
8 +
9 +         // Vector subtraction
10 +        Vector3 operator-(const Vector3& other) const;
11 +
12 +        // Calculate length squared
13 +        float LengthSquared() const;
14 + };
15 +
```

```
... @@ -0,0 +1,16 @@
```

```
1 + #include "pch.h"
2 + #include "Vector3.h"
3 +
4 + // Constructor
5 + Vector3::Vector3(float x, float y, float z) : x(x), y(y), z(z) {}
6 +
7 + // Vector subtraction
8 + Vector3 Vector3::operator-(const Vector3& other) const {
9 +     return Vector3(x - other.x, y - other.y, z - other.z);
10 + }
11 +
12 + // Calculate the squared length
13 + float Vector3::LengthSquared() const {
14 +     return x * x + y * y + z * z;
15 + }
16 +
```

Simulation Lab 1/Sphere.h

... @@ -0,0 +1,17 @@

```
1 + #pragma once
2 + #include "Vector3.h"
3 +
4 + class Sphere {
5 + public:
6 +     Vector3 center;
7 +     float radius;
8 +
9 +     // Constructor
10 +     Sphere(const Vector3& center, float radius);
11 +
12 +     // Method to check collision with another sphere
13 +     bool CollideWith(const Sphere& other) const;
14 + };
15 +
16 +
17 +
```

```

... @@ -0,0 +1,19 @@
1 + #include "pch.h"
2 + #include "Sphere.h"
3 +
4 + // Constructor
5 + Sphere::Sphere(const Vector3& center, float radius) : center(center), radius(radius)
6 +
7 + // Check collision with another sphere
8 + bool Sphere::CollideWith(const Sphere& other) const {
9 +     // Calculate the squared distance between centers
10 +     float distanceSquared = (center - other.center).LengthSquared();
11 +
12 +     // Calculate the squared sum of radii
13 +     float radiiSum = radius + other.radius;
14 +     float radiiSumSquared = radiiSum * radiiSum;
15 +
16 +     // Check if the squared distance is less than or equal to squared radii sum
17 +     return distanceSquared <= radiiSumSquared;
18 + }
19 +

```

```

Sample-Test1/test.cpp

5 + TEST(SphereSphereCollision, NoIntersectionCentreAtOrigin) {
6 +     Sphere sphereA(Vector3(0, 0, 0), 1);
7 +     Sphere sphereB(Vector3(5, 0, 0), 1);
8 +     EXPECT_FALSE(sphereA.CollideWith(sphereB)) << "Spheres at (0,0,0) and (5,0,0) with radius 1 should not collide.";
9 + }
10 +
11 + TEST(SphereSphereCollision, NoIntersectionOffsetCentre) {
12 +     Sphere sphereA(Vector3(3, 3, 3), 2);
13 +     Sphere sphereB(Vector3(10, 10, 10), 2);
14 +     EXPECT_FALSE(sphereA.CollideWith(sphereB)) << "Spheres at (3,3,3) and (10,10,10) with radius 2 should not collide.";
15 + }
16 +
17 + TEST(SphereSphereCollision, OverlappingCentreAtOrigin) {
18 +     Sphere sphereA(Vector3(0, 0, 0), 2);
19 +     Sphere sphereB(Vector3(2, 0, 0), 2);
20 +     EXPECT_TRUE(sphereA.CollideWith(sphereB)) << "Spheres at (0,0,0) and (2,0,0) with radius 2 should overlap.";
21 + }
22 +
23 + TEST(SphereSphereCollision, OverlappingOffsetCentre) {
24 +     Sphere sphereA(Vector3(5, 5, 5), 3);
25 +     Sphere sphereB(Vector3(8, 5, 5), 3);
26 +     EXPECT_TRUE(sphereA.CollideWith(sphereB)) << "Spheres at (5,5,5) and (8,5,5) with radius 3 should overlap.";
27 + }
28 +
29 + TEST(SphereSphereCollision, FullyContainedCentreAtOrigin) {
30 +     Sphere sphereA(Vector3(0, 0, 0), 5);
31 +     Sphere sphereB(Vector3(1, 0, 0), 1);
32 +     EXPECT_TRUE(sphereA.CollideWith(sphereB)) << "A larger sphere fully containing a smaller one should collide.";
33 + }

```

```

Microsoft Visual Studio Debug
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\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 console when debugging stops.
Press any key to close this window . . .

```

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 (P_L) and the purple direction of the line (D_L). The orange dot represents a point on the line for a specific value of m .

Solution:

Vector.h

```
... @@ -0,0 +1,15 @@
1 + #pragma once
2 + class Vector3 {
3 +     public:
4 +         float x, y, z;
5 +
6 +         // Constructor
7 +         Vector3(float x = 0, float y = 0, float z = 0);
8 +
9 +         // Vector subtraction
10 +         Vector3 operator-(const Vector3& other) const;
11 +
12 +         // Calculate length squared
13 +         float LengthSquared() const;
14 + };
15 +
```

Vector.cpp

```
... @@ -0,0 +1,16 @@
1 + #include "pch.h"
2 + #include "Vector3.h"
3 +
4 + // Constructor
5 + Vector3::Vector3(float x, float y, float z) : x(x), y(y), z(z) {}
6 +
7 + // Vector subtraction
8 + Vector3 Vector3::operator-(const Vector3& other) const {
9 +     return Vector3(x - other.x, y - other.y, z - other.z);
10 + }
11 +
12 + // Calculate the squared length
13 + float Vector3::LengthSquared() const {
14 +     return x * x + y * y + z * z;
15 + }
16 +
```

Line.h

```
1 #pragma once
2 #include "Vector3.h"
3
4 class Line {
5 public:
6     Vector3 point;
7     Vector3 direction;
8
9     Line(const Vector3& point, const Vector3& direction); // Constructor
10
11
12     float closestDistance(const Vector3& P_G) const; // Method
13 };
14
15
```

Line.cpp

```
1  #include "pch.h"
2  #include "Line.h"
3
4  // Constructor
5  Line::Line(const Vector3& point, const Vector3& direction)
6  {
7      : point(point), direction(direction.normalize()) {}
8
9  // Compute shortest distance
10 float Line::closestDistance(const Vector3& P_G) const {
11     Vector3 V = P_G - point; // Vector from P_L to P_G
12     Vector3 projection = direction * (V.dot(direction)); // Projection of V onto D_L
13     Vector3 P_A = V - projection; // Perpendicular vector
14     return P_A.length(); // Distance is magnitude of perpendicular vector
15 }
16
```

Test.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 Debug
[=====] 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 console when debugging stops.
Press any key to close this window . . .
```

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
... @@ -1,5 +1,6 @@
1 1 #pragma once
2 2 #include "Vector3.h"
3 + #include "Line.h"
3 4
4 5 class Sphere {
5 6 public:
@@ -10,7 +11,10 @@ class Sphere {
10 11     Sphere(const Vector3& center, float radius);
11 12
12 13     // Method to check collision with another sphere
13 - bool CollideWith(const Sphere& other) const;
14 + //bool CollideWith(const Sphere& other) const;
15 +
16 +     //to check if a line intersects with the sphere
17 + bool intersects(const Line& line) const;
14 18 };
15 19
16 20
```

Sphere.cpp:

```
Simulation Lab 1/Sphere.cpp
6 6
7 - // Check collision with another sphere
8 - bool Sphere::CollideWith(const Sphere& other) const{
9 -     // Calculate the squared distance between centers
10 -     //float distanceSquared = (center - other.center).LengthSquared();
11 -
12 -     // Calculate the squared sum of radii
13 -     float distanceSquared = (other.center - center).lengthSquared();
14 -     float radiusSum = radius + other.radius;
15 -
16 -     // Check if the squared distance is less than or equal to squared ra
17 -     //return distanceSquared <= radiiSumSquared;
18 -     return distanceSquared <= (radiusSum * radiusSum);
7 + bool Sphere::intersects(const Line& line) const
8 + {
9 +     float distance = line.closestDistance(center);
10 +     return distance <= radius;;
```

```

82  TEST(SphereLineIntersection, NoIntersection) {
83      Line line(Vector3(5, 5, 5), Vector3(1, 0, 0));
84      Sphere sphere(Vector3(0, 0, 0), 3);
85      bool expected = false;
86      bool actual = sphere.intersects(line);
87
88      std::cout << "Test: NoIntersection\n";
89      std::cout << "Expected: " << (expected ? "true" : "false")
90          << ", Actual: " << (actual ? "true" : "false") << "\n";
91      EXPECT_FALSE(sphere.intersects(line));
92  }
93
94  TEST(SphereLineIntersection, PassesThroughSphere) {
95      Line line(Vector3(10, 0, 0), Vector3(-1, 0, 0));
96      Sphere sphere(Vector3(10, 0, 0), 5);
97      bool expected = true;
98      bool actual = sphere.intersects(line);
99
100     std::cout << "Test: PassesThroughSphere\n";
101     std::cout << "Expected: " << (expected ? "true" : "false")
102         << ", Actual: " << (actual ? "true" : "false") << "\n";
103     EXPECT_TRUE(sphere.intersects(line));
104  }
105
106  TEST(SphereLineIntersection, LineStartsInsideSphere) {
107      Line line(Vector3(3, 2, 2), Vector3(1, 0, 0));
108      Sphere sphere(Vector3(2, 2, 2), 5);
109      bool expected = true;
110      bool actual = sphere.intersects(line);
111
112      std::cout << "Test: LineStartsInsideSphere\n";
113      std::cout << "Expected: " << (expected ? "true" : "false")
114          << ", Actual: " << (actual ? "true" : "false") << "\n";
115      EXPECT_TRUE(sphere.intersects(line));
116  }
117
118  TEST(SphereLineIntersection, LinePassesThroughCenter) {
119      Line line(Vector3(-5, 0, 0), Vector3(1, 0, 0));
120      Sphere sphere(Vector3(0, 0, 0), 3);
121      bool expected = true;
122      bool actual = sphere.intersects(line);
123
124      std::cout << "Test: LinePassesThroughCenter\n";

```

```
Microsoft Visual Studio Debug Console
[=====] 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\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 console when debugging stops.
Press any key to close this window . . .
```

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 $\mathbf{P}_p + S\mathbf{P}_b + T\mathbf{P}_r$

```
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;  
};
```

```

// 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(x: 0, y: 0, z: 0);
}

// Print

```

Simulation Lab 1/Plane.h

```
...    @@ -0,0 +1,14 @@
1  + #pragma once
2  + #include "pch.h"
3  + #include "Vector3.h"
4  +
5  + class Plane {
6  + public:
7  +     Vector3 point;    // A point on the plane
8  +     Vector3 normal;  // Normal vector of the plane
9  +
10 +     Plane(const Vector3& point, const Vector3& normal);
11 +     float closestDistance(const Vector3& P_G) const;
12 + };
13 +
14 +
```

Simulation Lab 1/Plane.cpp

```
...    @@ -0,0 +1,13 @@
1  + #include "pch.h"
2  + #include "Plane.h"
3  + #include <cmath>
4  + Plane::Plane(const Vector3& point, const Vector3& normal)
5  +     : point(point), normal(normal.normalize()) {}
6  +
7  + // Compute shortest distance from a point to the plane
8  + float Plane::closestDistance(const Vector3& P_G) const {
9  +     // Vector from plane point to general point
10 +     Vector3 V = P_G - point;
11 +     // Perpendicular distance using dot product
12 +     return std::fabs(V.dot(normal));
13 + }
```



```
Sample-Test1 (Global Scope)
133 //.....
134 TEST(PointPlaneDistance, PointAbovePlane) {
135     Plane plane(Vector3(0, 0, 0), Vector3(0, 0, 1));
136     Vector3 point(2, 3, 5);
137
138     float expected = 5.0;
139     float actual = plane.closestDistance(point);
140
141     std::cout << "Test: PointAbovePlane\n";
142     std::cout << "Expected: " << expected << ", Actual: " << actual << "\n";
143
144     EXPECT_NEAR(actual, expected, 0.01);
145 }
146
147 TEST(PointPlaneDistance, PointBelowPlane) {
148     Plane plane(Vector3(0, 0, 0), Vector3(0, 0, 1));
149     Vector3 point(2, 3, -4);
150
151     float expected = 4.0;
152     float actual = plane.closestDistance(point);
153
154     std::cout << "Test: PointBelowPlane\n";
155     std::cout << "Expected: " << expected << ", Actual: " << actual << "\n";
156
157     EXPECT_NEAR(actual, expected, 0.01);
158 }
159
160 TEST(PointPlaneDistance, PointOnPlane) {
161     Plane plane(Vector3(1, 1, 1), Vector3(1, 1, 1).normalize());
162     Vector3 point(0, 2, 1);
163
164     float expected = 0.0;
165     float actual = plane.closestDistance(point);
166
167     std::cout << "Test: PointOnPlane\n";
168     std::cout << "Expected: " << expected << ", Actual: " << actual << "\n";
169
170     EXPECT_NEAR(actual, expected, 0.01);
171 }
172
173 int main(int argc, char** argv) {
174     ::testing::InitGoogleTest(&argc, argv);
175     return RUN_ALL_TESTS();
176 }
```

```
Microsoft Visual Studio Debug: x + v
[=====] 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\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 console when debugging stops.
Press any key to close this window . . .|
```

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 than 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;
}
```

```

177
178     TEST(SpherePlaneCollision, SphereAbovePlane_NoCollision) {
179         Plane plane(Vector3(0, 0, 0), Vector3(0, 0, 1));
180         Sphere sphere(Vector3(0, 0, 5), 3);
181
182         bool expected = false;
183         bool actual = sphere.planeIntersects(plane);
184
185         std::cout << "Test: SphereAbovePlane_NoCollision\n";
186         std::cout << "Expected: " << (expected ? "true" : "false")
187             << ", Actual: " << (actual ? "true" : "false") << "\n";
188
189         EXPECT_EQ(actual, expected);
190     }
191
192     TEST(SpherePlaneCollision, SphereTouchesPlane) {
193         Plane plane(Vector3(0, 0, 0), Vector3(0, 0, 1));
194         Sphere sphere(Vector3(0, 0, 3), 3);
195
196         bool expected = true;
197         bool actual = sphere.planeIntersects(plane);
198
199         std::cout << "Test: SphereTouchesPlane\n";
200         std::cout << "Expected: " << (expected ? "true" : "false")
201             << ", Actual: " << (actual ? "true" : "false") << "\n";
202
203         EXPECT_EQ(actual, expected);
204     }
205
206     TEST(SpherePlaneCollision, SphereIntersectsPlane) {
207         Plane plane(Vector3(0, 0, 0), Vector3(0, 0, 1));
208         Sphere sphere(Vector3(0, 0, 2), 3);
209
210         bool expected = true;
211         bool actual = sphere.planeIntersects(plane);
212
213         std::cout << "Test: SphereIntersectsPlane\n";
214         std::cout << "Expected: " << (expected ? "true" : "false")
215             << ", Actual: " << (actual ? "true" : "false") << "\n";
216
217         EXPECT_EQ(actual, expected);
218     }
219

```