

ACCB5017 CW2 - 2112389

by Joshua Piecko

Submission date: 08-Jan-2023 09:57PM (UTC+0000)

Submission ID: 194007563

File name: 33382_Joshua_Piecko_ACCB5017_CW2_-_2112389_473919_2063092461.docx (122.53K)

Word count: 8077

Character count: 42541

University of Wales Trinity Saint David University

[ACCB5017] Games Architecture & Mechanics

[Assignment 2]

Joshua Piecko

Student Number: 2112389

8th January 2023

Table of Contents

[Title Page](#) p. 1

Table of Contents

1. [Introduction](#) p. 3
 2. [Distance Between Two Points](#) pp. (4-5).
 - 2.1 [Point To Point Distance](#) p. 4
 - 2.2 [Point to Point Distance Squared](#) p. 4
 3. [A Plane](#) pp. (6-11).
 - 3.1 [The Plane Class](#) p. 6
 - 3.2 [Normalizing Plane Normal](#) p. 7
 - 3.3 [Point to Plane Distance](#) pp. (8-9).
 - 3.4 [Point Position Relative to Plane](#) p. 10
 - 3.5 [Calculating Planes Normal](#) p. 11
 4. [Sphere To Sphere Collision Detection](#) pp. (12-13).
 - 4.1 [The Sphere Class](#) p. 12
 - 4.2 [Collision Detection](#) p. 13
 5. [Point to Plane and Sphere to Plane Collision Detection](#) pp. (14-15).
 - 5.1 [Point to Plane Collisions](#) p. 14
 - 5.2 [Sphere to Plane Collisions](#) p. 15
 6. [Conclusion](#) p. 16
- [References](#) p. 17

1. Introduction

This report presents the findings of a programming assignment completed as part of the Computing (Games Development) course at UWTSD. The assignment focused on the development of methods and classes for vector operations, plane geometry, and collision detection in the context of game development. Four tasks were completed in total, involving the implementation of: (1) a method to calculate the distance between two points (pToPDistance) and the distance squared between two points (pToPDistanceSqrd), (2) a Plane class with methods for normalizing the plane normal vector, calculating the distance of a point from the plane, and determining whether a point is on, in front of, or behind the plane, (3) a Sphere class and a collision detection method for detecting collisions between two sphere bounding volumes, and (4) methods to test for collisions between a point and a plane (pointToPlane) and between a sphere and a plane (sphereToPlane).

Through this report, the aim was to demonstrate an understanding of the technical principles necessary for game development, as well as the ability to utilize appropriate languages, tools, and techniques. The results of the relevant implementations and testing are presented in detail in the following sections.

2. Distance Between Two Points

2.1 Point to Point Distance

2.1.1 Method

The method pToPDistance calculates the point-to-point distance between a given vector v1 and the current vector object. This distance is calculated using the formula: $\sqrt{(x1 - x2)^2 + (y1 - y2)^2 + (z1 - z2)^2}$ [1], where (x1, y1, z1) and (x2, y2, z2) are the components of the two vectors [1].

First, the differences between the x, y, and z components of the two vectors are calculated. Then, the point-to-point distance is calculated using the formula and stored in the variable 'dist'. The distance is calculated by taking the square root of the sum of the squared differences of the x, y, and z components. Finally, the method returns the point-to-point distance to the caller by returning the value of the 'dist' variable.

2.1.2 Implementation

```
/*
** Calculates the point-to-point distance between this vector and the given vector.
** The point-to-point distance is the straight-line distance between the two points
** represented by the vectors.
** It can be calculated using the formula:  $\sqrt{(x1 - x2)^2 + (y1 - y2)^2 + (z1 - z2)^2}$ , where (x1, y1, z1) and (x2, y2, z2) are the components of the two vectors.
*/
float Vector::pToPDistance(Vector& v1)
{
    // Calculate the difference between the x, y, and z components of the two
    // vectors.
    float diffX = (v1.x - x);
    float diffY = (v1.y - y);
    float diffZ = (v1.z - z);
    // Calculate the point-to-point distance using the formula:  $\sqrt{(x1 - x2)^2 + (y1 - y2)^2 + (z1 - z2)^2}$ .
    float dist;
    dist = std::sqrt(std::pow(diffX, 2) + std::pow(diffY, 2) + std::pow(diffZ,
2));
    return dist; // Returns calculated distance
}
```

2.1.3 Results

Test Name	Test Number	Expected Result	Actual Result	Pass/Fail
Vector pToPDistance Method	1	1.732051	1.732051	Pass
Vector pToPDistance Method	2	3.316625	3.316625	Pass

[Table 1: pToPDistance Method Results]

2.2 Point to Point Distance Squared

2.2.1 Method

The method `pToPDistanceSqr` calculates the squared point-to-point distance between a given vector `v1` and the current vector object using the following formula: $(x1 - x2)^2 + (y1 - y2)^2 + (z1 - z2)^2$ [1].

This formula calculates the straight-line distance between the two points represented by the vectors. It does this by finding the difference between the x, y, and z components of the two vectors and then squaring those differences [1]. The squared differences are then added together to give the squared point-to-point distance [1].

2.2.2 Implementation

```
/*
** Calculates the squared point-to-point distance between this vector and the given
vector.
** The squared point-to-point distance is the square of the straight-line distance
between the two points represented by the vectors.
** It can be calculated using the formula: (x1 - x2)^2 + (y1 - y2)^2 + (z1 - z2)^2,
where (x1, y1, z1) and (x2, y2, z2) are the components of the two vectors.
*/
float Vector::pToPDistanceSqr(Vector& v1)
{
    // Calculate the difference between the x, y, and z components of the two
vectors.
    float diffX = (v1.x - x);
    float diffY = (v1.y - y);
    float diffZ = (v1.z - z);
    // Calculate the squared point-to-point distance using the formula: (x1 -
x2)^2 + (y1 - y2)^2 + (z1 - z2)^2.
    float dist;
    dist = (std::pow(diffX, 2) + std::pow(diffY, 2) + std::pow(diffZ, 2));
    return dist; // Returns distance calculated
}
```

2.2.3 Results

Test Name	Test Number	Expected Result	Actual Result	Pass/Fail
Vector <code>pToPDistanceSqr</code> Method	1	3	3	Pass
Vector <code>pToPDistanceSqr</code> Method	2	11	11	Pass

[Table 2: `pToPDistanceSqr` Method Results]

3. A Plane

3.1 The Plane Class

3.1.1 Implementation

```

#ifndef PLANE_H
#define PLANE_H
#include "Vector_Functions.h"
#include "Sphere.h"

class Plane {

public:
    // Constructors
    Plane(Vector myNormal, Vector myP0); // Constructor that initializes the plane with a
    normal and point on the plane
    Plane(void); // Default constructor
    // Member Functions
    void calculateNormal(Vector v1, Vector v2); // Calculates the normal of the plane
    based on two points on the plane
    void normalizeNormal(); // Normalizes the normal of the plane (makes its length equal
    to 1)
    float pointToPlaneDistance(Vector point); // Calculates the distance from a given
    point to the plane
    std::string getPointPosition(Vector point); // Returns a string indicating whether
    the given point is above, below, or on the plane
    bool CollidesWithSphere(Sphere& s1); // Returns true if the plane collides with a
    given sphere, otherwise false
    bool CollidesWithPoint(Vector& point); // Returns true if the plane collides with a
    given point, otherwise false

    Vector getNormal(void) const { return normal; }; // Returns the normal of the plane.
    void setp0(Vector v1) { p0 = v1; } // Sets point on the plane.
    Vector getp0() const { return p0; }; // Returns the point on the plane.

private:
    Vector normal;
    Vector p0;
};

#endif // PLANE_H

```

3.2 Normalizing Plane Normal

3.2.1 Method

Normalization is a common operation in vector mathematics, and it refers to the process of scaling a vector to have a magnitude (length) of 1 [1]. This is often useful when working with vectors because it allows them to be compared more easily, since the magnitude of a normalized vector is always 1 [1].

To normalize a vector, we divide each of its components by its magnitude (length) [1]. The magnitude (length) of a vector can be calculated using the formula: $\sqrt{x^2 + y^2 + z^2}$ [1]. This formula calculates the square root of the sum of the squares of the vector's components, which gives us the vector's magnitude.

The normalizeNormal method in the Plane class follows this process to normalize the normal vector of the plane. It first calculates the magnitude of the normal vector using the formula, and then divides the x, y, and z components of the normal vector by this magnitude. This scales the normal vector to have a magnitude of 1, effectively normalizing it.

3.2.2 Implementation

```

/*
**Normalizes the normal of the plane.
**Normalization is the process of scaling a vector to have a magnitude (length) of 1.
**This is often useful when working with vectors because it allows them to be compared
more easily.
**The normal is normalized by dividing each of its x, y, and z components by its
magnitude (length).
** The magnitude (length) is calculated using the formula: sqrt(x^2 + y^2 + z^2).
*/
void Plane::normalizeNormal() {
    // Calculate the magnitude (length) of the normal using the formula: sqrt(x^2 + y^2 +
z^2).
    float length;
    length = std::sqrt(pow(normal.getX(), 2) + pow(normal.getY(), 2) + pow(normal.getZ(),
2));
    // Divide the x, y, and z components of the normal by its magnitude to normalize it.
    normal.setX(normal.getX() / length);
    normal.setY(normal.getY() / length);
    normal.setZ(normal.getZ() / length);
}

```

3.2.3 Results

Test Name	Test Number	Expected Result	Actual Result	Pass/Fail
Normalizing Plane Normal	1	(X: -0.408248 Y: 0.816496 Z: -0.408248)	(X: -0.408248 Y: 0.816497 Z: -0.408248)	Pass
Normalizing Plane Normal	2	(X: 0.742781 Y: 0.557086 Z: 0.371390)	(X: 0.742781 Y: 0.557086 Z: 0.371391)	Pass

[Table 3: Normalizing Plane Normal Results]

3.3 Point to Plane distance

3.3.1 Method

The pointToPlaneDistance method in the Plane class is used to calculate the distance from a given point to the plane. The distance from a point to a plane can be calculated using the following formula: $\text{distance} = (A * x + B * y + C * z - D) / \sqrt{A^2 + B^2 + C^2}$ [1] where (A, B, C) is the normal of the plane, D is the dot product between the normal and the p0 point of the plane, and (x, y, z) are the components of the given point [1].

This formula uses the normal of the plane to find the shortest distance from the point to the plane. The dot product between the normal and the p0 point of the plane, D, is a measure of how far the p0 point is from the origin. The formula then divides the result by the magnitude of the normal vector to normalize the distance, since the normal vector may not have a magnitude of 1.

The pointToPlaneDistance method in the Plane class follows this process to calculate the distance from the given point to the plane. It first extracts the x, y, and z components of the normal vector and calculates the dot product between the normal and the p0 point of the plane. It then uses these values, along with the components of the given point, in the formula to calculate the distance. Finally, it returns this distance as a float value.

3.3.2 Implementation

```
// Calculate the distance from the given point to the plane.
// Returns the distance as a float value.
float Plane::pointToPlaneDistance(Vector point) {
    float distance; // Used to store the distance.

    // Extract the x, y, and z components of the normal.
    float A = normal.getX();
    float B = normal.getY();
    float C = normal.getZ();

    // Calculate the dot product between the normal and the p0 point of the plane.
    float D = normal.DotProd(p0);

    /*
    ** Calculates the distance from the given point to the plane using the following
    formula:
    ** distance = (A * x + B * y + C * z - D) / sqrt(A^2 + B^2 + C^2)
    ** where (A, B, C) is the normal of the plane and D is the dot product between the
    normal and the p0 point of the plane.
    ** The x, y, and z values of the point are passed as arguments to the method and are
    used to calculate the distance from the point to the plane.
    */
    distance = (A * point.getX() + B * point.getY() + C * point.getZ() - D) / std::sqrt(A
* A + B * B + C * C);

    // Return the calculated distance.
    return distance;
}
```

3.3.3 Results

Test Name	Test Number	Expected Result	Actual Result	Pass/Fail
Point to Plane Distance	1	6	6	Pass
Point to Plane Distance	2	1.73205	1.732051	Pass
Point to Plane Distance	3	3	3	Pass

[Table 4: Point to Plane Distance Results]

3.4 Point Position Relative to Plane

3.4.1 Method

The `getPointPosition` method in the `Plane` class is used to determine the position of a given point relative to the plane. To determine the position of a point relative to a plane, we can use the distance between the point and the plane. The `getPointPosition` method in the `Plane` class follows this process to determine the position of the given point relative to the plane. It first calculates the distance from the point to the plane using the `pointToPlaneDistance` method. This method uses the normal vector of the plane and the dot product between the normal and the `p0` point of the plane to calculate the distance between the point and the plane.

Once the distance has been calculated, the `getPointPosition` method compares it to a small tolerance value, 0.0001 in this case. If the absolute value of the distance is less than this tolerance value, the point is “on plane”. If the distance is greater than zero, the point is “in front of plane”. If the distance is less than or equal to zero, the point is “behind plane”. The method returns the appropriate string value based on the calculated distance.

3.4.2 Implementation

```
// Returns point position relative to the plane.
std::string Plane::getPointPosition(Vector point) {
    // Calculate the distance from the point to the plane.
    float distance = pointToPlaneDistance(point);
    // If the absolute value of the distance is less than a small tolerance value (0.0001
    in this case), return "on plane".
    if (abs(distance) < 0.0001) {
        return "on plane";
    }
    // If the distance is greater than zero, return "in front of plane".
    else if (distance > 0) {
        return "in front of plane";
    }
    // If the distance is less than or equal to zero, return "behind plane".
    else {
        return "behind plane";
    }
}
```

3.4.3 Results

Test Name	Test Number	Expected Result	Actual Result	Pass/Fail
Plane Get Point Position	1	on plane	on plane	Pass
Plane Get Point Position	2	in front of plane	in front of plane	Pass
Plane Get Point Position	3	behind plane	behind plane	Pass

[Table 5: Plane Relative Position Results]

3.5 Calculating a Planes Normal Given Two Vectors

3.5.1 Method

The calculateNormal method in the Plane class is used to calculate the normal vector of the plane using two given vectors. The normal vector of a plane is a vector that is perpendicular to the plane and has a magnitude equal to the area of the parallelogram formed by the plane [1, 2].

To calculate the normal vector of a plane, we can use the cross product of two vectors that lie on the plane [1, 2]. The cross product is a vector that is perpendicular to both input vectors and has a magnitude equal to the area of the parallelogram they form [1, 2].

The calculateNormal method in the Plane class uses this approach to calculate the normal vector of the plane. It takes two Vector objects as arguments, representing the vectors that lie on the plane. It then calculates the cross product between these two vectors and stores it in the normal member of the Plane object. This effectively calculates the normal vector of the plane. The method does not return any value; it simply modifies the normal vector of the Plane object in place.

3.5.2 Implementation

```
/*
** Calculates the normal of the plane using the given vectors.
** The normal is calculated as the cross product between the two given vectors.
** The cross product is a vector that is perpendicular to both of the input vectors and
** has a magnitude equal to the area of the parallelogram they form.
*/
void Plane::calculateNormal(Vector v1, Vector v2) {
    // Calculate the cross product between the two vectors and store it in the normal
    // member of the Plane object.
    normal = v1.Cross(v2);
}
```

3.5.3 Results

Test Name	Test Number	Expected Result	Actual Result	Pass/Fail
Calculate Plane Normal	1	(X: -1 Y: 2 Z: -1)	(X: -1.000000 Y: 2.000000 Z: -1.000000)	Pass
Calculate Plane Normal	1	(X: 1 Y: -2 Z: 1)	(X: 1.000000 Y: -2.000000 Z: 1.000000)	Pass

[Table 6: Calculating Plane Normal Results]

4. Sphere to Sphere Collision Detection

4.1 The Sphere Class

4.1.1 Implementation

```
#ifndef SPHERE_H
#define SPHERE_H

#include "Vector_Functions.h"
#include <iostream>
#include <cmath>

class Sphere {
public:
    // Constructors.
    Sphere(float x, float y, float z, float r); // Constructor initializes the
    center point and radius of the sphere.
    Sphere(Vector& v1, float r); // Constructor initializes the center point
    and radius of the sphere from a Vector object and a float value.

    // Member functions.
    bool CollidesWith(Sphere& s1); // Returns true if this sphere collides
    with the given sphere, false otherwise.
    void setCenter(Vector& v1) { sCenter = v1; } // Set the center of the
    sphere to the given Vector object.
    Vector getCenter() { return sCenter; } // Get the center of the sphere as a
    Vector object.
    void setRadius(float r) { sRadius = r; } // Set the radius of the sphere to
    the given float value.
    float getRadius() { return sRadius; } // Get the radius of the sphere as a
    float value.

private:
    Vector sCenter; // Center point of the sphere.
    float sRadius; // Radius of the sphere.
};

#endif // SPHERE_H
```

4.2 Collision Detection

4.2.1 Method

The CollidesWith method in the Sphere class is used to determine whether this sphere collides with another given sphere. A collision between two spheres occurs when the distance between their centre points is less than or equal to the sum of their radii [3, 4].

To calculate the distance between two points, we can use the pToPDistance method, which calculates the distance between two points using the Euclidean distance formula. The Euclidean distance between two points is the length of the shortest path between the points, calculated using the Pythagorean theorem [3, 4].

The CollidesWith method in the Sphere class uses the pToPDistance method to calculate the distance between the centre points of the two spheres. It then compares this distance to the sum of the radii of the spheres [3, 4]. If the distance is less than or equal to the sum of the radii, the spheres collide [3, 4].

4.2.2 Implementation

```
// Returns true if this sphere collides with the given sphere, false otherwise.
bool Sphere::CollidesWith(Sphere& s1) {
    // Calculate the distance between the two center points.
    float distance = sCenter.pToPDistance(s1.sCenter);
    // If the distance is less than or equal to the sum of the two radii, the spheres
    collide.
    return distance <= sRadius + s1.sRadius;
}
```

4.2.3 Results

Test Name	Test Number	Expected Result	Actual Result	Pass/Fail
Sphere Collisions	1	Collision Detected	Collision Detected	Pass
Sphere Collisions	2	No Collision Detected	No Collision Detected	Pass

[Table 7: Sphere to Sphere Collision Results]

5. Point to Plane and Sphere to Plane Collision Detection

5.1 Point to Plane Collisions

5.1.1 Method

The CollidesWithPoint method in the Plane class is used to determine whether a given point collides with the plane. A collision between a point and a plane occurs when the distance between the point and the plane is less than or equal to zero [5].

To calculate the distance between a point and a plane, we can use the pointToPlaneDistance method [5]. This method calculates the distance between a point and a plane using the normal vector of the plane and the dot product between the normal and the p0 point of the plane [5].

The CollidesWithPoint method in the Plane class uses the pointToPlaneDistance method to calculate the distance between the point and the plane. It then compares this distance to zero. If the distance is less than or equal to zero, the point collides with the plane [5]. The method returns a Boolean value indicating whether the point collides with the plane.

5.1.2 Implementation

```
bool Plane::CollidesWithPoint(Vector& point) {
    // Calculate the distance between the point and the plane.
    float distance = pointToPlaneDistance(point);
    // If the distance is less than or equal to 0, the point collides with the plane.
    return distance <= 0;
}
```

5.1.3 Results

Test Name	Test Number	Expected Result	Actual Result	Pass/Fail
Point to Plane Collision Detection	1	FALSE	0	Pass
Point to Plane Collision Detection	2	TRUE	1	Pass

[Table 8: Point to Plane Collision Results]

5.2 Sphere to Plane Collisions

5.2.1 Method

A collision between a sphere and a plane occurs when the distance between the centre point of the sphere and the plane is less than or equal to the radius of the sphere [6].

To calculate the distance between a point and a plane, we can use the dot product between the normal vector of the plane and the point [6], as well as the dot product between the normal and the p0 point of the plane [6]. The dot product is a scalar value that represents the projection of one vector onto another [6]. It is calculated using the following formula: $\text{dot\_product} = A_x * B_x + A_y * B_y + A_z * B_z$ [6], where A and B are the two vectors being considered and A_x, A_y, and A_z (and B_x, B_y, and B_z) are the x, y, and z components of the vectors, respectively [6].

The CollidesWithSphere method in the Plane class uses the dot product between the normal vector of the plane and the centre point of the sphere, as well as the dot product between the normal and the p0 point of the plane, to calculate the distance between the centre point of the sphere and the plane. Comparing this distance to the radius of the sphere. If the distance is less than or equal to the radius of the sphere, the sphere collides with the plane [6]. The method returns a Boolean value indicating whether the sphere collides with the plane.

5.2.2 Implementation

```
// Returns true if the sphere with the given center point and radius collides with this
// plane, false otherwise.
bool Plane::CollidesWithSphere(Sphere& s1) {
    // Calculate the distance between the center point of the sphere and the plane.
    float distance = normal.DotProd(s1.getCenter()) + (normal.DotProd(p0));
    // If the distance is less than or equal to the radius of the sphere, the sphere
    // collides with the plane.
    return distance <= s1.getRadius();
}
```

5.2.3 Results

Test Name	Test Number	Expected Result	Actual Result	Pass/Fail
Sphere-Plane Collision Detection	1	Collision Detected	Collision Detected	Pass
Sphere-Plane Collision Detection	2	Collision Detected	Collision Detected	Pass
Sphere-Plane Collision Detection	3	No Collision Detected	No Collision Detected	Pass

[Table 9: Sphere to Plane Collision Results]

6. Conclusion

The expansion and enhancement of the vector class for the game engine has been described. The distance between two points and distance squared between two points methods were successfully implemented in Task 1, using a sound understanding of the underlying mathematical concepts.

The Plane class was created in Task 2, and a series of methods were developed to manipulate and analyse planes. A method was implemented to normalize the plane normal vector, as well as a method to calculate the distance of a point from the plane. A method was also created to determine the position of a point relative to the plane, and a helper method for calculating a plane's normal given two vectors was included. In addition, a method for calculating the dot product of two vectors was implemented, which became relevantly useful in the point to plane distance calculation.

Task 3 involved conducting a thorough investigation into sphere-to-sphere collision detection and implementing a method for detecting collisions between two sphere bounding volumes. Geometry and algebra were applied to design an efficient and reliable collision detection algorithm.

In Task 4, methods for point to plane and sphere to plane collision detection were created, further extending the capabilities of the vector class for the games engine. These methods were tested thoroughly, and the results are included in the technical report. The test harness can be found in Appendix B.

Overall, this technical report represents a significant contribution to the expansion and improvement of the vector class for the games engine. Mathematical and software engineering principles were applied to design and implement effective classes and methods, and the reliability and accuracy of the work was demonstrated through thorough testing. All test results can be found in Appendix D.

REFERENCES

- [1] F. Dunn and I. Parberry, 3D math primer for graphics and game development. Boca Raton: CRC Press, 2012.
- [2] C. Ericson, Real-time collision detection. Amsterdam: Elsevier, 2005.
- [3] G. Bergen, Collision detection in interactive 3D environments. Amsterdam: Elsevier, 2004.
- [4] V. J. M. van and L. M. Bishop, Essential mathematics for games and interactive applications. Boca Raton: CRC Press/Taylor & Francis Group, 2016.
- [5] "12.5 lines and planes," whitman.edu. [Online]. Available: https://www.whitman.edu/mathematics/calculus_online/section12.05.html. [Accessed: 08-Jan-2023].
- [6] Collision detection, 31-Oct-2003. [Online]. Available: <http://www.edenwaith.com/products/pige/tutorials/collision.php>. [Accessed: 08-Jan-2023].
- [Tables 1-9] J. Piecko, Swansea, UWTSO, 2023.

Appendix A: Existing Vector Class .cpp and .h files

File: Vector.cpp:

```
#include "vector.h"

    //construct
Vector::Vector(void){};
Vector::Vector(float myx, float myy, float myz)
{
    x= myx;
    y= myy;
    z= myz;
}
void Vector::Scale(float s)
{
    x= s*x;
    y= s*y;
    z= s*z;
}
void Vector::Add(Vector & v1, Vector & v2)
{
    v1.x= v2.x + v1.x;
    v1.y= v2.y + v1.y;
    v1.z= v2.z + v1.z;
}
Vector Vector::getVector(void)
{
    return Vector(x,y,z);
}
void Vector::Add(Vector & v1)
{
    x= x + v1.x;
```

```
    y= y + v1.y;
    z = z + v1.z;
}
```

File: Vector.h:

```
/****** (note x,y,z are not private-should they be?)
#ifndef VECTOR_H
#define VECTOR_H

class Vector

{
public:

    //construct
    Vector(float x, float y, float z);
    Vector(void);
    void Scale(float s);
    void Add(Vector & v1);
    void Add(Vector & v1, Vector & v2);
    Vector getVector(void);

//private:    //these are currently public - should they be
private?

    float x;
    float y;
    float z;
};
#endif
```

Appendix B: VectorPlayground, Test Harness

File: VectorPlayground.cpp:

```
#include <iostream>
#include <cmath>
#include <string>
#include <fstream>
#include "Vector_Functions.h"
#include "Plane.h"
#include "Sphere.h"

// Declare a global variable of type std::ofstream
std::ofstream myfile;

// Define a function called PrintTestResults that takes five
arguments:
void PrintTestResults(std::string description, int testNum,
std::string expectedResult, std::string actualResult, std::string
passFail) {
    // Print the values of the arguments to the console with some
additional string literals
    std::cout << description << ", Test: " << testNum << "\n";
    std::cout << "Expected Result: " << expectedResult << "\n";
    std::cout << "Actual Result: " << actualResult << "\n";
    std::cout << "Pass/Fail: " << passFail << "\n";
    std::cout << "\n";

    // Open the file "TestResults.csv" in append mode
    myfile.open("TestResults.csv", std::ios_base::app);

    // Write the values of the five arguments to the file in CSV
format, with each argument separated by a comma
    // and each row terminated by a newline character
```

```
myfile << description << "," << testNum << "," << expectedResult  
<< "," << actualResult << "," << passFail << std::endl;
```

```
// Close the file
```

```
myfile.close();
```

```
}
```

```
// Task 1 Relevant Method Testing
```

```
void VecScaleMethodTests();
```

```
void VecAddMethodTests();
```

```
void VecCrossMethodTests();
```

```
void VecDotProdMethodTests();
```

```
void VecPToPDistanceTests();
```

```
void VecPToPDistanceSqrTests();
```

```
// Task 2 Relevant Method Testing
```

```
void CalculatePlaneNormalTests();
```

```
void NormalizePlaneNormalTests();
```

```
void PointToPlaneDistanceTests();
```

```
void PlaneGetPointPositionTests();
```

```
// Task 3 Relevant Method Testing
```

```
void SphereCollisionTests();
```

```
// Task 4 Relevant Method Testing
```

```
void SpherePlaneCollisionTests();
```

```
void CollidesWithPointTests();
```

```
int main()
```

```
{
```

```

std::cout << "--- Preparing Test Schedule ---" << "\n";
VecScaleMethodTests();
VecAddMethodTests();
VecCrossMethodTests();
VecDotProdMethodTests();
VecPToPDistanceTests();
VecPToPDistanceSqrTests();
std::cout << "--- Testing Phase 1 Completed ---" << "\n";
CalculatePlaneNormalTests();
NormalizePlaneNormalTests();
PointToPlaneDistanceTests();
PlaneGetPointPositionTests();
std::cout << "--- Testing Phase 2 Completed ---" << "\n";
SphereCollisionTests();
std::cout << "--- Testing Phase 3 Completed ---" << "\n";

SpherePlaneCollisionTests();
CollidesWithPointTests();
std::cout << "--- Testing Phase 4 Completed ---" << "\n";

return 0;
}

```

```

void VecScaleMethodTests() {
    std::cout << "--- Running Vector Scale Tests... ---" << "\n";

    // Scale Test 1
    Vector v1(1, 2, 3);

```

```

v1.Scale(2);

std::cout << "Running Vector Scale Test 1..." << "\n";

if (v1.getX() == 2 && v1.getY() == 4 && v1.getZ() == 6) {

    PrintTestResults("Scaling a Vector Test", 1, "(X: 2 Y: 4 Z:
6)", "(X: " + std::to_string(v1.getX()) + " Y: " +
std::to_string(v1.getY()) + " Z: " + std::to_string(v1.getZ()) +
")", "Pass");

}

else {

    PrintTestResults("Scaling a Vector Test", 1, "(X: 2 Y: 4 Z:
6)", "(X: " + std::to_string(v1.getX()) + " Y: " +
std::to_string(v1.getY()) + " Z: " + std::to_string(v1.getZ()) +
")", "Fail");

}

// Scale Test 2

Vector v2(2, 3, 4);

v2.Scale(-1);

std::cout << "Running Vector Scale Test 2..." << "\n";

if (v2.getX() == -2 && v2.getY() == -3 && v2.getZ() == -4) {

    PrintTestResults("Scaling a Vector", 2, "(X: 2 Y: 4 Z: 6)",
"(X: " + std::to_string(v2.getX()) + " Y: " +
std::to_string(v2.getY()) + " Z: " + std::to_string(v2.getZ()) +
")", "Pass");

}

else {

    PrintTestResults("Scaling a Vector", 2, "(X: 2 Y: 4 Z: 6)",
"(X: " + std::to_string(v2.getX()) + " Y: " +
std::to_string(v2.getY()) + " Z: " + std::to_string(v2.getZ()) +
")", "Fail");

}

std::cout << "--- Scale Testing Complete ---" << "\n";

}

void VecAddMethodTests() {

```

```

std::cout << "--- Running Tests on Vector Add Methods ---" <<
"\n";

// Add Method 1 Test
Vector v1(1, 2, 3);
Vector v2(2, 3, 4);
v1.Add(v2);

if (v1.getX() == 3 && v1.getY() == 5 && v1.getZ() == 7) {
    PrintTestResults("Vector Add Methods", 1, "(X: 3 Y: 5 Z:
7)", "(X: " + std::to_string(v1.getX()) + " Y: " +
std::to_string(v1.getY()) + " Z: " + std::to_string(v1.getZ()) +
")", "Pass");
}
else {
    PrintTestResults("Vector Add Methods", 1, "(X: 3 Y: 5 Z:
7)", "(X: " + std::to_string(v1.getX()) + " Y: " +
std::to_string(v1.getY()) + " Z: " + std::to_string(v1.getZ()) +
")", "Fail");
}

// Add Method 2 Test
Vector v3(3, 4, 5);
Vector v4(4, 5, 6);
v3.Add(v3, v4);

if (v3.getX() == 7 && v3.getY() == 9 && v3.getZ() == 11) {
    PrintTestResults("Vector Add Methods", 2, "(X: 7 Y: 9 Z:
11)", "(X: " + std::to_string(v3.getX()) + " Y: " +
std::to_string(v3.getY()) + " Z: " + std::to_string(v3.getZ()) +
")", "Pass");
}
else {
    PrintTestResults("Vector Add Methods", 2, "(X: 7 Y: 9 Z:
11)", "(X: " + std::to_string(v3.getX()) + " Y: " +
std::to_string(v3.getY()) + " Z: " + std::to_string(v3.getZ()) +
")", "Fail");
}
}

```

```

void VecCrossMethodTests() {
    std::cout << "---- Running Vector Cross Testing ----" << "\n";

    // Vector Cross Test 1
    Vector v1(1, 2, 3);
    Vector v2(2, 3, 4);
    Vector v3 = v1.Cross(v2);

    if (v3.getX() == -1 && v3.getY() == 2 && v3.getZ() == -1) {
        PrintTestResults("Vector Cross Method", 1, "(X: -1 Y: 2 Z: -1)", "(X: " + std::to_string(v3.getX()) + " Y: " +
std::to_string(v3.getY()) + " Z: " + std::to_string(v3.getZ()) +
")", "Pass");
    }
    else {
        PrintTestResults("Vector Cross Method", 1, "(X: -1 Y: 2 Z: -1)", "(X: " + std::to_string(v3.getX()) + " Y: " +
std::to_string(v3.getY()) + " Z: " + std::to_string(v3.getZ()) +
")", "Fail");
    }

    // Vector Cross Test 2
    Vector v4(3, 2, 1);
    Vector v5(4, 3, 2);
    Vector v6 = v4.Cross(v5);

    if (v6.getX() == 1 && v6.getY() == -2 && v6.getZ() == 1) {
        PrintTestResults("Vector Cross Method", 2, "(X: 1 Y: -2 Z: 1)", "(X: " + std::to_string(v6.getX()) + " Y: " +
std::to_string(v6.getY()) + " Z: " + std::to_string(v6.getZ()) +
")", "Pass");
    }
    else {
        PrintTestResults("Vector Cross Method", 2, "(X: 1 Y: -2 Z: 1)", "(X: " + std::to_string(v6.getX()) + " Y: " +
std::to_string(v6.getY()) + " Z: " + std::to_string(v6.getZ()) +
")", "Fail");
    }
}

```

```
    }  
}  
  
void VecDotProdMethodTests() {  
  
    std::cout << "--- Running Vector Dot Product Testing ---" <<  
    "\n";  
  
    // Vector Dot Product Test 1  
    Vector v1(1, 2, 3);  
    Vector v2(2, 3, 4);  
    float dotProd = v1.DotProd(v2);  
    if (dotProd == 20) {  
        PrintTestResults("Vector Dot Product", 1, "20",  
std::to_string(dotProd), "Pass");  
    }  
    else {  
        PrintTestResults("Vector Dot Product", 1, "20",  
std::to_string(dotProd), "Fail");  
    }  
  
    // Vector Dot Product Test 2  
    Vector v3(3, 4, 5);  
    Vector v4(4, 5, 6);  
    dotProd = v3.DotProd(v4);  
    if (dotProd == 62) {  
        PrintTestResults("Vector Dot Product", 2, "62",  
std::to_string(dotProd), "Pass");  
    }  
    else {  
        PrintTestResults("Vector Dot Product", 2, "62",  
std::to_string(dotProd), "Fail");  
    }  
}
```

```

}

void VecPToPDistanceTests() {
    std::cout << "--- Testing Vector PToPDistance Method ---" <<
    "\n";

    const float tolerance = 0.00001; // Set tolerance value
    float dist;

    // Vector pToPDistance Method Test 1
    Vector v1(1, 2, 3);
    Vector v2(2, 3, 4);
    dist = v1.pToPDistance(v2);

    if (std::abs(dist - std::sqrt(3.0)) < tolerance) { // Compare
using tolerance value
        PrintTestResults("Vector pToPDistance Method", 1,
"1.732051", std::to_string(dist), "Pass");
    }
    else {
        PrintTestResults("Vector pToPDistance Method", 1,
"1.732051", std::to_string(dist), "Fail");
    }

    // Vector pToPDistance Method Test 2
    Vector v3(4, 3, 2);
    Vector v4(1, 2, 3);
    dist = v3.pToPDistance(v4);

    if (std::abs(dist - std::sqrt(11.0)) < tolerance) {
        PrintTestResults("Vector pToPDistance Method", 2,
"3.316625", std::to_string(dist), "Pass");
    }
}

```

```

        else {
            PrintTestResults("Vector pToPDistance Method", 2,
"3.316625", std::to_string(dist), "Pass");
        }
    }

void VecPToPDistanceSqrTests() {
    std::cout << "--- Testing Vector PToPDistance Method ---" <<
"\n";

    const float tolerance = 0.00001; // Set tolerance value
    float dist;

    // Vector pToPDistance Method Test 1
    Vector v1(1, 2, 3);
    Vector v2(2, 3, 4);
    dist = v1.pToPDistanceSqr(v2);

    if (std::abs(dist - 3.0) < tolerance) { // Compare using
tolerance value
        PrintTestResults("Vector pToPDistance Method", 1, "3.0",
std::to_string(dist), "Pass");
    }
    else {
        PrintTestResults("Vector pToPDistance Method", 1, "3.0",
std::to_string(dist), "Fail");
    }

    // Vector pToPDistance Method Test 2
    Vector v3(4, 3, 2);
    Vector v4(1, 2, 3);
    dist = v3.pToPDistanceSqr(v4);

    if (std::abs(dist - 11.0) < tolerance) {
        PrintTestResults("Vector pToPDistanceSqrMethod", 2, "11.0",
std::to_string(dist), "Pass");
    }
}

```

```

    }
    else {
        PrintTestResults("Vector pToPDistanceSqrd Method", 2,
            "11.0", std::to_string(dist), "Fail");
    }
}

/*
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
*/

void CalculatePlaneNormalTests() {

    std::cout << "--- Testing Plane Calculate Normal Method ---" <<
        "\n";

    // Calculating Plane Normal Test 1
    Vector v1(1, 2, 3);
    Vector v2(2, 3, 4);
    Plane plane1;
```

```

    plane1.calculateNormal(v1, v2);

    Vector tNormal1 = plane1.getNormal();

    if (tNormal1.getX() == -1 && tNormal1.getY() == 2 &&
tNormal1.getZ() == -1) {

        PrintTestResults("Calculate Plane Normal", 1, "(X: -1 Y: 2
Z: -1)", "(X: " + std::to_string(tNormal1.getX()) + " Y: " +
std::to_string(tNormal1.getY()) + " Z: " +
std::to_string(tNormal1.getZ()) + ")", "Pass");

    }

    else {

        PrintTestResults("Calculate Plane Normal", 1, "(X: -1 Y: 2
Z: -1)", "(X: " + std::to_string(tNormal1.getX()) + " Y: " +
std::to_string(tNormal1.getY()) + " Z: " +
std::to_string(tNormal1.getZ()) + ")", "Fail");

    }

    // Calculating Plane Normal Test 2

    Vector v3(3, 2, 1);

    Vector v4(4, 3, 2);

    Plane plane2;

    plane2.calculateNormal(v3, v4);

    Vector tNormal2 = plane2.getNormal();

    if (tNormal2.getX() == 1 && tNormal2.getY() == -2 &&
tNormal2.getZ() == 1) {

        PrintTestResults("Calculate Plane Normal", 1, "(X: 1 Y: -2
Z: 1)", "(X: " + std::to_string(tNormal2.getX()) + " Y: " +
std::to_string(tNormal2.getY()) + " Z: " +
std::to_string(tNormal2.getZ()) + ")", "Pass");

    }

    else {

        PrintTestResults("Calculate Plane Normal", 1, "(X: 1 Y: -2
Z: 1)", "(X: " + std::to_string(tNormal2.getX()) + " Y: " +
std::to_string(tNormal2.getY()) + " Z: " +
std::to_string(tNormal2.getZ()) + ")", "Fail");

    }

}

```

```

void NormalizePlaneNormalTests() {
    const float tolerance = 0.00001; // Set tolerance value

    std::cout << "--- Testing Normalizing Plane Normal Method ---"
    << "\n";

    // Normalizing Plane Normal Test 1
    Vector normal1(-1, 2, -1);
    Vector p01(2, 3, 4);

    Plane plane1(normal1, p01);
    plane1.normalizeNormal();
    Vector tNormal1 = plane1.getNormal();

    if (tNormal1.getX() - 0.408248 < tolerance && tNormal1.getY() -
    0.816496 < tolerance && tNormal1.getZ() - -0.408248 < tolerance) {
        PrintTestResults("Normalizing Plane Normal", 1, "(X: -
    0.408248 Y: 0.816496 Z: -0.408248)", "(X: " +
    std::to_string(tNormal1.getX()) + " Y: " +
    std::to_string(tNormal1.getY()) + " Z: " +
    std::to_string(tNormal1.getZ()) + ")", "Pass");
    }
    else {
        PrintTestResults("Normalizing Plane Normal", 1, "(X: -
    0.408248 Y: 0.816496 Z: -0.408248)", "(X: " +
    std::to_string(tNormal1.getX()) + " Y: " +
    std::to_string(tNormal1.getY()) + " Z: " +
    std::to_string(tNormal1.getZ()) + ")", "Fail");
    }

    // Normalizing Plane Normal Test
    Vector normal2(4, 3, 2);
    Vector p02(2, 3, 4);

```

```

Plane plane2(normal2, p02);
plane2.normalizeNormal();
Vector tNormal2 = plane2.getNormal();

if (tNormal2.getX() - 0.742781 < tolerance && tNormal2.getY() -
0.557086 < tolerance && tNormal2.getZ() - 0.371390 < tolerance) {
    PrintTestResults("Normalizing Plane Normal", 2, "(X:
0.742781 Y: 0.557086 Z: 0.371390)", "(X: " +
std::to_string(tNormal2.getX()) + " Y: " +
std::to_string(tNormal2.getY()) + " Z: " +
std::to_string(tNormal2.getZ()) + ")", "Pass");
}
else {
    PrintTestResults("Normalizing Plane Normal", 2, "(X:
0.742781 Y: 0.557086 Z: 0.371390)", "(X: " +
std::to_string(tNormal2.getX()) + " Y: " +
std::to_string(tNormal2.getY()) + " Z: " +
std::to_string(tNormal2.getZ()) + ")", "Fail");
}
}

void PointToPlaneDistanceTests() {
    const float tolerance = 0.0001; // Set tolerance value

    std::cout << "--- Testing Point to Plane Distance Method ---" <<
    "\n";

    // Point to Plane Distance Test 1
    Vector normal1(0, 0, 1);
    Vector p01(-1, 2, -3);
    Plane plane1(normal1, p01);
    Vector point1(2, -1, 3);
    float tDistance1 = plane1.pointToPlaneDistance(point1);
    if (tDistance1 - 6 < tolerance) {
        PrintTestResults("Point to Plane Distance", 1, "6",
std::to_string(tDistance1), "Pass");
    }
}

```

```

    }
    else {
        PrintTestResults("Point to Plane Distance", 1, "6",
std::to_string(tDistance1), "Fail");
    }

    // Point to Plane Distance Test 2
    Vector normal2(-1, -1, -1);
    Vector p02(-2, -2, -2);
    Plane plane2(normal2, p02);
    Vector point2(-3, -3, -3);
    float tDistance2 = plane2.pointToPlaneDistance(point2);
    if (tDistance2 - 1.73205 < tolerance) {
        PrintTestResults("Point to Plane Distance", 2, "1.73205",
std::to_string(tDistance2), "Pass");
    }
    else {
        PrintTestResults("Point to Plane Distance", 2, "1.73205",
std::to_string(tDistance2), "Fail");
    }

    // Point to Plane Distance Test 3
    Vector normal3(1, 0, 0);
    Vector p03(-1, 2, 3);
    Plane plane3(normal3, p03);
    Vector point3(2, 1, -1);
    float tDistance3 = plane3.pointToPlaneDistance(point3);
    if (tDistance3 - 3 < tolerance) {
        PrintTestResults("Point to Plane Distance", 3, "3",
std::to_string(tDistance3), "Pass");
    }
    else {
        PrintTestResults("Point to Plane Distance", 3, "3",
std::to_string(tDistance3), "Fail");
    }

```

```

    }
}

void PlaneGetPointPositionTests() {
    std::cout << "---- Testing Plane Get Point Position Method ----"
    << "\n";

    // Test 1

    Plane p1(Vector(1, 2, 3), Vector(0, 0, 0)); // Plane with
    normal (1, 2, 3) and distance 0.

    Vector point1(0, 0, 0); // Point on plane.

    std::string expectedResult1 = "on plane";
    std::string actualResult1 = p1.getPointPosition(point1);

    if (expectedResult1 == actualResult1) {
        PrintTestResults("Plane Get Point Position", 1,
        expectedResult1, actualResult1, "Pass");
    }
    else {
        PrintTestResults("Plane Get Point Position", 1,
        expectedResult1, actualResult1, "Fail");
    }

    // Test 2

    Plane p2(Vector(1, 2, 3), Vector(0, 0, 0)); // Plane with
    normal (1, 2, 3) and distance 0.

    Vector point2(1, 0, 0); // Point in front of plane.

    std::string expectedResult2 = "in front of plane";
    std::string actualResult2 = p2.getPointPosition(point2);

    if (expectedResult2 == actualResult2) {
        PrintTestResults("Plane Get Point Position", 2,
        expectedResult2, actualResult2, "Pass");
    }
}

```

[illegible]

```

//
//
//
//
*/

void SphereCollisionTests() {
    std::cout << "--- Testing Sphere to Sphere Collision Method ---"
    << "\n";

    // Sphere Collision Test 1

    Sphere s1(0, 0, 0, 1); // Sphere with center at (0, 0, 0) and
    radius 1.

    Sphere s2(2, 0, 0, 1); // Sphere with center at (2, 0, 0) and
    radius 1.

    if (s1.CollidesWith(s2)) {
        PrintTestResults("Sphere Collisions", 1, "Collision
        Detected", "Collision Detected", "Pass");
    }
    else {
        PrintTestResults("Sphere Collisions", 1, "Collision
        Detected", "No Collision Detected", "Fail");
    }

    // Sphere Collision Test 2

    Vector v1(0, 0, 0);
    Vector v2(3, 0, 0);
    Sphere s3(v1, 1); // Sphere with center at (0, 0, 0) and radius
1.
    Sphere s4(v2, 1); // Sphere with center at (3, 0, 0) and radius
1.

    if (s3.CollidesWith(s4) == false) {

```

```
        PrintTestResults("Sphere Collisions", 2, "No Collision  
Detected", "No Collision Detected", "Pass");
```

```
    }
```

```
    else {
```

```
        PrintTestResults("Sphere Collisions", 2, "No Collision  
Detected", "Collision Detected", "Fail");
```

```
    }
```

```
}
```

```
/*
```

```
//
```

```
//
```

```
//
```

```
//
```

```
//
```

```
//
```

```
//          END OF TASK 3
```

```
//
```

```
//
```

```
//
```

```
//
```

```
//
```

```
//
```

```
*/
```

```
void SpherePlaneCollisionTests() {
```

```
    std::cout << "--- Testing Sphere-Plane Collision Detection ---"  
<< "\n";
```

```
Plane p1(Vector(-1, 2, -1), Vector(1, 2, 3)); // Plane with
normal (-1, 2, -1) and p0 (1, 2, 3).
```

```
// Sphere to Plane Collision Test 1
```

```
Vector center1(0, 0, 0); // Center point of sphere 1.
```

```
float radius1 = 1; // Radius of sphere 1.
```

```
Sphere s1(center1, radius1);
```

```
if (p1.CollidesWithSphere(s1)) {
```

```
    PrintTestResults("Sphere-Plane Collision Detection", 1,
"Collision Detected", "Collision Detected", "Pass");
```

```
}
```

```
else {
```

```
    PrintTestResults("Sphere-Plane Collision Detection", 1,
"Collision Detected", "No Collision Detected", "Fail");
```

```
}
```

```
// Sphere to Plane Collision Test 2
```

```
Vector center2(-2, 4, -2); // Center point of sphere 2.
```

```
float radius2 = 12; // Radius of sphere 2.
```

```
Sphere s2(center2, radius2);
```

```
if (p1.CollidesWithSphere(s2)) {
```

```
    PrintTestResults("Sphere-Plane Collision Detection", 2,
"Collision Detected", "Collision Detected", "Pass");
```

```
}
```

```
else {
```

```
    PrintTestResults("Sphere-Plane Collision Detection", 2,
"Collision Detected", "No Collision Detected", "Fail");
```

```
}
```

```
// Sphere to Plane Collision Test 3
```

```

Vector center3(0, 2, 0); // Center point of sphere 3.
float radius3 = 1; // Radius of sphere 3.
Sphere s3(center3, radius3);

if (!p1.CollidesWithSphere(s3)) {
    PrintTestResults("Sphere-Plane Collision Detection", 3, "No
Collision Detected", "No Collision Detected", "Pass");
}
else {
    PrintTestResults("Sphere-Plane Collision Detection", 3, "No
Collision Detected", "Collision Detected", "Fail");
}

}

void CollidesWithPointTests() {
    std::cout << "--- Testing CollidesWithPoint Method ---" << "\n";
    const float tolerance = 0.00001; // Set tolerance value
    bool result;

    // CollidesWithPoint Test 1
    Vector p2Normal(-1, -1, -1);
    Vector p2P0(1, 1, 1);
    Plane p2(p2Normal, p2P0);
    Vector point2(-1, -1, -1);
    result = p2.CollidesWithPoint(point2);

    if (result == false) { // Compare using tolerance value
        PrintTestResults("Point to Plane Collision Detection", 1,
"false", std::to_string(result), "Pass");
    }
    else {

```

```
        PrintTestResults("Point to Plane Collision Detection", 1,
"false", std::to_string(result), "Fail");
    }

    // CollidesWithPoint Test 2
    Vector p3Normal(-1, 0, 0);
    Vector p3P0(-1, 0, 0);
    Plane p3(p3Normal, p3P0);
    Vector point3(-1, 0, 0);
    result = p3.CollidesWithPoint(point3);

    if (result == true) { // Compare using tolerance value
        PrintTestResults("Point to Plane Collision Detection", 2,
"true", std::to_string(result), "Pass");
    }
    else {
        PrintTestResults("Point to Plane Collision Detection", 2,
"true", std::to_string(result), "Fail");
    }
}
```

Appendix C: Completed Project

File: Vector_Functions.cpp:

```
#include "Vector_Functions.h"
#include <cmath>
#include <iostream>

// Constructors.

// Default Constructor.
Vector::Vector(void) {}

// Constructor initializes the x, y, and z coordinates of the
Vector.
Vector::Vector(float xInput, float yInput, float zInput)
{
    x = xInput;
    y = yInput;
    z = zInput;
}

// Member Functions

// Vector Scales by the given input value.
void Vector::Scale(float s)
{
    // Scales the Vectors x, y, and z coordinates by the given
input value.
    x = s * x;
    y = s * y;
    z = s * z;
}

// Adds the given Vector to the current Vector.
void Vector::Add(Vector& v1, Vector& v2)
{
    // Adds the x, y and z coordinates of the given Vector to the
current Vector.
    x = v2.x + v1.x;
    y = v2.y + v1.y;
    z = v2.z + v1.z;
}

// Adds the current Vector to the given Vector.
void Vector::Add(Vector& v1)
{
    // Adds the x, y and z coordinates of the current Vector to
the given Vector.
    x = x + v1.x;
    y = y + v1.y;
    z = z + v1.z;
}
```

```

}

/*
** Calculates the dot product between this vector and the given
vector.
** The dot product is a scalar value that is equal to the product of
the magnitudes of the two vectors and the cosine of the angle
between them.
** It can be calculated using the formula:  $x_1 * x_2 + y_1 * y_2 + z_1 * z_2$ , where  $(x_1, y_1, z_1)$  and  $(x_2, y_2, z_2)$  are the components of the
two vectors.
*/
float Vector::DotProd(Vector v1)
{
    // Calculate and return the dot product using the formula:  $x_1 * x_2 + y_1 * y_2 + z_1 * z_2$ .
    return x * v1.x + y * v1.y + z * v1.z;
}

/*
**
** Calculates the cross product between this vector and the given
vector.
** The cross product is a vector that is perpendicular to both of
the input vectors and has a magnitude equal to the area of the
parallelogram they form.
** It can be calculated using the formula:  $(y_1 * z_2 - z_1 * y_2, z_1 * x_2 - x_1 * z_2, x_1 * y_2 - y_1 * x_2)$ , where  $(x_1, y_1, z_1)$  and  $(x_2, y_2, z_2)$  are the components of the two vectors.
** Returns A Vector object representing the result of the cross
product.
*/
Vector Vector::Cross(Vector& v1)
{
    // Calculate the x, y, and z components of the cross product.
    float tx, ty, tz;
    tx = y * v1.z - z * v1.y;
    ty = z * v1.x - x * v1.z;
    tz = x * v1.y - y * v1.x;
    // Update the x, y, and z components of this vector
    return Vector(x, y, z);
}

/*
** Calculates the point-to-point distance between this vector and
the given vector.
** The point-to-point distance is the straight-line distance between
the two points represented by the vectors.
** It can be calculated using the formula:  $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2 + (z_1 - z_2)^2}$ , where  $(x_1, y_1, z_1)$  and  $(x_2, y_2, z_2)$  are the
components of the two vectors.
*/
float Vector::pToPDistance(Vector& v1)
{

```

```

        // Calculate the difference between the x, y, and z components
of the two vectors.
        float diffX = (v1.x - x);
        float diffY = (v1.y - y);
        float diffZ = (v1.z - z);
        // Calculate the point-to-point distance using the formula:
sqrt((x1 - x2)^2 + (y1 - y2)^2 + (z1 - z2)^2).
        float dist;
        dist = std::sqrt(std::pow(diffX, 2) + std::pow(diffY, 2) +
std::pow(diffZ, 2));
        return dist; // Returns calculated distance
    }

/*
** Calculates the squared point-to-point distance between this
vector and the given vector.
** The squared point-to-point distance is the square of the
straight-line distance between the two points represented by the
vectors.
** It can be calculated using the formula: (x1 - x2)^2 + (y1 - y2)^2
+ (z1 - z2)^2, where (x1, y1, z1) and (x2, y2, z2) are the
components of the two vectors.
*/
float Vector::pToPDistanceSqrd(Vector& v1)
{
    // Calculate the difference between the x, y, and z components
of the two vectors.
    float diffX = (v1.x - x);
    float diffY = (v1.y - y);
    float diffZ = (v1.z - z);
    // Calculate the squared point-to-point distance using the
formula: (x1 - x2)^2 + (y1 - y2)^2 + (z1 - z2)^2.
    float dist;
    dist = (std::pow(diffX, 2) + std::pow(diffY, 2) +
std::pow(diffZ, 2));
    return dist; // Returns distance calculated
}

```

File: Vector_Functions.h:

```
#ifndef VECTOR_FUNCTIONS_H
#define VECTOR_FUNCTIONS_H
#include <cmath>

class Vector {

public:

    // Constructors
    Vector(float x, float y, float z); // Constructs a Vector with
x, y, and z values
    Vector(void); // Constructs a default Vector with x, y, and z
values of 0

    // Member Functions
    void Scale(float s); // Scales the Vector by a scalar value s
    void Add(Vector& v1); // Adds a Vector v1 to the current
Vector object
    void Add(Vector& v1, Vector& v2); // Adds Vector v1 and Vector
v2 and stores the result in the current Vector object
    Vector Cross(Vector& v1); // Returns the cross product of the
current Vector object and Vector v1
    float DotProd(Vector v1); // Returns the dot product of the
current Vector object and Vector v1
    float pToPDistance(Vector& v1); // Returns the distance from
the current Vector object to Vector v1
    float pToPDistanceSqr(Vector& v1); // Returns the squared
distance from the current Vector object to Vector v1

    // Getter and Setter methods
    void setX(float newX) { x = newX; } // Sets the x value of the
Vector to newX
    float getX() const { return x; }; // Returns the x value of
the Vector
    void setY(float newY) { y = newY; } // Sets the y value of the
Vector to newY
    float getY() const { return y; }; // Returns the y value of
the Vector
    void setZ(float newZ) { z = newZ; } // Sets the z value of the
Vector to newZ
    float getZ() const { return z; }; // Returns the z value of
the Vector

private:
    float x;
    float y;
    float z;
};
#endif // VECTOR_FUNCTIONS_H
```

File: Plane.cpp:

```
#include "Vector_Functions.h"
#include "plane.h"
#include "Sphere.h"
#include <iostream>

//Constructs a Plane object with the given normal and point.
Plane::Plane(Vector myNormal, Vector myP0) {
    // Initialize the normal and p0 members of the object with the
    given values.
    normal = myNormal;
    p0 = myP0;
}

//Default constructor for the Plane class.
//This constructor does not initialize any members and leaves them
with their default values.
Plane::Plane(void) {}

/*
** Calculates the normal of the plane using the given vectors.
** The normal is calculated as the cross product between the two
given vectors.
** The cross product is a vector that is perpendicular to both of
the input vectors and has a magnitude equal to the area of the
parallelogram they form.
*/
void Plane::calculateNormal(Vector v1, Vector v2) {
    // Calculate the cross product between the two vectors and store
it in the normal member of the Plane object.
    normal = v1.Cross(v2);
}

/*
**Normalizes the normal of the plane.
**Normalization is the process of scaling a vector to have a
magnitude (length) of 1.
**This is often useful when working with vectors because it allows
them to be compared more easily.
**The normal is normalized by dividing each of its x, y, and z
components by its magnitude (length).
** The magnitude (length) is calculated using the formula:  $\sqrt{x^2 + y^2 + z^2}$ .
*/
void Plane::normalizeNormal() {
    // Calculate the magnitude (length) of the normal using the
formula:  $\sqrt{x^2 + y^2 + z^2}$ .
    float length;
    length = std::sqrt(pow(normal.getX(), 2) + pow(normal.getY(), 2)
+ pow(normal.getZ(), 2));
```

```

    // Divide the x, y, and z components of the normal by its
    magnitude to normalize it.
    normal.setX(normal.getX() / length);
    normal.setY(normal.getY() / length);
    normal.setZ(normal.getZ() / length);
}

// Calculate the distance from the given point to the plane.
// Returns the distance as a float value.
float Plane::pointToPlaneDistance(Vector point) {
    float distance; // Used to store the distance.

    // Extract the x, y, and z components of the normal.
    float A = normal.getX();
    float B = normal.getY();
    float C = normal.getZ();

    // Calculate the dot product between the normal and the p0 point
    of the plane.
    float D = normal.DotProd(p0);

    /*
    ** Calculates the distance from the given point to the plane
    using the following formula:
    ** distance = (A * x + B * y + C * z - D) / sqrt(A^2 + B^2 +
    C^2)
    ** where (A, B, C) is the normal of the plane and D is the dot
    product between the normal and the p0 point of the plane.
    ** The x, y, and z values of the point are passed as arguments
    to the method and are used to calculate the distance from the point
    to the plane.
    */
    distance = (A * point.getX() + B * point.getY() + C *
point.getZ() - D) / std::sqrt(A * A + B * B + C * C);

    // Return the calculated distance.
    return distance;
}

// Returns point poission relative to the plane.
std::string Plane::getPointPosition(Vector point) {
    // Calculate the distance from the point to the plane.
    float distance = pointToPlaneDistance(point);
    // If the absolute value of the distance is less than a small
    tolerance value (0.0001 in this case), return "on plane".
    if (abs(distance) < 0.0001) {
        return "on plane";
    }
    // If the distance is greater than zero, return "in front of
    plane".
    else if (distance > 0) {
        return "in front of plane";
    }
    // If the distance is less than or equal to zero, return "behind
    plane".
    else {

```

```
        return "behind plane";
    }
}

// Returns true if the sphere with the given center point and radius
// collides with this plane, false otherwise.
bool Plane::CollidesWithSphere(Sphere& s1) {
    // Calculate the distance between the center point of the sphere
    // and the plane.
    float distance = normal.DotProd(s1.getCenter()) +
    (normal.DotProd(p0));
    // If the distance is less than or equal to the radius of the
    // sphere, the sphere collides with the plane.
    return distance <= s1.getRadius();
}

bool Plane::CollidesWithPoint(Vector& point) {
    // Calculate the distance between the point and the plane.
    float distance = pointToPlaneDistance(point);
    // If the distance is less than or equal to 0, the point
    // collides with the plane.
    return distance <= 0;
}
```

File: Plane.h:

```

#ifndef PLANE_H
#define PLANE_H
#include "Vector_Functions.h"
#include "Sphere.h"

class Plane {

public:
    // Constructors
    Plane(Vector myNormal, Vector myP0); // Constructor that
    initializes the plane with a normal and point on the plane
    Plane(void); // Default constructor
    // Member Functions
    void calculateNormal(Vector v1, Vector v2); // Calculates the
    normal of the plane based on two points on the plane
    void normalizeNormal(); // Normalizes the normal of the plane
    (makes its length equal to 1)
    float pointToPlaneDistance(Vector point); // Calculates the
    distance from a given point to the plane
    std::string getPointPosition(Vector point); // Returns a string
    indicating whether the given point is above, below, or on the plane
    bool CollidesWithSphere(Sphere& s1); // Returns true if the
    plane collides with a given sphere, otherwise false
    bool CollidesWithPoint(Vector& point); // Returns true if the
    plane collides with a given point, otherwise false

    Vector getNormal(void) const { return normal; }; // Returns the
    normal of the plane.
    void setp0(Vector v1) { p0 = v1; } // Sets point on the plane.
    Vector getp0() const { return p0; }; // Returns the point on the
    plane.

private:
    Vector normal;
    Vector p0;
};

#endif // PLANE_H

```

File: Sphere.cpp:

```
#include "Sphere.h"
#include <iostream>
#include <cmath>

// Constructor initializes the center point and radius of the sphere
from a Vector object and a float value.
Sphere::Sphere(Vector& v1, float r) {
    sCenter = v1;
    sRadius = r;
}

// Constructor initializes the center point and radius of the
sphere.
Sphere::Sphere(float x, float y, float z, float r) {
    sCenter = Vector (x, y, z);
    sRadius = r;
}

// Returns true if this sphere collides with the given sphere, false
otherwise.
bool Sphere::CollidesWith(Sphere& s1) {
    // Calculate the distance between the two center points.
    float distance = sCenter.pToPDistance(s1.sCenter);
    // If the distance is less than or equal to the sum of the two
radii, the spheres collide.
    return distance <= sRadius + s1.sRadius;
}
```

File: Sphere.h:

```

#ifndef SPHERE_H
#define SPHERE_H

#include "Vector_Functions.h"
#include <iostream>
#include <cmath>

class Sphere {
public:
    // Constructors.
    Sphere(float x, float y, float z, float r); //
    Constructor initializes the center point and radius of the sphere.
    Sphere(Vector& v1, float r); // Constructor initializes
    the center point and radius of the sphere from a Vector object and a
    float value.

    // Member functions.
    bool CollidesWith(Sphere& s1); // Returns true if this
    sphere collides with the given sphere, false otherwise.
    void setCenter(Vector& v1) { sCenter = v1; } // Set the
    center of the sphere to the given Vector object.
    Vector getCenter() { return sCenter; } // Get the center
    of the sphere as a Vector object.
    void setRadius(float r) { sRadius = r; } // Set the
    radius of the sphere to the given float value.
    float getRadius() { return sRadius; } // Get the radius
    of the sphere as a float value.

private:
    Vector sCenter; // Center point of the sphere.
    float sRadius; // Radius of the sphere.
};

#endif // SPHERE_H

```

Appendix C: Complete Test Results Dataset

Test Name	Test Number	Expected Result	Actual Result	Pass/Fail
Scaling a Vector Test	1	(X: 2 Y: 4 Z: 6)	(X: 2.000000 Y: 4.000000 Z: 6.000000)	Pass
Scaling a Vector	2	(X: 2 Y: 4 Z: 6)	(X: -2.000000 Y: -3.000000 Z: -4.000000)	Pass
Vector Add Methods	1	(X: 3 Y: 5 Z: 7)	(X: 3.000000 Y: 5.000000 Z: 7.000000)	Pass
Vector Add Methods	2	(X: 7 Y: 9 Z: 11)	(X: 7.000000 Y: 9.000000 Z: 11.000000)	Pass
Vector Cross Method	1	(X: -1 Y: 2 Z: -1)	(X: -1.000000 Y: 2.000000 Z: -1.000000)	Pass
Vector Cross Method	2	(X: 1 Y: -2 Z: 1)	(X: 1.000000 Y: -2.000000 Z: 1.000000)	Pass
Vector Dot Product	1	20	20	Pass
Vector Dot Product	2	62	62	Pass
Vector pToPDistance Method	1	1.732051	1.732051	Pass
Vector pToPDistance Method	2	3.316625	3.316625	Pass
Vector pToPDistanceSqr Method	1	3	3	Pass
Vector pToPDistanceSqr Method	2	11	11	Pass
Calculate Plane Normal	1	(X: -1 Y: 2 Z: -1)	(X: -1.000000 Y: 2.000000 Z: -1.000000)	Pass
Calculate Plane Normal	1	(X: 1 Y: -2 Z: 1)	(X: 1.000000 Y: -2.000000 Z: 1.000000)	Pass
Normalizing Plane Normal	1	(X: -0.408248 Y: 0.816496 Z: -0.408248)	(X: -0.408248 Y: 0.816497 Z: -0.408248)	Pass
Normalizing Plane Normal	2	(X: 0.742781 Y: 0.557086 Z: 0.371390)	(X: 0.742781 Y: 0.557086 Z: 0.371391)	Pass
Point to Plane Distance	1	6	6	Pass
Point to Plane Distance	2	1.73205	1.732051	Pass
Point to Plane Distance	3	3	3	Pass
Plane Get Point Position	1	on plane	on plane	Pass
Plane Get Point Position	2	in front of plane	in front of plane	Pass
Plane Get Point Position	3	behind plane	behind plane	Pass
Sphere Collisions	1	Collision Detected	Collision Detected	Pass
Sphere Collisions	2	No Collision Detected	No Collision Detected	Pass
Sphere-Plane Collision Detection	1	Collision Detected	Collision Detected	Pass
Sphere-Plane Collision Detection	2	Collision Detected	Collision Detected	Pass

Sphere-Plane Collision Detection	3	No Collision Detected	No Collision Detected	Pass
Point to Plane Collision Detection	1	FALSE	0	Pass
Point to Plane Collision Detection	2	TRUE	1	Pass