

Assignment 2

by Jacob Evans

Submission date: 28-Dec-2024 02:38PM (UTC+0000)

Submission ID: 246913227

File name: 50439_Jacob_Evans_Assignment_2_628793_2065071358.docx (1.11M)

Word count: 1506

Character count: 7485



ACCB017 - GAMES ARCHITECTURE AND MECHANICS

Assignment 2

We are demonstrating sound understanding of the
architecture of games and physics engines and
demonstrating the application of these languages, tools and
techniques.

Jacob Evans
2311160@student.uwtsd.ac.uk

Contents

Part 1	2
Part 2	5
Part 3	7
Part 4	9
Conclusion.....	12
References	12
Appendix.....	13
Main code.....	13
Vector.cpp and Vector.h	14
Plane.cpp and Plane.h.....	15
Sphere.cpp and Sphere.h	16

Part 1

In part 1, we are asked to create a Vector class that allows us to store and manipulate two vectors/points. The vector provided in the example usage was $V1 (1\ 2\ 3)$ and $V2(2\ 3\ 4)$. We are asked specifically to find the distance between these points, as well as the distance between squared.

We first find the distance between two points by using the Euclidean Equation[1].

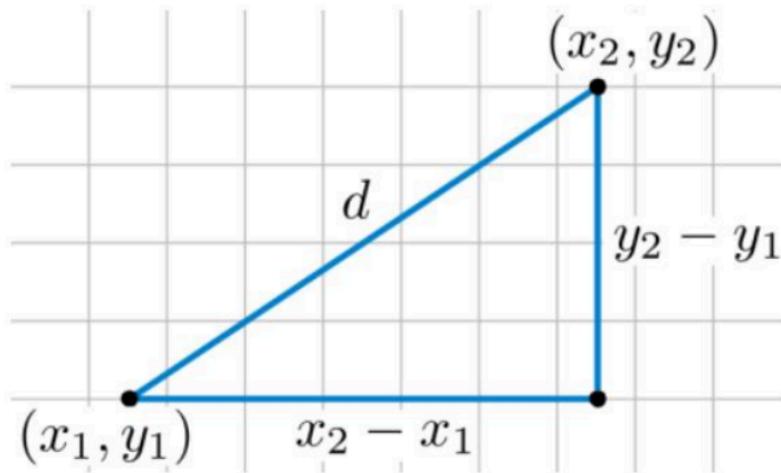


Figure 1: Two-dimensional Euclidean distance

FIGURE 1 - EXAMPLE OF DISTANCE IN 2D [1]

If we look at Figure 1, this is an example of Two-dimension Euclidean distance. We can see that to find the distance between these two points, we use Pythagoras' theorem of " $A^2 + B^2 = C^2$ ", where D is C^2 in figure 1.

We calculate the distance by subtracting the two points of X and two points of Y . The formula would look like this.

$$D = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

FIGURE 2 - EUCLIDEAN FORMULA [1]

This allows us to calculate the distance between two points. The same is true when working with 3D figures, like the Vectors provided (V1 and V2). We would input the values of z into the same formula like below.

$$\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2}$$

FIGURE 3 - PYTHAGORAS THEOREM 3D

```
int main()
{
    Vector v1(1, 2, 3); // Values for first position
    std::cout << "Vector 1 position: ";
    v1.display(); //displays these values
    Vector v2(2, 3, 4); //values for 2nd position
    std::cout << "Vector 2 position: ";
    v2.display(); //displays the 2nd position values
    float distance = v1.pToPDistance(v2); //calculates distance between v1 and v2 using Euclidean
    std::cout << "Distance Between two points: " << distance << std::endl; //displays results
    float distanceSqrd = v1.pToPDistanceSqrd(v2); //squares the distance between v1 and v2
    std::cout << "Distance Between two points squared: " << distanceSqrd << std::endl; //diplays distance squared
}
```

FIGURE 4 - MAIN FOR PART 1

The screenshot above is our main for part1. We set our Vector V1 and V2. “V1.display” calls on a function that simply displays the values of the vector. The same is done for “V2.display”. The next function we use is “v1.pToPDistance(v2)”.

This function allows us to bring values from the vector into our function found in the Vector.h header file. It simply then prints/displays the output for both the distance between the two points and the distance squared.

```

void Vector::display()
{
    std::cout << x << ", " << y << ", " << z << std::endl;
}

float Vector::pToPDistance(Vector& dist)
{
    float distX = dist.x - x;
    float distY = dist.y - y;
    float distZ = dist.z - z;
    float totalDist = (distX*distX) + (distY*distY) + (distZ*distZ);
    return std::sqrt(totalDist);
}

float Vector::pToPDistanceSqr(Vector& dist)
{
    float distX = dist.x - x;
    float distY = dist.y - y;
    float distZ = dist.z - z;
    return ((distX * distX) + (distY * distY) + (distZ * distZ));
}

```

FIGURE 5 - SCREENSHOT OF VECTOR CLASS ADDITION

These are the three new additional functions added to answer part 1. We see that the two functions only slightly differ. We see that the function calculates the distance between v2 and v1 and each value of X, Y and Z. The total distance is then calculated using the Euclidean formula in figure 3 without the root of the total distance. In the example case, the total distance would come out to be "3". To find the distance, we then square root the distance, as seen when the function returns the distance to main.

The same is done in "pToPDistanceSqr", except since the distance was asked to be squared, we can just return the total distance without square rooting the distance.

```

Vector 1 position: 1, 2, 3
Vector 2 position: 2, 3, 4
Distance Between two points: 1.73205
Distance Between two points squared: 3

```

FIGURE 6 - OUTPUT OF MAIN FOR PART 1

This is our output in the console, which displays both the distance and the distance squared between the two points. Using a calculator, we can see that the difference between the two points would be "1 + 1 +

1” and then square rooting, coming out as 1.73205.

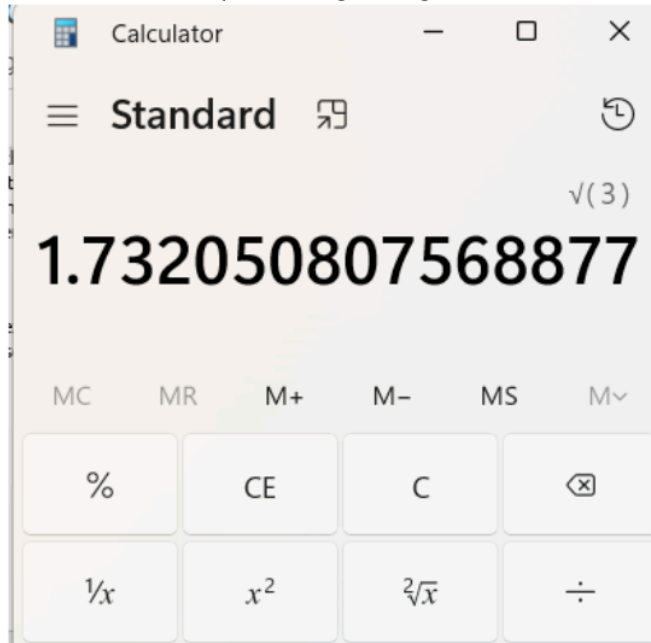


FIGURE 7 - CALCULATOR AS PROOF OF FORMULA

We can simply change the values of the vectors in the main, which would also change the calculation in the output.

Part 2

We are then asked to create a plane class, which allows us to “normalise” a Vector and represent a point P0 on the plane. The term “Normalising” means to process the length of the vector into something “more comparable”. In terms of games, this allows us to move omnidirectionally without slowing down/speeding up [2].

```

#include <plane.h>
#include <math>
#include <vector.h>
#include <plane.h>

Plane::Plane(const Vector& normal, const Vector& pointOnP) {
    normal(normal), pointOnP(pointOnP) {
        normalise();
    }

void Plane::normalise() {
    float magnitude = std::sqrt(normal.x * normal.x + (normal.y * normal.y) + (normal.z * normal.z)); // normalise vector so we have unit length
    normal.x /= magnitude; //vector values are then divided by the magnitude, giving a normal vector
    normal.y /= magnitude;
    normal.z /= magnitude;
}

float Plane::pointDistance(const Vector& point) const {
    Vector difference = point - pointOnP; //this finds the difference between the test point vector and the point on the plane
    return normal.dot(difference);
}

float Plane::pointPosition(const Vector& point) const {
    float distance = pointDistance(point); //this finds out where the point is compared to the plane, if > 0, its above, if == 0, its on, and if it's less than 0 (or anything else), its behind the plane
    return distance;
}

bool Plane::pointCollision(const Vector& point) const {
    return pointDistance(point); //checks if point is on plane
}

Vector Plane::calculateNormal(const Vector& normal, const Vector& testPoint) { //dot product calculation which allows us to find the normal of two vectors
    return normal.cross(testPoint);
}

```

FIGURE 8 - PLANE.CPP

To normalise, we first calculate the magnitude of the vector. We have used the normal vector (0 1 0) as our test in main. We calculate the magnitude of the vector using the same formula seen in

Figure 3 (Pythagoras theorem). We then can divide each value of the vector (X, Y, Z) by the magnitude creating a normal vector, which can be used for when the dot product formula is implemented later.

We then must find the distance between the test point vector and the point on the plane vector. We do this in the “pointDistance” function, which calculates the difference between the two vectors. This new vector is saved as “difference”. The dot product is then implemented using both the normal vector and the new “difference” vector. The dot product will provide us with information on where the point is located, as it will provide with an integer between 1 and -1. This is then returned and used in “pointPosition”, which allows us to indicate where the point is located. If the point distance is greater than 1, then the point is located above the plane. If the point distance is 0, then the point is located on the plane. If the point distance is less than 0, then the point is behind the plane [3].

Definition

The **dot product** of vectors $\mathbf{u} = \langle u_1, u_2, u_3 \rangle$ and $\mathbf{v} = \langle v_1, v_2, v_3 \rangle$ is given by the sum of the products of the components

$$\mathbf{u} \cdot \mathbf{v} = u_1 v_1 + u_2 v_2 + u_3 v_3. \quad (2.3)$$

FIGURE 9 - DOT PRODUCT[3]

Definition

Let $\mathbf{u} = \langle u_1, u_2, u_3 \rangle$ and $\mathbf{v} = \langle v_1, v_2, v_3 \rangle$. Then, the **cross product** $\mathbf{u} \times \mathbf{v}$ is vector

$$\begin{aligned} \mathbf{u} \times \mathbf{v} &= (u_2 v_3 - u_3 v_2) \mathbf{i} - (u_1 v_3 - u_3 v_1) \mathbf{j} + (u_1 v_2 - u_2 v_1) \mathbf{k} \\ &= \langle u_2 v_3 - u_3 v_2, -(u_1 v_3 - u_3 v_1), u_1 v_2 - u_2 v_1 \rangle. \end{aligned} \quad (2.9)$$

FIGURE 10 - CROSS PRODUCT[3]

The cross product of two vectors is perpendicular to the vectors. We can use this calculation to then see if the point vector collides with the perpendicular of these vectors.

```
#ifndef PLANE_H
#define PLANE_H

#include "Vector.h"

class Plane {
public:
    Vector normal;
    Vector pointOnP;

    Plane(const Vector& normal, const Vector& pointOnP);

    void normalise();
    float pointDistance(const Vector& point) const;
    float pointPosition(const Vector& point) const;

    bool pointCollision(const Vector& point) const;

    static Vector calculateNormal(const Vector& v3, const Vector& v4);
};

#endif PLANE_H
```

FIGURE 11 - PLANE.H

As seen in figure 10, this is the header file for Plane.h. Class Plane has been created with the necessary functions.

Part 3

In part three, we are asked to create a collision detection between two spheres, using a Sphere class.

```
1  #pragma once
2  #include <iostream>
3  #include <cmath>
4  #include "Vector.h"
5
6  class Sphere
7  {
8  public:
9      Vector centre; // centre of the sphere
10     float radius; // radius of the sphere
11
12     Sphere(const Vector& centre, float radius);
13
14     float calcVolume() const; // calculate volume of the sphere
15
16     bool checkVolumeCollision(const Sphere& second) const; // checks collision with both spheres
17
18 }
```

FIGURE 12 - PART 3 SPHERE.H HEADER

As seen in the screenshot above, we can see the sphere class that has been created. We see that the constructor allows us to pass both the radius and centre from main into the class.

We then have a function that allows us to calculate the volume of the spheres, using the radius from the testing spheres centre vector and radius (see figure). We can display then display the volume of the sphere in main. We also then have a function that allows us to check for a collision between the two spheres based on the distance between the centres of the spheres.

```

1  #include "Sphere.h"
2  #include <cmath>
3  #include <iostream>
4  #include "Vector.h"
5  #define PI 3.14159 //defining PI makes it easier to calculate volume later
6
7  Sphere::Sphere(const Vector& centre, float radius)
8  : centre(centre), radius(radius) {}
9
10 float Sphere::calcVolume() const
11 {
12     return (4.0f / 3.0f) * PI * std::pow(radius, 3); //calculates the volume, pow raises the power of the radius to 3
13 }
14
15 bool Sphere::checkVolumeCollision(const Sphere& second) const
16 {
17     float distX = second.centre.x - centre.x;
18     float distY = second.centre.y - centre.y;
19     float distZ = second.centre.z - centre.z;
20     float distance = std::sqrt((distX * distX) + (distY * distY) + (distZ * distZ));
21     return distance <= (radius + second.radius); //
22 }
23

```

FIGURE 13 - PART 3 SPHERE.CPP

In figure 9, we see how the functions are implemented. The “calcVolume” function takes the radius provided in the main and uses the formula to calculate volume of a sphere. PI has been defined in case of further use later as this is a constant. We then have the “std::pow” taken from the <cmath> library, which allows us to raise the radius to the power of 3, as seen in the formula below.

$$V = \frac{4}{3} \pi r^3$$

FIGURE 14 - FORMULA FOR VOLUME OF SPHERE

Figure 9 also displays the “checkVolumeCollison” function. The distance between the two centres of the spheres are calculated using the same calculations used in part 1. The difference between each value point (X, Y, Z) is first calculated (as seen as “distX, distY, distZ). Each value point is then squared, while being added which gives us the distance between the centre of both spheres.

The distance is then returned as long as the distance is not larger than the distance of both sphere radii. If the distance is less than the addition of both sphere radii, then this would suggest that the spheres have collided, providing our collision check.

```

Vector centre1(2.0f, 7.0f, 9.0f);
Sphere sphere1(centre1, 5.0f);

Vector centre2(8.0f, 0.0f, 0.0f);
Sphere sphere2(centre2, 4.0f);

std::cout << "\nVolume of the sphere 1: " << sphere1.calcVolume() << " cubic units." << std::endl;
std::cout << "\nVolume of the sphere 2: " << sphere2.calcVolume() << " cubic units." << std::endl;

// Check if the spheres collide
if (sphere1.checkVolumeCollision(sphere2)) {
    std::cout << "Mission success. The spheres have collided." << std::endl;
}
else {
    std::cout << "Mission failed. The spheres do not collide." << std::endl;
}

```

FIGURE 15 - MAIN FOR PART 3

As we can see, we have our sphere centres ready to be tested. These are passed through the constructor allow for the functions to work. We then display the volume of both spheres in cubic units. If the spheres collided or do not collide, there is a message displayed as seen below.

```
Volume of the sphere 1: 523.598 cubic units.
```

```
Volume of the sphere 2: 268.082 cubic units.
Mission failed. The spheres do not collide.
```

```
Volume of the sphere 1: 113.097 cubic units.
```

```
Volume of the sphere 2: 4188.79 cubic units.
Mission success. The spheres have collided.
```

This was done by changing the values of the centre of the spheres, as well as changing the radius value on both spheres. Testing looks to be correct, as it indicates that value changes do influence both the size of the sphere, and the collision check.

Part 4

Using the same pointDistance function to calculate the distance the point is from the plane provided in main as the point test vector.

```

bool Sphere::spherePlaneCollision(const Plane& plane) const
{
    float distance = plane.pointDistance(centre);
    return distance <= radius;
}

```

This function “spherePlaneCollision” passes the plane distance from the plane calculated in “pointDistance” through and then detects whether this distance collides with the radius of the sphere.

```
Vector centre1(7.0f, 5.0f, 9.0f);
Sphere sphere1(centre1, 5.0f);

Vector centre2(8.0f, 7.0f, 7.0f);
Sphere sphere2(centre2, 10.0f);

std::cout << "Volume of the sphere 1: " << sphere1.calcVolume() << " cubic units." << std::endl;
std::cout << "Volume of the sphere 2: " << sphere2.calcVolume() << " cubic units." << std::endl;

// check if the spheres collide with each other
if (sphere1.checkVolumeCollision(sphere2)) {
    std::cout << "Mission success. The spheres have collided." << std::endl;
}
else {
    std::cout << "Mission failed. The spheres do not collide." << std::endl;
}
// check if sphere collides with plane
if (sphere1.spherePlaneCollision(plane)) {
    std::cout << "Sphere touches the plane. Nice!" << std::endl;
}
else {
    std::cout << "No touchy with plane, sad." << std::endl;
}
```

When sphere1 with radius 5.0 is located 5 units away from “point on plane” (0, 0, 0) the sphere and plane touch, outputting the if statement when this is correct as seen below.

```
Vector 1 position: 1, 2, 3
Vector 2 position: 2, 3, 4
Distance Between two points: 1.73205
Distance Between two points squared: 3
Point Distance: 5
Point is above plane.
Volume of the sphere 1: 523.598 cubic units.
Volume of the sphere 2: 4188.79 cubic units.
Mission success. The spheres have collided.
Sphere touches the plane. Nice.!
```

If we are to change either the radius of sphere1 or the value of vector y in the point on the plane vector (example 0, -1, 0), then the output would not show a collision, as seen below.

```
Vector 1 position: 1, 2, 3  
Vector 2 position: 2, 3, 4  
Distance Between two points: 1.73205  
Distance Between two points squared: 3  
Point Distance: 5  
Point is above plane.  
Volume of the sphere 1: 268.082 cubic units.  
Volume of the sphere 2: 4188.79 cubic units.  
Mission success. The spheres have collided.  
No touchy with plane, sad.
```

The testing looks to be correct.

Conclusion

We can see how important classes are when coding and implementing calculations by calling on functions and passing correct information through constructors, from one class to another. This allows us to save time in the future, as we can change one value to allow for the result we are looking for, especially when implementing classes into games architecture.

References

References

- [1] Helzner Cynthia. "Euclidean Distance | Calculation, Formula & Examples." [https://study.com/academy/lesson/euclidean-distance-calculation-formula-examples.html#:~:text=Euclidean%20distance%20in%20two%20dimensions%20is%20given%20by%20D%20%3D%20\(%20x,coordinates%20of%20the%20two%20points.https://study.com/academy/lesson/euclidean-distance-calculation-formula-examples.html](https://study.com/academy/lesson/euclidean-distance-calculation-formula-examples.html#:~:text=Euclidean%20distance%20in%20two%20dimensions%20is%20given%20by%20D%20%3D%20(%20x,coordinates%20of%20the%20two%20points.https://study.com/academy/lesson/euclidean-distance-calculation-formula-examples.html) (accessed Dec 27).
- [2] "Introduction to vector math for game developers | GDQuest." https://school.gdquest.com/courses/learn_2d_gamedev_godot_4/to_space_and_beyond/bonus_vectors_101_for_gamedevs#normalizing-a-vector (accessed Dec 27).
- [3] E. "Herman and G. Strang, "Calculus Volume 3," vol. 3, pp.132-150, March 2016. Accessed: Dec 28, 2024, [Online]. Available: https://assets.openstax.org/oscms-prodcms/media/documents/Calculus_Volume_3_-_WEB.pdf

Appendix

Main code

```
Vector v1(1, 2, 3); // Values for first position
std::cout << "Vector 1 position: ";
v1.display(); //displays these values
Vector v2(2, 3, 4); //values for 2nd position
std::cout << "Vector 2 position: ";
v2.display(); //displays the 2nd position values
float distance = v1.pToPDistance(v2); //calculates distance between v1 and v2 using Euclidean
std::cout << "Distance Between two points: " << distance << std::endl; //displays results
float distanceSqr = v1.pToPDistanceSqr(v2); //squares the distance between v1 and v2
std::cout << "Distance Between two points squared: " << distanceSqr << std::endl; //displays distance squared

Vector normal(0, 1, 0);
Vector pointOnPlane(0, 0, 0);
Plane plane(normal, pointOnPlane);

Vector testPoint(0, 5, 0);
std::cout << "Point Distance: " << plane.pointDistance(testPoint) << std::endl;
if (plane.pointDistance(testPoint) > 1)
{
    std::cout << "Point is above plane." << std::endl;
}
else if (plane.pointDistance(testPoint) == 0)
{
    std::cout << "Point is on plane." << std::endl;
}
else
{
    std::cout << "Point is behind plane." << std::endl;
}

Vector centre1(7.0f, 7.0f, 9.0f);
Sphere sphere1(centre1, 3.0f);

Vector centre2(8.0f, 7.0f, 7.0f);
Sphere sphere2(centre2, 10.0f);

std::cout << "Volume of the sphere 1: " << sphere1.calcVolume() << " cubic units." << std::endl;
std::cout << "Volume of the sphere 2: " << sphere2.calcVolume() << " cubic units." << std::endl;

// check if the spheres collide with each other
if (sphere1.checkVolumeCollision(sphere2)) {
    std::cout << "Mission success. The spheres have collided." << std::endl;
}
else {
    std::cout << "Mission failed. The spheres do not collide." << std::endl;
}

// check if sphere collides with plane
if (sphere1.spherePlaneCollision(plane)) {
    std::cout << "Sphere touches the plane. Nice!" << std::endl;
}
else {
    std::cout << "No touchy with plane, sad." << std::endl;
}
```

Vector.cpp and Vector.h

```

5 //construct
6 Vector::Vector(void) {};
7 ~ Vector::Vector(float myx, float myy, float myz)
8 {
9     x = myx;
10    y = myy;
11    z = myz;
12 }
13
14 void Vector::Scale(float s)
15 {
16     x = s * x;
17     y = s * y;
18     z = s * z;
19 }
20
21 void Vector::Add(Vector& v1, Vector& v2)
22 {
23     v1.x = v2.x + v1.x;
24     v1.y = v2.y + v1.y;
25     v1.z = v2.z + v1.z;
26 }
27
28 Vector Vector::getVector(void)
29 {
30     return Vector(x, y, z);
31 }
32
33 void Vector::display()
34 {
35     std::cout << x << " ", << y << " ", << z << std::endl;
36 }
37
38 float Vector::pToDistance(Vector& dist)
39 {
40     float distX = dist.x - x;
41     float distY = dist.y - y;
42     float distZ = dist.z - z;
43     float totalDist = (distX*distX) + (distY*distY) + (distZ*distZ);
44     return std::sqrt(totalDist);
45 }
46
47 float Vector::pToDistanceSqrd(Vector& dist)
48 {
49     {
50         float distX = dist.x - x;
51         float distY = dist.y - y;
52         float distZ = dist.z - z;
53         return ((distX * distX) + (distY * distY) + (distZ * distZ));
54     }
55 }
56
57 // Subtract two vectors
58 Vector Vector::operator-(const Vector& sub) const {
59     return Vector(x - sub.x, y - sub.y, z - sub.z);
60 }
61
62 // Dot product
63 float Vector::dot(const Vector& dot) const {
64     return x * dot.x + y * dot.y + z * dot.z;
65 }
66
67 // Cross product of the two vectors
68 Vector Vector::cross(const Vector& cross) const {
69     return Vector(
70         (y * cross.z) - (z * cross.y), //cross product x
71         (x * cross.z) - (x * cross.z), //cross product y
72         (x * cross.y) - (y * cross.x) // cross product z
73     );
74 }

```

```

1  #ifndef VECTOR_H
2  #define VECTOR_H
3
4  class Vector
5  {
6
7  public:
8
9      //construct
10     Vector(float x, float y, float z);
11     Vector(void);
12     void Scale(float s);
13     void Add(Vector& v1);
14     void Add(Vector& v1, Vector& v2);
15     Vector getVector(void);
16     void display();
17
18     float pToPDistance(Vector& dist);
19     float pToPDistanceSqr(Vector& v1);
20     Vector operator-(const Vector& other) const;
21
22     float dot(const Vector& other) const;
23     Vector cross(const Vector& other) const;
24
25     float x;
26     float y;
27     float z;
28 };
29 #endif
30

```

Plane.cpp and Plane.h

```

1  #include "Plane.h"
2  #include "Vector.h"
3  #include "Plane.h"
4
5  ~Plane::Plane(const Vector& normal, const Vector& pointOnP) {
6      normalise();
7      pointOnP(pointOnP);
8  }
9
10 void Plane::normalise() {
11     float magnitude = sqrt((normal.x * normal.x) + (normal.y * normal.y) + (normal.z * normal.z)); // normalize vector so we have unit length
12     normal.x /= magnitude; //vector values are then divided by the magnitude, giving a normal vector
13     normal.y /= magnitude;
14     normal.z /= magnitude;
15 }
16
17 ~Plane::Plane::pointDistance(const Vector& point) const {
18     Vector difference = point - pointOnP; //This finds the difference between the test point vector and the point on the plane
19     return normal.dot(difference);
20 }
21
22 ~Plane::Plane::pointPosition(const Vector& point) const {
23     float distance = pointDistance(point); //This finds out where the point is compared to the plane, if > 0, its above, if = 0, its on, and if it's less than 0 (or anything else), its behind the plane
24     return distance;
25 }
26
27 ~Plane::Plane::pointCollision(const Vector& point) const {
28     return pointDistance(point); //checks if point is on plane
29 }
30
31 ~Plane::Plane::calculateNormal(const Vector& normal, const Vector& testPoint) { //Dot product calculation which allows us to find the normal of two vectors
32     return normal.cross(testPoint);
33 }
34

```

```

1  #ifndef PLANE_H
2  #define PLANE_H
3
4  #include "Vector.h"
5
6  class Plane {
7  public:
8      Vector normal;
9      Vector pointOnP;
10
11      Plane(const Vector& normal, const Vector& pointOnP);
12
13      void normalise();
14      float pointDistance(const Vector& point) const;
15      float pointPosition(const Vector& point) const;
16
17      bool pointCollision(const Vector& point) const;
18
19      static Vector calculateNormal(const Vector& v3, const Vector& v4);
20 };
21
22 #endif PLANE_H

```

Sphere.cpp and Sphere.h

```

#include <math>
#include <iostream>
#include "Vector.h"
#define PI 3.14159 //defining PI makes it easier to calculate volume later

Sphere::Sphere(const Vector& centre, float radius)
: centre(centre), radius(radius) {}

float Sphere::calcVolume() const
{
    return (4.0f / 3.0f) * PI * std::pow(radius, 3); //calculates the volume, pow raises the power of the radius to 3
}

bool Sphere::checkVolumeCollision(const Sphere& second) const
{
    float distX = second.centre.x - centre.x;
    float distY = second.centre.y - centre.y;
    float distZ = second.centre.z - centre.z;
    float distance = std::sqrt((distX * distX) + (distY * distY) + (distZ * distZ));
    return distance <= (radius + second.radius); //
}

bool Sphere::spherePlaneCollision(const Plane& plane) const
{
    float distance = plane.pointDistance(centre);
    return distance <= radius;
}

```

```

#pragma once
#include <iostream>
#include <cmath>
#include "Vector.h"
#include "Plane.h"

class Sphere
{
public:
    Vector centre; // centre of the sphere
    float radius; // radius of the sphere

    Sphere(const Vector& centre, float radius);

    float calcVolume() const; // calculate volume of the sphere

    bool checkVolumeCollision(const Sphere& second) const; // checks collision with both spheres
    bool spherePlaneCollision(const Plane& plane) const;
};

```

Assignment 2

GRADEMARK REPORT

FINAL GRADE

GENERAL COMMENTS

73 / 100

PAGE 1

PAGE 2

PAGE 3



Comment 1

In academic writing, particularly in scientific and technical papers, the use of the first person (e.g., "I", "we") is generally discouraged as it can make the writing appear subjective and less formal. The focus of academic work should be on the research, analysis, or findings rather than the researcher. Using an objective tone ensures that the argument or evidence takes precedence over personal opinions or experiences, which enhances the perceived credibility and professionalism of the work. For example, instead of saying, "I conducted the experiment," you could write, "The experiment was conducted," which maintains an impersonal and objective stance. To improve in this area, aim to use passive voice judiciously or rephrase sentences to emphasise the action rather than the actor. Additionally, consider adopting phrases like "the research indicates" or "it was observed" to convey findings without self-referencing. Reviewing published papers in your field can provide valuable examples of how to maintain an objective tone while ensuring clarity and precision.

PAGE 4

PAGE 5

PAGE 6

PAGE 7

PAGE 8

PAGE 9

PAGE 10

PAGE 11



Comment 2

lol

PAGE 12

PAGE 13

PAGE 14

PAGE 15



Comment 3

for the add operation perhaps it would have been better to add the class object to the object passed in the argument such as

```
Add(Vector& v1)
{
    this->x += v1.x;
    this->y += v1.y;
    this->z += v1.z;
}
```



Comment 4

another approach to this is to overload the ostream like

```
std::ostream& operator<<(std::ostream& os, const Vector3D& vec) {
    os << "(" << vec.x << ", " << vec.y << ", " << vec.z << ")";
    return os;
}
```



Comment 5

This implementation of operator overloading would have been ideal for the additional operation performed above.

PAGE 16

PAGE 17

PAGE 18
