

resit for asignment

by Teifion Gibson

Submission date: 31-Mar-2025 01:35PM (UTC+0100)

Submission ID: 254311907

File name: 50533_Teifion_Gibson_resit_for_asignment_652458_325934554.docx (937.84K)

Word count: 3084

Character count: 14643

Games Architecture Assignment 2

Student Name : Teifion Gibson
Student Number : 2304294

Contents

Task 1	3
Task 2	5
Task 3	10
Task 4	13
References	16
Appendix	17
Main ccp file	17
Vector Header file	18
Vector ccp file	18
Plane Header file	19
Plane ccp file	20
Sphere Header file	22
Sphere ccp file	23

Task 1

The assignment brief outlines asks us to create a point to point distance calculator function within the premade vector class. To do this we need to create a point to point distance function within the Class file, this function is called PToPDistance. additionally the brief tells us to also implement a function to square this result, therefore we are also adding a additional function within the same class called PToPDistanceSqrt.

```
vector<vector> getVector(void);
float PToPDistance(Vector& P1, Vector& P2);
float PToPDistanceSqrt(Vector& P1, Vector& P2);
```

Fig 1 : Vector class added functions

Next We implement the Functions into a CCP file, first we implement the PToPDistance function, then using the Method outlined from the assignment brief to get the distance between the two points, we then return the calculated variable.

```
float Vector::PToPDistance(Vector& P1, Vector& P2)
{
    float DistBetween = 0;
    DistBetween = pow((P1.x - P2.x), 2) + pow((P1.y - P2.y), 2) + pow((P1.z - P2.z), 2);

    return DistBetween;
}
```

Fig 2 : PToPDistance function

This is repeated within the PToPDistanceSqrt Function, we first calculate the distance between, and then we then use the sqrt function to square the result. We then return this variable.

```
float Vector::PToPDistanceSqrt(Vector& P1, Vector& P2)
{
    float DistBetween=0;
    DistBetween = pow((P1.x - P2.x), 2) + pow((P1.y - P2.y), 2) + pow((P1.z - P2.z), 2);
    DistBetween = sqrt(DistBetween);
    return DistBetween;
}
```

Fig 3 : PToPDistanceSqrt function

Finally we run the code in the main file to confirm it functions correctly. As shown in the image below the code when ran calculates the distance between the two vector points and returns the

distance between it first, then the code returns the square of the distance between.

```
#include <iostream>
#include "Vector.h"

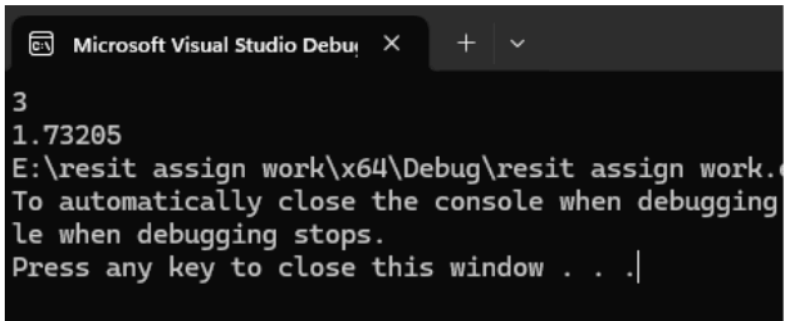
void main()
{
    float dist;
    Vector Vector1(1, 2, 3); //creation of first vector
    Vector Vector2(2, 3, 4);

    dist = Vector1.PToPDistance(Vector2, Vector1);
    std::cout << dist << "\n";

    dist = Vector1.PToPDistanceSqrt(Vector2, Vector1);

    std::cout << dist;
```

Fig 4 : Main CCP file



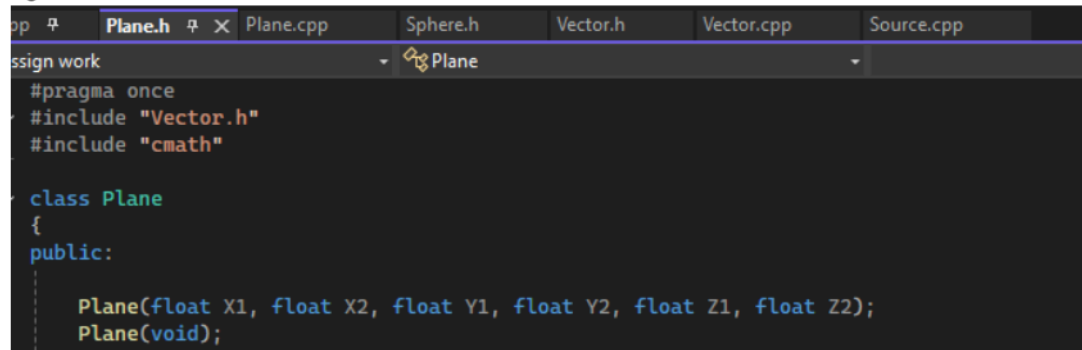
The screenshot shows the Microsoft Visual Studio Debug Console window. The title bar reads "Microsoft Visual Studio Debug Console". The console output is as follows:

```
3
1.73205
E:\resit assign work\x64\Debug\resit assign work.
To automatically close the console when debugging
le when debugging stops.
Press any key to close this window . . .|
```

Fig 5 : Code ran calculations

Task 2

Task 2 outlines the creation of a plane class. This class is going to be created using two vectors for two points in 3d space. First we define the class within the header file. Next we define the constructors. This function is called plane, and defines two points X1, Y1 and Z1, this is done again for the second XYZ values with 2 instead of 1.



```

#pragma once
#include "Vector.h"
#include "cmath"

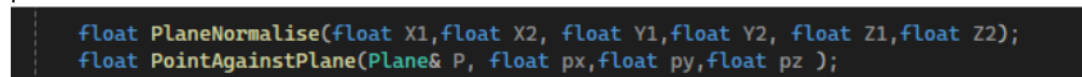
class Plane
{
public:
    Plane(float X1, float X2, float Y1, float Y2, float Z1, float Z2);
    Plane(void);

```

Fig 6 : Plane Class constructors

Next the vector class declares the plane class functions. The first function we define is the plane normalise function, this function will be used to find the normal vector and value. This function will take the XYZ 1 and XYZ 2. This is because this function also can be used to define the plane. This function will return the normal, as well as define a 3rd vector for the plane.

The point against plane function will be used to find the distance that a point is from the plane. This function will take in the plane class and an additional point, this is the point we will be using to find the distance from the plane. This function will return the distance of the point from the plane.



```

float PlaneNormalise(float X1, float X2, float Y1, float Y2, float Z1, float Z2);
float PointAgainstPlane(Plane& P, float px, float py, float pz );

```

Fig 7 : Plane Normalise and Point against Plane functions defined

The last function we are creating within this class is the Plane above or below function. This is a function which will take in the Plane and the points distance and return whether the point is above, on or below the plane.

Finally within the plane we initialize the vector that we will need. First is the preset vectors which are x1, x2, y1, y2, z1 and z2. Then we define the normal vector which is stored as A, B and C,

Finally we then create a Variable for the normal value.

```
float PlaneAboveorBelow(Plane& P, float PointDistance);

// initial vector
float x1,x2;
float y1,y2;
float z1,z2;

//calc vector
float A, B, C;

//normal
float Normal;
```

Fig 8 : plane class Variables declaration

Next we are outlining the Plane CCP file. This file will be where the plane classes functions are implemented.

The first item we will be discussing is the plane constructor and destructor. All these functions do is write the two points into the Plane class and define the plane object. This object will be called when using the point against plane function.

```
#include "Plane.h"

Plane::Plane(float X1, float X2, float Y1, float Y2, float Z1, float Z2)
{
    x1 = X1; x2 = X2;
    y1 = Y1; y2 = Y2;
    z1 = Z1; z2 = Z2;
}

Plane::Plane(void){}
```

Fig 9 : Plane constructor and destructor ccp file

The next function is the plane normalization function. This function takes in two vectors, and calculates the plane's normal vector and the plane normal. First the function calculates the XY and Z of the normal vector. This is called a, b, c within the code. This is then saved into the plane class.

Next the plane normalises the vector. The following functions are derived from [2], this reference is a video that outlines how to find a plane normal then normalise the vector. This is done in the code by using $Y1 \cdot Z2 - Z1 \cdot Y2$, in this case this finds the X normal position. The Axis used is different for Each axis vector. To find normal y, replace Y with z and Z with X, and to find z replace Y with x and Z with y. The code then writes these positions to the ABC within the plane class.

```

float Plane::PlaneNormalise(float X1, float X2, float Y1, float Y2, float Z1, float Z2)
{
    x1 = X1; x2 = X2;
    y1 = Y1; y2 = Y2;
    z1 = Z1; z2 = Z2;

    float a, b, c;
    // calculates normal vector
    a = Y1 * Z2 - Z1 * Y2; // a = x
    b = Z1 * X2 - X1 * Z2; // b = y
    c = X1 * Y2 - Y1 * X2; // c = z

    //saves normal vector in class
    A = a;
    B = b;
    C = c;
}

```

Fig 10 : Plane normalise function finding normal vector

Finally we find the normal by multiplying the normal vector by the first vector - the second vector, this is done for each point. These are added together to return a single value. This value is the plane's normal. The code then returns this Value.

```

// normalise vector
Normal = a * (X1 - X2) + b * (Y1 - Y2) + c * (Z1 - Z2);

return Normal;
}

```

Fig 11 : Plane normalise function calculation of normal.

The next part of the Plane header file is the point against plane function. This function finds the distance of the point from the plane. First we create some temporary values for a, b and c then we need to increase a, b and c using a power of 2. This is then written into the temp a,b,c values. Next we add the points xyz values, and the normal. Then divide it by the square root of a b and c [1]. This gives us the distance, which is then returned by the function.

```

float Plane::PointAgainstPlane(Plane& P, float px, float py, float pz)
{
    //this equation is for this is Ax + By + Cz + D DEVIDE BY root(A^2+B^2+C^2) the first ABC is the point.
    // the second ABC is the calculated point D is the normal this returns points distance
    float a, b, c; float distance = 0;
    a = pow(P.A, 2); b = pow(P.B, 2); c = pow(P.C, 2);
    distance = px + py + pz + P.Normal / sqrt(a + b + c);

    return distance;
}

```

Fig 12 : Point against plane function code

The last function of the plane class is the point above or below the plane function. This function Uses the plane class and a float which is the point distance. Then it is a simple if statement which if the point distance is greater than 0 it prints above, if it is equal to 0 it prints on plane and otherwise the function prints below plane.

```

float Plane::PlaneAboveorBelow(Plane& P, float PointDistance)
{
    if (PointDistance > 0)
    {
        std::cout << "Above";
    }
    else if (PointDistance == 0)
    {
        std::cout << "on plane";
    }
    else
    {
        std::cout << "below plane";
    }

    return 0;
}

```

Fig 13 : Plane above or below function

shown below is the classes implementation into the main CCP file. First we include the Plane header file and the Vector header files, next we define the vectors and the base variables that are needed within the code.

Next for point to plane distance we need to declare the plane and input vector 1 and 2 into the variables.

Then we use the normalise plane function to find the normal vector of the plane.

Finally we then use the plane point against the plane function, so this finds the distance that the point is away from the plane. Finally we call the above or below function to check if the point is above or below the plane.

```

void main()
{
    float dist, diam1 = 5, diam2 = 5, Distance;
    bool collision = 0;
    Vector Vector1(1, 1, 2); //creation of first vector
    Vector Vector2(2, 3, 2);
    Vector Vector3(20, 4, -10);
}

```

Fig 14 : main function variable declarations

```

//Point to plane distance
//declare plane
Plane P(Vector1.x, Vector2.x , Vector1.y,Vector2.y, Vector1.z, Vector2.z);
//Normalise plane
P.PlaneNormalise(Vector1.x, Vector2.x, Vector1.y, Vector2.y, Vector1.z, Vector2.z);
//Find point to plane distance
Distance = P.PointAgainstPlane(P, Vector3.x, Vector3.y, Vector3.z);
//print distance
std::cout << Distance;
P.PlaneAboveorBelow(P, Distance);

```

Fig 15 : point to plane distance calculation within main ccp file

As shown below are the printed values of the distance of the point against the plane and whether it is above or below the plane.

```

14Above
E:\resit assign work\x6

```

Fig 16 : returned calculations

Task 3

Task 3 asks us to create a sphere class that can store the information about a sphere, and then check if the spheres are colliding.

The first thing we'll be discussing is the creation of the sphere header file. This header file will contain the needed functions and values for the sphere object. Within the class we have created two functions. The first one is the sphere object constructor and sphere object destructor.

The next function is the sphere collision check, this takes in two sphere objects, and then returns whether the spheres are close enough to be colliding. Finally the class exclusive variables are the xyz and the diameter of the sphere.

```
#pragma once
#include "Vector.h"
#include "cmath"

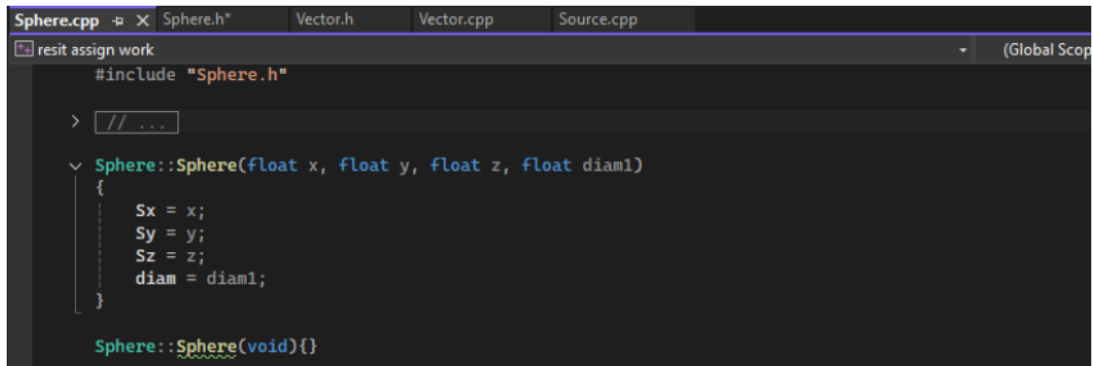
class Sphere
{
public:
    //const
    Sphere(float x, float y, float z, float diam1);
    Sphere(void);

    // ...

    bool SphereCollisionCheck(Sphere& S1, Sphere& S2);
    float Sx;
    float Sy;
    float Sz;
    float diam;
};
```

Fig 17 : Sphere class header file

The first part of the CCP file for the sphere class is the sphere object constructor. This writes the inputted xyz and diameter values and attaches them to the sphere object. The sphere destructor is left blank.



```

Sphere.cpp  X  Sphere.h*  Vector.h  Vector.cpp  Source.cpp
resit assign work  (Global Scop

#include "Sphere.h"

> // ...

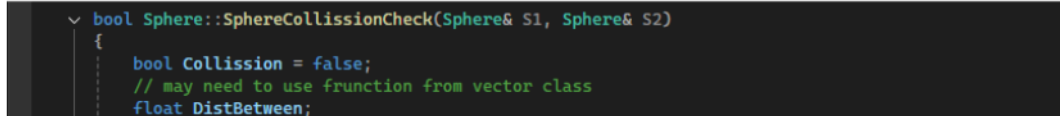
Sphere::Sphere(float x, float y, float z, float diam1)
{
    Sx = x;
    Sy = y;
    Sz = z;
    diam = diam1;
}

Sphere::~Sphere(void){}

```

Fig 18 : sphere ccp file constructor and destructor

The next part of the CCP file is the sphere collision check function. This function takes in two sphere objects, these are S1 and S2. Next we create a boolean within the function called collision and default it to false. This will be updated and returned. Then declare a float called distance between.



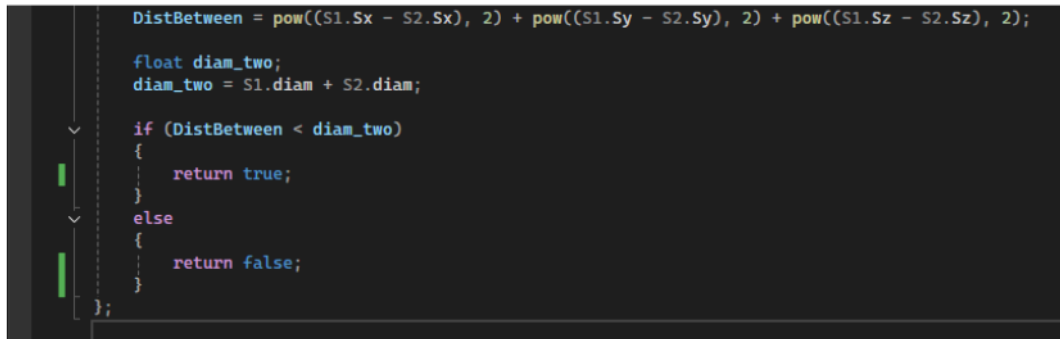
```

bool Sphere::SphereCollisionCheck(Sphere& S1, Sphere& S2)
{
    bool Collision = false;
    // may need to use function from vector class
    float DistBetween;
}

```

Fig 19 : Sphere collision check function value declaration

Now for the calculation we first find the distance between the two vectors using the distance between two points vectors. Next we float a value called diam two, this will be the two sphere objects diameters added together, then utilizing a if statement, if the distance between is less than the diam two value it will return true, meaning a collision occurred otherwise the value returns false.



```

DistBetween = pow((S1.Sx - S2.Sx), 2) + pow((S1.Sy - S2.Sy), 2) + pow((S1.Sz - S2.Sz), 2);

float diam_two;
diam_two = S1.diam + S2.diam;

if (DistBetween < diam_two)
{
    return true;
}
else
{
    return false;
}
};

```

Fig 20 : Sphere collision check calculation and collision check

Now for the code implementation within the main file we first need to include the sphere class and vector class into the CCP file. Next we float the needed variables for the sphere object, these are the collision Boolean, the two diameter and two vectors.

```
void main()
{
    float dist, diam1 = 5, diam2 = 5;
    bool collision = 0;
    Vector Vector1(0, 0, 1); //creation of first vector
    Vector Vector2(2, 3, 1);
```

Fig 21 : Main ccp variable declaration

Next we Create the two sphere objects, and name them Sphere 1 and Sphere 2. The input vectors xyz and diameter are defined to the two spheres. Next we run the collision check function and write true or false to the collision check.

```
//Sphere to sphere collision

Sphere Sphere1(Vector1.x, Vector1.y, Vector1.z, diam1);
Sphere Sphere2(Vector2.x, Vector2.y, Vector2.z, diam2);

//is not updateing returned variable
collision = Sphere1.SphereCollisionCheck(Sphere1, Sphere2);

std::cout << collision;
```

Fig 22 : Sphere to sphere collision check function and value print

Finally the code prints the collision boolean. If it is 0 (false) it means a collision does not occur and if 1 (true) it means the spheres were close enough for a collision to occur.

```
1
E:\resit as
```

Fig 23 : collision check output.

Task 4

Task 4 outlines the creation of two functions, the first function this task asks us to create is a function that checks if a point has collided with the plane. First we need to create the function into the plane header file

```
bool PointCollisionCheck(Plane& P, float px, float py, float pz);
```

Fig 24 : point collision check function in class file

For this we call the point against function to find the distance the point is from the plane, we will then use this to check if the point collides with the plane.

```
bool Plane::PointCollisionCheck(Plane& P, float px, float py, float pz)
{
    float distance;
    distance = Plane::PointAgainstPlane(P, px, py, pz);
```

Fig 25 : point collision check code 1

Then if the distance is equal to 0 the function prints the word collision, as if it is at 0 the point is on the plane and therefore has collided and returns true. Otherwise, no collision is printed and the function returns false. This is because if it is above 0 the point is above the plane, and if it is below 0 the point is below the plane.

```
    if (distance == 0)
    {
        std::cout << "COLLISSION OCCURED";
        return true;
    }
    else
    {
        std::cout << "No Collission";
        return false;
    }
}
```

Fig 26 : point collision check code 2

To utilise this function within the main ccp file, we need to call the function as shown below.

```
P.PointCollisionCheck(P, Vector3.x, Vector3.y, Vector3.z);
```

Fig 27 : point collision check called in ccp file

```
No Collission
```

Fig 28 : point collision check output

The second part of task 4 asks us to now expand this into another function within the sphere class that checks for collision between a sphere and a plane. This function is within the sphere class. And takes input of a sphere object, a point, called abc (this is because the abc needs to be the plane's normal vector, which in the plane class is abc), and the normal of the plane.

```
float diam;
bool SphereAgainstPlane(Sphere& S, float a, float b, float c, float normal);
};
```

Fig 29 : sphere against plane function

Therefore first we need to find the distance of the plane, multiply the normal vector by the power of 2, then we divide the square root of the powered by two numbers by the position of the sphere and the normal added together. This gives the distance between the plane and the sphere.

```
bool Sphere::SphereAgainstPlane(Sphere& S, float a, float b, float c, float normal)
{
    float X, Y, Z;
    X = pow(a, 2); Y = pow(b, 2); Z = pow(c, 2);
    float distance = (S.Sx + S.Sy + S.Sz + normal / sqrt(X + Y + Z));

    float checkSp = distance - S.diam;

    if (checkSp < 0)
    {
        std::cout << "Collission!";
        return true;
    }
    else
    {
        std::cout << "no.";
        return false;
    }
}
```

Fig 30 : Sphere against plane function code

Shown below is the function being utilised within the main ccp file, and returning whether it has collided or not.

```
float dist, //diam1 = 5,
diam2 = 1000, Distance;
bool collission = 0;
Vector Vector1(1, 1, 0); //creation of first vector
Vector Vector2(1, 1, 1);
Vector Vector3(10, 10, 10);
```

Fig 31 : main cpp file variables

```
//declare plane
Plane P(Vector1.x, Vector2.x, Vector1.y, Vector2.y, Vector1.z, Vector2.z);
//Normalise plane
P.PlaneNormalise(Vector1.x, Vector2.x, Vector1.y, Vector2.y, Vector1.z, Vector2.z);
```

Fig 32 : Plane creation and normalise in main file

```
Sphere Sphere1(Vector3.x, Vector3.y, Vector3.z, diam2);  
Sphere1.Sphere::SphereAgainstPlane(Sphere1, P.A, P.B, P.C, P.Normal);
```

Fig 33 : Sphere object creation and Sphere against plane function call

Collission!

Fig 34 : Sphere against plane function result

References

[1]

"Distance Between Point and Plane - Formula, Proof, Examples," *Cuemath*.
<https://www.cuemath.com/geometry/distance-between-point-and-plane/>

[2]

Khan Academy, "Normal vector from plane equation | Vectors and spaces | Linear Algebra | Khan Academy," *Youtu.be*, 2025. <https://youtu.be/gw-4wltP5tY?si=BZrCapl42TMLmN9Q>
(accessed Mar. 31, 2025).

Appendix

Main ccp file

```
#include <iostream>
#include "Vector.h"
#include "Plane.h"
#include "Sphere.h"

void main()
{
    float dist, //diam1 = 5,
    diam2 = 1000, Distance;
    bool collision = 0;
    Vector Vector1(1, 1, 0); //creation of first vector
    Vector Vector2(1, 1, 1);
    Vector Vector3(10, 10, 10);
    /*dist = Vector1.PToPDistance(Vector2, Vector1);
    std::cout << dist << "\n";

    dist = Vector1.PToPDistanceSqrt(Vector2, Vector1);

    std::cout << dist;*/

    //Sphere to sphere collision

    //Sphere Sphere1(Vector1.x, Vector1.y, Vector1.z, diam1);
    //Sphere Sphere2(Vector2.x, Vector2.y, Vector2.z, diam2);

    //collision = Sphere1.SphereCollisionCheck(Sphere1, Sphere2);

    //std::cout << collision;

    //Point to plane distance
    //declare plane
    Plane P(Vector1.x, Vector2.x, Vector1.y, Vector2.y, Vector1.z, Vector2.z);
    //Normalise plane
    P.PlaneNormalise(Vector1.x, Vector2.x, Vector1.y, Vector2.y, Vector1.z, Vector2.z);
    //Find point to plane distance
    Distance = P.PointAgainstPlane(P, Vector3.x, Vector3.y, Vector3.z);
    //print distance
    std::cout << Distance;
    P.PlaneAboveorBelow(P, Distance);
```

```
//P.PointCollisionCheck(P, Vector3.x, Vector3.y, Vector3.z);
Sphere Sphere1(Vector3.x, Vector3.y, Vector3.z, diam2);

Sphere1.Sphere::SphereAgainstPlane(Sphere1, P.A, P.B, P.C, P.Normal);
```

```
}
```

Vector Header file

```
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);
    float PToPDistance(Vector& P1, Vector& P2);
    float PToPDistanceSqrt(Vector& P1, Vector& P2);

    //private: //these are currently public – should they be private?
    float x;
    float y;
    float z;

};
```

Vector ccp file

```
#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);
}
float Vector::PToPDistance(Vector& P1, Vector& P2)
{
    float DistBetween = 0;
    DistBetween = pow((P1.x - P2.x), 2) + pow((P1.y - P2.y), 2) + pow((P1.z - P2.z), 2);

    return DistBetween;
}
float Vector::PToPDistanceSqrt(Vector& P1, Vector& P2)
{
    float DistBetween=0;
    DistBetween = pow((P1.x - P2.x), 2) + pow((P1.y - P2.y), 2) + pow((P1.z - P2.z), 2);
    DistBetween = sqrt(DistBetween);
    return DistBetween;
}
void Vector::Add(Vector& v1)
{
    x = x + v1.x;
    y = y + v1.y;
    z = z + v1.z;
}

```

Plane Header file

```

#pragma once
#include "Vector.h"
#include "cmath"

```

```

class Plane
{

```

public:

```

Plane(float X1, float X2, float Y1, float Y2, float Z1, float Z2);
Plane(void);

float PlaneNormalise(float X1, float X2, float Y1, float Y2, float Z1, float Z2);
float PointAgainstPlane(Plane& P, float px, float py, float pz);
float PlaneAboveorBelow(Plane& P, float PointDistance);

// initial vector
float x1, x2;
float y1, y2;
float z1, z2;

// calc vector
float A, B, C;

// normal
float Normal;
bool PointCollisionCheck(Plane& P, float px, float py, float pz);

```

};

Plane ccp file

```

#include <iostream>
#include "Plane.h"

```

```

Plane::Plane(float X1, float X2, float Y1, float Y2, float Z1, float Z2)
{
    x1 = X1; x2 = X2;
    y1 = Y1; y2 = Y2;
    z1 = Z1; z2 = Z2;
}

```

```

Plane::Plane(void){}

```

```

float Plane::PlaneNormalise(float X1, float X2, float Y1, float Y2, float Z1, float Z2)
{
    x1 = X1; x2 = X2;
    y1 = Y1; y2 = Y2;
    z1 = Z1; z2 = Z2;

    float a, b, c;

```

```

// calculates normal vector
a = Y1 * Z2 - Z1 * Y2; // a = x
b = Z1 * X2 - X1 * Z2; // b = y
c = X1 * Y2 - Y1 * X2; // c = z

//saves normal vector in class
A = a;
B = b;
C = c;

// normalise vector
Normal = a * (X1 - X2) + b * (Y1 - Y2) + c * (Z1 - Z2);

return Normal;
}

float Plane::PointAgainstPlane(Plane& P, float px, float py, float pz)
{
    //this equation is for this is Ax + By + Cz + D DEVIDE BY root(A^2+B^2+C^2) the first
    ABC is the point.
    // the second ABC is the calculated point D is the normal this returns points distance
    float a, b, c; float distance = 0;
    a = pow(P.A, 2); b = pow(P.B, 2); c = pow(P.C, 2);
    distance = px + py + pz + P.Normal / sqrt(a + b + c);

    return distance;
}

float Plane::PlaneAboveorBelow(Plane& P, float PointDistance)
{
    if (PointDistance > 0)
    {
        std::cout << "Above";
    }
    else if (PointDistance == 0)
    {
        std::cout << "on plane";
    }
    else
    {
        std::cout << "below plane";
    }
}

```

```

        return 0;
    }

    bool Plane::PointCollisionCheck(Plane& P, float px, float py, float pz)
    {
        float distance;
        distance = Plane::PointAgainstPlane(P, px, py, pz);

        if (distance >= 0)
        {
            std::cout << "COLLISSION OCCURED";
            return true;
        }
        else
        {
            std::cout << "No Collission";
            return false;
        }
    }
}

```

Sphere Header file

```

#pragma once
#include "Vector.h"
#include "cmath"

class Sphere

{
public:

    //const
    Sphere(float x, float y, float z, float diam1);
    Sphere(void);

    //this would be better
    //float DeclareSphere(Vector& (point), float diam);

    bool SphereCollisionCheck(Sphere& S1, Sphere& S2);
    float Sx;
    float Sy;
    float Sz;
    float diam;
    bool SphereAgainstPlane(Sphere& S, float a, float b, float c, float normal);

```

```
};
```

Sphere ccp file

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

//better if i can make work
//float Sphere::DeclareSphere(Vector& (point), float mydiam){    Sx = point.x;  Sy = point.y;
    Sz = point.z;  diam = mydiam;};

Sphere::Sphere(float x, float y, float z, float diam1)
{
    Sx = x;
    Sy = y;
    Sz = z;
    diam = diam1;
}

Sphere::Sphere(void){}

bool Sphere::SphereCollissionCheck(Sphere& S1, Sphere& S2)
{
    //bool Collission = false;
    // may need to use frunction from vector class
    float DistBetween;
    DistBetween = pow((S1.Sx - S2.Sx), 2) + pow((S1.Sy - S2.Sy), 2) + pow((S1.Sz -
S2.Sz), 2);

    float diam_two;
    diam_two = S1.diam + S2.diam;

    if (DistBetween < diam_two)
    {
        return true;
    }
    else
    {
        return false;
    }
}

bool Sphere::SphereAgainstPlane(Sphere& S, float a, float b, float c, float normal)
{

```

```
float X, Y, Z;  
X = pow(a, 2); Y = pow(b, 2); Z = pow(c, 2);  
float distance = (S.Sx + S.Sy + S.Sz + normal / sqrt(X + Y + Z));  
  
float checkSp = distance - S.diam;  
  
if (checkSp < 0)  
{  
    std::cout << "Collision!";  
    return true;  
}  
else  
{  
    std::cout << "no.";  
    return false;  
}  
}
```

resit for assignment

GRADEMARK REPORT

FINAL GRADE

60/100

GENERAL COMMENTS

Great work Teifion.

A few points to mention about your report; In academic report writing, the use of personal pronouns such as "we" is generally avoided to maintain an objective and formal tone. Academic writing aims to present information, arguments, and findings in a way that is neutral and evidence-based, rather than reflective of personal opinion or involvement. By avoiding "we", the focus remains on the subject matter rather than the author, which enhances the credibility and impartiality of the report. This convention also aligns with the broader academic goal of producing work that can be critically evaluated on the basis of logic and evidence, rather than perceived personal bias or collective authorship.

Including references that confirm how the equations work would have significantly strengthened the report by demonstrating that the mathematical models used are grounded in established theory or prior research. Such references provide validation for the methods applied, showing that the equations are not arbitrary but are supported by academic or scientific precedent. This not only adds credibility to the analysis but also allows readers to verify the correctness and relevance of the approach. Furthermore, citing authoritative sources helps place the work within the broader context of the field, enabling others to trace the origins of the equations and understand their intended use, assumptions, and limitations.

Since the full code has been included in the appendix, it would have been more effective to reference it directly from the main body of the report rather than inserting a separate screenshot for each discussed function. Referencing the appendix maintains a cleaner and more concise structure within the main text, allowing the reader

to focus on the analysis or explanation without being interrupted by large visual inserts. It also ensures that the code is presented consistently in one place, which is easier to read and more practical for those who may want to review the implementation in its entirety. This approach is more in line with academic conventions and helps to avoid redundancy while still giving the reader access to the relevant technical detail.

PAGE 1

PAGE 2

PAGE 3

PAGE 4

PAGE 5

PAGE 6

PAGE 7



Comment 1

An issue here with the order of operations.

here it adds the coordinates directly and applies the division **only to P.Normal**, due to operator precedence. It should be:

$$\text{distance} = (P.A * px + P.B * py + P.C * pz + P.Normal) / \text{sqrt}(a + b + c);$$

PAGE 8

PAGE 9

PAGE 10

PAGE 11

PAGE 12



Comment 2

the same order of operations bug here