

Assignment 2 Report

by Sam Mifsud

Submission date: 02-Jan-2025 10:02AM (UTC+0000)

Submission ID: 247004909

File name: 50540_Sam_Mifsud_Assignment_2_Report_628793_146283186.docx (564.13K)

Word count: 3082

Character count: 16362

Assignment 2 Report

GAMES ARCHITECTURE AND MECHANICS
(ACCB5017)
SAMUEL MIFSUD

Table of Contents

Vector class changes	2
Task 1 – Distance Between 2 Points	3
Task 2 – Plane Class	8
Task 3 – Sphere Class with Collision.....	12
Task 4 – Point to Plane and Sphere to Plane Collision	15
Appendix	16
Main CPP file	16
Vector.h file	17
Vector.cpp file	18
Plane.h file	21
Plane.cpp file	21
Sphere.h file	23
Sphere.cpp file.....	24
References	26

Vector class changes

The vector class provided in the assignment brief includes comments asking if the x, y and z attributes in the class should be private instead of public. Private class attributes are an example of the encapsulation concept of programming. In encapsulation, you restrict access to certain data, limiting access only to or through its parent and its methods [1], in this case the vector class. This allows you to control exactly how and where data can be modified or read. 

Using private variables has several advantages. In some situations, it may be necessary to prevent writing to certain attributes at all from outside of the class, such as for any attribute used in calculations, to prevent them from being incorrectly modified. In many cases using methods for changing and reading variables could enforce rules for how they are used. And in other examples, such as the normalized vector for the plane class, can allow for operations to be performed before they are saved, in this case normalizing the vector.

In the current implementation of the vector class there is not much of an advantage to using private attributes for x, y and z as the only things that are going to be done with them are getting and setting. However, as it is considered good practice to use private variables where possible I did so and created getter and setter methods for each attribute.

```
#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);
    float getX(); float getY(); float getZ();
    void setX(float newVal); void setY(float newVal); void setZ(float newVal);
private:
    float x;
    float y;
    float z;
};

#endif
```

Fig 1. The modified vector class header file, adapted from the version in the assignment brief, showing the getter and setter methods I created, own work

```

//Getters, return the relevant value
float Vector::getX()
{
    return this->x;
}
float Vector::getY()
{
    return this->y;
}
float Vector::getZ()
{
    return this->z;
}
//Setters, change the relevant value
void Vector::setX(float newVal)
{
    this->x = newVal;
}
void Vector::setY(float newVal)
{
    this->y = newVal;
}
void Vector::setZ(float newVal)
{
    this->z = newVal;
}

```

Fig 2. The code for the getter and setter methods I created for the vector class, own work

Task 1 – Distance Between 2 Points

To calculate the distance between 2 points in 3-dimensional space Pythagoras' Theorem can be used [2]. The distances on each axis can be considered as a cube, with each side length of the cube being equal to the corresponding axis' distance, and the two points being opposite corners of the cube.

In the diagram below the 2 vectors, V1 and V2 are marked on opposite corners of the cube. The length of the cube is the x axis distance between them, width the y axis and height the z axis, with lengths of 3, 2 and 1 respectively.

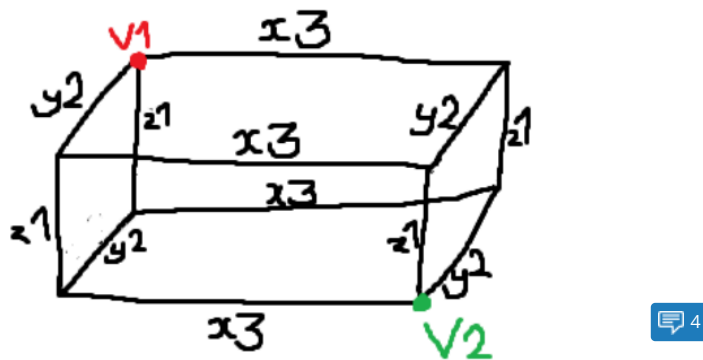


Fig 3. Visualisation of 2 vectors as a cube with side lengths of axis distances, own work

In the diagram below, the light blue line (d) represents the shortest distance between V1 and V2, a direct straight line between the two vectors. This is the distance the point-to-point calculation function needs to calculate.

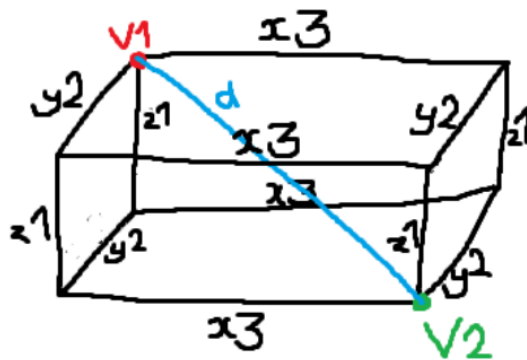


Fig 4. Visualisation of the distance (d) to be calculated between the two vectors, own work

If we consider the distance as being the hypotenuse of a right-angled triangle (shown in dark blue in the diagram below) we can use Pythagoras' Theorem to calculate its length. The height of the triangle is the distance z, or 1, but the length is currently unknown.

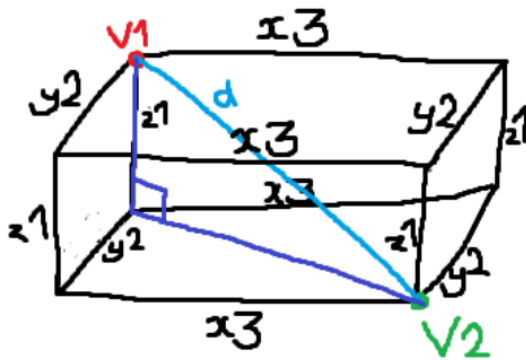


Fig 5. Visualisation of a right-angled triangle with distance d as the hypotenuse, own work

To calculate the unknown length for the dark blue triangle, we can consider it as the hypotenuse of another right-angled triangle (shown in purple in the diagram below) with height y (2) and length x (3). Using Pythagoras' Theorem with this triangle, we can calculate the unknown length of the blue triangle.

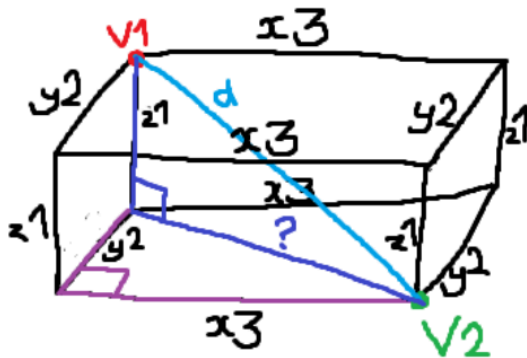


Fig 6. Visualisation of a second triangle (purple) which can be used to calculate the unknown length of the blue triangle, own work

Using Pythagoras' Theorem ($C^2 = A^2 + B^2$) we can calculate the length of line $?$. Substituting in the values from the diagram, we get the equation $?^2 = 3^2 + 2^2$, so $?^2 = 13$. We do not need to calculate $?$ at this stage, as the next stage is to find $?^2 + z^2$ to calculate distance d^2 . So, to calculate distance d we need the square root of 14, which comes out to 3.7416...

To simplify this equation into one step, we can say that d is the square root of $(x^2 + y^2 + z^2)$ [2].

I implemented this as a method in the vector class which takes a reference to the second vector to measure the distance to, and a Boolean value to allow the program to calculate both distance and squared distance, as requested by the brief. The method function and declaration are shown in the diagrams below:

```
float Vector::p2pDistanceTo(Vector& vector2, bool squared)
{
    //Formula for calculating distance in 3D space is sqrt(dx^2 + dy^2 + dz^2), d=distance of (axis), for value squared omit the sqrt portion.
    //3D Pythagorean Theorem - X^2 + Y^2 + Z^2 = d^2, d = above equation
    //Calculating the distance in each axis
    float dx = vector2.getX() - this->getX();
    float dy = vector2.getY() - this->getY();
    float dz = vector2.getZ() - this->getZ();

    //Distance Squared
    float dTotal = dx * dx + dy * dy + dz * dz;

    if (squared) //If distance is squared the function will return and exit and not run the second return, otherwise it will run the second return only.
    {
        return dTotal; //The function exits upon a return so will not run the code after this line, which eliminates the need for an else statement
    }
    return sqrt(dTotal); //If distance is squared the previous return will be skipped and this one will run instead
}
```

Fig 7. Method function to calculate point to point distance and point to point distance squared for 2 vectors, own work

```
float p2pDistanceTo(Vector& vector2, bool squared = false); //squared is set to false by default so it does not have to be set unless the user wants the value squared
```

Fig 8. Method declaration in the vector.h file, own work

To use this method, I created 2 testing vectors in the main function and called it through vector1, with vector2 as the input, both with and without the squared argument, printing their outputs each time.

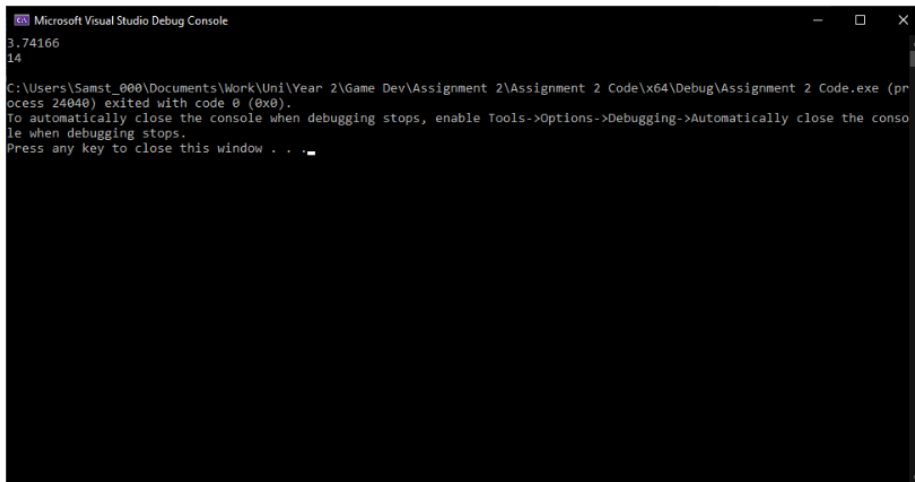
```
#include <iostream>
#include "vector.h"

int main()
{
    Vector vector1(1, 2, 3);
    Vector vector2(2, 4, 6);

    std::cout << vector1.p2pDistanceTo(vector2) << std::endl; //P2P distance calculation
    std::cout << vector1.p2pDistanceTo(vector2, true) << std::endl; //P2P distance squared calculation
}
```

Fig 9. Testing code in the main function for the p2pDistanceTo function, own work

The outputs of the function match the results calculated manually in the visualised example above, as shown in the console output below.



```
Microsoft Visual Studio Debug Console
3.74166
14
C:\Users\Samst_000\Documents\Work\Uni\Year 2\Game Dev\Assignment 2\Assignment 2 Code\x64\Debug\Assignment 2 Code.exe (process 24040) exited with code 0 (0x0).
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when debugging stops.
Press any key to close this window . . .
```

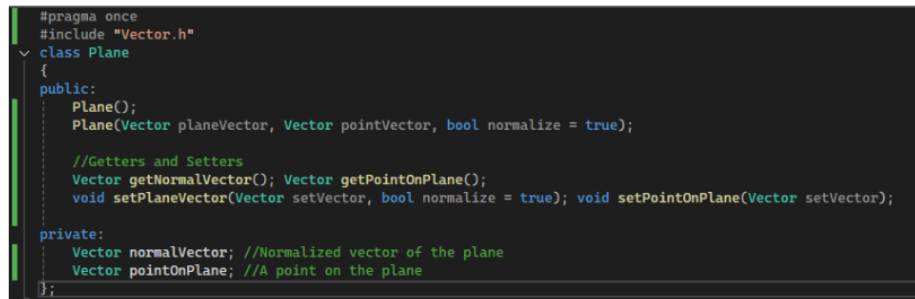
Fig 10. Output of the testing code, showing the results of the point-to-point distance and point to point distance squared calculations, own work

The code can handle negative numbers, or vector1 being smaller than vector2, without issue. This is because, when the distances are squared, they will always produce a positive number due to negative numbers multiplied by negative numbers producing positive numbers.

Task 2 – Plane Class

First, the plane requires a normal vector. A normal vector, also known as a unit or direction vector [3] is a vector with a magnitude (or distance) of 1 unit in the same direction as the un-normalized vector. To normalize a 3D vector like the ones used in this assignment you divide the x, y and z coordinates by the magnitude, which is the square root of $(X^2 + Y^2 + Z^2)$. A normalized vector shows how much each value in the vector, in this case the x, y and z coordinates, change over 1 unit of distance [3] [4] [5] [6].

To begin with I created a basic class for the plane. I gave it the plane normal and point on plane attributes and added getter and setter methods.

A screenshot of a code editor showing the header file for a Plane class. The code is written in C++ and includes a pragma once directive, an include for Vector.h, and the class definition for Plane. The class has two public constructors: a default constructor and one that takes a plane vector, a point vector, and a bool for normalization (defaulting to true). It also has public getter and setter methods for the normal vector and point on plane. The private section contains two member variables: a normalized vector and a point on the plane.

```
#pragma once
#include "Vector.h"
class Plane
{
public:
    Plane();
    Plane(Vector planeVector, Vector pointVector, bool normalize = true);

    //Getters and Setters
    Vector getNormalVector(); Vector getPointOnPlane();
    void setPlaneVector(Vector setVector, bool normalize = true); void setPointOnPlane(Vector setVector);

private:
    Vector normalVector; //Normalized vector of the plane
    Vector pointOnPlane; //A point on the plane
};
```

Fig 11. Plane.h file with declarations for basic getter and setter methods and attributes for normal and point on plane vectors, own work

The getter methods are as simple as in the other classes, however for the setter method for the plane vector and the constructor I implemented code to automatically normalize the input vector. The bool value "normalize" is set to true by default so that unless it is specifically set to false the input vector is normalized, but this step can be skipped if the vector is already normalized, such as if it is being copied from another plane class.

```

#include "Plane.h"
Plane::Plane() {}

Plane::Plane(Vector planeVector, Vector pointVector, bool normalize)
{
    pointOnPlane = pointVector;
    if (normalize) // If normalize is set to false a pre-normalized vector can be written directly, by default input vector is normalized automatically
    {
        planeVector.normalize();
    }
    normalVector = planeVector;
}

Vector Plane::getNormalVector()
{
    return normalVector;
}

Vector Plane::getPointOnPlane()
{
    return pointOnPlane;
}

void Plane::setPlaneVector(Vector setVector, bool normalize)
{
    if (normalize) // If normalize is set to false a pre-normalized vector can be written directly, by default input vector is normalized automatically
    {
        setVector.normalize();
    }
    normalVector = setVector;
}

void Plane::setPointOnPlane(Vector setVector)
{
    pointOnPlane = setVector;
}

```

Fig 12. Plane.cpp file with code for the constructor and the getter and setter methods, own work

I made the normalization function a method of the vector class as it could be useful elsewhere. To do this I added a method to the vector class which normalizes the vector and returns it. It uses the existing p2pDistanceTo method to calculate the distance from 0,0,0, or the length of the vector, and uses the 3D Pythagorean Theorem to calculate its magnitude. Then the x y and z values are divided by the magnitude, and the result is returned.

```

Vector normalize();

```

Fig 13. Method declaration for the normalize function in the Vector.h file, own work

```

Vector Vector::normalize()
{
    Vector origin(0, 0, 0);
    float magnitude = p2pDistanceTo(origin); // Magnitude is determined by the length of the vector, in this case its distance from the origin (0,0,0)
    return Vector(x/magnitude, y/magnitude, z/magnitude);
}

```

Fig 14. Method code for normalizing the vector in the Vector.cpp file, own work

To test the code, I utilised an online unit vector calculator [7] to normalize a vector of (1,2,3) and then did the same in my own program. Its output is shown in the image below.

The calculator will find the unit vector in the direction of the vector $\langle 1, 2, 3 \rangle$, with steps shown.

$\vec{u}$:

Comma-separated.

If the calculator did not compute something or you have identified an error, or you have a suggestion/feedback, please [contact us](#).

CALCULATE

CLEAR ALL

YOUR INPUT

Find the unit vector in the direction of $\vec{u} = \langle 1, 2, 3 \rangle$.

SOLUTION

The magnitude of the vector is $|\vec{u}| = \sqrt{14}$ (for steps, see [magnitude calculator](#)).

The unit vector is obtained by dividing each coordinate of the given vector by the magnitude.

Thus, the unit vector is $\vec{e} = \left\langle \frac{\sqrt{14}}{14}, \frac{\sqrt{14}}{7}, \frac{3\sqrt{14}}{14} \right\rangle$ (for steps, see [vector scalar multiplication calculator](#)).

ANSWER

The unit vector in the direction of $\langle 1, 2, 3 \rangle$ is $\left\langle \frac{\sqrt{14}}{14}, \frac{\sqrt{14}}{7}, \frac{3\sqrt{14}}{14} \right\rangle \approx \langle 0.267261241912424, 0.534522483824849, 0.801783725737273 \rangle$.

Fig 15. Unit vector calculation from online calculator, [7]

I then created a vector in the main cpp file, used the normalization method on it and printed its resulting x, y and z values.

```
//Test for normalizing a vector
Vector normalVector(1, 2, 3);
normalVector = normalVector.normalize();
std::cout << "Normalization Test: " << normalVector.getX() << ", " << normalVector.getY() << ", " << normalVector.getZ() << std::endl;
```

Fig 16. Vector normalization calculation code test, own work

The terminal output from the program was rounded to fewer decimal places, however the values match the ones output from the online calculator, suggesting that the code is functional.

```
3.74166  
14  
Normalization Test: 0.267261, 0.534522, 0.801784
```

Fig 17. Vector Normalization code test output which matches the values from the online calculator, own work

Task 3 – Sphere Class with Collision

A sphere is a 3-dimensional shape, with one continuous face. While any sphere in real life is imperfect, in a hypothetically perfect sphere every point on its face is the same distance from a point at its centre, and this distance is its radius [8]. This makes it easy to check if the sphere is colliding with something – if a point is closer to the centre of the sphere than its radius it must be colliding with it.

Calculating a collision between two spheres is also simple. If the distance between the two sphere's centre points is less than their combined radii then they must be in collision. Checking if this is the case can be done quite simply using the preexisting point to point distance calculation from the vector class and an extra function in the sphere class I will now create.

To begin with, I created a new class for the sphere and some basic attributes, a vector to represent the centre point of the sphere and a float to represent its radius. I then created simple getter and setter methods for those attributes.

```
#pragma once
#include "Vector.h"
class Sphere
{
public:
    //Constructor
    Sphere(Vector center, float radius);
    Sphere();

    //Setter and getter methods
    void setPos(Vector newPos); Vector getPos();
    void setRadius(float newRadius); float getRadius();

private:
    Vector sphereCenter;
    float sphereRadius;
};
```

Fig 18. Sphere.h file with attributes and basic methods, own work

For the setRadius method I decided to add a sanity check to ensure that the radius value was greater than or equal to 0. A negative sized sphere cannot exist and therefore should not be allowed in the code. While a sphere of 0 radius also could not exist in real life, there could be situations where it would be useful to set the radius to 0, so I allowed for this in the code.

```

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

Sphere::Sphere() {};
Sphere::Sphere(Vector center, float radius)
{
    sphereCenter = center;
    sphereRadius = radius;
}

void Sphere::setPos(Vector newPos)
{
    sphereCenter = newPos;
}

Vector Sphere::getPos()
{
    return sphereCenter;
}

void Sphere::setRadius(float newRadius)
{
    if (newRadius >= 0) //Sanity check, sphere should never have a negative radius
    {
        sphereRadius = newRadius;
        return;
    }
    std::cout << "Error: Radius cannot be less than 0" << std::endl;
}

float Sphere::getRadius()
{
    return sphereRadius;
}

```

Fig 19. Sphere.cpp file with code for the basic methods implemented so far, own work

To implement the sphere-to-sphere collision detection I created another method which takes a pointer to the sphere to check for a collision with as an input. The method returns a bool, true if a collision is detected and false if not.

```

bool s2sCollisionCheck(Sphere& sphere2); //Check if this sphere is colliding with another sphere
private:
    Vector sphereCenter;

```

Fig 20. Method to detect sphere to sphere collisions in class header file, own work

For the method code I retrieved the centre point of the other sphere, then used the p2pDistanceTo method I implemented in the vector class to find the distance between both spheres' centres. I then calculated their combined radii and checked if the distance between their centres was smaller than their combined radii. If it is, the method returns true, otherwise it returns false.

```

bool Sphere::s2sCollisionCheck(Sphere& sphere2)
{
    Vector pos2 = sphere2.getPos();
    float distance = this->sphereCenter.p2pDistanceTo(pos2);
    float totalRad = this->sphereRadius + sphere2.getRadius();

    if (distance < totalRad)
    {
        return true; //The function exits upon a return so will not run the code after this line, which eliminates the need for an else statement
    }
    return false;
}

```

Fig 21. Sphere to Sphere collision detection code in class cpp file, own work

To test the Sphere class, I created 2 spheres in the main cpp file. I used the two vectors from the earlier vector test as their centre points, and radii of 0.5 and 1.5 respectively. This would put the distance between them at 2.0, and the distance between the two vectors, as shown earlier, is approximately 3.7, so the two spheres would not be colliding and the function should output "false", which will print as "0" in the terminal. I then set sphere2's centre point to the same as sphere1's centre point so they would collide, and a "1" should be printed in the terminal.

```

Vector vector1(1, 2, 3);
Vector vector2(2, 4, 6);

std::cout << vector1.p2pDistanceTo(vector2) << std::endl; //P2P distance calculation
std::cout << vector1.p2pDistanceTo(vector2, true) << std::endl; //P2P distane squared calculation

//Creating the 2 spheres
Sphere sphere1(vector1, 0.5);
Sphere sphere2(vector2, 1.5);

//Checking the collision, should output a 0
std::cout << "Test 1: " << sphere1.s2sCollisionCheck(sphere2) << std::endl;

//Setting the center points to the same value so the 2 spheres should be colliding
sphere2.setPos(sphere1.getPos());
std::cout << "Test 2: " << sphere1.s2sCollisionCheck(sphere2) << std::endl; //Should output 1

```

Fig 22. Code for testing the sphere class, own work

The sphere class and sphere to sphere collision detection method appear to be working as expected, and the terminal outputs match the expected outputs.

```

3.74166
14
Test 1: 0
Test 2: 1

C:\Users\Samst_000\Documents\Work\Uni\Year 2\Game Dev\Assignment 2\Assignment 2 Code\x64\Debug\Assignment 2 Code.exe (process 8548) exited with code 0 (0x0).
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when debugging stops.
Press any key to close this window . . .

```

Fig 23. Output of testing, showing expected results, own work

Task 4 – Point to Plane and Sphere to Plane Collision

Appendix

Main CPP file

```
#include <iostream>

#include "Vector.h"
#include "Plane.h"
#include "Sphere.h"

int main()
{
    Vector vector1(1, 2, 3);
    Vector vector2(2, 4, 6);

    std::cout << vector1.p2pDistanceTo(vector2) << std::endl; //P2P distance
    calculation

    std::cout << vector1.p2pDistanceTo(vector2, true) << std::endl; //P2P distane
    squared calculation

    //Creating the 2 spheres
    Sphere sphere1(vector1, 0.5);
    Sphere sphere2(vector2, 1.5);

    //Test for normalizing a vector
    Vector normalVector(1, 2, 3);
    normalVector = normalVector.normalize();

    std::cout << "Normalization Test: " << normalVector.getX() << ", " <<
    normalVector.getY() << ", " << normalVector.getZ() << std::endl;

    //Checking the collision, should output a 0
    std::cout << "Test 1: " << sphere1.s2sCollisionCheck(sphere2) << std::endl;
```

```

        //Setting the center points to the same value so the 2 spheres should be colliding
        sphere2.setPos(sphere1.getPos());
        std::cout << "Test 2: " << sphere1.s2sCollisionCheck(sphere2) <<
        std::endl; //Should output 1
    }

```

Vector.h file

```

#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);

```

```

    float getX(); float getY(); float getZ();

```

```

    void setX(float newVal); void setY(float newVal); void setZ(float newVal);

```

```

    float p2pDistanceTo(Vector& vector2, bool squared = false); //squared is set to
false by default so it does not have to be set unless the user wants the value
squared

```

```

    Vector normalize();

```

```

private:

```

```

    float x;

```

```

    float y;

```

```

    float z;

```

```

};

```

```
#endif
```

Vector.cpp file

```
#include "Vector.h"
```

#include <cmath>//square root function does not work here unless cmath is included, despite working fine elsewhere in the same program when not specifically included.

```
//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);
```

```

    }

    //Getters, return the relevant value
    float Vector::getX()
    {
        return this->x;
    }
    float Vector::getY()
    {
        return this->y;
    }
    float Vector::getZ()
    {
        return this->z;
    }

    //Setters, change the relevant value
    void Vector::setX(float newVal)
    {
        this->x = newVal;
    }
    void Vector::setY(float newVal)
    {
        this->y = newVal;
    }
    void Vector::setZ(float newVal)
    {
        this->z = newVal;
    }

    float Vector::p2pDistanceTo(Vector& vector2, bool squared)
    {

```

```

//Formula for calculating distance in 3D space is sqrt(dX^2 + dY^2 + dZ^2),
d=distance of (axis), for value squared omit the sqrt portion.

//3D Pythagorean Theorem - X^2 + Y^2 + Z^2 = d^2, d = above equation

//Calculating the distance in each axis
float dX = vector2.getX() - this->getX();
float dY = vector2.getY() - this->getY();
float dZ = vector2.getZ() - this->getZ();

//Distance Squared
float dTotal = dX * dX + dY * dY + dZ * dZ;

if (squared)//If distance is squared the function will return and exit and not run
the second return, otherwise it will run the second return only.
{
    return dTotal;//The function exits upon a return so will not run the code
after this line, which eliminates the need for an else statement
}

return sqrt(dTotal);//If distance is squared the previous return will be skipped
and this one will run instead
}

Vector Vector::normalize()
{
    Vector origin(0, 0, 0);

    float magnitude = p2pDistanceTo(origin);//Magnitude is determined by the
length of the vector, in this case its distance from the origin (0,0,0)

    return Vector(x/magnitude, y/magnitude, z/magnitude);
}

void Vector::Add(Vector& v1)
{
    x = x + v1.getX();

```

```
        y = y + v1.getY();  
        z = z + v1.getZ();  
    }
```

Plane.h file

```
#pragma once  
#include "Vector.h"  
  
class Plane  
{  
public:  
    Plane();  
    Plane(Vector planeVector, Vector pointVector, bool normalize = true);  
  
    //Getters and Setters  
    Vector getNormalVector(); Vector getPointOnPlane();  
    void setPlaneVector(Vector setVector, bool normalize = true); void  
    setPointOnPlane(Vector setVector);  
  
    float p2planeDistance(Vector point);  
private:  
    Vector normalVector; //Normalized vector of the plane  
    Vector pointOnPlane; //A point on the plane  
};
```

Plane.cpp file

```
#include "Plane.h"  
  
Plane::Plane() {}  
  
Plane::Plane(Vector planeVector, Vector pointVector, bool normalize)  
{
```

```

        pointOnPlane = pointVector;
        if (normalize)//If normalize is set to false a pre-normalized vector can be
        written directly, by default input vector is normalized automatically
        {
            planeVector.normalize();
        }
        normalVector = planeVector;
    }

```

```

Vector Plane::getNormalVector()
{
    return normalVector;
}

```

```

Vector Plane::getPointOnPlane()
{
    return pointOnPlane;
}

```

```

void Plane::setPlaneVector(Vector setVector, bool normalize)
{
    if (normalize)//If normalize is set to false a pre-normalized vector can be
    written directly, by default input vector is normalized automatically
    {
        setVector.normalize();
    }
    normalVector = setVector;
}

```

```

void Plane::setPointOnPlane(Vector setVector)

```

```
{  
    pointOnPlane = setVector;  
}  
  
float Plane::p2planeDistance(Vector point)  
{  
    return 0.0f;  
}
```

Sphere.h file

```
#pragma once  
#include "Vector.h"  
class Sphere  
{  
public:  
    //Constructor  
    Sphere(Vector center, float radius);  
    Sphere();  
  
    //Setter and getter methods  
    void setPos(Vector newPos); Vector getPos();  
    void setRadius(float newRadius); float getRadius();  
  
    bool s2sCollisionCheck(Sphere& sphere2); //Check if this sphere is colliding  
    with another sphere  
private:  
    Vector sphereCenter;  
    float sphereRadius;  
};
```

Sphere.cpp file

```
#include "Sphere.h"
```

```
#include <iostream>
```

```
Sphere::Sphere() {};
```

```
Sphere::Sphere(Vector center, float radius)
```

```
{
```

```
    sphereCenter = center;
```

```
    sphereRadius = radius;
```

```
}
```

```
void Sphere::setPos(Vector newPos)
```

```
{
```

```
    sphereCenter = newPos;
```

```
}
```

```
Vector Sphere::getPos()
```

```
{
```

```
    return sphereCenter;
```

```
}
```

```
void Sphere::setRadius(float newRadius)
```

```
{
```

```
    if (newRadius >= 0)//Sanity check, sphere should never have a negative  
radius
```

```
    {
```

```
        sphereRadius = newRadius;
```

```
        return;
```

```
    }
```

```

        std::cout << "Error: Radius cannot be less than 0" << std::endl;
    }

    float Sphere::getRadius()
    {
        return sphereRadius;
    }

    bool Sphere::s2sCollisionCheck(Sphere& sphere2)
    {
        Vector pos2 = sphere2.getPos();
        float distance = this->sphereCenter.p2pDistanceTo(pos2);
        float totalRad = this->sphereRadius + sphere2.getRadius();

        if (distance < totalRad)
        {
            return true; //The function exits upon a return so will not run the code
            after this line, which eliminates the need for an else statement
        }
        return false;
    }

```

References

- [F. Di Giacomo, M. Abbadi, A. Cortesi, P. Spronck, G. Costantini and G. Maggiore, "High
1 performance encapsulation and networking in Casanova 2," *Entertainment Computing*, vol. 20, pp.
] 25-41, 2017.
- [J.-M. Lévy-Leblond, "N-Dimensional Variations on Themes of Pythagoras, Euclid, and
2 Archimedes," *The Mathematical intelligencer*, vol. 26, no. 3, pp. 43-53, 2004.
]
- [E. W. Weisstein, "Unit Vector," Wolfram MathWorld, [Online]. Available:
3 [https://mathworld.wolfram.com/UnitVector.html#:~:text=A%20unit%20vector%20is%20a,as%20the](https://mathworld.wolfram.com/UnitVector.html#:~:text=A%20unit%20vector%20is%20a,as%20the%20(finite)%20vector%20..)
] [%20\(finite\)%20vector%20..](https://mathworld.wolfram.com/UnitVector.html#:~:text=A%20unit%20vector%20is%20a,as%20the%20(finite)%20vector%20..) [Accessed 02 01 2025].
- ["Unit Vector," Cuemath, [Online]. Available: <https://www.cuemath.com/calculus/unit-vector/>.
4 [Accessed 02 01 2025].
]
- ["3.4: The Unit Vector in 3-Dimensions and Vectors in Standard Position," LibreTexts Mathematics,
5 [Online]. Available:
] [https://math.libretexts.org/Bookshelves/Applied_Mathematics/Mathematics_for_Game_Developers](https://math.libretexts.org/Bookshelves/Applied_Mathematics/Mathematics_for_Game_Developers_(Burzynski)/03%3A_Vectors_in_Three_Dimensions/3.04%3A_The_Unit_Vector_in_3-Dimensions_and_Vectors_in_Standard_Position)
_ (Burzynski)/03%3A_Vectors_in_Three_Dimensions/3.04%3A_The_Unit_Vector_in_3-
Dimensions_and_Vectors_in_Standard_Position. [Accessed 02 01 2025].
- ["Unit Vector," BYJU'S, [Online]. Available: <https://byjus.com/maths/unit-vector/>. [Accessed 02 01
6 2025].
]
- ["Unit Vector Calculator," emathhelp, [Online]. Available:
7 <https://www.emathhelp.net/calculators/linear-algebra/unit-vector-calculator/>. [Accessed 02 01
] 2025].
- [T. E. o. E. Britannica, "sphere," Encyclopedia Britannica, 04 12 2024. [Online]. Available:
8 <https://www.britannica.com/science/sphere>. [Accessed 01 01 2025].
]

Assignment 2 Report

GRADEMARK REPORT

FINAL GRADE

GENERAL COMMENTS

A fantastic attempt, what happened for task 4?

Apart from that, You smashed it. Great work Sam.

60/100

PAGE 1

PAGE 2

PAGE 3



Comment 1

Good catch.



Comment 2

Good implementation

PAGE 4



Comment 3

The wording here could have been polished up a bit.

PAGE 5



Comment 4

Good effort, a man after my own heart for ms paint diagrams. There are tools that make this easier such as desmos

PAGE 6

PAGE 7

PAGE 8

PAGE 9



Comment 5

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 10

PAGE 11

PAGE 12

PAGE 13

PAGE 14

PAGE 15

PAGE 16

PAGE 17

PAGE 18

PAGE 19

PAGE 20

PAGE 21

PAGE 22

PAGE 23

PAGE 24

PAGE 25

PAGE 26

PAGE 27
