

Assignment 2 - Jamie Richards

0852036 - ACCB5017

by Jamie Richards

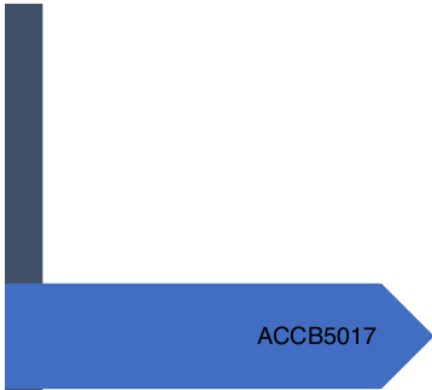
Submission date: 11-Jan-2025 06:24PM (UTC+0000)

Submission ID: 248046511

File name: 31393_Jamie_Richards_Assignment_2_-_Jamie_Richards_0852036_-_ACCB5017_628793_1590477091.docx (200.53K)

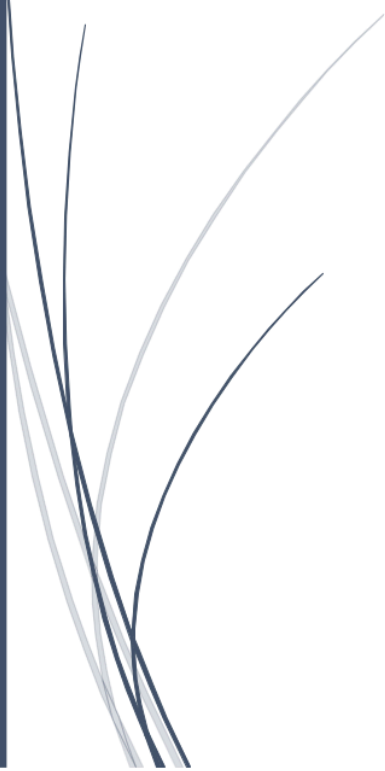
Word count: 2661

Character count: 13463



ACCB5017

Games Architecture and Mechanics



Jamie Richards (0852036)

ACCB5017 – GAMES ARCHITECTURE AND MECHANICS

0

Table of Contents

1 – Introduction	Page - 2
2 – Task 1 – Distance	Page - 2
2.1 – Implementation and Testing	Page - 3
2.2 – Squared	Page - 7
3 – Task 2 – Plane	Page - 9
3.1 – Implementation and Testing	Page - 9
3.2 – Helper Functions	Page - 12
4 – Task 3 – Sphere with Collision Detection	Page - 12
4.1 – Collision Detection	Page - 12
4.2 – Implementation and Testing	Page - 13
5 – Task 4 – Plane and Sphere Collision Detection	Page - 17
6 – Conclusion	Page - 17
7 – References	Page - 18
8 – Appendix	Page - 19

1 - Introduction

This report will consist of four tasks involving working with a vector class for a games engine coded in C++. Each task will detail the mathematical operations involved and display the code used for the task with its implementation and testing stages.

A basic vector class provided for this report can be found in the Appendix section.

2 - Task 1 – Distance

Task one will be to use the game engine vector to measure the distance between two points, using the appendix as a template which has already created a basic vector class for this task.



Figure 1 above taken from the math insight website illustrates what is a vector, the below statement reiterates this in words.

“A vector is an object that has a magnitude and a direction. Geometrically, we can picture a vector as a directed line segment, whose length is the magnitude of the vector and with an arrow indicating the direction. The direction of the vector is from its tail to its head.”

The above statement cited from the math insight website written by D Q Nykamp perfectly defines what a vector is, explaining the how a vector can be viewed. [1]

2.1 – Implementation and Testing

Below are the three files used to implement this code.

Soure.cpp file

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

int main()
{
    // Create two vectors for Version 1 and Version 2 or V1 V2
    Vector v1(3.0, 4.0, 5.0);
    Vector v2(1.0, 1.0, 1.0);

    // Subtract v2 from v1
    Vector result = v1.Subtract(v2);

    // Print the result
    std::cout << "Result of subtraction: ("
        << result.getVector().x << ", "
        << result.getVector().y << ", "
        << result.getVector().z << ")\n";

    // Calculate the length of the resulting vector
    std::cout << "Length of resulting vector: "
        << result.Length() << "\n";

    return 0;
}
```

Vector.cpp file

```
#include "vector.h"

#include <cmath> // math functions


// Construct
Vector::Vector(void) : x(0), y(0), z(0) {}


Vector::Vector(float myx, float myy, float myz)
{
    x = myx;
    y = myy;
    z = myz;
}


// Scale the vector
void Vector::Scale(float s)
{
    x = s * x;
    y = s * y;
    z = s * z;
}


// Add two vectors allowing a version 1 and version 2
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;
}
```

```

// Get a copy of the current vector
Vector Vector::getVector(void)
{
    return Vector(x, y, z);
}

// Add a vector to the current vector
void Vector::Add(Vector& v1)
{
    x = x + v1.x;
    y = y + v1.y;
    z = z + v1.z;
}

// Subtract vector version 2 from vector version 1 and return the result as a
completely new vector
Vector Vector::Subtract(Vector& v2)
{
    Vector result(x - v2.x, y - v2.y, z - v2.z);
    return result;
}

// Calculate the length
float Vector::Length(void)
{
    return sqrt(x * x + y * y + z * z);
}

```

Vector.h file

```

#ifndef VECTOR_H
#define VECTOR_H

```

```

class Vector
{
public:
    // Constructor
    Vector(float x, float y, float z);
    Vector(void);

    // Methods to manipulate vectors
    void Scale(float s);
    void Add(Vector& v1);
    void Add(Vector& v1, Vector& v2);
    Vector getVector(void);

    // New methods for subtraction and length
    Vector Subtract(Vector& v2); // Subtract two vectors
    float Length(void);         // Calculate the length of the vector

    // Vector components
    float x;
    float y;
    float z;
};

#endif

```

The above Vector.h file in the appendix had the x, y and z floats private making the program unable to access them, therefore these were made public.

The basic principle of this code is to subtract the x,y and z elements of two vectors which will then produce the length of a third vector, revealing the distance between two points. In order to do this firstly the class needed to be modified to allow the subtraction to occur and furthermore allow another method to calculate the length or magnitude of the final vector.

It can be seen from the implementation of the coding files above where vectors were created, namely version 1 and 2 or v1,v2. The subtraction of v2 from v1 allowed the return of a new vector which when using the below formula to calculate the length gave the distance between two points.

$$\text{Length} = \sqrt{x^2 + y^2 + z^2}$$

2.2 – Squared

Next in this task a new method will be presented which will calculate the point-to-point distance squared.

To add this method the following code will be added to **vector.cpp** file

```
// New method to calculate squared
float Vector::DistanceSquared(Vector& v2)
{
    float dx = x - v2.x;
    float dy = y - v2.y;
    float dz = z - v2.z;
    return dx * dx + dy * dy + dz * dz;
}
```

While adding the line to the **vector.h** file

```
float DistanceSquared(Vector& v2); // Calculate squared distance between two
vectors
```

This allows the program to calculate the squared distance between the current vector holding the length and v2 vector.

The new **source.cpp** file will now look like this:

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

```

int main()
{
    // Create two vectors (representing two points)
    Vector v1(3.0, 4.0, 5.0);
    Vector v2(1.0, 1.0, 1.0);

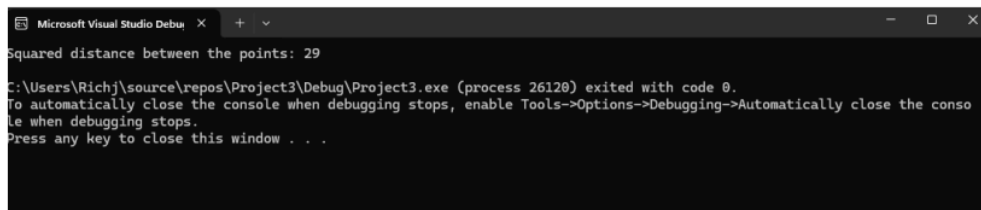
    // Calculate the squared distance between v1 and v2
    float distSquared = v1.DistanceSquared(v2);

    // Output the result
    std::cout << "Squared distance between the points: " << distSquared << std::endl;

    return 0;
}

```

Following a test of the code the outcome illustrates the distance between points as 29, as shown below.



The screenshot shows a console window titled "Microsoft Visual Studio Debug" with a dark background. The output text is as follows:

```

Squared distance between the points: 29
C:\Users\Richj\source\repos\Project3\Debug\Project3.exe (process 26120) exited with code 0.
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 . . .

```

3 - Task 2 – Plane Class

In this task a plane class will be created using two attributes, implementation and testing in this task will involve a method for normalizing the plane, a method for calculating the distance of a point from the plane and a helper method which will calculate the plane's normality within two vectors.

Before beginning this task, a class is definition for an object in programming, it can be viewed as container which holds values used towards its purpose, often used with subclasses, each with their own instructions or tasks.[2]

As briefly mentioned above to begin this task a plane class will be created with two attributes:

- Vector which will present as the normal plane
- Point also a vector but represents a point on the plane such as the back or front.

A number of different methods will be needed to implement the tasks instructions.

- A helper method - this is where several calculations will be made such as the distance from a point on the plane or seeing if the point is at the front for back of the plane.
- Distance – As briefly mentioned above this is where a point on the plane will be measured.
- Normality of the Plane – This will require the use of two vectors and will normalise the plane.

3.1 – Implementation and Testing

Creating class for plane will include the methods shown above and look something like this:

```
#include "vector.h"
```

```

class Plane
{
public:
    // takes a point P0 on the plane
    Plane(Vector normal, Vector point);

    // method to normalize the plane
    void NormaliseNormal();

    // calculate the distance from a point to the plane
    float DistanceFromPoint(Vector& point);

    // calculate where the point on the plane is
    void TestPointPosition(Vector& point);

    // helper method to calculate the two vectors
    float DotProduct(Vector& v1, Vector& v2);

    // another helper method to calculate the normal plane
    Vector CalculateNormal(Vector& v1, Vector& v2);

};

#endif

```

As illustrated in the code above the implementation of the different methods are put into place.

In the main.cpp file the code will look like this:

```

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

int main()
{
    // create the needed vectors
    Vector pointOnPlane(1.0f, 2.0f, 3.0f); // this is the point on the plane
    Vector v1(1.0f, 0.0f, 0.0f); // the first vector or v1
    Vector v2(0.0f, 1.0f, 0.0f); // second vector or v1

    // calculating the v1 vector and point on plane
    Plane plane(v1, pointOnPlane);

    // this is the normalising the plane
    plane.NormaliseNormal();

    Vector testPoint(1.0f, 2.0f, 5.0f); // the test point
    plane.TestPointPosition(testPoint);

    // calculate the distance from the point
    float distance = plane.DistanceFromPoint(testPoint);
    std::cout << "The distance from the point to the plane is: " << distance <<
    std::endl;

    return 0;
}

```

Here the calculations are made for the distance to the point and the vector to point on the plane.

Unfortunately testing did not correctly work on the calculations and failure with implementing the correct code.

3.2 – Helper Functions

Helper functions as the name suggests help main functions to perform more efficiently and even give names to the calculations involved in functions, for instance using a helper function can improve a programmer's code and even make it reusable. Often when calculations are used multiple times multiple helper functions will be used again also. [3]

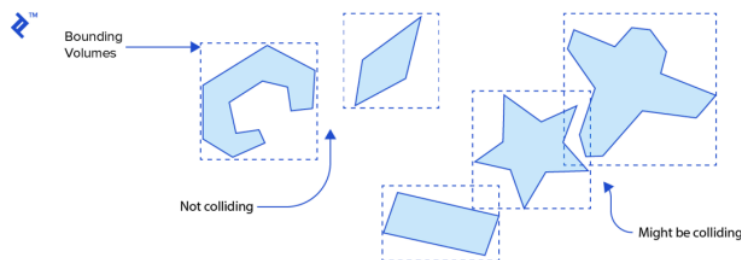
4 – Task 3 – Sphere Class with Collision Detection

This task will involve the creation of a sphere class and the exploration of a collision detection, the sphere class will hold a method for collision detection and will detect a collision between two sphere volumes.

4.1 – Collision Detection

Collision defines two objects hitting each other, however the detection of collision is used in the world of gaming where these collisions between objects are detected in code and produced, vast amount of examples of this can be produced such as characters on characters or character and terrain, how a character responds when walking into a wall etc.

Created using algorithms collision detection, the calculation within determines when or where two or more objects interact or collide. Figure 2 below is an excellent representation of this and was taken from the toptal website. [4]



4.2 – Implementation and Testing

To begin two attributes are needed for the sphere class, these will be a vector for the middle of the sphere and a float which will give the radius of the sphere. In order to detect if the spheres will collide a calculation can be made where the distance between the middle of each sphere will be less than or equal to the size of their radiuses, if they are then that means a collision has occurred.

A formula will be needed to calculate the above situation, this will look like this:

$$\text{Dist}(1,2) = \sqrt{(x_2-x_1)^2 + (y_2-y_1)^2 + (z_2-z_1)^2}$$

Figure 3 above illustrates the formula that will be used in calculating if a collision has occurred, taken from inch calculator website. [5]

Below is the **Sphere.h** file which creates the sphere class.

```
#ifndef SPHERE_H
#define SPHERE_H

#include "vector.h"

class Sphere {
public:

    Sphere(Vector centre, float radius);

    // this will check if there is a collision
    bool checkCollision(const Sphere& otherSphere) const;

    Vector getCentre() const;
```

```

        float getRadius() const;

private:
    Vector centre;
    float radius;
};

#endif

```

In the above code it can be seen the method for checking if the two spheres collide.

The **sphere.cpp** file will look like this:

```

#include "sphere.h"
#include <cmath>

Sphere::Sphere(Vector centre, float radius) {
    this->centre = centre;
    this->radius = radius;
}

// checking if there is a collision
bool Sphere::checkCollision(const Sphere& otherSphere) const {
    Vector diff;
    diff.Add(this->centre.getVector(), otherSphere.getCentre().getVector());

    // this checks the distance between the middle of each sphere
    float distanceSquared = diff.x * diff.x + diff.y * diff.y + diff.z * diff.z;
}

```

```

float radiiSum = this->radius + otherSphere.getRadius();
float radiiSumSquared = radiiSum * radiiSum;

// checking if less than or equal too
return distanceSquared <= radiiSumSquared;
}

```

```

Vector Sphere::getCentre() const {
    return centre;
}

```

```

float Sphere::getRadius() const {
    return radius;
}

```

The above code implementing the check for collision as well as calculating the radius of the spheres.

Finally, to implement the test the main code will look like:

```

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

int main() {
    // Create two spheres
    Vector centre1(0.0f, 0.0f, 0.0f);
    Vector centre2(2.0f, 0.0f, 0.0f);
}

```

```

Sphere sphere1(centre1, 1.0f); // Radius 1
Sphere sphere2(centre2, 1.0f); // Radius 1

// check for collision
if (sphere1.checkCollision(sphere2)) {
    std::cout << "There was a collision!" << std::endl;
}
else {
    std::cout << "There was no collision." << std::endl;
}

return 0;
}

```

The output should determine if their spheres are colliding or not colliding, due to the figures entered for the spheres radius the output should indicate there was a collision due to the middle of one sphere being 2.0 and the radius being 2.0 meaning the spheres would touch or collide.

However unfortunately no output was produced due to errors in the implementation, possibly where adding another vector incorrectly, a solution to this may be by adding a subtraction method of the vectors.

Regardless of the outcome there are room for improvements by using more detailed distance functions or even a three-dimensional environment, making this more complex.

5 - Task 4 – Plane and Sphere Collision Detection

In this task two further methods will be implemented to the collision test, first a test for the collision between a point and a plane class followed by a sphere and plane collision.

In order to implement the point to plane collision test the Plane class needed to be specified so the plane can be represented by a point on the plane, if this was done in three-dimensional the following formula can be used:

$$Ax + By + Cz + D = 0$$

Meaning the ABC factors are the planes vectors and the D would be persistent.

The formula then to calculate the distance between the point and the plane would be like that in task 3 but mixed with the above formula:

$$\text{Distance} = \frac{Ax_1 + By_1 + Cz_1 + D}{\sqrt{A^2 + B^2 + C^2}}$$

$$A^2 + B^2 + C^2$$

Also briefly mentioned in task 3, to determine a collision between a plane and a sphere would be to calculate the position of the middle of sphere and a point on the plane, again if less than or equal to resulting in a collision or not.

6 - Conclusion

Through the series of tasks in this report the focus has been on implementing and testing different collision detection algorithms using mathematical concepts such as vectors. It can be seen that a vector plays an essential role in game development as well as the vector class in collision detection.

Even though most of the tests were unsuccessful due to error handling looking back a lot due to improper initialisation the concepts did demonstrate the application of these methods in game development, especially game engines and physics.

7 – References

[1] Nykamp DQ. Nykamp DQ. Image: A vector - math insight. Image: A vector - math insight. "A Vector." "A Vector." Available: <https://mathinsight.org/image/vector>.

[2] Anonymous . *What is class? | Definition from TechTarget*. Available: <https://www.techtarget.com/whatis/definition/class>.

[3] Anonymous (-12-06 00:30:51+00:00). *What are the Helper functions ?*. Available: <https://www.geeksforgeeks.org/what-are-the-helper-functions/>.

[4] Anonymous . *Video Game Physics Tutorial - Part II: Collision Detection for Solid Objects | Toptal®*. Available: <https://www.toptal.com/game/video-game-physics-part-ii-collision-detection-for-solid-objects>.

[5] Anonymous . *Distance Between Two Points Calculator*. Available: <https://www.inchcalculator.com/distance-calculator/>.

8 – Appendix

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:

```

#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

```

Assignment 2 - Jamie Richards 0852036 - ACCB5017

GRADEMARK REPORT

FINAL GRADE

60/100

GENERAL COMMENTS

Great work, Jamie!

Your sphere class implementation was very close to being correct. With a few adjustments, it could work exactly as you intended. Let me explain where the key issues lie and how to address them.

The Pythagorean theorem is essential for calculating the Euclidean distance between two points in 3D space. It involves finding the square root of the sum of the squared differences of the coordinates:

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

The first issue in your implementation was with the `diff.add` operation. To compute the difference between the coordinates of the two spheres, you need to subtract the corresponding components of the two centres (e.g., $x_2 - x_1$, $y_2 - y_1$, $z_2 - z_1$).

After correctly computing the differences, you need to square these values and sum them up to get the distance squared. For example:

$$\text{distance squared} = (x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2$$

The next step is to calculate the actual distance by taking the square root of the distance squared. This step was missing in your implementation.

Checking Against radiiSum: Once you have the distance, you can compare it with the sum of the

radii of the two spheres (radiiSum)
to determine if the spheres intersect. Specifically:

If $\text{distance} < \text{radiiSum}$ the spheres are intersecting.
Otherwise, they are not.

Overall, this was a great attempt! With these
corrections,
your implementation should work as expected.

PAGE 1

PAGE 2

PAGE 3

PAGE 4

PAGE 5

PAGE 6

PAGE 7

PAGE 8

PAGE 9

PAGE 10

PAGE 11

PAGE 12

PAGE 13

PAGE 14

PAGE 15

PAGE 16

PAGE 17

PAGE 18

PAGE 19
