

Assignment 2

by William Baker Oliver

Submission date: 27-Dec-2024 09:13PM (UTC+0000)

Submission ID: 246904869

File name: 50534_William_Baker_Oliver_Assignment_2_628793_599797074.docx (1.48M)

Word count: 2280

Character count: 11281

Games Architecture and Mechanics

Assignment 2: Vectors For Game Engine

William Baker Oliver

Student Number: 2314716

Computing (Games Development)

Contents

Introduction	3
Vectors	3
Planes.....	9
Spheres	14
Testing.....	16
Conclusion.....	17
External Links.....	18
Bibliography	19

Introduction

For this assignment, the task was to write code for a vector class in C++, for a video game engine. Vectors are crucial in both the 2D and 3D areas of video games, being the essential backbone of object placement and the spaces that they are in. This assignment discusses the work done for the game engine, how the code functions, the mathematical concepts behind the code and testing of the engine's code.

Vectors

A Vector, according to Encyclopedia Britannica, is "a quantity that has both magnitude and direction" [1]. Vectors are important for video game development as it helps us represent the game space. Take for example some 3D spheres with different heights and locations. You need points to represent where they are in the 3D world. Flat surfaces, such as planes, also need to be defined, such as their length. We can use vectors to give information on the points and locations of objects, as well as the scale of objects. For both 2D and 3D games, vectors are essential.

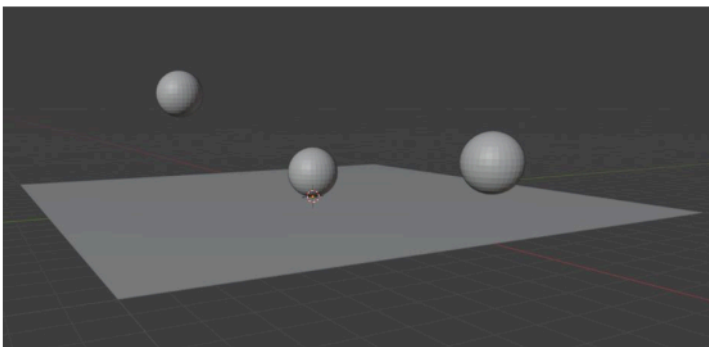
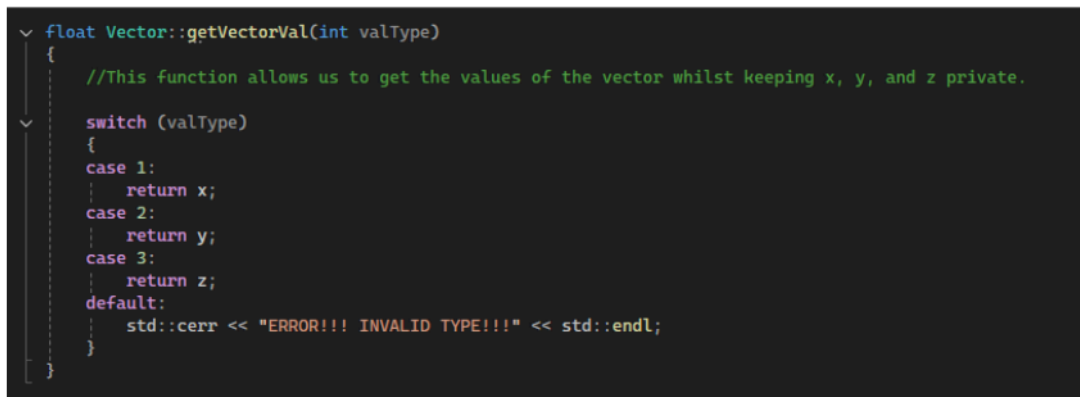


Fig. 1: A picture of 3D spheres in different locations, as well as a plane. 3D scene made in Blender.

For this task, a Vector class was already made for the engine. However, the x, y, and z variables were initially public in the class. Typically, it is best practice to keep all variables like this private for data integrity and security, so they were moved to be private. If we want to access these values, however, we can create a function within the Vector class which gets the specified values needed and returns them. This allows us to use the values elsewhere whilst keeping the original values secure. This was added to the class.

A screenshot of a code editor showing the implementation of the `getVectorVal` function in the `Vector` class. The code is written in C++ and uses syntax highlighting. It features a switch statement to handle different values of `valType` (1 for x, 2 for y, 3 for z) and a default case that outputs an error message to `std::cerr`.

```
float Vector::getVectorVal(int valType)
{
    //This function allows us to get the values of the vector whilst keeping x, y, and z private.

    switch (valType)
    {
        case 1:
            return x;
        case 2:
            return y;
        case 3:
            return z;
        default:
            std::cerr << "ERROR!!! INVALID TYPE!!!" << std::endl;
    }
}
```

Fig. 2: A picture of the `getVectorVal` function in the `Vector` class. Taken from Visual Studio 2022.

The class contained multiple functions, such as adding a vector to another vector, scaling up a vector with a given amount, a `getVector` function which returns a `Vector`, as well as the constructor and destructor needed for the class to work. The initial code provided is shown below.

```

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

class Vector
{
public:
    //construct
    Vector(float x, float y, float z);
    Vector(void);
    void Scale(float s);
    void Add(Vector& v1);
    void Add(Vector& v1, Vector& v2);
    Vector getVector(void);
    //private:    //these are currently public - should they be private?
    float x;
    float y;
    float z;
};
#endif

```

Fig. 3: A picture of the initial code in “Vector.h”. Taken from Visual Studio 2022.

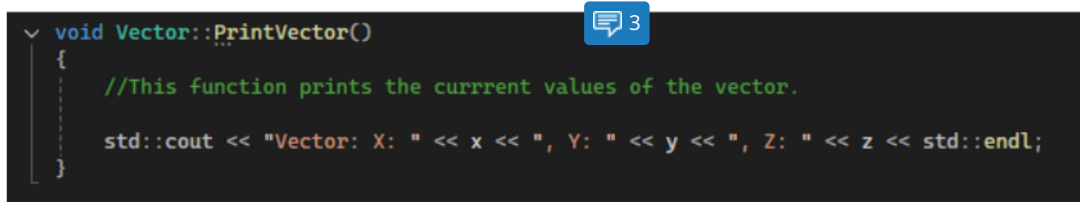
```

#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;
}

```

Fig. 4: A picture of the initial code in “Vector.cpp”. Taken from Visual Studio 2022.

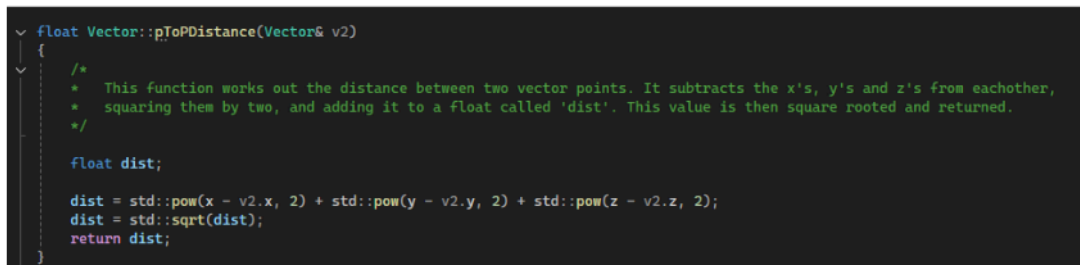
In addition to the `getVectorVal` function added and setting `x`, `y` and `z` to be private, other additions were made. A `PrintVector` function was added, to help with testing and to see the current values of the vector in the console.

A screenshot of a code editor showing the `PrintVector` function within the `Vector` class. The function is defined as `void Vector::PrintVector()` and contains a comment `//This function prints the current values of the vector.` followed by a `std::cout` statement that prints the values of `x`, `y`, and `z` in the format "Vector: X: ", "Y: ", and "Z: " followed by a newline character.

```
void Vector::PrintVector()
{
    //This function prints the current values of the vector.
    std::cout << "Vector: X: " << x << ", Y: " << y << ", Z: " << z << std::endl;
}
```

Fig.5: A picture of the `PrintVector` function in the `Vector` class. Taken from Visual Studio 2022.

One of the main requirements for this task was to implement a method which works out the distance between two vector points and returns the distance, as well as a method which squares the result by 2. Both methods were added to the `Vector` class. The methods take in a vector and then using the current vector and the vector taken in, it works out the distance then returns a value. The `pToPDistanceSqr` calls the `pToPDistance` method, applies it to a float, then uses `cmath` functions to square it by 2.

A screenshot of a code editor showing the `pToPDistance` function within the `Vector` class. The function is defined as `float Vector::pToPDistance(Vector& v2)` and contains a multi-line comment explaining its purpose: to calculate the distance between two vector points by subtracting their x, y, and z components, squaring the results, summing them, and then taking the square root. The code implements this logic using `std::pow` and `std::sqrt` functions.

```
float Vector::pToPDistance(Vector& v2)
{
    /*
     * This function works out the distance between two vector points. It subtracts the x's, y's and z's from eachother,
     * squaring them by two, and adding it to a float called 'dist'. This value is then square rooted and returned.
     */
    float dist;

    dist = std::pow(x - v2.x, 2) + std::pow(y - v2.y, 2) + std::pow(z - v2.z, 2);
    dist = std::sqrt(dist);
    return dist;
}
```

Fig. 6: A picture of the `pToPDistance` method in the `Vector` class. Taken from Visual Studio 2022.

```

float Vector::pToPDistanceSqrd(Vector& v2)
{
    //This function uses the pToPDistance function and squares it.

    float dist;

    dist = pToPDistance(v2);
    dist = std::pow(dist, 2);
    return dist;
}

```

Fig. 7: A picture of the pToPDistanceSqrd method in the Vector class. Taken from Visual Studio 2022.

The formula for calculating the distance is: $d = \sqrt{(x-x)^2 + (y-y)^2 + (z-z)^2}$. To give an example of how this method works, consider 2 vectors: one being [1,2,3] and the other being [4,5,6]. The way that we can calculate the distance is by first taking away each x, y and z from both vectors, which will leave us with the values [-3, -3, 3]. We then put these values under a square root, square each value individually by two and add them together, which leaves us with $\sqrt{27}$. The square root of 27 is equal to 5.19615242271, which is our distance value between the two points.

Other functions than the point-to-point distance method were added. One of these was a function which finds the dot product of two vectors. This will be needed for other classes in the engine, so it was implemented here.

```

float Vector::getDotProd(Vector& v2)
{
    /*
     * This function works out the dot product by using both the current vector in the class and another vector, by multiplying the
     * x's, y's and z's together to return the product.
     */

    float dotProd = (x * v2.x) + (y * v2.y) + (z * v2.z);
    std::cout << "Dot Product: " << dotProd << std::endl;
    return dotProd;
}

```

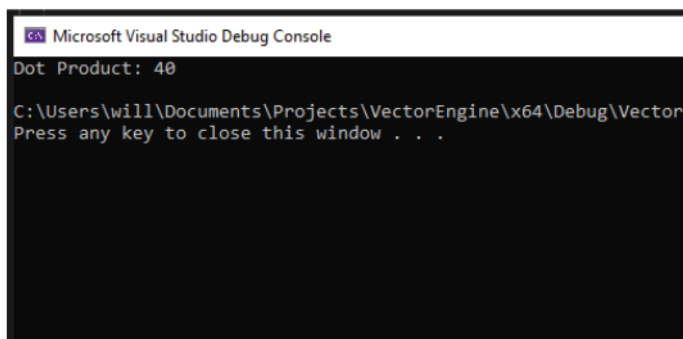
Fig. 8: A picture of the getDotProd method in the Vector class. Taken from Visual Studio 2022.

The dot product, according to Game Physics Cookbook, represents “the directional relation of the vectors”, with the method returning “a scalar value”, as opposed to a vector value [2]. The formula for working out the dot product is shown below.

$$\vec{A} \cdot \vec{B} = \sum_{i=0}^n A_i B_i$$

Fig. 9: A picture showing the formula for working out the dot product of two vectors. Taken from [2].

In our code, the method works by taking the x, y and z values of two vectors, then multiplying each x, y and z value separately. Adding them together produces the dot product. For example, the vectors [2, 1, 4] and [3, 6, 7] return a dot product of 40.



```

Microsoft Visual Studio Debug Console
Dot Product: 40
C:\Users\will\Documents\Projects\VectorEngine\x64\Debug\Vector
Press any key to close this window . . .

```

Fig. 10: A picture of the console outputting the results of the getDotProd method. Taken from Visual Studio 2022.

Another method added to the Vector class for later use was the getLength method, which gets the length/magnitude of the vector. The method for calculating the length/magnitude of the vector is identical to how point-to-point distance is calculated, except it takes in only 1 vector. For example, the length/magnitude of the vector [The formula for working out the length is: $\sqrt{x^2 + y^2 + z^2}$.

```

float Vector::getLength()
{
    /*
     * This function works out the length by squaring each part of the vector by 2 and adding it together,
     * then works out the square root of the combined values to return the length result.
     */

    float length = std::pow(x, 2) + std::pow(y, 2) + std::pow(z, 2);
    length = std::sqrt(length);
    std::cout << "Length Of Vector: " << length << std::endl;
    return length;
}

```

Fig. 11: A picture of the getLength function in the Vector class. Taken from Visual Studio 2022.

Planes

Another requirement was to add a class for planes. A plane, according to Wolfram Mathworld, is “a two-dimensional doubly ruled surface spanned by two linearly independent vectors.” [3]. They can allow us to represent flat surfaces in our game world. For this class, a few functions were required to be implemented. One of these functions was a method to get the plane normal vector, taking in two vectors. The way we get the plane normal vector is by working out the cross product of the two vectors. The code takes in the two vectors, works out the cross product of both, and returns the plane normal.

```

void Plane::getPlaneNormal(Vector& v1, Vector& v2)
{
    //This gets the plane normal vector, taking two vectors and using the cross product formula.

    int ni = (v1.getVectorVal(2) * v2.getVectorVal(3)) - (v1.getVectorVal(3) * v2.getVectorVal(2));
    int nj = (v1.getVectorVal(3) * v2.getVectorVal(1)) - (v1.getVectorVal(1) * v2.getVectorVal(3));
    int nk = (v1.getVectorVal(1) * v2.getVectorVal(2)) - (v1.getVectorVal(2) * v2.getVectorVal(1));

    Vector nV(ni, nj, nk);
    planeNormal = nV.getVector();
    std::cout << "Plane Normal: ";
    planeNormal.PrintVector();
}

```

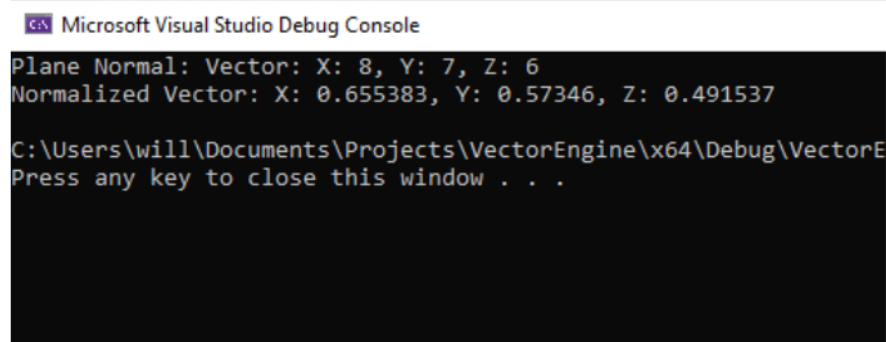
Fig. 12: A picture of the getPlaneNormal function in the Plane class. Taken from Visual Studio 2022.

The cross product, or vector product, differs from the dot product. Where the dot product “multiplies two vectors together to give a scalar value, the vector product multiplies them to get another vector.” [4]. So, we can use the cross product method to obtain a new vector, in this case the plane normal. The method for working out the cross product is shown below.

$$\mathbf{a} \circ \mathbf{b} = \begin{bmatrix} a_x \\ a_y \\ a_z \end{bmatrix} \circ \begin{bmatrix} b_x \\ b_y \\ b_z \end{bmatrix} = \begin{bmatrix} a_x b_x \\ a_y b_y \\ a_z b_z \end{bmatrix}$$

Fig. 13: A picture of the formula for working out the cross product of two vectors. Taken from [4].

Another function required for the plane class was a method to normalize the vector normal. Normalization makes the length/magnitude of the vector equal to exactly 1, whilst keeping its direction [5]. For example, if you were to take the vector [8,7,6] and normalize it, it would end up as [0.655, 0.57, 0.49]. If you worked out the length of this vector, it would be exactly 1.



```
Microsoft Visual Studio Debug Console
Plane Normal: Vector: X: 8, Y: 7, Z: 6
Normalized Vector: X: 0.655383, Y: 0.57346, Z: 0.491537
C:\Users\will\Documents\Projects\VectorEngine\x64\Debug\VectorE
Press any key to close this window . . .
```

Fig. 14: A picture of the console outputting the normalized vector of [8, 7, 6]. Taken from Visual Studio 2022.

The formula for working out a normalized vector is shown below.

$$\hat{\mathbf{X}} \equiv \frac{\mathbf{X}}{|\mathbf{X}|}$$

Fig. 15: A picture of the formula for normalizing a vector. This applies to the x value of the vector. However, this formula also applies to y and z. Taken from [5].

The code works by finding the length/magnitude of the plane normal, then dividing the x, y and z values of the vector by the length. This gives us our normalized vector.

```
void Plane::normalizeVector()
{
    //This normalizes the plane vector by dividing x, y, and z by the total length of the vector, using getLength() which is implemented in the Vector class.
    //This makes the vector's length, or magnitude, equal to exactly 1.

    float length = planeNormal.getLength();

    x = planeNormal.getVectorVal(1) / length;
    y = planeNormal.getVectorVal(2) / length;
    z = planeNormal.getVectorVal(3) / length;

    Vector normalizedVector(x, y, z);

    planeNormal = normalizedVector.getVector();
    std::cout << "Normalized ";
    planeNormal.PrintVector();
}
```

Fig. 16: A picture of the normalizeVector function in the Plane class. Taken from Visual Studio 2022.

With the vector normal function in the class, we can use it for other functions: one for calculating the distance between a vector point to the plane, and to see if a vector point is on, behind or in front of the plane. To do this, a planeEquation function was added to work out the plane, which will be used for both functions. The equation for a plane is: $Ax + By + Cz + D = 0$. A, B and C represent the plane normal, and x, y and z represent the point, with D being a constant [6]. We can use the getDotProd function here to work out the plane.

```

float Plane::planeEquation(Vector& point)
{
    //This function is used for both calculating the point-to-plane distance, as well as finding out where the point is with regards to the plane.

    float plane = planeNormal.getDotProd(point);
    std::cout << "AX + BY + CZ + D = " << plane << std::endl;

    return plane;
}

```

Fig. 17: A picture of the planeEquation function in the Plane class. Taken from Visual Studio 2022.

If the plane normal is not normalized, it will tell us where the point is with regards to the plane. If $Ax + By + Cz + D = 0$, then the point is on the plane. If it's less than zero, it is behind the plane. If it is greater than zero, it is in front. We can use this information as well to find out if the point is colliding with the plane. If it's equal to 0, then it is on the plane and thus is colliding with the plane.

```

Microsoft Visual Studio Debug Console

Plane Normal: Vector: X: 1, Y: -6, Z: -10
Dot Product: -64
AX + BY + CZ + D = -64
This point is behind the plane.
The point is not colliding with the plane.

C:\Users\will\Documents\Projects\VectorEngine\x64
Press any key to close this window . . .

```

Fig. 18: A picture of the console showing an example of the plane equation, which shows us that the point [8, 7, 3] is behind the plane and is not colliding with the plane. Taken from Visual Studio 2022.

```

Float Plane::checkPointToPlane(Vector& point)
{
    //This function performs the plane equation and checks to see if it is less than, more than, or exactly zero, which tells us where it is on the plane.
    float check = planeEquation(point);
    if (check == 0)
    {
        std::cout << "This point is on the plane." << std::endl;
        return 1;
    }
    if (check < 0)
    {
        std::cout << "This point is behind the plane." << std::endl;
        return 2;
    }
    if (check > 0)
    {
        std::cout << "This point is in front of the plane." << std::endl;
        return 3;
    }
}

bool Plane::pToPlaneCollision(Vector& point)
{
    //This function checks to see if a point is touching the plane. If so, there is a collision.
    float collision = checkPointToPlane(point);
    if (collision == 1)
    {
        return true;
    }
    else
    {
        return false;
    }
}

```

Fig. 19: A picture of the pToPlaneCollision and checkPointToPlane functions in the Plane class. Taken from Visual Studio 2022.

If the plane normal has been normalized, then the result given from the equation will be the distance from the point to the plane [6].

```

float Plane::pToPlaneDistance(Vector& point)
{
    //This function uses the plane equation to return the point-to-plane distance using a normalized vector.
    normalizeVector();
    float dist = planeEquation(point);
    dist = dist / planeNormal.getLength();
    std::cout << "Distance from Plane: " << dist << std::endl;
    return dist;
}

```

Fig. 20: A picture of the pToPlaneCollision and checkPointToPlane functions in the Plane class. Taken from Visual Studio 2022.

```
Microsoft Visual Studio Debug Console
Plane Normal: Vector: X: 1, Y: -6, Z: -10
Length Of Vector: 11.7047
Normalized Vector: X: 0.0854358, Y: -0.512615, Z: -0.854358
Dot Product: -5.46789
AX + BY + CZ + D = -5.46789
Length Of Vector: 1
Distance from Plane: -5.46789

C:\Users\will\Documents\Projects\VectorEngine\x64\Debug\Vector
Press any key to close this window . . .
```

Fig. 21: A picture of the console showing the distance of the point [8, 7, 3] from the plane. Taken from Visual Studio 2022.

Spheres

In addition to the plane class, a sphere class was also required for the engine. For this class, a function to detect sphere-to-sphere collision as well as sphere-to-plane collision was required.

To work out whether two spheres are colliding with each other, we need to work out the distance between the two spheres' center points. If the distance is less than the radiuses, or radii combined, then a collision is happening between the two spheres. We can use the pToPDistance method here to work out the distance between the spheres, and then use an if statement to check whether the spheres are colliding. We can then return a Boolean value which says whether the spheres are colliding or not. A function called moveSphere was added to test the collision, which adds a value to the spheres' center point vector to simulate movement.

```
void Sphere::moveSphere(float x, float y, float z)
{
    //This function moves a sphere by creating a 'move' vector, which is then added to the current vector.

    Vector move(x, y, z);
    sv.Add(move);
    std::cout << "Sphere Current Position: ";
    sv.PrintVector();
}
```

Fig. 22: A picture of the moveSphere function in the Sphere class. Taken from Visual Studio 2022.

```
bool Sphere::isColliding(Sphere s)
{
    /*
     * This checks the distance of both of the sphere's central points, which in this case is their vectors. If the distance is less than both radiuses, or radii,
     * then there is a collision between both spheres and will return a true value.
     */

    sv.PrintVector();
    s.sv.PrintVector();

    float radii = radius + s.getRadius();
    std::cout << radii << std::endl;
    float dist = sv.pToPDistance(s.sv);
    std::cout << dist << std::endl;

    if (dist < radii)
    {
        return true;
    }
    else
    {
        return false;
    }
}
```

Fig. 23: A picture of the isColliding function in the Sphere class. Taken from Visual Studio 2022.

```
Microsoft Visual Studio Debug Console

Sphere Current Position: Vector: X: 2, Y: 3, Z: -1
Vector: X: 2, Y: 3, Z: -1
Vector: X: 2, Y: -3, Z: -1
4
6
Sphere Current Position: Vector: X: 2, Y: 2, Z: -1
Vector: X: 2, Y: 2, Z: -1
Vector: X: 2, Y: -3, Z: -1
4
5
Sphere Current Position: Vector: X: 2, Y: 1, Z: -1
Vector: X: 2, Y: 1, Z: -1
Vector: X: 2, Y: -3, Z: -1
4
4
Sphere Current Position: Vector: X: 2, Y: 0, Z: -1
Vector: X: 2, Y: 0, Z: -1
Vector: X: 2, Y: -3, Z: -1
4
3
Sphere has collided!

C:\Users\will\Documents\Projects\VectorEngine\x64\Debug\VectorEngine.exe
Press any key to close this window . . .
```

Fig. 24: A picture of the console showing an example of the sphere collision detection code being used. Taken from Visual Studio 2022.

For sphere-to-plane collision, we can use the `pToPlaneDistance` function from the plane class to get the distance of the centre point of the sphere to the plane. If the distance is less than the radius of the sphere, the sphere is colliding with the plane.

```
bool Sphere::sToPlaneCollision(Plane p)
{
    //This function uses the point-to-plane distance method in the Plane class, using the sphere's vector, then checks it with the radius.
    //If the distance is less than the radius of the sphere, a collision is occurring and it returns true.

    float dist = p.pToPlaneDistance(sv);

    if (dist < radius)
    {
        std::cout << "Sphere is colliding with plane!!!" << std::endl;
        return true;
    }
    else
    {
        return false;
    }
}
```

Fig. 25: A picture of `sToPlaneCollision` function in the Sphere class. Taken from Visual Studio 2022.

```
Microsoft Visual Studio Debug Console

Plane Normal: Vector: X: -5, Y: -3, Z: 8
Sphere Current Position: Vector: X: 2, Y: 1, Z: -1
Length Of Vector: 9.8995
Normalized Vector: X: -0.505076, Y: -0.303046, Z: 0.808122
Dot Product: -2.12132
AX + BY + CZ + D = -2.12132
Length Of Vector: 1
Distance from Plane: -2.12132
Sphere is colliding with plane!!!

C:\Users\will\Documents\Projects\VectorEngine\x64\Debug\Vector
Press any key to close this window . . .
```

Fig. 26: A picture of the console showing the sphere-to-plane collision detection code in use. Taken from Visual Studio 2022.

Testing

Once the code was completed, a series of tests were done with the functions to ensure that they were working as intended. These tests were given specific values and were to

return results calculated beforehand manually, to check that the code could also perform these functions correctly. For example, a function was created called `testPlaneNormal`, which checks whether the code will output an expected plane normal result by giving it two specific vectors. The code was run and the output provided was compared with the expected results, where it was shown to have passed. All these tests were logged in a test sheet document. These functions can be altered to further test the program and the engine.

Test 8: Plane Normal

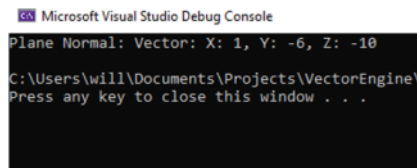
Code:

```
void testPlaneNormal()
{
    Vector v1(6, -4, 3);
    Vector v2(2, -3, 2);

    Plane plane1(v1, v2);
}
```

Expected Result: The code will work out the plane normal using the two vectors provided, which is [1, -6, -10].

Outcome:



Microsoft Visual Studio Debug Console

Plane Normal: Vector: X: 1, Y: -6, Z: -10

C:\Users\will\Documents\Projects\VectorEngine\

Press any key to close this window . . .

Test Verdict: PASSED

Fig. 27: A picture of the test sheet showing the plane normal test, with the code, expected results, outcome and test verdict. Taken from the Game Engine Test Sheet.

Conclusion

This code, which works with vectors, planes and sphere bounding volumes, hopefully can be used in the game engine as an important backbone for both rendering and handling game objects. Combined with a graphical API, these classes, functions and methods can help with computation for important functions for the game engine.

External Links

Game Engine Test Sheet: https://uowtsd-my.sharepoint.com/:w:/g/personal/2314716_student_uwtsd_ac_uk/EXJr7l5p0e1Lp4p6Go1enCsB1O4WJAXpLlnUbc9CbgBNpw?e=AM0h7u

Code: https://uowtsd-my.sharepoint.com/:u:/g/personal/2314716_student_uwtsd_ac_uk/Efil5129jodLu-XxvUyICXsBzPGnrqFH4kgCa777tMGKCA?e=hccnVj

Bibliography



- [1] Encyclopedia Britannica, "Vector," Encyclopedia Britannica, 20 December 2024. [Online]. Available: <https://www.britannica.com/science/vector-physics>. [Accessed 20 December 2024].
- [2] G. Szauer, Game Physics Cookbook, Birmingham: Packt Publishing, 2017, p. 11.
- [3] E. Weisstein, "Plane," Wolfram Mathworld, 18 December 2024. [Online]. Available: <https://mathworld.wolfram.com/Plane.html>. [Accessed 20 December 2024].
- [4] I. Millington, Game Physics Engine Development, Amsterdam: Elsevier, 2007, p. 32.
- [5] "Normalized Vector," Wolfram Mathworld, 18 December 2024. [Online]. Available: <https://mathworld.wolfram.com/NormalizedVector.html>. [Accessed 20 December 2024].
- [6] E. Weisstein, "Point-Plane Distance," Wolfram Mathworld, 18 December 2024. [Online]. Available: <https://mathworld.wolfram.com/Point-PlaneDistance.html>. [Accessed 20 December 2024].

Assignment 2

GRADEMARK REPORT

FINAL GRADE

GENERAL COMMENTS

thorough, comprehensive, well written.

Great work Will.

80 / 100

PAGE 1

PAGE 2

PAGE 3



Comment 1

In linear algebra, a coordinate vector is a representation of a vector as an ordered list of numbers (a tuple) that describes the vector in terms of a particular ordered basis

PAGE 4

PAGE 5



Comment 2

this could be updated to use operator overloading as follows:

```
Vector3D Vector3D::operator+(const Vector3D& other) const {  
    return Vector3D(x + other.x, y + other.y, z + other.z);  
}
```

PAGE 6



Comment 3

this could have been an operator overload

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

PAGE 7

PAGE 8

PAGE 9



Comment 4

Great research

PAGE 10

PAGE 11

PAGE 12

PAGE 13

PAGE 14

PAGE 15

PAGE 16

PAGE 17

PAGE 18

PAGE 19



Comment 5

A Bibliography is different to a reference list, where a reference list are things that have been referenced and a bibliography is essentially sources that have been read but not directly referenced. It can be considered a further reading list.