

Rhys Lewis Game Architecture Assignment

by Rhys Lewis

FILE	12025_RHYS_LEWIS_RHYS_LEWIS_GAME_ARCHITECTURE_ASSIGNMENT_234537_432132376.DOCX (1.07M)		
TIME SUBMITTED	07-FEB-2020 04:31PM (UTC+0000)	WORD COUNT	4633
SUBMISSION ID	119765440	CHARACTER COUNT	24105

Game Architecture and Mechanics Assignment 1

ACCA5017

RHYS LEWIS (1802146)

Table of Contents

Task 1	3
Euler's Method.....	3
Advantages:.....	3
Disadvantages:	3
Runge-Kutta Mid Point Method.....	3
Advantages:.....	3
Disadvantages:	3
Verlet Method.....	4
Advantages:.....	4
Disadvantages:	4
Comparison of Verlet and Euler	4
Euler Method Program	5
Task 2	7
Distance between two points	7
Investigation.....	7
Code implementation	7
Testing.....	7
Plane Class.....	11
Normalisation.....	11
Distance between point and plane	12
Task 3	14
Sphere Collision detection	14
Investigation.....	14
Code implementation	14
Testing.....	14
References	16
Appendix	17
Code	18
Car.h.....	18
Plane.h	20
Sphere.h	20
Vector.h.....	21
Car.cpp	22
Plane.cpp.....	Error! Bookmark not defined.
Source.cpp	25

Sphere.cpp	Error! Bookmark not defined.
Vector.cpp	25

Task 1

Euler's Method

Euler's method of numerical integration is an operation that calculates how values will change over time using timesteps. Euler's integration method calculates the path of an object by first adding up all the forces affecting it, the operation then divides that total by the mass to acquire a value for acceleration. The position is then changed by multiplying the current timestep with the current velocity. The total of this is added to the previous position. The velocity is then calculated by adding the multiplied sum of the current time step and the current acceleration. Then the process repeats with a new timestep. [3, pg. 611-614]

Advantages:

- Fairly easy to implement into code.

Disadvantages:

- Time complexity is $O(n)$, meaning the time needed is directly correlated to the number of timesteps. Larger programs would require far more time to process, making Euler's method resource-heavy. This would reduce performance as the machine's processor needs to dedicate more resources to Euler's method.
- Loses accuracy as timesteps get larger, in games that require precise calculations, such as shooters that have bullets with travel time, an inaccurate shot could deviate from any realism intended.

Runge-Kutta Mid Point Method

The Runge-Kutta integration method first uses Euler's method to calculate a midpoint, it then takes the details of the midpoint and uses them to calculate an accurate path from the beginning to the midpoint. [3, pg. 645-646]

Advantages:

- When implemented under the correct conditions, can be incredibly accurate due to its use of a pre-established midpoint.

Disadvantages:

- Time complexity is $O(n^2)$, meaning the time needed is the number of timesteps squared. This complexity is incredibly taxing and could drastically reduce performance.
- Incredibly poor error tolerance as the operation will cut directly from the beginning to the midpoint without attempting to divert from that path

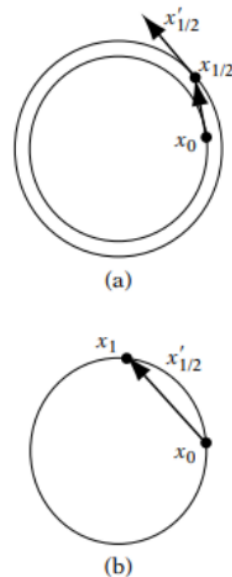


Figure 1:(a) Orbit example, showing first step of midpoint method: getting the midpoint derivative. (b) Orbit example, stepping with midpoint derivative to next estimate. Source: http://www.r-5.org/files/books/computers/algo-list/game-development/James_Van_Verth_Lars_Bishop-Essential_Mathematics-EN.pdf

Verlet Method

The Verlet integration method is used in molecular dynamics. Basic Verlet is from using the Taylor Series, a representation of a function with an infinite sum of terms which are calculated as derivatives of a single point [4]. Using the Taylor method for the previous step and combining it with the method for the current step.

Advantages:

- Can be useful to simulate small particles. This can be used to simulate the movement of soft objects like a rope. [3, pg. 647]
- Can be used for ragdoll physics
- Is time-invariant, it can be run backwards and forwards while ending in the same place

Disadvantages:

- Does not calculate velocity, meaning forces like friction will be incredibly difficult to calculate.
- Time complexity is $O(n^2)$, meaning the time needed is the number of timesteps squared. This complexity is incredibly taxing and could drastically reduce performance.

Comparison of Verlet and Euler

While Euler's method is far more efficient with a lower time complexity when compared to Verlet, Verlet is more useful for smaller calculations and, despite having no velocity calculation in its basic form, is more accurate than Euler, which requires smaller timesteps to maintain accuracy. When given the choice, the author would use Euler's to calculate more broad items and objects such as bullet trajectory while saving Verlet for more minute things such as ragdoll and rope physics.

Euler Method Program

```
void cCar::Euler()
{
    time = 0, Velocity = 0, TractiveForce = 0, DragForce = 0, Distance = 0, Acceleration = 0; //Set all variabls to 0
    while (Velocity > 0 || time <= 0.07) //Loop repeats until car stops, time <= 0.07 is there to gurantee the statement runs for the first time
    {
        //Sets drag to 0 if the car has stopped
        if (Velocity < 0)
        {
            DragForce = 0;
        }
        else
        {
            DragForce = (RollResistance * (Mass * Gravity)) + 0.5 * (DragCo * AirDensity * FrontArea * pow(Velocity, 2)); //Calculates Dragforce to slow car
        }
        Distance += Velocity * time; //Add current velocity * time to distance
        //Reduces forward force to 0 if the car has moved past the spring pushing it (8 meters)
        if (Distance > 8)
        {
            TractiveForce = 0;
        }
        else
        {
            TractiveForce = 1.5 * (16 - Distance); //Calculates foward force from spring
        }
        Acceleration = (TractiveForce - DragForce) / Mass; //Calculates total acceleration
        cout << time << " " << Distance << " " << Velocity << " " << Acceleration << endl; //Outputs results
        Velocity = (Acceleration * time) + Velocity; //Calculates current velocity

        time += 0.06; //Increases timestep
    }
}
```

Figure 2: Author's implementation of Euler's method

Above is the author's implementation of Euler's method, at the beginning of the program, all the variables needed are set to 0, constants such as gravity and mass of the vehicle are stored in the car class used. The program is set to run until the car comes to a stop (Velocity reaches 0). During each step, the method calculates the total drag force affecting the car by combining all the forces that would be affecting the car. In the project brief it is stated that the initial force of the car would be from a spring and after 8 meters, the tractive force gained would be lost, therefore, the system checks if the total distance (in meters) has surpassed 8 and once it has, reduces all tractive force to 0, otherwise, it applies a gradually depleting forward momentum. Total Acceleration is attained by subtracting the backward momentum from the forward and dividing it by the mass as a higher mass car would travel for less time due to its weight. Finally, the current velocity is calculated by adding the previous velocity to Acceleration multiplied by time, as acceleration begins to decrease and reach negative, the velocity will lower until an eventual stop. The time step then increases and the process repeats.

```
class cCar
{
private:
    float time;
    const float RollResistance = 0.21;
    const float Mass = 1;
    const float Gravity = 9.81;
    const float DragCo = 0.34;
    const float AirDensity = 1.2;
    const float FrontArea = 0.01;
    float Velocity, Acceleration, TractiveForce, DragForce, Distance;
public:
    cCar();
    ~cCar();
    void Euler();
    void Verlet();
};
```

Figure 3: Implemented Car class

When the program is executed and its values are placed into a bar graph, the following graphs are created.

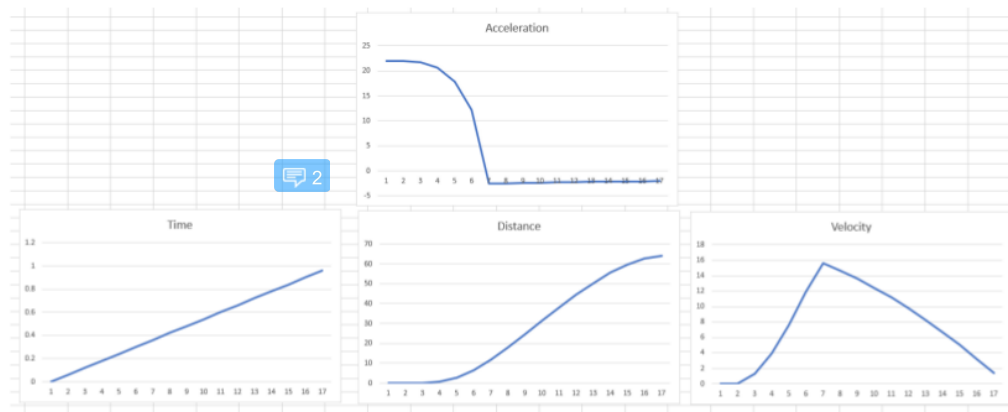


Figure 4: Graphs displaying results of author's program

The graphs show firstly, a linear progression of time, secondly a degrading acceleration given by the spring before its eventual depletion, a spike in velocity as acceleration continues to affect the car until the loss of acceleration causes the car to slow to a stop, and finally, the distance slowly beginning to climb at the beginning, but as velocity increases, distance rises faster until the complete loss of momentum slows the distance gain to a stop. The graphs contain visible lines and edges due to the nature of Euler being ineffective on larger timesteps, with smaller timesteps, the graph would smoothen and become more accurate.

Task 2

Distance between two points

Investigation

Before writing the formula into code, an understanding of the background is needed. The formula for the distance between two points in 3D space is as follows. [2, pg. 126-127]

$$\text{Distance}(\text{Point1}, \text{Point2}) = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2}$$

Figure 5: Formula for distance between 2 points

In this process, each point has its respective coordinate deducted from the other coordinates on the other point ($x_2 - x_1$). The result is squared as to make any number calculated a positive, for example, both $(3-2)^2$ and $(2-3)^2$ give you a result of 1 despite the second formula resulting in a negative value initially. Once the 3 coordinates have been deducted and squared, the results are added together and square rooted. The square root reduced the number to its initial value while keeping it as an absolute as distance cannot be negative.

Code implementation

```
void Vector::Distance(Vector & v1, Vector & v2)
{
    cout << "Distance: "<<pow(pow((v2.x - v1.x), 2) + pow((v2.y - v1.y), 2) + pow((v2.z - v1.z), 2), 0.5) << endl; //Calculates distance between 2 points
}
```

Figure 6: Author's implementation of the distance between 2 points

Implementing the formula for the distance between 2 points is a simple matter of converting the formula from equation formatting to C++ code. The pow function is used to multiply numbers to the power of a given number, for example, pow(45, 2) will multiply 45 to the power of 2, squaring it. Using this and knowing that the square root of a number can be calculated by multiplying to the power of 0.5, the author implemented the code by using the pow function on each initial calculation and then multiplied the results to the power of 0.5, square rooting the result. This solution emulates the initial formula perfectly and will output the same results as the initial equation provided the inputs are the same.

Testing

To test this formula, numbers are run through a paper test and then placed into the code, if the code is correct, the code should output the same numbers.

As the formulas all use the same input validation function, that function will only be tested here to avoid repetition.

Formula Tests

Test no.	Test inputs	Manual result	Program result	Explanation
1	P1 = (1, 5, 6) P2 = (2, 6, 4)	2.449	2.44949	$\sqrt{(2-1)^2 + (6-5)^2 + (4-6)^2}$ Sqrt($1^2+1^2+ -2^2$) Sqrt(1+1+4) Sqrt(6) 2.449

2	P1 = (-2, -8, -10) P2 = (-7, -34, -9)	26.495	26.4953	$\sqrt{(-7 - -2)^2 + (-34 - -8)^2 + (-9 - -10)^2}$ Sqrt(-5 ² + -26 ² + -1 ²) Sqrt(25+676+1) Sqrt(702) 26.495282599
3	P1 = (-4, 2, 0.6) P2 = (-2, 7, 1.8)	5.517246	5.51725	$\sqrt{(-2 - -4)^2 + (7 - 2)^2 + (1.8 - 0.6)^2}$ Sqrt(-2 ² + -5 ² + -1.2 ²) Sqrt(4+25+1.44) Sqrt(30.44) 5.5172

Input Validation

For each function that requires manual input, the ValueEntry function is called. The function first runs a loop going through each individual character of the user's input to check if it's a number, if the first character is a hyphen (representing negative) followed by a number or a period. If none of these are applicable, the system immediately sets the validity of the input to false as the only remaining possible inputs are illegal special characters. If this check is successful, the system then checks to see if the current char is a period, if it is the first entry in the input, it is an illegal character and valid is set to false as periods must have a prefix. However, if it isn't, the system checks if this is the only period in the input as numbers cannot include more than one period, it does this by setting a bool named "periodused" to true upon the first iteration of a period and if the system detects another, it deems the input as invalid. Once every individual character in the input has been individually analysed, the entire input is checked to see if it's either empty or just comprised of spaces. If either of these is true, the entry is deemed invalid.

Finally, the system checks if at any point the input has been deemed invalid, if it has, it outputs an error message and the user must input a value, if not, the input is saved and returned.

```
float Vector::ValueEntry(string message)
{
    //saves getline input to variable
    string input;
    float num;
    cout << message;
    while (true)
    {
        bool periodused = false;
        getline(cin, input);
        bool valid = true;
        for (int i = 0; i < input.length(); i++)
        {
            if (!isdigit(input[i]) || input[i] == '.' || (input[0] == '-' && !isdigit(input[1])))
            {
                if (input[i] == '.')
                {
                    if (input[0] == '-')
                    {
                        valid = false;
                        break;
                    }
                    else
                    {
                        if (periodused == false)
                        {
                            periodused = true;
                        }
                        else
                        {
                            valid = false;
                            break;
                        }
                    }
                }
            }
            else
            {
                valid = false;
                break;
            }
        }
        if (input == "" || input == " " && valid == true)
        {
            valid = false;
        }
        if (valid == false)
        {
            cout << "Please enter a valid value: ";
        }
        else
        {
            stringstream(input) >> num;
            return num;
        }
    }
}
```

Figure 7: Implementation of Input Validation

For these test conditions, accepting the input is signified by the system moving on to the next step while rejection is signified by an output of "Please enter a valid value: "

Test no.	Test Criteria	Wanted result	Actual result	Explanation
1	Input whole number	System accepts	Enter details for Vector 1: Enter x: 1 Enter y: System accepted	A whole number is a legal value, therefore, it will be accepted by the program.
2	Input negative number	System accepts	Enter details for Vector 1: Enter x: -2 Enter y: System accepted	A negative number is a legal value; therefore, it will be accepted by the program.
3	Input decimal number	System accepts	Enter details for Vector 1: Enter x: 2.5 Enter y: System accepted	A decimal number is a legal value; therefore, it will be accepted by the program.
4	Input "."	System rejects	Enter details for Vector 1: Enter x: . Please enter a valid value: System rejected	The system will initially accept the period but will then check to see if it's the first entry and upon seeing that it is, will reject the input.
5	Input "This is a test"	System rejects	Enter details for Vector 1: Enter x: This is a test Please enter a valid value: System rejected	The system will detect that the first character is a letter and invalidate the entire input.
6	Input "--"	System rejects	Enter details for Vector 1: Enter x: -- Please enter a valid value: System rejected	The system will detect that the hyphen is not followed by a number and will reject the input.
7	Input ".5"	System rejects	Enter details for Vector 1: Enter x: .5 Please enter a valid value: System rejected	The system will initially accept the period but will then check to see if it's the first entry and upon seeing that it is, will reject the input.
8	Input "2.2.2"	System rejects	Enter details for Vector 1: Enter x: 2.2.2 Please enter a valid value:	The system will initially accept

			System rejected	both periods but upon validating the first, will set the "periodused" bool to true, meaning that upon accepting the second, the system will check if the bool is true and upon seeing that it is, will reject the input.
9	Input nothing	System rejects	Enter details for Vector 1: Enter x: Please enter a valid value: System rejected	After analysing the individual characters (of which there are none) the system will check if the input is just a space or blank and upon seeing that it's blank, the system will reject the input.
10	Input just a space	System rejects	Enter details for Vector 1: Enter x: Please enter a valid value: System rejected	After analysing the individual characters (of which there are none) the system will check if the input is just a space or blank and upon seeing that it's a space the system will reject the input.
11	Input negative decimal number	System accepts	Enter details for Vector 1: Enter x: -0.5 Enter y: System accepted	A negative decimal number is a legal value; therefore, it will be accepted by the program.

Plane Class

The brief given asks for 2 methods for which the plane should be responsible. For this task, the author implemented the equation to find the distance between a point in 3d space and a plane and the normalisation of a plane.

Normalisation

Investigation

In 2D and 3D computer graphics, normalisation is often used to perform operations [1], normalisation is also used in the distance between point and plane. To find the normalised figures, each coordinate in the vector is divided by the total length of the plane's normal.

$$\text{normalised}N = N/Nlength$$

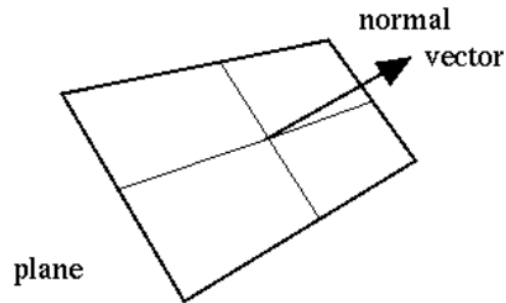


Figure 8: Plane example. source: <https://math.stackexchange.com/questions/659088/3d-game-pitch-yaw-roll-of-a-point>

Code implementation

```
float* cPlane::NormalisePlane(Vector vector, bool PointToPlane)
{
    //Checks if the function is being called as part of the point plane distance formula, if it isn't calls method to assign values to n
    if (PointToPlane == false)
    {
        assignDetails(vector);
    }
    //Declare variables
    float Normal[3];
    float nLength;

    nLength = pow(pow(vector.x, 2) + pow(vector.y, 2) + pow(vector.z, 2), 0.5); //Calculates length
    //Normalises n
    Normal[0] = vector.x / nLength;
    Normal[1] = vector.y / nLength;
    Normal[2] = vector.z / nLength;

    //if the normalisation isn't linked to the point plane distance formula, output results and return null
    if (PointToPlane == false)
    {
        cout << endl;
        for (int i = 0; i < 3; i++)
        {
            cout << Normal[i] << endl;
        }
        return nullptr;
    }
    return Normal;
}
```

Figure 9: Author's code implementation of normalisation

In the author's code, the normalisation operation is set as its own function. This function can be utilised both by itself and as a part of the distance between point and plane equation. The normalised values are stored in an array of 3 values, one for each coordinate of the vector(x, y, z). If

the function is being used for the plane distance equation, the values in Normal are returned. Otherwise, the values are simply outputted.

Testing

To test this formula, numbers are run through a paper test and then placed into the code, if the code is correct, the code should output the same numbers.

Test no.	Test inputs	Manual result	Program result	Explanation	Result
1	n = (7, 3, 4)	Length: 8.602 Normal x: 0.8137642409 Normal y: 0.3487561032 Normal z: 0.4650081376	Length: 8.6023 Normal x: 0.813733 Normal y: 0.348743 Normal z: 0.464991	$\text{Sqrt}(7^2+3^2+4^2)$ $\text{Sqrt}(74)$ 8.6023 $7/8.602 = 0.8137$ $3/8.602 = 0.3487$ $4/8.602 = 0.4649$	PASS
2	n = (-1, -3, -5)	Length: 5.916 Normal x: -0.1690331305 Normal y: -0.5070993915 Normal z: -0.8451656525	Length: 5.916 Normal x: -0.169031 Normal y: -0.507093 Normal z: -0.845154	$\text{Sqrt}(-1^2-3^2-5^2)$ $\text{Sqrt}(24)$ 5.916 $-1/5.916 = -0.169$ $-3/5.916 = -0.507$ $-5/5.916 = -0.845$	PASS
3	n = (4, -8, 2)	Length: 9.165 Normal x: 0.4364429896 Normal y: -0.8728859793 Normal z: 0.2182214948	Length: 9.165 Normal x: 0.436436 Normal y: -0.872872 Normal z: 0.218218	$\text{Sqrt}(4^2-8^2+2^2)$ $\text{Sqrt}(84)$ 9.165 $4/9.165 = -0.436$ $-8/9.165 = -0.873$ $2/9.165 = -0.218$	PASS

Distance between point and plane

Investigation

The distance between point and plane is needed for collision detection as the point in space could be representative of a sphere with a radius and the system may need to calculate the distance to see if the sphere and plane surface collide.

The simplified formula, given the normal is normalised, for the distance between a plane and point in 3D space is as follows [2, pg. 126-127]: -

$$t = (n \cdot x) - d$$

Where d = the dot product of the normalised normal on the plane and the point on the plane.

x = the point in 3d space

n = the normalised normal on the plane.

t = the shortest distance.

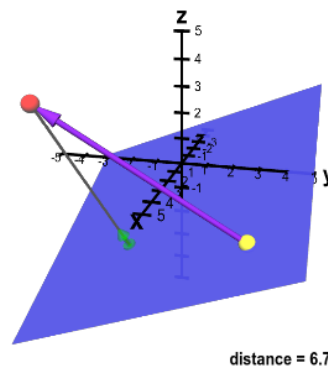


Figure 10: Example of equation. Source: https://mathinsight.org/distance_point_plane

Code implementation

```
float cPlane::DistanceBetweenPointandPlane(int x[3])
{
    //Define variables
    float *nNormalised = NULL;
    float D1;
    float D2;

    nNormalised = NormalisePlane(n, true); //Normalise n, true represents that it is being used in the point plane distance formula

    //Calculate both distances
    D1 = -((nNormalised[0] * P.x) + (nNormalised[1] * P.y) + (nNormalised[2] * P.z));
    D2 = (nNormalised[0] * x[0]) + (nNormalised[1] * x[1]) + (nNormalised[2] * x[2]);

    return fabs(D1 + D2);
}
```

Figure 11: Implementation of distance between point and plane

Here is the author's implementation of the formula, the normalisation of the normal is done in the normalisation function mentioned earlier. D1 is equal to the original formula's d . D2 is equal to $(n.x)$. The value is then made absolute to guarantee that the distance is a positive number.

Testing

To test the equation, the author will first calculate the distance manually and then use the program to run the same calculation. As the

Test no.	Test inputs	Manual result	Program result	Explanation	Result
1	x = (2, 3, 2) n = (1, 2, 1) p = (12, 0, 5)	Length: 13 Shortest Distance: 1.3077	Length: 13 Shortest Distance: 1.30769	Sqrt($7^2+3^2+4^2$) Sqrt(74) 8.6023 $7/8.602 = 0.8137$ $3/8.602 = 0.3487$ $4/8.602 = 0.4649$	Pass
2	x = (1, 2, 3) n = (1, 2, 3) p = (4, 5, 6)	Length: 3.7 Shortest Distance: 4.86	Length: 3.74166 Shortest Distance: 4.8107	Sqrt($-1^2-3^2-5^2$) Sqrt(24) 5.916 $-1/5.916 = -0.169$ $-3/5.916 = -0.507$ $-5/5.916 = -0.845$	Pass (Program is slightly more precise as handwritten calculations abbreviate the numbers to make the process easier.)

Task 3

Sphere Collision detection

Investigation

Calculating if two spheres in 3D space are colliding is a matter of using the information discussed earlier on the distance between two points and comparing the distance to the combined sum of the radii which will be present on the sphere. Below is an example of pseudocode showing how the system will work.

```
if(radius1 + radius2 < Distance), output "no collision"

else, output "collision"
```

Figure 12: Pseudocode demonstrating sphere collision

In the program, two spheres touching (edges being present on the same point) will count as a collision.

Code implementation

```
void cSphere::SphereCollision(cSphere Sphere2)
{
    float Distance;
    Distance = pow(pow(position.x - Sphere2.position.x, 2) + pow(position.y - Sphere2.position.y, 2) + pow(position.z - Sphere2.position.z, 2), 0.5); //Calculates distance between 2 spheres
    cout <<endl<< "Distance: " <<Distance<< endl;
    cout << "Combined radii: " << radius + Sphere2.radius << endl << endl; //outputs the combined radii of both spheres

    if (Distance > (radius + Sphere2.radius)) //checks if the total radii is smaller than the Distance between both spheres
    {
        cout << "No collision. Remaining distance is: " <<Distance - (radius + Sphere2.radius) << endl<<endl; //outputs no collision and shows remaining distance between both spheres
    }
    else
    {
        cout << "Collision Detected" << endl; //outputs collision
    }
}
```

Figure 13: Author's implementation of Sphere collision

Above is the author's implementation of the sphere collision method. In this code, the program will take the details of both spheres, use the theory mentioned before to calculate the distance between both points. The system will then add both radii to create a combined radius which is then compared to the distance, if the distance is greater, the system outputs a collision, otherwise, the system outputs that there is no collision and the remaining distance.

Testing

The testing process for this section will include estimations of what is supposed to happen compared to the actual results provided by the program.

Test No.	Values	Expected Result	Actual Result	Explanation
1	Point1 = (1, 2, 3) Radius1 = (5) Point2 = (1, 2, 20) Radius2= (5)	Distance: 17 Combined Radius: 10 Output: "No collision. Remaining distance is: 7"	Distance: 17 Combined Radius: 10 Output: "No collision. Remaining distance is: 7"	The combined radius is less than the distance, therefore the system outputs the remaining distance (Distance -(radius - radius2)).

2	Point1 = (2, 4, 3) Radius1 = (5) Point2 = (2, 4, 13) Radius2= (5)	Distance: 10 Combined Radius: 10 Output: "Collision"	Distance: 10 Combined Radius: 10 Output: "Collision Detected"	As the distance and radii are equal, the spheres are touching and the system detects a collision.
3	Point1 = (10, 11, 5) Radius1 = (5) Point2 = (2, 8, 7) Radius2= (20)	Distance: 8.77 Combined Radius: 25 Output: "Collision"	Distance: 8.77496 Combined Radius: 25 Output: "Collision"	As the distance is less than the radii, the system outputs a collision.

References

- [1] M. Kesson, "Vector Normalizing," 2002. [Online]. Available:
<http://www.fundza.com/vectors/normalize/>.
- [2] C. Ericson, Real time collision detection, Morgan Kaufman, 2005.
- [3] V. V. Bishop, Essential Mathematics for Games and Interactive Applications, A Programmers Guide, Morgan Kaufmann, 2015.
- [4] R. Roy, "The Discovery of the Series Formula for π by Leibniz, Gregory, Nilakantha," *Mathematics Magazine*, p. 17, 11.

Appendix

Point to plane

$$x = 2, 3, 7$$

$$p = 1, 2, 1$$

$$n = 12, 0, 5$$

$$\text{length} = |N| = \sqrt{12^2 + 0^2 + 5^2}$$

$$|N| = 13$$

$$\text{Normal } n = N^A = N / |N| = (12/13, 0/13, 5/13)$$

$$-(N^A \cdot P_0) = -((12/13)1 + (0)2 + (5/13)1)$$

$$-(N^A \cdot P_0) = -17/13$$

$$x \cdot N^A = (2(12/13) + 3(0) + 7(5/13))$$

$$x \cdot N^A = 34/13$$

$$x \cdot N^A + -(N^A \cdot P_0) = 17/13$$

$$x = 1, 2, 3$$

$$p = 4, 5, 6$$

$$n = 1, 2, 3$$

$$|N| = \sqrt{1^2 + 2^2 + 3^2} = 3.7$$

$$N^A = n / |N| = (1/3.7, 2/3.7, 3/3.7)$$

$$(0.27, 0.54, 0.81)$$

$$-(N^A \cdot P_0) = -(0.27 * 4) + (0.54 * 5) + (0.81 * 6) = -8.64$$

$$x \cdot N^A = ((1 * 0.27) + (2 * 0.54) + (3 * 0.81)) = 3.78$$

$$3.78 + -8.64 = -4.86$$

Figure 14: Author tests point to plane equation on paper with 2 examples

Distance between 2 points

$$P1 = x, y, z = 1, 2, 3$$

$$P2 = x, y, z = 4, 5, 6$$

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

$$\text{Distance} = \sqrt{(4-1)^2 + (5-2)^2 + (6-3)^2}$$

$$\text{Distance} = \sqrt{17}$$

$$\text{Distance} = 5.196$$

$$P1 = 17, 4, 1$$

$$P2 = 5, 3, 8$$

$$\text{Distance} = \sqrt{(5-17)^2 + (3-4)^2 + (8-1)^2} = 13.928$$

Figure 15: Hand calculations of distance between 2 points

Code

Car.h

```
#pragma once
#include <iostream>
#include <string>

using namespace std;

class cCar
{
private:
    //Defining constant figures
    const float RollResistance = 0.21; //Roll resistance applies friction to car
    const float Mass = 1; //Mass of the car
    const float Gravity = 9.81; //Downward force to slow the car
    const float DragCo = 0.34; //Used to quantify fluid drag
    const float AirDensity = 1.2; //Air Density of the air effecting the car
    const float FrontArea = 0.01; //Width of the car
    //Defining variables
    float time, Velocity, Acceleration, TractiveForce, DragForce, Distance;
public:
    //Defining methods
```

```
    cCar();          //Constuctor  
    ~cCar();         //Destructor  
    void Euler();  
};
```

Plane.h

```
#pragma once
#include "Vector.h"
#include <iostream>
#include <sstream>
#include <string>
using namespace std;

class cPlane : Vector
{
private:
    //Define variables
    Vector P;    //point on plane
    Vector n;    //plane

    //define methods
    void AssignN();
    float DistanceBetweenPointandPlane(int X[3]);
public:
    void AssignDistanceValues();
    float * NormalisePlane(Vector vector, bool PointToPlane);
};
```

Sphere.h

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

class cSphere : Vector
{
private:
    //Define variables
    Vector position;
    float radius;
public:
    //Define methods
    void AssignSphereValues();
    void SphereCollision(cSphere Sphere2);
};
```

Vector.h

```
#pragma once
#ifndef VECTOR_H
#define VECTOR_H
#include <string>
using namespace std;

class Vector
{
public:
    //construct
    Vector(float x, float y, float z);
    Vector(void);
    float ValueEntry(string message);
    void Scale(float s);
    void Add(Vector & v1);
    void Distance(Vector & v1, Vector & v2);
    void Add(Vector & v1, Vector & v2);
    Vector getVector(void);
    void assignDetails(Vector & v1);
    float x;
    float y;
    float z;
};
#endif
```

Car.cpp

```
#include "Car.h"

cCar::cCar()
{
    //Declare variables
    time = 0, Velocity = 0, TractiveForce = 0, DragForce = 0, Distance = 0;
}

cCar::~cCar()
{
}

void cCar::Euler()
{
    time = 0, Velocity = 0, TractiveForce = 0, DragForce = 0, Distance = 0,
    Acceleration = 0;    //Set all variables to 0
    while (Velocity > 0 || time <= 0.07) //Loop repeats until car stops, time <= 0.07
    is there to guarantee the statement runs for the first time
    {
        //Sets drag to 0 if the car has stopped
        if (Velocity < 0)
        {
            DragForce = 0;
        }
        else
        {
            DragForce = (RollResistance* (Mass*Gravity)) +
            0.5*(DragCo*AirDensity*FrontArea*pow(Velocity, 2));    //Calculates Dragforce to slow
            car
        }
        Distance += Velocity * time;    //Add current velocity * time to
        distance
        //Reduces forward force to 0 if the car has moved past the spring
        pushing it (8 meters)
        if (Distance > 8)
        {
            TractiveForce = 0;
        }
        else
        {
            TractiveForce = 1.5*(16 - Distance);    //Calculates foward
            force from spring
        }
        Acceleration = (TractiveForce - DragForce) / Mass;    //Calculates
        total acceleration
        cout << time<<" "<< Distance<<" "<< Velocity<<" "<< Acceleration <<
        endl;    //Outputs results
        Velocity = (Acceleration*time) + Velocity;    //Calculates current
        velocity

        time += 0.06; //Increases timestep
    }
}
```

Plane.cpp

```
#pragma once
#include <iostream>
#include <sstream>
#include <iostream>
#include <sstream>
#include <string>
#include "Plane.h"
#include <math.h>
using namespace std;

void cPlane::AssignN()
{
    //Enters cooderinates of n using methods
    n.x = n.ValueEntry("Enter n1: ");
    n.y = n.ValueEntry("Enter n2: ");
    n.z = n.ValueEntry("Enter n3: ");
}

void cPlane::AssignDistanceValues()
{
    //defining variables
    int X[3];
    float Dis;

    //Enters cooderinates of X using methods
    X[0] = ValueEntry("Enter X1: ");
    X[1] = ValueEntry("Enter X2: ");
    X[2] = ValueEntry("Enter X3: ");

    cout << endl;
    //Enters cooderinates of P using methods
    P.x = ValueEntry("Enter P1: ");
    P.y = ValueEntry("Enter P2: ");
    P.z = ValueEntry("Enter P3: ");

    cout << endl;

    AssignN();    //Assigns values of N

    //Assigns variable Dis to the distance between both points and outputs it
    Dis = DistanceBetweenPointandPlane(X);
    cout << "Shortest Distance: " << Dis << endl;
    return;
}

float* cPlane::NormalisePlane(Vector vector, bool PointToPlane)
{
    //Checks if the function is being called as part of the point plane distance
    formula, if it isn't calls method to assign values to n
    if (PointToPlane == false)
    {
        assignDetails(vector);
    }
    //Declare variables
    float Normal[3];
    float nLength;

    nLength = pow(pow(vector.x, 2) + pow(vector.y, 2) + pow(vector.z, 2), 0.5);
    //Calculates length
```

```

        cout << endl << "Length: " << nLength << endl << endl;
        //Normalises n
        Normal[0] = vector.x / nLength;
        Normal[1] = vector.y / nLength;
        Normal[2] = vector.z / nLength;

        //If the normalisation isn't linked to the point plane distance formula, output
        results and return null
        if (PointToPlane == false)
        {
            cout << endl;
            //cout all normals
            for (int i = 0; i < 3; i++)
            {
                cout << Normal[i] << endl;
            }
            return nullptr;    //Return nullptr
        }
        return Normal;
    }

float cPlane::DistanceBetweenPointandPlane(int X[3])
{
    //Define variables
    float *nNormalised = NULL;
    float D1;
    float D2;

    nNormalised = NormalisePlane(n, true);    //Normalise n, true represents that it
    is being used in the point plane distance formula

    //Calculate both distances
    D1 = -((nNormalised[0] * P.x) + (nNormalised[1] * P.y) + (nNormalised[2] *
P.z));
    D2 = (nNormalised[0] * X[0]) + (nNormalised[1] * X[1]) + (nNormalised[2] *
X[2]);

    return fabs(D1 + D2);    //Return absolute distance
}

```

Source.cpp

```
#include <iostream>
#include <sstream>
#include <string>
#include "vector.h"
#include "Sphere.h"
#include "Plane.h"
#include "Car.h"
using namespace std;

int main(int argc, char *argv())
{
    //Defining variables
    cSphere Sphere1;
    cSphere Sphere2;
    cPlane testPlane;
    Vector testVector1;
    Vector testVector2;
    Vector planeVector;
    cCar testCar;
    stringstream stream;
    string input;
    int choice, choice2;
    bool loop = true;

    //Menu to select options, used to access different parts of the program
    while (true) //infinite loop unless exited through case 5
    {
        cout << endl << "Please select an option: " << endl
        << "1. Calculate distance between point in space and plane" <<
endl
        << "2. Find the distance between 2 points" << endl
        << "3. Normalise a Vector" << endl
        << "4. Sphere collision detection" << endl
        << "5. Car simulation (Euler)" << endl
        << "6. Exit" << endl;
        //sets input to variable choice
        getline(cin, input);
        stream.str(input);
        stream >> choice;
        stream.clear();

        switch (choice)
        {
            case 1:
                testPlane.AssignDistanceValues(); //Calls method to begin Point
to plane calculation process
                break;
            case 2:
                //Assigns values to Vectors
                cout << "Enter details for Vector 1: " << endl;
                testVector1.assignDetails(testVector1);
                cout << "Enter details for Vector 2: " << endl;
                testVector2.assignDetails(testVector2);

                testVector1.Distance(testVector1, testVector2); //Calls method to
calculate distance
                break;
            case 3:
                testPlane.NormalisePlane(planeVector, false); //Calls method to
normalise plane, true signifies that this isn't part of the point to plane equation
```

```

        break;
    case 4:
        //Enter details of Spheres
        cout << "Enter details of Sphere 1: " << endl;
        Sphere1.AssignSphereValues();
        cout << endl<<"Enter details of Sphere 2: " << endl;
        Sphere2.AssignSphereValues();

        Sphere1.SphereCollision(Sphere2); //Calls method to compare
        Sphere to another Sphere
        break;
    case 5:
        testCar.Euler();
        break;
    case 6:
        exit(0); //Ends program
        break;
    default:
        cout << "please enter a valid option" << endl; //Tells
        user that that input was invalid
        break;
    }
}
}

```

Sphere.cpp

```
#pragma once
#include <iostream>
#include <sstream>
#include <iostream>
#include <sstream>
#include <string>
#include "Sphere.h"
using namespace std;

void cSphere::AssignSphereValues()
{
    //Calls method to enter details of a sphere
    position.x = position.ValueEntry("Enter x: ");
    position.y = position.ValueEntry("Enter y: ");
    position.z = position.ValueEntry("Enter z: ");

    cout << "Enter radius (more than 0.0)" << endl;
    while (radius <= 0) //Keeps asking for radius if user tries to input something
equal to or lower than 0
    {
        radius = position.ValueEntry("Enter Radius: ");
    }
}

void cSphere::SphereCollision(cSphere Sphere2)
{
    float Distance;
    Distance = pow(pow((position.x - Sphere2.position.x), 2) + pow((position.y -
Sphere2.position.y), 2) + pow((position.z - Sphere2.position.z), 2), 0.5);
    //Calculates distance between 2 spheres
    cout << endl << "Distance: " << Distance << endl;
    cout << "Combined radius: " << radius + Sphere2.radius << endl << endl;
    //outputs the combined radii of both spheres

    if (Distance > (radius + Sphere2.radius)) //checks if the total radii is smaller
than the Distance between both spheres
    {
        cout << "No collision. Remaining distance is: " << Distance - (radius +
Sphere2.radius) << endl << endl; //outputs no collision and shows remaining distance
between both spheres
    }
    else
    {
        cout << "Collision Detected" << endl; //outputs collision
    }
    radius = 0;
    Sphere2.radius = 0;
}
```

Vector.cpp

```
#include "vector.h"
#include <iostream>
#include <sstream>
#include <string>
using namespace std;
//construct

//Code provided by lecturer
Vector::Vector(void) {}

float Vector::ValueEntry(string message)
{
    //saves getline input to variable
    string input;
    float num;
    cout << message;
    while (true)
    {
        bool periodused = false;
        getline(cin, input);
        bool valid = true;
        for (int i = 0; i < input.length(); i++)
        {
            if (isdigit(input[i]) || input[i] == '.' || (input[0] == '-' &&
(isdigit(input[1]))))
            {
                if (input[i] == '.')
                {
                    if (input[0] == '.')
                    {
                        valid = false;
                        break;
                    }
                    else
                    {
                        if (periodused == false)
                        {
                            periodused = true;
                        }
                        else
                        {
                            valid = false;
                            break;
                        }
                    }
                }
            }
        }
        else
        {
            valid = false;
            break;
        }
    }
    if (input == "" || input == " " && valid == true)
    {
        valid = false;
    }
    if (valid == false)
    {

```

```

        cout << "Please enter a valid value: ";
    }
    else
    {
        stringstream(input) >> num;
        return num;
    }
}

void Vector::assignDetails(Vector & v1)
{
    //assigns values to vector
    v1.x = ValueEntry("Enter x: ");
    v1.y = ValueEntry("Enter y: ");
    v1.z = ValueEntry("Enter z: ");
}

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;
}

//Code Written by author

void Vector::Distance(Vector & v1, Vector & v2)
{
    cout << "Distance: "<<pow(pow((v2.x - v1.x), 2) + pow((v2.y - v1.y), 2) +
pow((v2.z - v1.z), 2), 0.5) << endl;    //Calculates distance between 2 points
}

```

Rhys Lewis Game Architecture Assignment

GRADEMARK REPORT

FINAL GRADE

GENERAL COMMENTS

Instructor

69/100

PAGE 1

PAGE 2

PAGE 3

PAGE 4



Comment 1

reference this please

PAGE 5

PAGE 6

PAGE 7



Comment 2

Label your graphs' axes and include units. -

PAGE 8

PAGE 9

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

PAGE 28

PAGE 29

PAGE 30

GRADING FORM: SMU GENERIC_GJ

RHYS LEWIS

69

RESEARCH-INFORMED LITERATURE

Some good refs here. Key topics covered and



discussed well.

7

KNOWLEDGE AND UNDERSTANDING



Report provides a detailed account of methods and requisite SE skills demonstrated in code. Good.

15

ANALYSIS



Methods for integration contrasted . Back up claims/assertions with refs.. Test schedule provided that suitably tests the code. Good. Design of code solution shows good analysis of problem.

14.5

PRACTICAL APPLICATION AND DEP.



Some methods do more than theory should - don't output results in a method that just calculates a value. Outputs to console only - not to file. Only Euler's method documented.

25.5

SKILLS FOR PROFESSIONAL PRAC.



Detailed report - well structured. Refs good.

7