

1602835 Jake Russell Game architecture

by Jake Russell

FILE	12019_JAKE_RUSSELL_1602835_JAKE_RUSSELL_GAME_ARCHITECTUR E_234537_554910917.PDF (590.9K)		
TIME SUBMITTED	07-FEB-2020 11:41PM (UTC+0000)	WORD COUNT	2425
SUBMISSION ID	119781642	CHARACTER COUNT	14375

Games Architecture and Mechanics

Jake Russell

1602835

Table of Contents

Task 1	3
Euler's method	3
Task 2	5
Section 1:.....	5
Section 2.1:	5
Section 2.2.....	7
Task 3	7
References	8
Appendix	8
Dragster.cpp.....	8
Sphere.cpp	9
Plane.cpp.....	9
Sphere.h.....	10
Main.cpp	10
Dragster.h	12
Vector.cpp.....	12
Plane.h	13
Vector.h.....	13

Task 1

Euler's method

Euler's method is a numerical integration used for calculating the physics of an object, according to Dr. Abebe Geletu the advantages of Euler's method are as follows "Eulers method is simple and direct" and it "Can be used for nonlinear IVPs" [1]. According to the same doctor the disadvantages are "it is less accurate and numerically unstable", "approximation error is proportional to the step size h . Hence, good approximation is obtained with a very small h . This requires a large number of time discretization leading to a larger computation time", and "Usually applicable to explicit differential equations" [1]. Based on what Dr. Geletu has stated Eulers is best used for shorter time spans and only when great accuracy is not required. Another method would be the verlet method, according to this document the advantages of using the verlet method are "it is straight forward" and "the storage requirements are modest" [2]. According to the same document the disadvantage is "the algorithm is of moderate precision" [2]. Comparing this to Euler's method the author would recommend using the Euler's as the verlet method is of big O notation (n^2) which is far worse run time than Euler's.

```
void Dragster::eulers()
{
    V = 0, Fd = 0, Ft = 0, x = 0, t = 0, a = 0;
    cout << "Time    Distance    Velocity    Acceleration" << endl;
    while (V > 0 || t < 0.06) //while loop to calculate eulers
    {
        if (V < 0) //if velocity is less than 0 make drag force 0
        {
            Fd = 0;
        }
        else
        {
            Fd = (Cr*(m*g) + (0.5*(Cd*p*A*(V*V)))); //calculates drag force
        }
        x = x + (V*t); //calculate distance
        if (x > 8) //if distance is greater than 8 then traction force is 0
        {
            Ft = 0;
        }
        else
        {
            Ft = 1.5*(16 - x); // calculate traction force
        }
        a = (Ft - Fd) / m; //calculate acceleration

        cout << t << "    " << x << "    " << V << "    " << a << endl; //output values
        V = V + (a * t); //calculate velocity

        t += 0.06; //increment time by 0.06
    }
}
```

Figure 1: Author's implementation of a simulation of a dragster

The code above was taken from the author's solution, it creates a function called euler's which performs the formulas in the assignment brief along with if statements which makes the drag force 0 if the velocity is less than 0 and the traction force 0 if the distance is greater than 8. This is done to have the dragster come to a stop in a reasonable time.

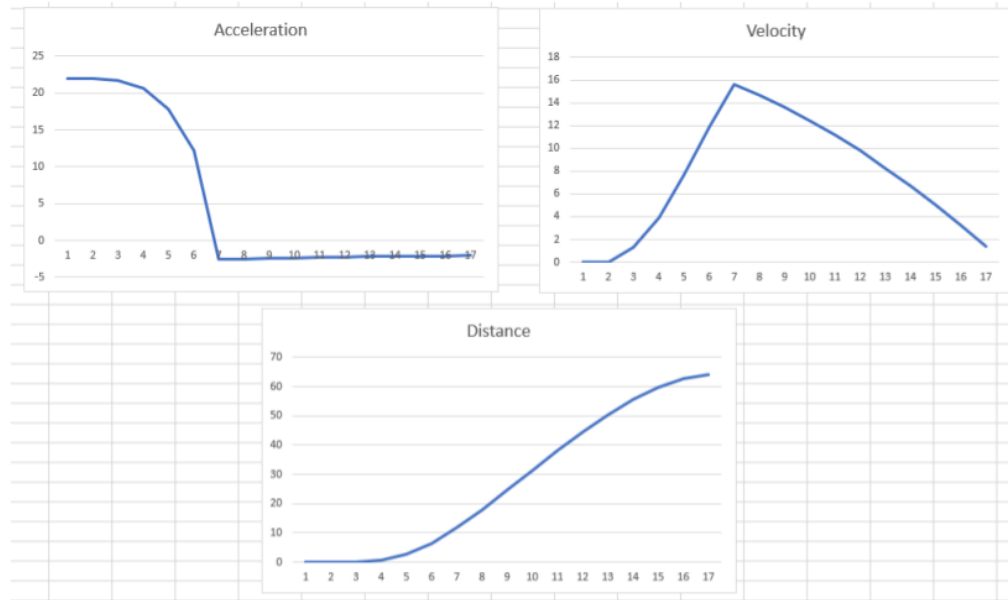


Figure 2: Graphed results of simulation

The graphs above are the plotted results of the simulation. The acceleration graph shows a high start to a rapid decline until the acceleration is negative, this was expected as the dragster was supposed to rapidly slow down once reaching a distance of 8. The velocity graph rapidly increases and then decreases relatively quick until stopped, this was expected as the initial acceleration would make the dragster quickly gain speed but once this distance is 8 the acceleration is negative making the dragster come to a halt. The distance graph gradually increases until tapering off towards the end, this was expected as the force from the initial acceleration allow the dragster to keep moving until the drag force eventually makes the car slow down enough to make it move forward gradually and then to a stop.

Task 2

Section 1:

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

Figure 3: Formula for distance between two points. Source: https://www.mathwarehouse.com/algebra/distance_formula/index.php

The formula above is the calculation for the distance between two points. You subtract the x, y and z coordinates from one point with the x, y and z points of another, and then square the new x, y and z coordinates, add them together and finally square root the result. This is especially useful in a games mechanics as it is used to find out whether two objects have collided. For example, in a game which involves shooting guns, you would calculate the distance between the point of the bullet being fired and target, and if they reach a distance of 0 you know that the bullet hit the target.

```
float Vector::ToPDistance(Vector & v2)
{
    return sqrt(pow((x - v2.x),2) + pow((y - v2.y),2) + pow((z - v2.z),2));
}
```

Figure 4: Author's implementation of the formula

The code above was taken from the author's solution, it essentially carries out the distance between two points formula. The function is sent the two points, it then gets the difference between each point, squares them, and then square roots the sum of those values.

Test	Coordinate Inputs	Expected Result	Actual Result
1	1,2,1 1,2,1	0	0
2	1,2,1 1,2,2	1	1
3	1,1,1 10,10,10	15.5.885	15.5885

Section 2.1:

$$\begin{aligned} d &= |\mathbf{v} \cdot \mathbf{n}| \\ &= |(x_1 - x_0, y_1 - y_0, z_1 - z_0) \cdot \mathbf{n}| \\ &= \frac{|A(x_1 - x_0) + B(y_1 - y_0) + C(z_1 - z_0)|}{\sqrt{A^2 + B^2 + C^2}} \end{aligned}$$

Figure 5: Formula for distance from point to plane. Source: https://mathinsight.org/distance_point_plane

The above formula is how to calculate the distance from one point to a plane. To get the distance you dot product $\mathbf{v}$ and $\mathbf{n}$, where $\mathbf{v}$ is the difference between the x, y, and z coordinates of the point off the plane and the x, y and z coordinates of the point on the plane, and where $\mathbf{n}$ is the normalisation of the normal vector. This is useful when figuring out whether an object is colliding with a plane. For example, if an item is dropped, the equation would tell you that item is on the floor if the distance between the item and the plane representing the floor is 0.

```

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

using namespace std;

Plane::Plane()
{
}

Plane::~Plane()
{
}

float Plane::planeEquation(Vector & v3) //function which does the plane equation
{
    Vector Norm;
    //subtract the point off the plane by the point on the plane
    v.x = (v3.x - o.x);
    v.y = (v3.y - o.y);
    v.z = (v3.z - o.z);
    Norm = normalise(n); //Set the vector "Norm" by the normalised normal vector
    return abs((v.x * Norm.x) + (v.y * Norm.y) + (v.z * Norm.z)); //returns the calculated distance
}

```

Figure 6: Author's implementation of the formula

The code above was taken from the author's solution. The function called planeEquation, creates a vector called "Norm", then gets the values for v by getting the difference between the x, y, and z coordinates of the point off the plane and the x, y and z coordinates of the point on the plane. It then get's the Norm vector by calling the normalise function sending the normal vector. It then returns the absolute of the dot product of v and n.

Test	Numerical values inputted	Expected Value	Actual Value
1	Normal: 1,1,1 On the plane: 1,1,1 Off the plane: 1,1,1	0	0
2	Normal: 1,1,1 On the plane: 0,0,0 Off the plane: 10,10,10	17.3	17.3205
3	Normal: 2,2,2 One the plane: 10,10,10 Off the plane: 1,1,1	15.59	15.5885

Section 2.2

$$\hat{u} = \frac{\vec{u}}{||\vec{u}||}$$

Figure 7: Formula to normalise a vector. Source: <https://www.khanacademy.org/computing/computer-programming/programming-natural-simulations/programming-vectors/a/vector-magnitude-normalization>

The above formula is to normalise a vector, it does this by taking the vector and dividing it by its length, the length being Pythagoras' theorem [1] of the vector. This is because dividing a vector by the length will make the length of the vector 1. In the context of this program this is used in order to calculate the distance from a point to a plane. In a gaming context this is used to allow the vector's magnitude to be changed without affecting the direction, this means you can have an object move at an exact speed despite the direction it is travelling.

```
Vector Plane::normalise(Vector & v1) //function to normalise a vector
{
    float length;
    Vector Norm;
    length = sqrt((v1.x * v1.x) + (v1.y * v1.y) + (v1.z * v1.z)); //calculates length of a vector
    Norm.x = v1.x / length;
    Norm.y = v1.y / length;
    Norm.z = v1.z / length;
    return Norm;
}
```

Figure 8: Author's implementation of the formula

The above code was taken from the author's solution, the function called normalise, get the length if the vector sent to the function, it then divides the x, y, and z coordinates of that vector by the length and saves the calculated results into a vector call "Norm" and then returns the Norm vector.

Task 3

To find out whether two spheres are touching you need to find the distance between the two points of the spheres and compare the value to the sum of the two sphere's radii. The author assumes that if the distance is less than or equal to the sum of the radii, then the spheres are colliding. The formula as mentioned above to find the distance between two points.

```
void Sphere::Collision(Sphere & s2) //function to calculate collision
{
    if (point.ToPDistance(s2.point) <= (radius + s2.radius)) // if the distance between two points is less than or equal to the sum of the radii then out put there is a collision
    {
        cout << "The Sphere's are colliding" << endl;
    }
    else
    {
        cout << "The Sphere's are not colliding" << endl;
    }
}
```

Figure 9: Author's implementation of the method

The code above uses an if statement to compare the distance between two points with the sum of their radii, if the distance is less than or equal to the sum then it outputs to the user that the spheres are colliding, else if outputs to the user that they are not colliding.

References

- [1] D. A. Geletu, "tu-ilmenau," [Online]. Available: https://www.tu-ilmenau.de/fileadmin/media/simulation/Lehre/div/Lec_Slides3.pdf. [Accessed 7 February 2020].
- [2] [Online]. Available: <http://students.iitk.ac.in/projects/wiki/lib/exe/fetch.php?media=2014as:verlet.pdf>. [Accessed 7 February 2020].
- [3] "bbc," [Online]. Available: <https://www.bbc.co.uk/bitesize/guides/z3g9q6f/revision/3>. [Accessed 7 February 2020].

Appendix

Dragster.cpp

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

using namespace std;

Dragster::Dragster()
{
}

Dragster::~Dragster()
{
}

void Dragster::eulers()
{
    V = 0, Fd = 0, Ft = 0, x = 0, t = 0, a = 0;
    cout << "Time Distance Velocity Acceleration" << endl;
    while (V > 0 || t < 0.06) //while loop to calculate eulers
    {
        if (V < 0) //if velocity is less than 0 make drag force 0
        {
            Fd = 0;
        }
        else
        {
            Fd = (Cr*(m*g) + (0.5*(Cd*p*A*(V*V)))); //calculates drag force
        }
        x = x + (V*t); //calculate distance
        if (x > 8) //if distance is greater than 8 then traction force is 0
        {
            Ft = 0;
        }
        else
        {
            Ft = 1.5*(16 - x); // calculate traction force
        }
        a = (Ft - Fd) / m; //calculate acceleration
    }
}
```

```

        cout << t << "          " << x << "          " << V << "          " << a << endl; //output
values
        V = V + (a * t); //calculate velocity
        t += 0.06; //increment time by 0.06
    }
}

```

Sphere.cpp

```

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

using namespace std;
Sphere::Sphere()
{
}

Sphere::~Sphere()
{
}

void Sphere::Collision(Sphere & s2) //function to calculate collision
{
    if (point.ToPDistance(s2.point) <= (radius + s2.radius)) // if the distance
between two points is less than or equal to the sum of the radii then out put there is
a collision
    {
        // This program assumes two spheres touching is a collision

        cout << "The Sphere's are colliding" << endl;
    }
    else
    {
        cout << "The Sphere's are not colliding" << endl;
    }
}

```

Plane.cpp

```

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

using namespace std;

Plane::Plane()
{
}

Plane::~Plane()
{
}

float Plane::planeEquation(Vector & v3) //function which does the plane equation
{
    Vector Norm;
    //subtract the point off the plane by the point on the plane
    v.x = (v3.x - o.x);
    v.y = (v3.y - o.y);
    v.z = (v3.z - o.z);
}

```

```

        Norm = normalise(n); //Set the vector "Norm" by the normalised normal vector
        return abs((v.x * Norm.x) + (v.y * Norm.y) + (v.z * Norm.z)); //returns the
        calculated distance

```

```

}

```

```

Vector Plane::normalise(Vector & v1) //function to normalise a vector

```

```

{
    float length;
    Vector Norm;
    length = sqrt((v1.x * v1.x) + (v1.y * v1.y) + (v1.z * v1.z)); //calculates
    length of a vector
    Norm.x = v1.x / length;
    Norm.y = v1.y / length;
    Norm.z = v1.z / length;
    return Norm;
}

```

```

Sphere.h

```

```

#pragma once
#include "Vector.h"
//creat sphere class with inheritance from class vector and create the variables
class Sphere: Vector
{
public:
    Sphere();
    ~Sphere();
    Vector point;
    float radius;
    void Collision(Sphere & s2);

private:
};

```

```

Main.cpp

```

```

#include <iostream>
#include <sstream>
#include <string>
#include "Vector.h"
#include "Plane.h"
#include "Sphere.h"
#include "Dragster.h"

//create the variables and functions
using namespace std;
string input;
void getCoords(Vector & v1);
Vector v1;
Vector v2;
Vector v3;
Plane p1;
Sphere s1;
Sphere s2;
Dragster d1;
float r1;
float r2;
int main()

```

```

{
    //create a switch statement around the variable "choice" to make a user menu
    int choice;
    cout << "Press the appropriate number for the operation you wish to do: " <<
endl;
    cout << "1. Dragster" << endl;
    cout << "2. Distance between two vectors" << endl;
    cout << "3. Plane" << endl;
    cout << "4. Sphere" << endl;
    getline(cin, input);
    stringstream(input) >> choice;
    switch (choice)
    {
        case 1:
            d1.eulers(); //call the function "eulers" from the dragster class
            break;
        case 2:
            //get the coordinates from the user
            cout << "Please enter the coordinates for the first vector" << endl;
            getCoords(v1);
            cout << "Please enter the coordinates for the second vector" << endl;
            getCoords(v2);
            cout << v1.ToPDistance(v2) << endl; //call the function ToPDistance from
the vector class and output the result
            break;
        case 3:
            //get the coordinates from the user
            cout << "Please enter the coordinates for the normal" << endl;
            getCoords(p1.n);
            cout << "Please enter the coordinates for the point on the plane" <<
endl;
            getCoords(p1.o);
            cout << "Please enter the coordinates for the point off the plane" <<
endl;
            getCoords(v3);
            cout << p1.planeEquation(v3) << endl; //call the function ToPDistance
from the vector class and output the result
            break;
        case 4:
            //get the coordinates and the radius from the user
            cout << "Please enter the coordinates of the point for the first sphere"
<< endl;
            getCoords(s1.point);
            cout << "Please enter the radius of the first sphere" << endl;
            getline(cin, input);
            stringstream(input) >> s1.radius;
            cout << "Please enter the coordinates of the point for the first sphere"
<< endl;
            getCoords(s2.point);
            cout << "Please enter the radius of the first sphere" << endl;
            getline(cin, input);
            stringstream(input) >> s2.radius;
            s1.Collision(s2); // call the collision function from the sphere class
            break;
    }

    system("pause");
    return 0;
}
void getCoords(Vector & v1) //function to get coordinates from users
{

```

```

        cout << "x coordinate: " << endl;
        getline(cin, input);
        stringstream(input) >> v1.x;
        cout << "Y coordinate: " << endl;
        getline(cin, input);
        stringstream(input) >> v1.y;
        cout << "Z coordinate: " << endl;
        getline(cin, input);
        stringstream(input) >> v1.z;
    }

```

Dragster.h

```

#pragma once
class Dragster //create dragster class
{
public: //create the variables and functions
    Dragster();
    ~Dragster();
    const float Cr = 0.21;
    const float m = 1;
    const float g = 9.81;
    const float p = 1.2;
    const float A = 0.01;
    const float Cd = 0.34;
    float a;
    float V;
    float Fd;
    float Ft;
    float x;
    float t;
    void eulers();
private:
};

```

Vector.cpp

```

#include "vector.h"
using namespace std;
#include <string>
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;
}
float Vector::ToPDistance(Vector & v2) //function to calculate the distance between
two points
{
    return sqrt(pow((x - v2.x),2) + pow((y - v2.y),2) + pow((z - v2.z),2));
}

```

Plane.h

```

#pragma once
#include "Vector.h"

//create a Plane class
class Plane
{
public: //create the variables and functions
    Plane();
    ~Plane();
    float planeEquation(Vector & v3);
    Vector normalise(Vector & v1);
    Vector n;
    Vector o;
    Vector v;
};

```

Vector.h

```

#ifndef VECTOR_H
#define VECTOR_H

class Vector
{
public:

    Vector(float x, float y, float z);
    Vector(void);
    void Scale(float s);
    void Add(Vector & v1);
    void Add(Vector & v1, Vector & v2);
    float ToPDistance(Vector & v2);
    Vector getVector(void);
    float x;
    float y;
    float z;
};
#endif
#pragma once

```

1602835 Jake Russell Game architecture

GRADEMARK REPORT

FINAL GRADE

GENERAL COMMENTS

Instructor

59/100

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

GRADING FORM: SMU GENERIC_GJ

JAKE RUSSELL

58.50

RESEARCH-INFORMED LITERATURE

 Key topics identified and discussed. Refs online - avoid these - use established texts such as Bishop and van Verth.

5.5

KNOWLEDGE AND UNDERSTANDING

 Essential ideas covered in report. Code produces results which need checking - total time used at each time step - should use the time step Δt (eg 0.06 seconds).

12.5

ANALYSIS

 See above. Results graphed and discussed - could be more thorough/clear here. Otherwise a good discussion.

12.5

PRACTICAL APPLICATION AND DEP.

 Output result to a file. Wrong time used. Vector and sphere class completed. Good.

25

SKILLS FOR PROFESSIONAL PRAC.

 Report structured. Avoid on-line refs.

3

