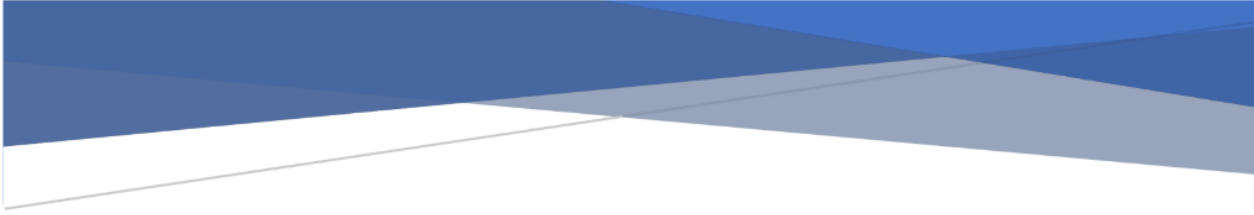


Kyle Lovell Report

by Kyle Lovell

FILE	12002_KYLE_LOVELL_KYLE_LOVELL_REPORT_234537_510505897.DOC X (248.09K)		
TIME SUBMITTED	07-FEB-2020 10:32PM (UTC+0000)	WORD COUNT	2570
SUBMISSION ID	119779106	CHARACTER COUNT	12125



GAMES ARCHITECTURE AND MECHANICS

ACCA5017

Kyle Lovell (1802143)
1802143@student.uwtsd.ac.uk

Contents

Task 1	2
Euler	2
Backward Euler	2
Author Code (Euler's Method)	2
Results	3
Task 2	4
Point to Point Distance	4
Plane Equation	4
Dot Product	5
Task 3	6
Sphere Collision Detection	6
Appendix	7
Code	7
Source.cpp	7
Header.h	11
Vector.cpp	11
Vector.h	13
Plane.cpp	13
Plane.h	14
Sphere.cpp	14
Sphere.h	15
Bibliography	15

Task 1

Euler

Euler's method is a numerical integrator, the simplest of the Runge-Kutta family of numerical integrators, and the most basic explicit method, used to approximate the result of a differential equation. It is used when a computer cannot use a different differential equation to find the exact equation. An advantage of Euler's method is that, when compared to other numerical integrators, it is quite simple to implement. A disadvantage of Euler's is that it is less accurate than other numerical integrators and the approximation error is proportional to the step size. In the case of the author's code, this error would be proportional to the value of time, which steps in 0.06s increments.

Backward Euler

The Backward Euler method is also one of the simplest numerical integrators of the Runge-Kutta family of numerical integrators. The main difference between the Backward Euler, and the Euler method, is that the Euler method is an explicit method.

Author Code (Euler's Method)

```
void dragsterSimulation()
{
    cout << "Time" << "\t" << "Distance" << "\t" << "Velocity" << endl;
    float distance = 0, velocity = 0, acceleration = 0, time = 0;
    float area = 0.01, density = 1.2, mass = 1, gravity = 9.81;
    float cd = 0.34, cr = 0.21, fd = 0, ft = 0;

    while (velocity > 0 || time <= 0.06)
    {
        if (velocity < 0)
            fd = 0;
        else
            fd = (cr * mass * gravity) + (0.5 * cd * density * area * pow(velocity, 2));

        distance += (velocity * time);
        if (distance > 8)
            ft = 0;
        else
            ft = 1.5*(16 - distance);

        acceleration = (ft - fd) / mass;

        cout << time << "\t" << distance << "\t" << "\t" << velocity << endl;

        velocity += (acceleration * time);

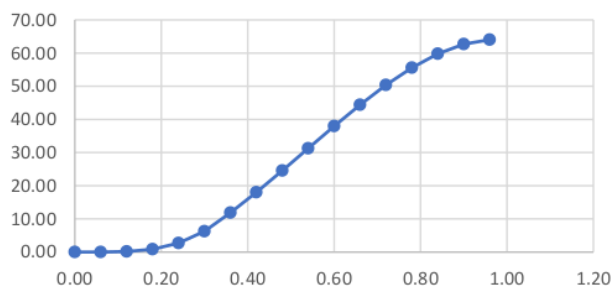
        time += 0.06;
    }
}
```

Results

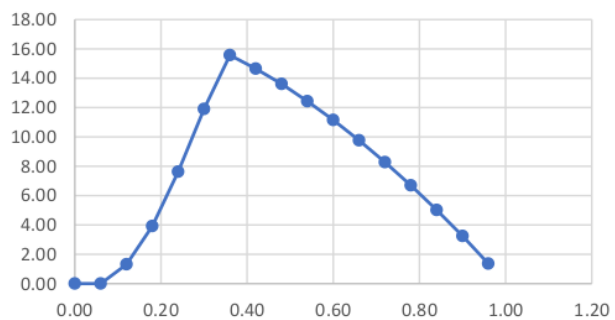
Time	Distance	Velocity
0	0	0
0.06	0	0
0.12	0.157967	1.31639
0.18	0.863625	3.92032
0.24	2.69499	7.63068
0.3	6.26426	11.8976
0.36	11.8709	15.574
0.42	18.0257	14.6542
0.48	24.556	13.605
0.54	31.2709	12.4349
0.6	37.9621	11.1521
0.66	44.4063	9.76381
0.72	50.3648	8.27579
0.78	55.5845	6.69192
0.84	59.7961	5.01378
0.9	62.7123	3.24022
0.96	64.0245	1.36686

Please press ENTER to return to the main menu.

Distance



Velocity



Task 2

Point to Point Distance

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

Figure 1: Equation for the distance between two points [1]

One of the basic equations in linear algebra is the distance between two points in 3D space.

The equation:

- Calculates the difference in the x components, squared
 - $(x_2 - x_1)^2$ – called X
- Calculates the difference in the y components, squared
 - $(y_2 - y_1)^2$ – called Y
- Calculates the difference in the z components, squared
 - $(z_2 - z_1)^2$ – called Z
- Calculates the sum of the previous values
 - $X + Y + Z$ – called sum
- Square roots the previous result
 - $\sqrt{\text{sum}}$

Plane Equation

$$\begin{aligned} d &= |\mathbf{v} \cdot \mathbf{n}| \\ &= |(\mathbf{x}_1 - \mathbf{x}_0, \mathbf{y}_1 - \mathbf{y}_0, \mathbf{z}_1 - \mathbf{z}_0) \cdot \mathbf{n}| \\ &= \frac{|A(\mathbf{x}_1 - \mathbf{x}_0) + B(\mathbf{y}_1 - \mathbf{y}_0) + C(\mathbf{z}_1 - \mathbf{z}_0)|}{\sqrt{A^2 + B^2 + C^2}}. \end{aligned}$$

Figure 2: Equation for the distance between a point and plane [2]

The plane equation, pictured above, is used for calculating the shortest distance between a plane and a point in 3D space.

The equation:

- Calculates the distance between the point in 3D space (x_1, y_1, z_1) and the point on the plane (x_0, y_0, z_0) – labelled as v.
 - This is done by subtracting the x, y, and z components of the point on the plane from the x, y, and z components of the point in 3D space
 - $(x_1 - x_0), (y_1 - y_0), (z_1 - z_0)$
- Normalises the normal to the plane (A, B, C) – labelled as n.
 - This is done by working out the length of the normal,
 - $\sqrt{A^2 + B^2 + C^2}$
 - And dividing each component of the normal by the length of the normal
 - $(A/L), (B/L), (C/L)$
- Calculates the dot product of v and n – labelled as d.
 - $(A * v_x), (B * v_y), (C * v_z)$

The reason this calculation is performed in video games is for collision detection in 3D space.

```

float Plane::PlaneEquation(Vector & p)
{
    // works out the length of the normal to the plane
    float length = sqrt(pow(normal.x, 2) + pow(normal.y, 2) + pow(normal.z, 2));

    // creates a normalised n vector from the normal and its length
    Vector n(normal.x / length, normal.y / length, normal.z / length);

    // creates a v vector, which is the vector distance between the point in space and the point on the plane
    Vector v(p.x - point.x, p.y - point.y, p.z - point.z);

    // performs a dot product on vectors v and n, stores as distance
    float distance = DotProduct(v, n);

    // if distance is negative, makes it a positive value
    if (distance < 0)
        distance *= -1;

    return distance;
}

```

As shown by the screenshot above, the author has developed a C++ function that calculates the shortest distance between a plane and a point in 3D space. The step-by-step process is shown using the comments included with the code.

Dot Product

$$\mathbf{a} \cdot \mathbf{b} = a_x \times b_x + a_y \times b_y + a_z \times b_z$$

A necessary equation in the calculations of the plane equation is the Dot Product, pictured above. It is the sum of the products of each component of the two vectors given.

The equation:

- Multiplies the x components
 - $(x_0 \times x_1)$ – Save as X
- Multiplies the y components
 - $(y_0 \times y_1)$ – Save as Y
- Multiplies the z components
 - $(z_0 \times z_1)$ – Save as Z
- Add the three results together
 - $(X + Y + Z)$

A use of Dot Product in video games is to find the difference in rotation between two objects. You would do this by performing a dot product on normal vectors of each object. For example, a result of 1 means they are of the same rotation.

```
// performs a dot product on the given v1 and v2 vectors
float Plane::DotProduct(Vector & v1, Vector & v2)
{
    return ((v1.x*v2.x) + (v1.y*v2.y) + (v1.z*v2.z));
}
```

As shown by the screenshot above, the author has developed a C++ function that calculates the Dot Product of two vectors. To do this, the program finds sum of the product of the x, y, and z components of each vector, and returns the input.

Task 3

Sphere Collision Detection

As part of the Sphere class in their code, the author has created a Collision Detection function.

The function:

- Calculates the distance between the origins of the spheres
 - Uses [Point-to-Point Distance](#) method
- Calculates the sum of the radii of the spheres
 - $(r_0 + r_1)$
- Checks if the distance is more than the sum of the radii
- If true, spheres are not colliding.

```
// checks if two spheres are colliding and, if they are, returns true
// if not, returns false
bool Sphere::CollisionDetection(Sphere s2)
{
    if (origin.pToPDistance(s2.origin) <= (radius + s2.radius))
        return true;
    else
        return false;
}
```

As shown by the screenshot above, the author has developed a C++ function that checks if the spheres are colliding. To do this, the program calls the pToPDistance method, passes it the origin Vector of the second sphere, and checks if this is less than or equal to the sum of the radii. If the distance is less than or equal to the sum, the program returns true. If the distance is more than the sum, the program returns false.

Appendix

Code

Source.cpp

```
/*
Vectors and Planes assignment code
Kyle Lovell
*/

#include "Header.h"

int floatLength(float distance)
{
    int count = 0;
    while (true)
    {
        if (to_string(distance)[count] != NULL)
            count++;
        else
            break;
    }
    return count;
}

void dragsterSimulation()
{
    cout << "Time" << '\t' << "Distance" << '\t' << "Velocity" << endl;
    float distance = 0, velocity = 0, acceleration = 0, time = 0;
    float area = 0.01, density = 1.2, mass = 1, gravity = 9.81;
    float cd = 0.34, cr = 0.21, fd = 0, ft = 0;

    while (velocity > 0 || time <= 0.06)
    {
        if (velocity < 0)
            fd = 0;
        else
            fd = (cr * mass * gravity) + (0.5 * cd * density * area *
pow(velocity, 2));

        distance += (velocity * time);
        if (distance > 8)
            ft = 0;
        else
            ft = 1.5*(16 - distance);

        acceleration = (ft - fd) / mass;

        cout << time << '\t' << distance << '\t' << '\t' << velocity << endl; //
outputs results, ignore this

        velocity += (acceleration * time);

        time += 0.06;
    }

    // function to check if user input is a float
    // if it is a float, outputs true
    // if it's a string or null value, outputs false
    bool numCheck(string input)
    {

```

```

        for (int i = 0; i < input.length(); i++)
        {
            if (!isdigit(input[i]) && input[i] != '.' && input[i] != '-')
            {
                cout << "ERROR: Please enter a number" << endl;
                return false;
            }
        }

        if (input == "")
        {
            cout << "ERROR: Please enter a number" << endl;
            return false;
        }
        return true;
    }

    // function to get float value from user
    float getNum(string message)
    {
        string input;
        float value;

        do
        {
            cout << message;
            getline(cin, input);
        } while (!numCheck(input));

        stringstream(input) >> value;
        return value;
    }

    // function to check if user input is within legal range
    bool rangeCheck(int choice, int min, int max)
    {
        if (choice < min || choice > max)
        {
            cout << "ERROR: Please enter a value between " << min << " and " << max;
            return false;
        }
        else
            return true;
    }

    int main()
    {
        // initialise variables, arrays, and classes
        int choice;
        float values[6], value[2], *values2[6];
        Vector v1, v2, v3;
        Plane p1;
        Sphere s1, s2;

        while (true)
        {
            do
            {
                // main menu
                cout << string(50, '\n');
                cout << "What would you like to do?" << endl

```

```

        << "1. Point-to-Point Distance" << endl
        << "2. Point-to-Plane Distance" << endl
        << "3. Sphere Collision Detection" << endl
        << "4. Dragster Simulation" << endl
        << "5. Exit" << endl;

        choice = getNum("Choice: ");

    } while (rangeCheck(choice, 1, 5) == false);

    cout << endl;

    switch (choice)
    {
    case 1:
        // Point-to-Point Distance
        for (int i = 0; i < 2; i++)
        {
            cout << "Please enter values for Vector " << i + 1 <<
endl;

            for (int j = 0; j < 3; j++)
            {
                values[(i * 3) + j] = getNum("Enter value: ");
            }
            cout << endl;
        }
        v1.SetValues(values[0], values[1], values[2]);
        v2.SetValues(values[3], values[4], values[5]);

        cout << "The distance between these two points is: " <<
v1.pToPDistance(v2) << endl;
        cout << "Please press ENTER to return to the main menu.";

        cin.get();
        break;
    case 2:
        //Point-to-Plane Distance
        cout << "Please enter the values for the plane" << endl <<
"Normal" << endl;

        for (int i = 0; i < 3; i++)
        {
            values[i] = getNum("Enter value: ");
        }
        cout << endl << "Origin" << endl;
        for (int i = 0; i < 3; i++)
        {
            values[i + 3] = getNum("Enter value: ");
        }
        v1.SetValues(values[0], values[1], values[2]);
        v2.SetValues(values[3], values[4], values[5]);
        p1.SetValues(v1, v2);

        cout << endl << "Please enter the values for the point" << endl;
        for (int i = 0; i < 3; i++)
        {
            values[i] = getNum("Enter value: ");
        }
        v1.SetValues(values[0], values[1], values[2]);

        cout << endl << "The distance between this point and plane is: "
<< p1.PlaneEquation(v1) << endl;

```

```

        if (p1.SideCheck(v1) == -1)
        {
            cout << "It is behind the plane." << endl;
        }
        if (p1.SideCheck(v1) == 0)
        {
            cout << "It is in front of the plane." << endl;
        }

        cout << "Please press ENTER to return to the main menu.";
        cin.get();
        break;
    case 3:
        // Sphere Collision Detection
        for (int i = 0; i < 2; i++)
        {
            cout << "Please enter the values for sphere " << i + 1 <<
endl << "Origin" << endl;
            for (int j = 0; j < 3; j++)
            {
                values[(i * 3) + j] = getNum("Enter value: ");
            }
            value[i] = getNum("Enter radius: ");
            cout << endl;
        }
        v1.SetValues(values[0], values[1], values[2]);
        v2.SetValues(values[3], values[4], values[5]);

        s1.SetValues(v1, value[0]);
        s2.SetValues(v2, value[1]);

        if (s1.CollisionDetection(s2))
            cout << "These spheres are colliding." << endl;
        else
            cout << "These spheres are not colliding." << endl;

        cout << "Please press ENTER to return to the main menu.";
        cin.get();

        break;
    case 4:
        dragsterSimulation();

        cout << "Please press ENTER to return to the main menu.";
        cin.get();

        break;
    case 5:
        //Exit Program
        cout << "Please press ENTER to exit the program.";
        cin.get();
        exit(0);
        break;
    }
}

/*float values[3];*/

/*cout << "Please enter values for first Vector" << endl;
for (int i = 0; i < 3; i++)

```

```

    {
        values[i] = getNum("Enter value: ");
    }
    Vector v1(values[0], values[1], values[2]);*/

    /*cout << "Please enter values for second Vector" << endl;
    for (int i = 0; i < 3; i++)
    {
        values[i] = getNum();
    }
    Vector v2(values[0], values[1], values[2]);

    cout << v1.pToPDistance(v2) << endl;*/

    /*Vector normal(12, 0, 5);
    Vector point(1, 2, 1);

    Plane p1(normal, point);

    cout << p1.PlaneEquation(v1) << endl;*/

    /*Vector v1(0, 0, 0);
    Vector v2(0, 0, 5);

    Sphere s1(v1, 5.0);
    Sphere s2(v2, 5.0);

    if (s1.CollisionDetection(s2))
    {
        cout << "True";
    }
    else
        cout << "False";*/

    cin.get();
    return 0;
}

```

Header.h

```
#pragma once
```

```

#include <string>
#include <sstream>
#include <iostream>

```

```

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

```

```
using namespace std;
```

Vector.cpp

```

// Allows the user of "Header.h" code in this file
#include "Header.h"

```

```

//construct
Vector::Vector(void) {};

```

```
// Vector Constructor
```

```

Vector::Vector(float myx, float myy, float myz)
{
    // Sets x, y, and z as given values myx, myy, and myz, respectively
    x = myx;
    y = myy;
    z = myz;
}

// Vector method, SetValues
void Vector::SetValues(float myx, float myy, float myz)
{
    // Sets x, y, and z as given values myx, myy, and myz, respectively
    x = myx;
    y = myy;
    z = myz;
}

// Vector method, Scale
void Vector::Scale(float s)
{
    // Multiplies x, y, and z by given value s
    x = s * x;
    y = s * y;
    z = s * z;
}

// Vector method, Add (called when 2 Vectors are passed)
void Vector::Add(Vector & v1, Vector & v2)
{
    // Sets v1.x as v2.x + v1.x
    v1.x = v2.x + v1.x;

    // Sets v1.y as v2.y + v1.y
    v1.y = v2.y + v1.y;

    // Sets v1.z as v2.z + v1.z
    v1.z = v2.z + v1.z;
}

// Vector method, getVector
Vector Vector::getVector(void)
{
    // Returns Vector with values x, y, and z
    return Vector(x, y, z);
}

// Vector method, Add (called when 1 Vector is passed)
void Vector::Add(Vector & v1)
{
    // Sets x as x + v1.x
    x = x + v1.x;

    // Sets y as y + v1.y
    y = y + v1.y;

    // Sets z as z + v1.z
    z = z + v1.z;
}

float Vector::pToPDistance(Vector & v2)
{

```

```

        return pow(pow((x - v2.x), 2) + pow((y - v2.y), 2) + pow((z - v2.z), 2), 2);
    }

```

Vector.h

```

#pragma once
#ifndef VECTOR_H
#define VECTOR_H

class Vector
{
public:

    //construct
    Vector(float x, float y, float z);
    void SetValues(float myx, float myy, float myz);
    Vector(void);
    void Scale(float s);
    void Add(Vector & v1);
    float pToPDistance(Vector & v2);
    void Add(Vector & v1, Vector & v2);
    Vector getVector(void);

    float x;
    float y;
    float z;
};
#endif

```

Plane.cpp

```

#include "Plane.h"
#include "Header.h"

//construct
Plane::Plane(void) {};

// Sets normal as given n vector, and point as given p vector
Plane::Plane(Vector & n, Vector & p)
{
    normal = n;
    point = p;
}

// performs a dot product on the given v1 and v2 vectors
float Plane::DotProduct(Vector & v1, Vector & v2)
{
    return ((v1.x*v2.x) + (v1.y*v2.y) + (v1.z*v2.z));
}

float Plane::PlaneEquation(Vector & p)
{
    // works out the length of the normal to the plane
    float length = sqrt(pow(normal.x, 2) + pow(normal.y, 2) + pow(normal.z,
2));

    // creates a normalised n vector from the normal and its length
    Vector n(normal.x / length, normal.y / length, normal.z / length);

    // creates a v vector, which is the vector distance between the point in
space and the point on the plane
    Vector v(p.x - point.x, p.y - point.y, p.z - point.z);

```

```

        // performs a dot product on vectors v and n, stores as distance
        float distance = DotProduct(v, n);

        // if distance is negative, makes it a positive value
        if (distance < 0)
            distance *= -1;

        return distance;
    }

// method to set values of Plane instance
void Plane::SetValues(Vector n, Vector o)
{
    // Sets x, y, and z as given values myx, myy, and myz, respectively
    point = o;
    normal = n;
}

// tells the user whether the point in space is in front of or behind the plane
int Plane::SideCheck(Vector p)
{
    Vector v(p.x - point.x, p.y - point.y, p.z - point.z);
    return signbit(DotProduct(normal, v)); // -1 = behind, 0 = front
}

Plane.h
#pragma once
#include "Header.h"
#ifndef PLANE_H
#define PLANE_H

class Plane
{
public:

    //construct
    Plane(Vector & normal, Vector & point);
    Plane(void);

    float DotProduct(Vector & v1, Vector & v2);
    void SetValues(Vector n, Vector point);
    int SideCheck(Vector p);
    float PlaneEquation(Vector & p);
    Vector normal;
    Vector point;
};
#endif

Sphere.cpp
#include "Header.h"

//construct
Sphere::Sphere(void) {};

// Sets origin as given o vector, and radius as given r float
Sphere::Sphere(Vector o, float r)
{
    origin = o;
    radius = r;
}

```

```

// method to set values of Sphere instance
void Sphere::SetValues(Vector v1, float r)
{
    origin = v1;
    radius = r;
}

// checks if two spheres are colliding and, if they are, returns true
// if not, returns false
bool Sphere::CollisionDetection(Sphere s2)
{
    if (origin.pToPDistance(s2.origin) <= (radius + s2.radius))
        return true;
    else
        return false;
}

```

Sphere.h

```

#pragma once
#include "Header.h"
#ifndef SPHERE_H
#define SPHERE_H

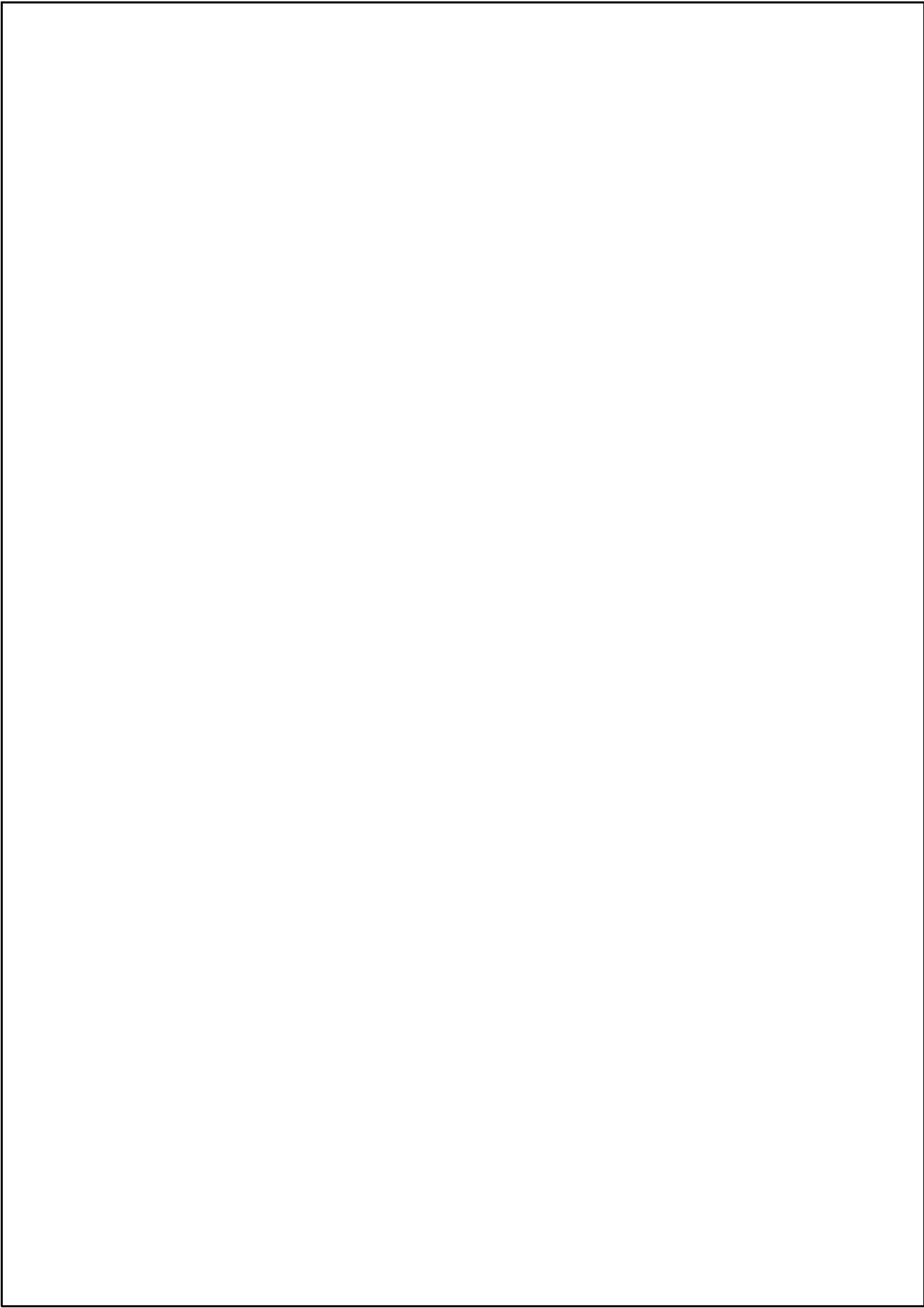
class Sphere
{
public:

    //construct
    Sphere(Vector o, float r);
    void SetValues(Vector v1, float r);
    bool CollisionDetection(Sphere s2);
    Sphere(void);
private:
    Vector origin;
    float radius;
};
#endif

```

Bibliography

- [1] Brilliant, "3D Coordinate Geometry - Distance," [Online]. Available: <https://brilliant.org/wiki/3d-coordinate-geometry-distance/>.
- [2] D. Q. Nykamp, "Distance from point to plane," [Online]. Available: https://mathinsight.org/distance_point_plane.



Kyle Lovell Report

GRADEMARK REPORT

FINAL GRADE

GENERAL COMMENTS

Instructor

56/100

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

PAGE 14

PAGE 15

PAGE 16

PAGE 17

KYLE LOVELL

55.50

RESEARCH-INFORMED LITERATURE



Refs used but few. Also exclusively on-line - avoid this. Use a good text at the very least - eg Bishop and van Verth. Key topics covered. Some more detail/clarity here would help also.

5

KNOWLEDGE AND UNDERSTANDING



Essential ideas covered. Some good discussion of integrators and some context provided.

13

ANALYSIS



Code design solves essential problems - analysis here could be deeper. Some discussion of integration methods included and some results provided but with little discussion/analysis.

11

PRACTICAL APPLICATION AND DEP.



Essential tasks coded. Could have included additional methods. Write results to file please. Commenting light in several places.

24

SKILLS FOR PROFESSIONAL PRAC.



Refs online- avoid this. Report structured..

2.5