

1800242 - GAM Report

by Benjamin Kiddie

FILE	12005_BENJAMIN_KIDDIE_1800242_- _GAM_REPORT_234537_234680375.DOCX (685.58K)		
TIME SUBMITTED	07-FEB-2020 11:55PM (UTC+0000)	WORD COUNT	4302
SUBMISSION ID	119782403	CHARACTER COUNT	22114

**Games Architecture and
Mechanics – Games Physics**

Name: Ben Kiddie

Student Number: 1800242

Submitted: 07/02/2020

Contents

Task One – Dragster Simulation	1
Integration Methods	2
Euler’s Method.....	4
Implementation.....	4
Testing	6
Midpoint Method	7
Task Two – Vector Class	8
Distance Between Two Points	9
Investigation.....	9
Implementation.....	10
Testing	11
Planes.....	15
Investigation.....	15
Implementation.....	16
Testing	18
Task Three – Sphere to Sphere Collision Detection	24
Investigation.....	25
Implementation.....	26
Testing	27
Bibliography	30
Appendix	31

Task One – Dragster Simulation

Task one asks the author to research different integration methods used in gaming and to compare the relative advantages and disadvantages of the given methods in a gaming context. Integration of the Euler Method is necessary, as is another method of the author's choice.

Integration Methods

According to Bishop & Van Verth (pages 611-614) [1], two known methods of integration used in game development can include:

- Euler's Method
- Midpoint Method

Euler's method requires that we have an initial velocity and position. A difference in time, usually found by finding the time taken between frames displayed, is used as the time step. The method then finds the total forces acting on an object, which calculates acceleration when divided by mass of the object. The next velocity and displacement values can then be calculated.

Advantages of Euler:

1. Low memory and CPU intensity.
2. Easy to implement.

Disadvantages of Euler:

1. Large forces will deviate the approximation from the actual value.
2. Larger time steps magnify approximation errors.
3. Drastic changes in force will return truncation errors.
4. Complexity $O(h)$.

In the context of games, Euler is able to approximate the values of an object well enough when given a small timestep, and is easy to implement, making it widely used, however it is constrained by its flaws. This method works well for modelling large quantities of particles in games, as the inaccuracies of particles using Euler calculations is hardly noticed in the grand scheme of things. The lower intensity allows for a greater number of particles to be handled at a given time.

The midpoint method uses Euler to step into the halfway point of the time interval, i.e. $h/2$. The values calculated here are used once the method steps backwards to the starting location, and the values found at the midpoint are used to move across the interval at h .

Advantages of Midpoint:

1. For $h < 1$, midpoint is more accurate than Euler.
2. Greater error tolerance.

Disadvantages of Midpoint:

1. Complexity $O(h^2)$.
2. More memory and CPU intensive.

Due to the higher limitations, midpoint is better suited to smaller bodies that are in motion, such as bullets, which necessitate smoother traversal of the projectile for a sense of realism in shooting games.

Euler's Method

Implementation

A class for Euler integration is used to instantiate a new object that runs on this method.

```
class Euler
{
public:
    Euler();
    void DragsterSimulation();

private:
    float mDragForce,
          mTractiveForce,
          mCurrentDisplacement,
          mNextDisplacement,
          mTimeElapsed,
          mTimeStep,
          mRollingResistanceCoefficient,
          mMass,
          mGravity,
          mDragCoefficient,
          mAirDensity,
          mFrontalVehicleArea,
          mCurrentVelocity,
          mNextVelocity,
          mAcceleration;

    int mStep;
};
```

The function for dragster simulation will simulate the motion of a car being pushed by a spring until the car is at rest.

```

Euler::Euler() // Constructor for Euler
{
    mTimeElapsed = 0; // Represents
    mTimeStep = 0.06; // The increment
    mStep = 0; // Tracks the number
    mCurrentDisplacement = 0; // Current
    mNextDisplacement = 0; // The next
    mCurrentVelocity = 0; // Represents
    mNextVelocity = 0; // Represents
    mAcceleration = 0; // Represents
    mRollingResistanceCoefficient = 0
    mMass = 1; // Mass of the car
    mGravity = 9.81; // Acceleration
    mDragCoefficient = 0.34;
    mAirDensity = 1.2; // Density of
    mFrontalVehicleArea = 0.01; // Area
}

```

```

while (mCurrentVelocity > 0 || mTimeElapsed == 0) // Enter loop either when the car is moving, or at initial position
{
    if (mCurrentDisplacement > 8) // When car's displacement is greater than the length of the spring
    {
        mTractiveForce = 0; // Spring no longer applies a tractive force to the car
    }
    else // Car's displacement is within the length of the spring
    {
        mTractiveForce = 1.5 * (16 - mCurrentDisplacement); // Spring applies tractive force
    }

    if (mCurrentVelocity < 0) //
    {
        mDragForce = 0;
    }
    else
    {
        mDragForce = mRollingResistanceCoefficient * mMass * mGravity + (0.5 * mDragCoefficient) * mAirDensity * mFrontalVehicleArea * (mCurrentVelocity * mCurrentVelocity);
    }

    mAcceleration = (mTractiveForce - mDragForce) / mMass;

    mCurrentDisplacement += ((mCurrentVelocity + mNextVelocity) / 2) * mTimeStep; // s = (u + v / 2) * t
    cout << mTimeElapsed << " " << mCurrentDisplacement << " " << mCurrentVelocity << " " << mAcceleration << endl;

    mNextVelocity = mCurrentVelocity + (mAcceleration * mTimeStep);
    mCurrentVelocity = mNextVelocity;
    mTimeElapsed += mTimeStep; // Increments the time by the time step
    mStep++; // Increments step counter by 1 each loop
}

```

The values used in the simulation of the car's motion are initialised in the constructor, while the dragster function runs a while loop which stops once the velocity of the car reaches 0, i.e. has entered a position of rest.

Testing

As shown below in figure 1, the results of the Euler implementation describe the motion of the car from the initial position to a resting position:

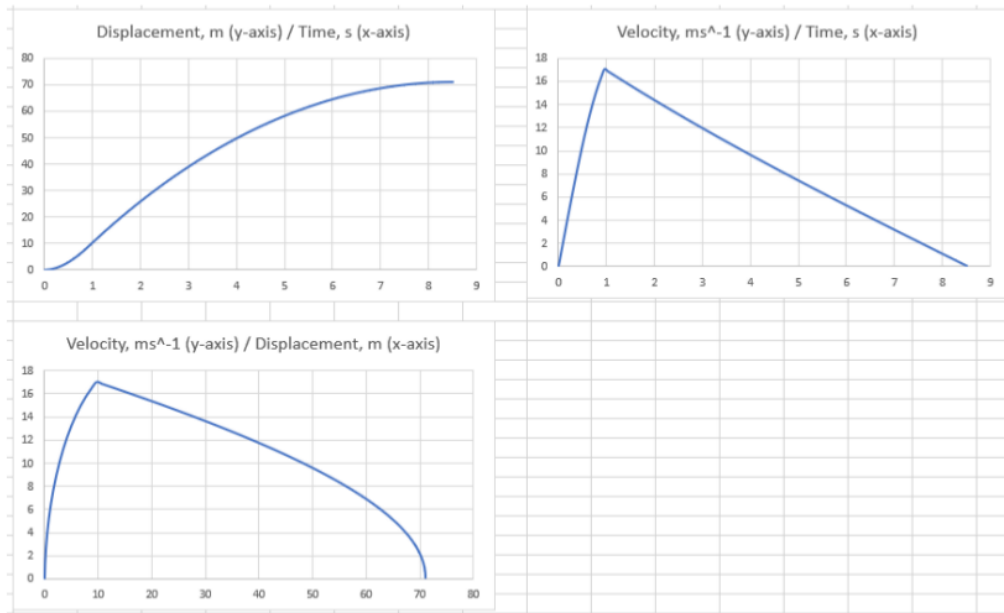


Figure 1 Graphs detailing the Euler Method simulation

The graphs detail the car starting with a great burst of speed due to the spring, followed by a steady decline in speed as friction consistently acts on the car.

Midpoint Method

Task Two – Vector Class

This task asks that a vector class is implemented, which will be worked on to include created and tested functions necessary for functionality. In addition, a plane class needs to be made, of which is responsible for two methods.

Distance Between Two Points

Investigation

The distance between two points finds the absolute value for the sum of differences between the components of two coordinates.

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

The subtraction of the first coordinate from the second provides the distance between them. Summation of these distances squared, when rooted, will provide an absolute value representing the distance between the two points.

Implementation

Distance between two points was implemented as a function of the vector class, as seen below:

```
float Vector::DistanceBetweenTwoPoints(Vector & v1) // Finds the distance between this vector and a vector which is passed in
{
    return sqrt(((v1.x - x) * (v1.x - x)) + (v1.y - y) * (v1.y - y) + (v1.z - z) * (v1.z - z)); // Returns the calculated distance
}
```

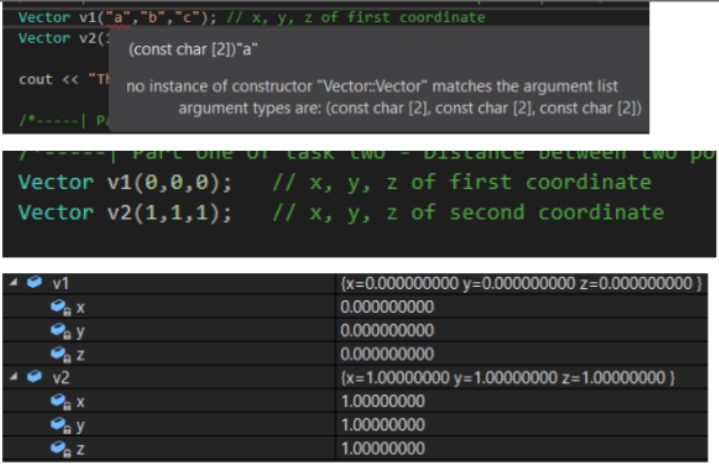
Figure 2 Implementation of distance between two points

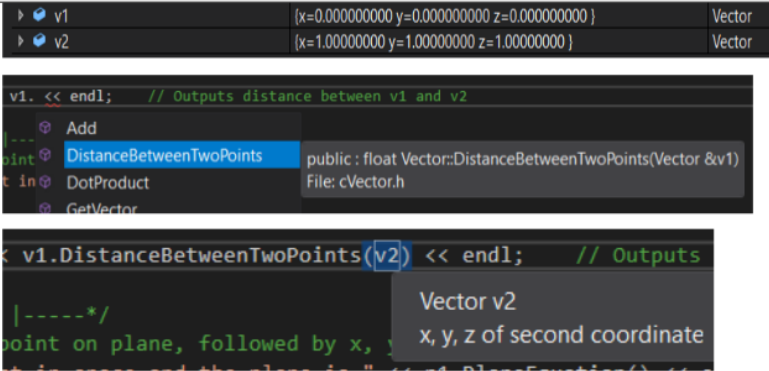
This function integrates the formula outlined in the investigation section. The first point (the current vector object) represents (x, y, z), while the second vector object is passed in as an argument, representing (v1.x, v1.y, v1.z). The distance between the x, y and z components of both point one and point two is found, then multiplied by the same value, effectively squaring it. The square distances are then summed together, followed by being passed into the function sqrt(), which finds the square root of a given number. The output is always positive, as the squaring process yields absolute values.

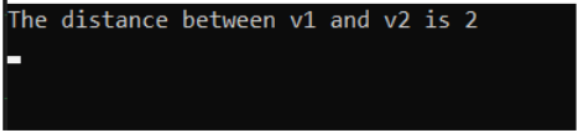
Testing

From here on, across the rest of this report, testing will follow this template:

Test Case Number (e.g. Test 1, Test 5)	Test Description (e.g. testing numerical and non-numerical inputs)
Prerequisites (i.e. what prior conditions need to be met to trigger the test case)	Expected Results (i.e. what the author expects each test to yield)
Actual Results (i.e. what results were yielded from the test)	Status (e.g. successful, failure, may include notes)

Test 1	The vector object can be constructed using numerical values
N/A	The vector object accepts three numerical values as arguments and constructs the object.
 <pre> Vector v1("a","b","c"); // x, y, z of first coordinate Vector v2((const char [2])"a" cout << "T no instance of constructor "Vector::Vector" matches the argument list argument types are: (const char [2], const char [2], const char [2]) /*----- P // ===== Part One of task two = Distance between two po Vector v1(0,0,0); // x, y, z of first coordinate Vector v2(1,1,1); // x, y, z of second coordinate v1 x 0.000000000 y 0.000000000 z 0.000000000 v2 x 1.000000000 y 1.000000000 z 1.000000000 </pre> A vector can be instantiated given three values that act as the x,y,z properties.	PASS

Test 2	The function for distance between two points is accessible from a vector object
At least two vectors are established	A vector object can call the distance between two points function and successfully pass in a different vector as an argument
 The screenshot shows a C++ IDE with two Vector objects, v1 and v2, and a code snippet. v1 is initialized with (0,0,0) and v2 with (1,1,1). The code snippet shows v1 calling the DistanceBetweenTwoPoints function on v2. A tooltip for v2 shows its coordinates (1, 1, 1).	PASS
The function for distance between points can be called from a vector object successfully. Other vector objects can be passed in.	

Test 3	Running the distance between points function, with a vector passed in, returns the absolute distance.
Two or more vectors instantiated	The function will output absolute value of distance between points. The test vectors for this case will be: 1. V1 (2,0,0), V2 (0,0,0) 2. V1 (0,-5,0), V2 (0,0,0) 3. V1 (0,0,0), V2 (0,0,0) 4. V1 (0,0,0), V2 (-10,-10,-10) 5. V1 (0,0,0), V2 (0,100,-100)
As seen in the outputs, the function successfully returns the absolute distance between two points. 1. $\sqrt{(2-0)^2 + (0-0)^2 + (0-0)^2} = 2$ 2. $\sqrt{(0-0)^2 + (-5-0)^2 + (0-0)^2} = 5$ 3. $\sqrt{(0-0)^2 + (0-0)^2 + (0-0)^2} = 0$ 4. $\sqrt{(0-(-10))^2 + (0-(-10))^2 + (0-(-10))^2} = 17.3205$ 5. $\sqrt{(0-0)^2 + (0-100)^2 + (0-(-100))^2} = 141.4214$ 	PASS

<pre>hThe distance between v1 and v2 is 5 E -</pre>	
<pre>hThe distance between v1 and v2 is 0 - -</pre>	
<pre>hThe distance between v1 and v2 is 17.3205 - -</pre>	
<pre>hThe distance between v1 and v2 is 141.421 - h</pre>	

Planes

Investigation

A plane is an infinitely thin surface in 3d space. The task calls for two methods for which this class is responsible. The first is the plane equation, which finds the shortest point between a point in space and a point on the plane, given the normalized vector of a normal to the plane, according to C. Ericson (pages 126 – 127) [1]:

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

Where t is the shortest distance between a point in space and plane, Q is a point in space, X is a point on plane, n is the normal vector, $d = n \cdot X$, the equation simplifies to $t = n \cdot Q - n \cdot X$ for a normalised normal.

The second method is detecting whether a given line connecting two coordinates crosses through the plane. The points A , B , along with a point on the plane P , and the normal n . The formula is as follows:

$$t = \frac{(n \cdot X - n \cdot A)}{n \cdot (B - A)}$$

Where $0 \leq t \leq 1$ means that the line intersects with the plane, while outside these bounds means that the line does not intersect with the plane.

Implementation

A class to hold the properties and functions of the plane was created. Both a point on the plane and coordinate of a normal are needed for the functions, however the plane equation asks that a point in space be provided.

```
class Plane : Vector
{
public:
    Plane(float popX, float popY, float popZ,
          float normX, float normY, float normZ); // Plane constructor

    float PlaneEquation(Vector & pointInSpace); // Finds the shortest distance between a point in space and the plane
    bool LinePlaneIntersect(Vector & v1, Vector & v2);

private:
    Vector mPointOnPlane; // Holds the x, y, z properties for a coordinate that lies on this plane
    Vector mNormal; // Holds the x, y, z properties for the normal to the plane
};
```

Upon instantiating a plane object, values for a point on the plane and coordinates of a normal are given by the user. The functions for the plane class, PlaneEquation and LinePlaneIntersect, can be called from any plane, where the PlaneEquation takes in a random vector as a point in space.

The function for the plane equation requires that, to find the shortest distance between the plane and a point, the user provides said point as an argument.

```
float Plane::PlaneEquation(Vector & pointInSpace) // Calculates the shortest distance
{
    Vector normalisedNorm = mNormal.NormaliseVector(mNormal.LengthOfLine());
    float normalisedNormDotPOP = normalisedNorm.DotProduct(mPointOnPlane); //
    float normalisedNormDotPIS = normalisedNorm.DotProduct(pointInSpace); //
    return (normalisedNormDotPIS - normalisedNormDotPOP); // Returns the value
}
```

As the approach described in the investigation details that a normalised normal be used, the first calculation to be made is normalisation of the normal, $\hat{N}$. The plane equation is then calculated as $\hat{N}.pointInSpace - \hat{N}.pointOnPlane$.

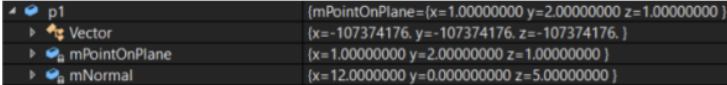
The LinePlaneIntersect function determines whether a line connecting two given vectors intersect with the plane, where the vectors making the line are passed in.

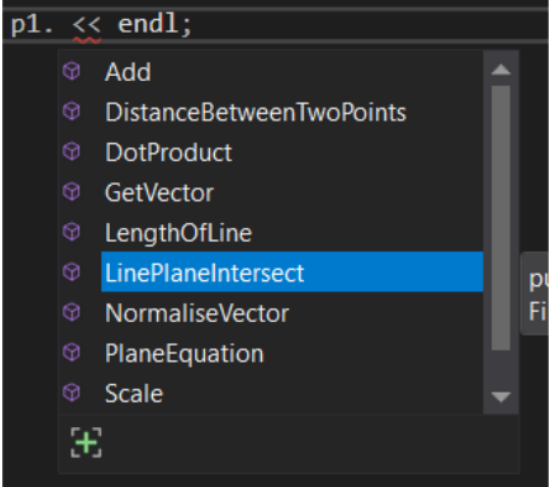
```
bool Plane::LinePlaneIntersect(Vector & v1, Vector & v2) // v1 represents 'A', v2 represents 'B',
{
    Vector ab = v2.SubtractVector(v1); // Vector AB = B - A
    float t = (mNormal.DotProduct(mPointOnPlane) - mNormal.DotProduct(v1)) / mNormal.DotProduct(ab);

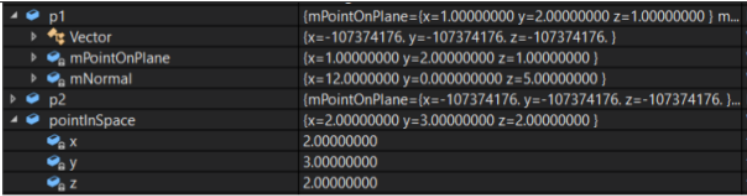
    if (t >= 0 && t <= 1)
    {
        return true;
    }
    else
    {
        return false;
    }
}
```

Using the approach given in the investigation, the vector AB must first be found. This is done by subtracting the components of vector A from B, i.e. $(B-A)$. The value that determines if the line intersects, t , is found by subtracting $n \cdot A$ from $n \cdot P$ and dividing by $n \cdot AB$.

Testing

Test 1	A plane object can be instantiated by passing in x,y,z values for the point on the plane and a normal vector
N/A	Passing in six values (two x,y,z values) that create the point on plane and normal vectors to the plane constructor instantiates a plane object
<pre>/*----- Part two Plane p1(1,2,1, 12,0,5); Vector mNormal; Vector mPointOnPlane;</pre>  Passing in six numerical values creates a plane object of components mPointOnPlane and mNormal.	PASS

Test 2	The functions of a plane object are accessible
A plane object must be instantiated	Creating a plane object allows the user to access the functions PlaneEquation and LinePlaneIntersect
 The PlaneEquation and LinePlaneIntersect functions are accessible through an instantiated plane object	PASS

Test 3	The plane equation function finds the shortest distance between a point in space and the plane
A plane object and point in space vector are instantiated	Passing the vector for a point in space into the plane equation function as an argument calculates and returns the shortest distance between the point and plane
 <pre> p1 (mPointOnPlane=[x=1.00000000 y=2.00000000 z=1.00000000] m... > Vector {x=-107374176, y=-107374176, z=-107374176. } > mPointOnPlane {x=1.00000000 y=2.00000000 z=1.00000000 } > mNormal {x=12.00000000 y=0.000000000 z=5.000000000 } > p2 (mPointOnPlane=[x=-107374176, y=-107374176, z=-107374176.]... > pointInSpace {x=2.00000000 y=3.00000000 z=2.00000000 } > x 2.00000000 > y 3.00000000 > z 2.00000000 </pre> • $\widehat{mNormal} = \frac{12}{13}, 0, \frac{5}{13}$ • $\widehat{mNormal} \cdot mPointInSpace = \frac{17}{13}$ • $\widehat{mNormal} \cdot mPointOnPlane = \frac{34}{13}$	PASS

- $\hat{n} \cdot \text{PointInSpace} - \hat{n} \cdot \text{PointOnPlane} = \frac{17}{13}$

The shortest distance between the point in space and the plane is 1.30769

p1	{mPointOnPlane={x=5.00000000 y=3.00000000 z=4.00000000 }
Vector	{x=-107374176. y=-107374176. z=-107374176. }
mPointOnPlane	{x=5.00000000 y=3.00000000 z=4.00000000 }
mNormal	{x=0.000000000 y=-1.00000000 z=-10.0000000 }
p2	{mPointOnPlane={x=-107374176. y=-107374176. z=-107374176. }
pointInSpace	{x=0.000000000 y=5.00000000 z=10.0000000 }
x	0.000000000
y	5.00000000
z	10.0000000

- $m\widehat{Normal} = 0, -\frac{1}{\sqrt{101}}, -\frac{10}{\sqrt{101}}$
- $m\widehat{Normal} \cdot m\text{PointInSpace} = -4.2787$
- $m\widehat{Normal} \cdot m\text{PointOnPlane} = -10.4479$
- $\hat{n} \cdot \text{PointInSpace} - \hat{n} \cdot \text{PointOnPlane} = -6.1692$

The shortest distance between the point in space and the plane is -6.16923

Test 4	The LinePlaneIntersect function can be used to find whether or not a line intersects with a plane
A plane object and two vector objects must be instantiated	A plane object can run the function LinePlaneIntersect given two vector coordinates as arguments. The function returns 'true' for an intersection of the line AB, 'false' for no intersection. The test cases are (note: in the program, v1 = vector A, v2 = vector B): • A(0,0,0), B(0,10,0), P(0,0,0), n(0,1,0) • A(1,1,1), B(2,2,2), P(0,0,0), N(0,1,0)

this	0x008ffd50 {mPointOnPlane={x=0.000000000 y=0.000000000 z=0.000000000}}
Vector	{x=-107374176, y=-107374176, z=-107374176 }
mPointOnPlane	{x=0.000000000 y=0.000000000 z=0.000000000 }
mNormal	{x=0.000000000 y=1.000000000 z=0.000000000 }
ab	{x=0.000000000 y=10.00000000 z=0.000000000 }
t	0.000000000
v1	{x=0.000000000 y=0.000000000 z=0.000000000 }
v2	{x=0.000000000 y=10.00000000 z=0.000000000 }

- $AB = (0, 10, 0)$
- $mNormal \cdot mPointOnPlane = 0$
- $-mNormal \cdot A = 0$
- $mNormal \cdot AB = 1$
- $t = \frac{0-0}{1} = 0$

The line intersects with the plane

this	0x0133f980 {mPointOnPlane={x=0.000000000 y=0.000000000 z=0.000000000}}
Vector	{x=-107374176, y=-107374176, z=-107374176 }
mPointOnPlane	{x=0.000000000 y=0.000000000 z=0.000000000 }
mNormal	{x=0.000000000 y=1.000000000 z=0.000000000 }
ab	{x=1.000000000 y=1.000000000 z=1.000000000 }
t	-1.000000000
v1	{x=1.000000000 y=1.000000000 z=1.000000000 }
v2	{x=2.000000000 y=2.000000000 z=2.000000000 }

- $AB = (1, 1, 1)$
- $mNormal \cdot mPointOnPlane = 0$
- $-mNormal \cdot A = -1$
- $mNormal \cdot AB = 1$
- $t = \frac{0-1}{1} = -1$

The line does not intersect with the plane

Task Three – Sphere to Sphere Collision Detection

This task asks that the user implement sphere to sphere collision detection by implementing a class. The vector class may be used here.

Investigation

To detect the collision between spheres, the centrepoint and radius of each sphere is needed. In this collision, the spheres either touch, or they don't. If they don't touch, there is a gap between the spheres, else they make contact. This is represented below:

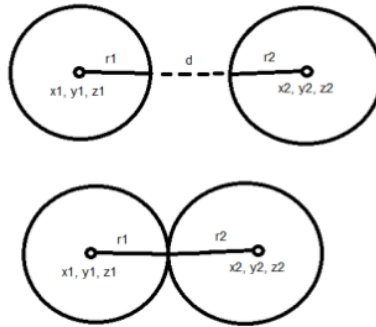


Figure 3 Diagram showing both cases of sphere to sphere collision

As can be seen here, to know if there is a gap between the spheres, we must first know the distance between the two centre points. This can be found using the formula for distance between points described in task two:

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

This provides the total distance between the spheres. If the distance between spheres is less than the length of the spheres' radii summed, then the spheres have collided. If the distance is greater than radii summed, they have not collided. This is justified by the approach taken by Bishop & Van Verth (pages 560 – 561) [2], which also looks to compare the distance between spheres with sum of radii to detect collision between spheres.

Implementation

A class 'cSphere' was created to hold the properties of a sphere. The centre point being a vector of x,y,z properties, and a radius are necessary.

```
class Sphere : Vector
{
public:
    Sphere(float centreX, float centreY, float centreZ, float radius); // Sphere constructor
    bool CollisionCheck(Sphere & s1); // Checks for collision between spheres

private:
    Vector mCentrePoint; // The centrepoint of the circle, being a vector of x, y, z properties
    float mRadius; // The radius of the circle from centrepoint to circumference
};
```

The constructor takes the x,y,z and radius values and creates the object as follows:

```
Sphere::Sphere(float centreX, float centreY, float centreZ, float radius)
{
    mCentrePoint = Vector(centreX, centreY, centreZ); // Sets the centrepoint
    mRadius = fabs(radius); // Sets the radius of the circle to the absolute value
```

The x,y,z values create the centre point vector, while the radius is set to the absolute value of what is passed in, as the radius cannot be of negative length.

The function that detects collision between spheres, 'CollisionCheck', will find whether it collides with another sphere that has the object passed in as an argument.

```
bool Sphere::CollisionCheck(Sphere & s1) // Checks the values of this sphere and a sphere
{
    float distanceBetweenSpheres = mCentrePoint.DistanceBetweenTwoPoints(s1.mCentrePoint);
    float totalRadii = mRadius + s1.mRadius; // Finds the total radius length of this sphere and s1
    if (distanceBetweenSpheres > totalRadii) // If the distance between the spheres is greater than the sum of radii, there is no collision
    {
        return false; // False = no collision detected
    }
    else // If the distance between spheres is equal to or less than the sum of radii, there is a collision
    {
        return true; // True = collision detected
    }
}
```

Initially, the distance between sphere radii is calculated. A float holds the value for the distance between the centre points of sphere 1 and 2. Once this value is found, the sum of radii is calculated. If the distance between centre points is greater than that of the sum of radii, we know there is no collision, hence returning false. The else statement describes the scenario where the sum of radii is greater than or equal to the distance between centre points of the spheres, where they do in fact collide.

Testing

Test 1	A sphere can be established
N/A	A sphere object can be established given values for the centre point of x,y,z and radius length as arguments
<pre> Sphere s1(-10, 0, 0, 10); Sphere s2(10, 0, 0, 10); Sphere s1(true, false, false, true); Sphere s2(10, 0, 0, 10); if (s1.CollisionCheck(s2) == true) Sphere s1("a", "b", "c", "d"); Sphere s2(10, 0, 0, 10); if (s1.CollisionCheck(s2) == true) { cout << "Collision detected" << endl; } Sphere s1(); Sphere s2(10, 0, 0, 10); if (s1.CollisionCheck(s2) == true) { cout << "Collision detected" << endl; } else { cout << "No collision" << endl; } </pre> As the above results show, a sphere object can only be made and utilised if the numerical values for x,y,z and radius are passed in. Booleans are treated as numerical values however, where true equals 1, false equals 0.	PASS

Test 2	The functions of a sphere are accessible
A sphere object is instantiated	The functions belonging to a sphere are accessible through an instantiated sphere
<pre> Sphere s1(-10,0,0,10); Sphere s2(10, 0, 0, 10); if (s1.CollisionCheck(s2) == true) { // Collision detected } // Collision detected </pre> <pre> bool Sphere::CollisionCheck(Sphere & s1) // Checks the values of this sphere and a sphere { float distanceBetweenSpheres = mCentrePoint.DistanceBetweenTwoPoints(s1.mCentrePoint); float totalRadii = mRadius + s1.mRadius; // Finds the total radius length of this sphere if (distanceBetweenSpheres > totalRadii) // If the distance between the spheres is greater than the sum of radii, then { return false; // False = no collision detected } else // If the distance between spheres is equal to or less than the sum of radii, then { return true; // True = collision detected } } </pre> The sphere can be instantiated (with values passed in to the constructor), and the functions are accessible via this object.	PASS

Test 3	The function for detecting sphere to sphere collision can be used to detect if two spheres are in contact or not
At least two sphere objects have been instantiated	The function for collision detection can be called from any sphere. This function detects if the sphere calling the function is colliding with another sphere that is passed in as an argument. The test values will be: 1. S1(-10,0,0,10),S2(10,0,0,10) 2. S1(-10,10,-10,5),S2(0,0,0,5) 3. S1(0,0,0,0),S2(0,0,0,false) 4. S1(0,0,0,10), S2(0,5,10,20)
As seen above, the tests return the expected result for each test value. 1. Distance = 20, radii sum = 20 2. Distance = 17.3205, radii sum = 10 3. Distance = 0, radii sum = 0 4. Distance = 11.1803, radii sum = 30 Sphere 1 and Sphere 2 collide_ Sphere 1 and Sphere 2 do not collide_ Sphere 1 and Sphere 2 collide_ Sphere 1 and Sphere 2 collide_	PASS

Bibliography

- [1] C. Ericson, Real-time Collision Detection, Amsterdam ; London : Morgan Kaufmann, 2005.
- [2] J. M. Van Verth and L. M. Bishop, Essential Mathematics For Games and Interactive Applications, Morgan Kaufmann, 2015.

Appendix

Appendix 1 – Code submission

```
#include <iostream>
#include <sstream>
#include <string>
#include "cEuler.h"
#include "cVector.h"
#include "cPlane.h"
#include "cSphere.h"

using namespace std;

int main()
{
    /*-----| Part one of task one - Euler Integration method implementation |-----*/

    Euler car1; // Instantiates a car object for simulating the movement of a car
    car1.DragsterSimulation(); // Runs the simulation function

    /*-----| Part two of task one - Other Integration method implementation |-----*/

    /*-----| Part one of task two - Distance between two points |-----*/
    Vector v1(0,0,0); // x, y, z of first coordinate
    Vector v2(1,1,1); // x, y, z of second coordinate

    cout << "The distance between v1 and v2 is " << v1.DistanceBetweenTwoPoints(v2)
    << endl; // Outputs distance between v1 and v2

    /*-----| Part two of task two - Plane class |-----*/
    Plane p1(5,3,4,
            0,-1,-10); // x, y, z of point on plane, followed by x, y, z
of point in space, followed by x, y, z of normal
    Vector pointInSpace(0,5,10); // Vector to be passed in to the plane
equation
    cout << "The shortest distance between the point in space and the plane is " <<
p1.PlaneEquation(pointInSpace) << endl; // Outputs the shortest distance between the
plane and the vector passed in

    Vector a(1,1,1);
    Vector b(2,2,2);
    Plane p2(0,0,0,
            0,1,0);

    if (p2.LinePlaneIntersect(a, b))
    {
        cout << "The line intersects with the plane" << endl;
    }
    else
    {
        cout << "The line does not intersect with the plane" << endl;
    }

    /*-----| Part three - Collision detection of two spheres |-----*/
    Sphere s1(0,0,0,10);
    Sphere s2(0,5,10,20);
    if (s1.CollisionCheck(s2) == true)
    {
```

```
        cout << "Sphere 1 and Sphere 2 collide";
    }
    else
    {
        cout << "Sphere 1 and Sphere 2 do not collide";
    }

    // End of main
    cin.get();
    return 0;
}
```

```

#pragma once

class Euler
{
public:
    Euler();
    void DragsterSimulation();

private:
    float mDragForce,
          mTractiveForce,
          mCurrentDisplacement,
          mNextDisplacement,
          mTimeElapsed,
          mTimeStep,
          mRollingResistanceCoefficient,
          mMass,
          mGravity,
          mDragCoefficient,
          mAirDensity,
          mFrontalVehicleArea,
          mCurrentVelocity,
          mNextVelocity,
          mAcceleration;

    int mStep;
};

```

```

#include "cEuler.h"
#include <iostream>
using namespace std;

Euler::Euler()      // Constructor for Euler class, initialises variables used in the
simulation
{
    mTimeElapsed = 0;    // Represents 'T' in SUVAT
    mTimeStep = 0.06;    // The increment of time between each calculation
    mStep = 0;          // Tracks the number of steps
    mCurrentDisplacement = 0; // Current displacement from origin
    mNextDisplacement = 0; // The next displacement to be calculated, delta
'S' ( used in SUVAT) can be found using current displacement
    mCurrentVelocity = 0; // Represents 'U' in SUVAT
    mNextVelocity = 0;    // Represents 'V' in SUVAT
    mAcceleration = 0;    // Represents 'A' in SUVAT
    mRollingResistanceCoefficient = 0.21;
    mMass = 1;            // Mass of the car
    mGravity = 9.81;      // Acceleration due to gravity
    mDragCoefficient = 0.34;
    mAirDensity = 1.2;    // Density of air, acts against car movement
    mFrontalVehicleArea = 0.01; // Area of the car acted upon by the air
}

void Euler::DragsterSimulation() // Simulates the movement of a car from initial
position to resting position
{
    cout << "Time      Displacement      Velocity      Acceleration" <<
endl; // Details the layout of displayed values to the user

    while (mCurrentVelocity > 0 || mTimeElapsed == 0) // Enter loop either
when the car is moving, or at initial position
    {
        if (mCurrentDisplacement > 8) // When car's displacement is greater
than the length of the spring
        {
            mTractiveForce = 0; // Spring no longer applies a tractive force
to the car
        }
        else // Car's displacement is within the length of the spring
        {
            mTractiveForce = 1.5 * (16 - mCurrentDisplacement); // Spring
applies tractive force
        }

        if (mCurrentVelocity < 0) //
        {
            mDragForce = 0;
        }
        else
        {
            mDragForce = mRollingResistanceCoefficient * mMass * mGravity +
(0.5 * mDragCoefficient) * mAirDensity * mFrontalVehicleArea * (mCurrentVelocity *
mCurrentVelocity);
        }

        mAcceleration = (mTractiveForce - mDragForce) / mMass;

        mCurrentDisplacement += ((mCurrentVelocity + mNextVelocity) / 2) *
mTimeStep; // s = (u + v / 2) * t
    }
}

```

```

        cout << mTimeElapsed << " " << mCurrentDisplacement << "
" << mCurrentVelocity << " " << mAcceleration << endl;

        mNextVelocity = mCurrentVelocity + (mAcceleration * mTimeStep);
        mCurrentVelocity = mNextVelocity;
        mTimeElapsed += mTimeStep; // Increments the time by the time step
        mStep++; // Increments step counter by 1 each loop
    }
}

```

```

#pragma once

#ifndef VECTOR_H
#define VECTOR_H

class Vector // Instantiates a coordinate of x, y, z properties
{
public:
    /*-----| Constructors |-----*/
    Vector(float x, float y, float z); // Create vector object with values passed in
    Vector(void); // Create empty vector object

    void Scale(float s); // Uniform scaling of vector
    void Add(Vector & v1); // Adds the vector to another vector that is passed
in
    void Add(Vector & v1, Vector & v2); // Adds two vectors wich are
passed in
    Vector GetVector(void); // Returns the vector
    float DistanceBetweenTwoPoints(Vector & v1); // Finds the distance between
this vector and another vector passed in
    float LengthOfLine(); // Finds the length of this vector
    float DotProduct(Vector & v1); // Finds the dot product of this vector and
another vector passed in
    Vector NormaliseVector(float lineLength); // Normalises the vector given the
length
    Vector SubtractVector(Vector & v1); // Finds the distance between each
component of this vector and another

private:
    float x; // x coordinate of the vector
    float y; // y coordinate of the vector
    float z; // z coordinate of the vector
};
#endif

```

```

#include "cVector.h"
#include <math.h>

/*-----| Constructors for a vector |-----*/
Vector::Vector(void) {}; // Instantiates an empty vector when no values are passed
in

Vector::Vector(float myx, float myy, float myz) // Instantiates a vector using passed
in x, y, z values
{
    x = myx; // Sets this vector's x to myx
    y = myy; // Sets this vector's y to myy
    z = myz; // Sets this vector's z to myz
}

void Vector::Scale(float s) // Uniform scaling of this vector when a scaling factor is
passed in
{
    x = s * x; // Multiplies x by the scaling factor
    y = s * y; // Multiplies y by the scaling factor
    z = s * z; // Multiplies z by the scaling factor
}

void Vector::Add(Vector & v1, Vector & v2) // Adds together two vectors which are
passed in
{
    v1.x = v2.x + v1.x;
    v1.y = v2.y + v1.y;
    v1.z = v2.z + v1.z;
}

Vector Vector::GetVector(void) // Gets the x, y, z properties of this vector
{
    return Vector(x, y, z); // Returns this vector
}

void Vector::Add(Vector & v1) // Adds together this vector and a vector which is
passed in
{
    x = x + v1.x;
    y = y + v1.y;
    z = z + v1.z;
}

float Vector::DistanceBetweenTwoPoints(Vector & v1) // Finds the distance between
this vector and a vector which is passed in
{
    return sqrt(((v1.x - x) * (v1.x - x)) + (v1.y - y) * (v1.y - y) + (v1.z - z) *
(v1.z - z)); // Returns the calculated distance
}

float Vector::LengthOfLine() // Calculates the length of this vector
{
    return sqrt((x * x) + (y * y) + (z * z)); // Returns the calculated length value
}

float Vector::DotProduct(Vector & v1) // Calculates the dot product of this vector
and a vector passed in
{
    return ((x * v1.x) + (y * v1.y) + (z * v1.z)); // Returns the calculated dot
product
}

```

```
Vector Vector::NormaliseVector(float lineLength) // Normalises this vector given the
length of the vector
{
    return Vector(x / lineLength, y / lineLength, z / lineLength); // Returns this
vector with normalised values
}

Vector Vector::SubtractVector(Vector & v1)      // Subtracts a vector from this object
{
    return Vector(x - v1.x, y - v1.y, z - v1.z); // Returns the value for this
vector subtracting the vector passed in
}
```

```

#pragma once
#include "cVector.h"

class Plane : Vector
{
public:
    Plane(float popX, float popY, float popZ,
          float normX, float normY, float normZ); // Plane constructor

    float PlaneEquation(Vector & pointInSpace); // Finds the shortest distance
    between a point in space and the plane
    bool LinePlaneIntersect(Vector & v1, Vector & v2);

private:
    Vector mPointOnPlane; // Holds the x, y, z properties for a coordinate
    that lies on this plane
    Vector mNormal; // Holds the x, y, z properties for the normal to the
    plane
};

```

```

#include "cPlane.h"

Plane::Plane(float popX, float popY, float popZ,
             float normX, float normY, float normZ) // Constructor for the plane,
// utilising values passed in for the point on plane, point in space and normal vectors
{
    mPointOnPlane = Vector(popX, popY, popZ); // Defines mPointOnPlane with values
// passed in for x, y, z
    mNormal = Vector(normX, normY, normZ); // Defines mNormal with values passed
// in for x, y, z
}

float Plane::PlaneEquation(Vector & pointInSpace) // Calculates the shortest
// distance from a point in space to the plane
{
    Vector normalisedNorm = mNormal.NormaliseVector(mNormal.LengthOfLine()); //
// Creates a temporary vector that holds mNormal normalised
    float normalisedNormDotPOP = normalisedNorm.DotProduct(mPointOnPlane); //
// Calculates the dot product of normalised normal and point on a plane
    float normalisedNormDotPIS = normalisedNorm.DotProduct(pointInSpace); //
// Calculates the dot product of normalised normal and point in space

    return (normalisedNormDotPIS - normalisedNormDotPOP); // Returns the value for
// shortest distance from point in space to plane
}

bool Plane::LinePlaneIntersect(Vector & v1, Vector & v2) // v1 represents vector
// 'A', v2 represents vector 'B', this function returns whether a line intersects a plane
{
    Vector ab = v2.SubtractVector(v1); // Vector AB = B - A
    float t = (mNormal.DotProduct(mPointOnPlane) - mNormal.DotProduct(v1)) /
// mNormal.DotProduct(ab); // The value of t determines whether

    if (t >= 0 && t <= 1)
    {
        return true;
    }
    else
    {
        return false;
    }
}

```

```

#pragma once
#include "cVector.h"

class Sphere : Vector
{
public:
    Sphere(float centreX, float centreY, float centreZ, float radius); // Sphere
    constructor
    bool CollisionCheck(Sphere & s1); // Checks for collision between spheres

private:
    Vector mCentrePoint; // The centrepoint of the circle, being a vector of x, y,
    z properties
    float mRadius; // The radius of the circle from centrepoint to
    circumference
};

```

```

#include "cSphere.h"
#include <math.h>

Sphere::Sphere(float centreX, float centreY, float centreZ, float radius) //
Constructor for sphere properties, utilising passed in values of centrepoint x, y, z
and the length of the radius
{
    mCentrePoint = Vector(centreX, centreY, centreZ);    // Sets the centrepoint
vector to the x, y, z values passed in
    mRadius = fabs(radius);    // Sets the radius of the circle to the absolute
value of radius passed in, as the radius of a circle cannot be negative
}

bool Sphere::CollisionCheck(Sphere & s1) // Checks the values of this sphere and a
sphere passed in to see whether there is collision. A 'true' returned means collision,
'false' means no collision.
{
    float distanceBetweenSpheres =
mCentrePoint.DistanceBetweenTwoPoints(s1.mCentrePoint); // Finds the distance between
the centrepoints of this sphere and the sphere passed in
    float totalRadii = mRadius + s1.mRadius; // Finds the total radius length of
this sphere and the sphere passed in

    if (distanceBetweenSpheres > totalRadii) // If the distance between the spheres
is greater than the sum of radii, i.e. there is a gap between the spheres
    {
        return false; // False = no collision detected
    }
    else // If the distance between spheres is equal to or less than the sum of
radii, there is overlap between the spheres
    {
        return true; // True = collision detected
    }
}

```

1800242 - GAM Report

GRADEMARK REPORT

FINAL GRADE

GENERAL COMMENTS

70/100

Instructor

PAGE 1

PAGE 2

PAGE 3

PAGE 4



Comment 1

It's lower complexity than most.

PAGE 5



Comment 2

reference this and explain

PAGE 6

PAGE 7

PAGE 8

PAGE 9

PAGE 10

PAGE 11

PAGE 12

PAGE 13

PAGE 14

PAGE 15

PAGE 16

PAGE 17

PAGE 18

PAGE 19

PAGE 20

PAGE 21

PAGE 22

PAGE 23

PAGE 24

PAGE 25

PAGE 26

PAGE 27

PAGE 28

PAGE 29

PAGE 30

PAGE 31

PAGE 32

PAGE 33

PAGE 34

PAGE 35

PAGE 36

PAGE 37

PAGE 38

PAGE 39

GRADING FORM: SMU GENERIC_GJ

**BENJAMIN
KIDDIE**

69.50

RESEARCH-INFORMED LITERATURE



Key topics identified. 2 integration methods discussed.. Good refs used but include more!

7

KNOWLEDGE AND UNDERSTANDING



Some good work here. Report covers topics appropriately but only one integration method applied sadly.

14

ANALYSIS



Test plan provided and detailed tests included.

14.25

Good. Code uses OO SE practices. Discussion of methods pros and cons included - good. but double check some claims and back them up with refs to support them.

PRACTICAL APPLICATION AND DEP.



Only one method used to integrate v and x. Other tasks implemented as required. Good SE practices in evidence.

31

SKILLS FOR PROFESSIONAL PRAC.



Report structured. Refs used appropriately. Good quality refs - could be more.

3.25