

GAM Assessment 1a Writeup

by Rhys David Whatty

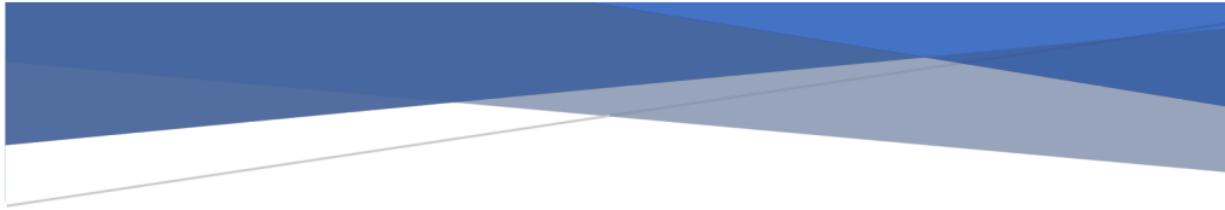
Submission date: 29-Nov-2020 12:34PM (UTC+0000)

Submission ID: 137707119

File name: 17862_Rhys_David_Whatty_GAM_Assessment_1a_Writeup_234537_1015896558.docx (518.05K)

Word count: 3123

Character count: 15299



PART 1A - GAMES PHYSICS

ACCB5017 - Games Architecture and Mechanics

Rhys Whatty
1900045

Contents

Introduction	2
Task 1 – Methods of Numerical Integration	3
Euler Method	3
Calculating through Euler	3
Advantages and disadvantages of Euler	3
Other Methods.....	4
Verlet Method.....	4
Explicit methods.....	4
Other types of Euler	6
Comparing Methods	7
Euler’s Method.....	7
Verlet Method.....	10
Differences in the graphs	11
Task 2 - Coding the games Engine.....	12
Brief.....	12
Distance and distance squared between two points.....	12
Code	12
Testing.....	12
Vector Plane.....	14
Code	14
Testing.....	15
Sphere-to-sphere collision detection.....	16
Code	16
Testing.....	17
Summary.....	18
Bibliography	19

Introduction

For a character to react realistically with the game world, it is necessary to apply physics which requires the game engine to calculate the equations of various forms of motions. For a games developer, this requires understanding of numerical integration methods.

The game engine uses equations of motion to work out the velocity of movement of a character. This is done in a continuous loop, and Harold Serrano describes it in these steps:

1. *Identify all forces and moments acting on the object.*
2. *Take the vector sum of all forces and moments.*
3. *Solve the equation of motion for linear and angular acceleration.*
4. *Integrate the acceleration with respect to time to find the linear and angular velocity.*
5. *Integrate the velocity with respect to time to find the linear and angular displacement.* [1]

The problem we face is that the computer only gets an approximate value using only numerical integration methods. The main models of numerical integration fall into 3 categories; explicit, implicit or semi-implicit. During this assignment, we will be investigating these categories, and then applying two types of numerical integration into practice and comparing them to see how different the results are.

Proceeding this will be the creation of a vector class for a games engine, in which several mathematical and mechanical operations will be explored and then coded, before being tested to confirm their functionality.

Task 1 – Methods of Numerical Integration

Euler Method

Calculating through Euler

The basic numerical integration method used in gaming is Euler's method, which is an example of an explicit model. Euler works by assuming a starting position and velocity and proceeding to calculate updated values after a set amount of time, or the time step value. These values are then used to calculate the next set of values after the same set amount of time, entering a loop. The Essential Mathematics for Games and Interactive Applications by Bishop and Van Verth describes the method that this can be calculated with the following formulas:

$$y(t+h) = y(t) + h y'(t) + (h^2/2) * y'' + \dots + (h^n/n!) * y^n(t) + \dots$$

We can rewrite this to compute the value for time step $l+1$, where the time from t_i to t_i+1 is h_i :

$$y_{i+1} = y_i + h_i y'_i + (h_i^2/2) * y''_i + \dots + (h_i^n/n!) * y^n_i + \dots$$

This assumes that we know all the values for the entire infinite series at step time i , which we don't... However, if h_1 is small enough and all values of y''_i are bounded, we can use an approximation instead:

$$\begin{aligned} y_{i+1} &\approx y_i + h_i y'_i \\ &\approx y_i + h_i f(t_i, y_i) \end{aligned} \quad [2]$$

In the final formula, we have a function $f(t_i, y_i)$ with an initial time and value, and these can be used to calculate tangent to the function's curve, which we can use to get the next approximate point in the vector field by travelling along the tangent for the distance of step h_i , however it is important to compute the new velocity after the new position with Euler, as you do not want the velocity to be prematurely overwritten. [2]

Advantages and disadvantages of Euler

While this is one of the simplest methods of numerical integration, accuracy can be a problem. Each calculation will create an error value, which is the difference between the calculation and the actual solution. As Dickinson says, you can make the time step value, the closer the estimate is to the actual solution, however this can become impractical as if the time step is too small, it creates thousands of calculations to be processed which slows down the speed of the calculation and will still leave a small error which will increase subsequent calculation. "Euler is mainly used when the impacts of the error value won't have a deciding impact on what happens next e.g. a sprinting character jumping then as the character lands it continues to sprint". [3]

Other Methods

Verlot Method

The final explicit method is known as Verlot Method, which is another popular method in developing games. It is a higher order integration method, making it more accurate, but still simple to implement. Rather than calculating the position from velocity like Euler, Verlot calculates the position from acceleration.

$$p_o(t + \Delta t) = 2p_o(t) - p_o(t - \Delta t) + \Delta t^2 p_o''(t) + \dots$$

One of the drawbacks of this method is that it works on constant step times, so it will not be as accurate if the step time varies. It also has an approximation value for velocity, so it would not be ideal for scenario where the velocity is constantly changing (such as a racing game). It is mainly used to simulate liquid flow, things falling or a bullet firing. [3] [4]

Explicit methods

Euler is the basis for the other main explicit methods of integration, which includes Midpoint and Runge-Kutta 4 (RK4).

Midpoint

Midpoint method is an order 2 method in which the tangent of the curve is calculated, which creates a new point halfway between the initial starting point and the new point at the end of the time step.

$$p_o(t + \Delta t) = p_o(t) + \Delta t * v\left(t + \frac{\Delta t}{2}, p_o + \frac{\Delta t}{2} v(t, p_o)\right)$$

This method is still not fully accurate, however the error range will be better than with Euler if the time step value is less than 1, as the error order is dependent on the square of the time step (Δt^2). [3] [4]

RK4

RK4 method is an order 4 method in which a tangent is generated and then used to generate a new tangent. This process is repeated until there are 4 tangents generated which are then weighed together to provide an average for the change in speed.

$$\begin{aligned}v_1 &= \Delta t * a(t, v(t)) \\v_2 &= \Delta t * a\left(t + \frac{\Delta t}{2}, v(t) + \frac{1}{2}v_1\right) \\v_3 &= \Delta t * a\left(t + \frac{\Delta t}{2}, v(t) + \frac{1}{2}v_2\right) \\v_4 &= \Delta t * a(t + \Delta t, v(t) + v_3) \\v(t + \Delta t) &= v(t) + \frac{v_1 + 2v_2 + 2v_3 + v_4}{6}\end{aligned}$$

RK4 is not ideal for calculating all types of physics in games, but it is good for simulation or when accuracy is important to what would happen next (such as a sniper shot). It is also used for planetary orbit and flight simulations. [3] [4]

Rodriguez did a comparison of these 3 methods in 2 different situations, in a velocity field where a particle has no momentum, and projectile motion, where it has momentum and force is applied to it.

With velocity fields, RK4 proved to be the most accurate, but as it requires 4 times more evaluations than Euler he felt that the small improvement in accuracy was not outweighed by the complexity and additional memory needed to run RK4. He felt that if precision was important, then Midpoint gives slightly more precision than Euler with 1 extra calculation.

For projectile Motion, surprisingly he found that Euler was slightly better than Midpoint and RK4, which performed almost identically. His conclusion was that in the context of video games, although you would expect it to be more accurate, RK4 is not always the best choice. [5]

Other types of Euler

There are 2 other versions of Euler that can be used. Implicit (or backwards Euler) calculates the derivative first and uses that to calculate the next estimation. It still has an error, but this is minimal, even with large time steps. The problem with it is there are further calculations to get the updated acceleration and velocity, which will make it slower. It is not a good method for rigid body movements, but Dickinson says that it is used within games for things such as springs, calculating friction, drag, and changes in acceleration. [3]

The final method I looked at is semi-implicit Euler. This method calculates both the velocity and position at a specific time. It uses explicit Euler techniques to calculate the velocity, but the position is done using the implicit method. This method is considered by Dickinson to be more accurate by Euler, but faster and easier to integrate than implicit Euler. It is mainly used to calculate rigid body movements. [3]

Comparing Methods

To see how different methods vary, two of them are going to be used on the same scenario to see how the results differ. The two methods being compared are Euler and Verlet. The scenario calls for a model of a dragster car's velocity to be measured over a period of time until it comes to a stop. For each, the model was coded in C++ that generated the results and outputted the results, and using those results, a graph was then created in Excel to demonstrate the change in velocity.

Euler's Method

Below is the code used to calculate the values in Euler's Method. It works around a while loop that repeats until the car's velocity drops under 0m/s, and on each rotation, the code will output the number of iterations, the total elapsed time, the new velocity, new acceleration and total distance covered.

```
while (loop == 0)
{
    if (currentVelocity > 0)
    {
        dragForce = (0.21 * 1 * 9.81) + (0.5 * 0.34 * 1.2 * 0.01 * (currentVelocity * currentVelocity));

        if (currentDist > 8)
        {
            tractForce = 0;
        }
        else
        {
            tractForce = 1.5 * (16 - currentDist);
        }

        accel = (tractForce - dragForce) / 1;
        time = time + 0.06;

        //Euler Method
        newDist = currentDist + (currentVelocity * time);
        newVelocity = currentVelocity + (accel * time);

        cout << "Time step: " << count << " (Time Elapsed: " << time << ")" << endl;
        cout << "Distance = " << newDist << endl;
        cout << "Velocity = " << newVelocity << endl;
        cout << "Acceleration = " << accel << endl << endl;

        currentDist = newDist;
        currentVelocity = newVelocity;
        count = count + 1;
    }
    else
    {
        //When the car stops, program ends
        dragForce = 0;
        cout << "Car halted" << endl;
        loop = 1;
        return 0;
    }
}
```

```
Time step: 1 (Time Elapsed: 0.06)
Distance = 0.6
Velocity = 11.3042
Acceleration = 21.7359

Time step: 2 (Time Elapsed: 0.12)
Distance = 1.9565
Velocity = 13.7977
Acceleration = 20.7792

Time step: 3 (Time Elapsed: 0.18)
Distance = 4.44008
Velocity = 17.1487
Acceleration = 18.6168

Time step: 4 (Time Elapsed: 0.24)
Distance = 8.55576
Velocity = 20.6719
Acceleration = 14.6799

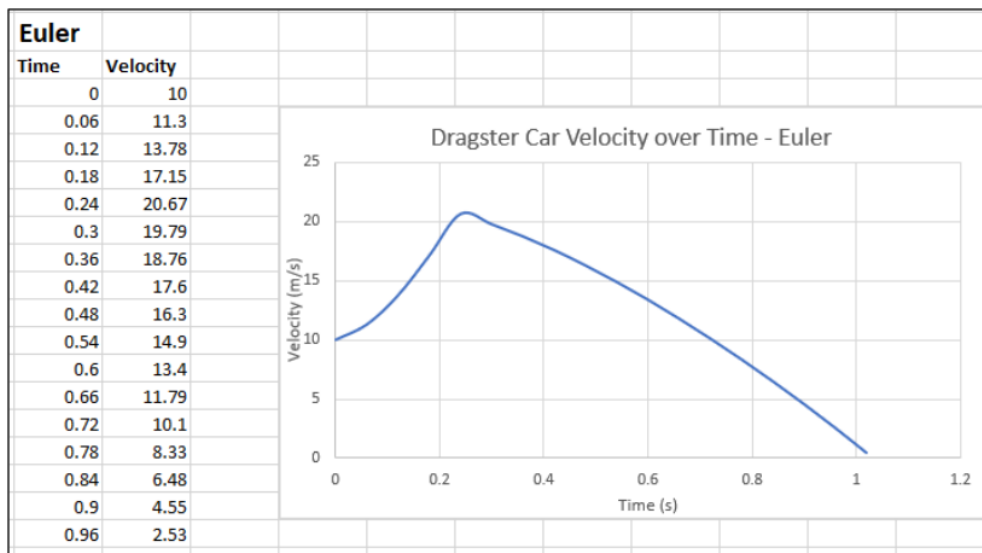
Time step: 5 (Time Elapsed: 0.3)
Distance = 14.7573
Velocity = 19.7923
Acceleration = -2.93184

Time step: 6 (Time Elapsed: 0.36)
Distance = 21.8825
Velocity = 18.763
Acceleration = -2.85924

Time step: 7 (Time Elapsed: 0.42)
Distance = 29.763
Velocity = 17.5961
Acceleration = -2.77828

Time step: 8 (Time Elapsed: 0.48)
Distance = 38.2091
Velocity = 16.3041
Acceleration = -2.69173
```

Above are the outputs from the code model (not the entire output), in which we can see at each step the elapsed time, Distance, Velocity and Acceleration. Below is this data plotted into an Excel graph.



The graph clearly shows the car quickly building momentum initially before its gradual deceleration down to 0m/s, last just over a second in length.

Verlet Method

Below is the code used to calculate the values in Verlet Method. Most of the code is shared from Euler's method, but is edited so that the velocity and distance are appropriately solved under the method.

```
//Calculate starting acceleration
dragForce = (0.21 * 1 * 9.81) + (0.5 * 0.34 * 1.2 * 0.01 * (currentVelocity * currentVelocity));
tractForce = 1.5 * (16 - currentDist);
accel = (tractForce - dragForce) / 1;

while (loop == 0)
{
    if (currentVelocity > 0)
    {
        dragForce = (0.21 * 1 * 9.81) + (0.5 * 0.34 * 1.2 * 0.01 * (currentVelocity * currentVelocity));

        if (currentDist > 8)
        {
            tractForce = 0;
        }
        else
        {
            tractForce = 1.5 * (16 - currentDist);
        }

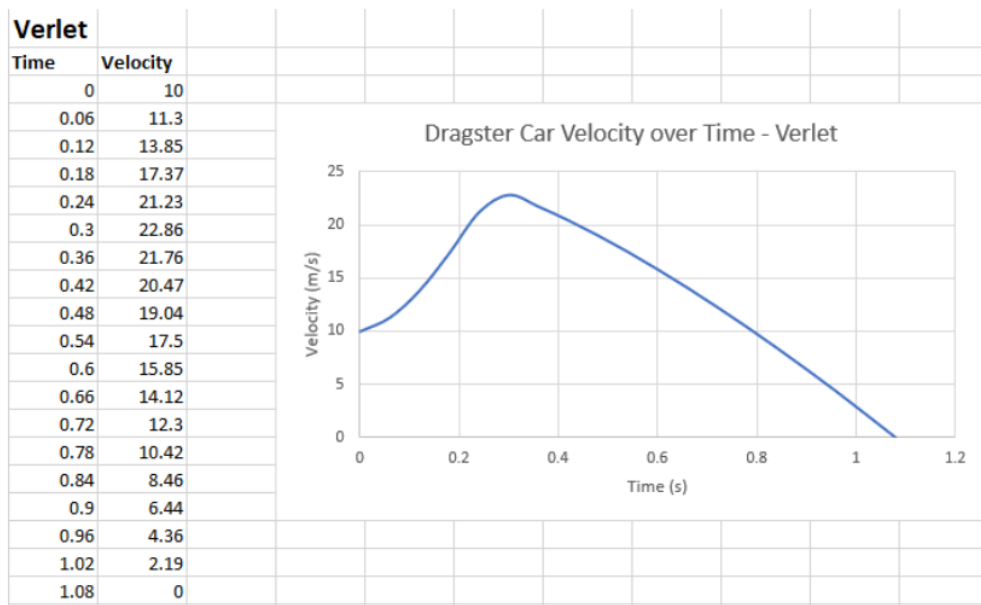
        time = time + 0.06;

        //Verlet Method
        midVelocity = currentVelocity + (accel * time / 2);
        newDist = currentDist + (midVelocity * time);
        newAccel = (tractForce - dragForce) / 1;
        newVelocity = midVelocity + (newAccel * time / 2);

        cout << "Time step: " << count << " (Time Elapsed: " << time << ")" << endl;
        cout << "Distance = " << newDist << endl;
        cout << "Velocity = " << newVelocity << endl;
        cout << "Acceleration = " << accel << endl << endl;

        currentDist = newDist;
        currentVelocity = newVelocity;
        accel = newAccel;
        count = count + 1;
    }
    else
    {
        //When the car stops, program ends
        dragForce = 0;
        cout << "Car halted" << endl;
        loop = 1;
        return 0;
    }
}
```

Below are the figures produced from the Verlet Method model of code and plotted onto a graph in Excel also. Both graphs do bare similar results but do have some differences which will be discussed.



Differences in the graphs

Both graphs bare a very similar shape, with a sharp increase in velocity initially as the car builds up its speed before the velocity rounds off at the top of the line before decreasing at a continuous gradual descent until velocity hits 0m/s. In terms of noticeable differences, Verlet has a higher maximum calculated velocity, peaking at 22.86m/s at 0.3s, where Euler's method only peaks at 20.67 at 0.24s. The Verlet graph also terminates at 0 at a later stage on the graph than the Euler graph does, with Euler terminating around the 1 second mark, and Verlet terminating at around 1.08 seconds. On top of that, it is noticeable that Verlet's method graph features a less severe curve at its climax, especially just after peak velocity, which would suggest that this is more accurate a representation, as the car is not having any other forces applied to it that would cause the graph to display unnaturally, especially if the vehicle in question is on a steady natural deceleration.

Task 2 - Coding the games Engine

Brief

The games engine requires 3 different additional mathematical operations to be coded. The 3 operations are calculating the distance and distance squared between 2 points, normalize a plane normal vector and then calculates the distance of a point from that plane, and a sphere-to-sphere collision detection program. For each of these, research into the mathematical theory will be required to determine an ideal method and formula to use, describe the function of the method, implement the code and test it.

Distance and distance squared between two points

For this step, the distance and square distance between two points in a 3D area will be calculated. The square distance can be calculated by calculating the sum of the difference between the x, y and z values of each vector squared, and then the result can be square rooted to solve for the distance, working similarly to Pythagorean maths. The main formula for the square distance will be $(\Delta x^2 + \Delta y^2 + \Delta z^2)$, and then will be square rooted for distance. The difference between values (Δ) can be calculated by taking away the value of one vector from the other (Vector2.x – Vector1.x). The solution of both distances will be the length between the two vector points. [2]

Code

```
//Defining Vector's 1 and 2 for use in the system
Vector const v1(3, 0, 1);
Vector const v2(5, 5, 4);
float dist = 0;
float distSq = 0;
cout << "v1 is: (" << v1.x << ", " << v1.y << ", " << v1.z << ")" << endl;
cout << "v2 is: (" << v2.x << ", " << v2.y << ", " << v2.z << ")" << endl;

cout << endl << endl;
//=====

//Task 1 - Distance between 2 points

//Calculating the distance
distSq = ((v2.x - v1.x) * (v2.x - v1.x)) + ((v2.y - v1.y) * (v2.y - v1.y)) + ((v2.z - v1.z) * (v2.z - v1.z));
cout << "The Square Distance between point 1 and 2 is " << distSq << " units" << endl;
dist = sqrt(distSq);
cout << "The distance between point 1 and 2 is " << dist << " units" << endl;

cout << endl << endl;
```

Testing

Below is a testing table listing all the tests I ran on the code for this task. Due to the fact that the vectors are set as constants and do not change throughout the code, therefore testing them as different values will not be relevant.

Test No.	Description of test	Expected Outcome	Actual Outcome
1	Testing the output of initial Vectors v1 and v2.	The code will output Vector v1's points, followed by Vector v2.	"v1 is: (3, 0, 1)" "v2 is: (5, 5, 4)"

2	Testing the formula correctly solves the distance squared by manual calculation for comparison.	The code correctly calculates the distance squared between vector v1 and v2.	distSq is set to 38, which is the correct solution
3	Testing the formula correctly solves the distance squared by manual calculation for comparison.	The code correctly calculates the distance between vector v1 and v2.	dist is set to 6.1644 (to 4 decimal places), which is the correct solution
4	Testing the output of the distance and distance squared.	The code will output the distance squared, followed by the distance	"The square distance between point 1 and 2 is 38 units" "The distance between point 1 and 2 is 6.16441 units"

Vector Plane

For this step, first the given values of a vector plane are normalized, and then the distance between a given point and the vector plane will be calculated. To first calculate the normalized plane vector, each value in the vector must be divided by the normal vector's length to provide a new value for the vector. With the normalized vector, calculating the distance of a point from the plane can be done. [6]

Code

```
//Task 2 - A Plane

//Part I - Normalizing a plane normal vector
//Defining the vector n
normal n;
n.nx = 1;
n.ny = 5;
n.nz = 3;
cout << "Normal 'n' is (" << n.nx << ", " << n.ny << ", " << n.nz << ")" << endl;

//Calculating Length (|n|)
float nLength = sqrt((n.nx * n.nx) + (n.ny * n.ny) + (n.nz * n.nz));
cout << "Length of 'n' is " << nLength << " units" << endl;

//Calculating normalised vector
float normalX = 1 / nLength;
float normalY = 5 / nLength;
float normalZ = 3 / nLength;
cout << "The normalized plane normal vector is (" << normalX << ", " << normalY << ", " << normalZ << ")" << endl;

//Part II - Calculate the distance of a point from the plane
//Defining point X, not on the plane
position vX;
vX.x = v1.x;
vX.y = v1.y;
vX.z = v1.z;

//Outputting the distance of point X from the plane
dist = pointToPlaneDist(vX, v1, normalX, normalY, normalZ);
if (dist < 0)
{
    dist = dist * -1;
}
cout << "Distance of the point from the plane is " << dist << " units" << endl;

cout << endl << endl;
```

```
float pointToPlaneDist(position vP, Vector v1, float normalX, float normalY, float normalZ)
{
    //Defining Point P0, which is on the plane
    position vP0, vNn;
    vP0.x = -1;
    vP0.y = 3;
    vP0.z = 4;

    //Calculate the distance from the plane (normal n.x + d)
    float normalNX = ( v1.x * (normalX) + v1.y * (normalY) + v1.z * (normalZ));
    float d = -1 * (-1 * (normalX) + 3 * (normalY) + 4 * (normalZ));
    return d + normalNX;
}
```

Testing

Below is a testing table listing all the tests I ran on the code for this task. Any tests that were relevant to previously tested parts (such as testing the initial vectors) will not be repeated in this table.

Test No.	Description of test	Expected Outcome	Actual Outcome
1	Testing whether the code successfully calculates the normal Length by comparing to a manual calculation.	The code correctly calculates the normal length nLength.	"Length of n is 5.91608"
2	Testing whether the code correctly calculates the normalized plane normal vector by comparing to a manual calculation.	The code correctly calculates the normalized vector.	"The normalized plane normal vector is (0.169031, 0.845154, 0.507093)"
3	Testing if the code successfully calculates normal NX + d	The code correctly calculates and returns normal NX + d from the structure	dist is outputted as 3.38062 units after being calculated correctly in the structure
4	Testing the if command to always make the distance a positive value	If the dist comes out of the structure as a negative value, it is multiplied by -1 to turn it positive	Values were changed so dist is outputted as a negative value from the structure, and it was changed to a positive value.

Sphere-to-sphere collision detection

For this step, two spheres will be generated in a 3D area and the code will test whether the two spheres collide with each other. The code for this will function similarly to the first task, where the distance between 2 points was calculated. Each sphere will have a centre point and a radius. First, as per the first task, the distance between the 2 centre points of the spheres will be calculated (square root of $\Delta x^2 + \Delta y^2 + \Delta z^2$). Once this length is calculated, the radii of both spheres will be deducted from that value and the remainder will determine if the spheres are touching (if the remainder is 0) or have collided (if the remainder is less than zero). [2]

Code

```
//Task 3 - Sphere to Sphere Collision detection
//Defining the sphere radii, with vectors v1 and v2 acting as the central point respectively
float r1 = 5;
float r2 = 7;

//Calculating the distance between the centre points of each sphere
dist = sqrt(((v2.x - v1.x) * (v2.x - v1.x)) + ((v2.y - v1.y) * (v2.y - v1.y)) + ((v2.z - v1.z) * (v2.z - v1.z)));

//Checking for collision
float collision = dist - r1 - r2;
if (collision > 0)
{
    cout << "Objects have not collided, " << collision << "units between spheres" << endl;
}
else
{
    if (collision == 0)
    {
        cout << "Objects are touching, " << endl;
    }
    else
    {
        cout << "Objects have collided" << endl;
    }
}

cout << endl << endl;
```

Testing

Below is testing table listing all the tests I ran on the code for this task. Any tests that were relevant to previously tested parts (such as testing the distance, as it was calculated in parts 1 and 2 with the same formula) will not be repeated in this table.

Test No.	Description of test	Expected Outcome	Actual Outcome
1	Tests whether the collision variable is correctly calculated and assigned.	Collision (which represents distance between the edges of the 2 spheres) will be equal to the distance between centre points minus the radii of each sphere.	Collision is equal to the distance between the centre points minus the radii (-5.83559)
2	Tests that the code correctly detects if the spheres have not collided.	Collision will be greater than 0 and the code will tell the user how far apart the spheres are.	"Objects have not collided, 3.16441 units between spheres" – radii were edited to 1 and 2 units.
3	Tests that the code correctly detects if the spheres are touching.	Collision will be equal to 0 and code will say objects are touching.	"Objects are touching" – radii were edited to 1 and 5.16441 units.
4	Tests that the code correctly detects if the objects have collided	Collision will be less than 0 and the code will say the objects have collided.	"Objects have collided"

Summary

Numerical integration plays an important in the mathematical side of game development, and there are a number of various types that each work differently which allow them to be a better or worse option for different situations, with some making more calculations for the sake of accuracy but at the cost of speed, which could start becoming an issue if a large number of calculating were having to be made every second. Comparing the Verlet and Euler method, the differences in the results can be seen, with Verlet seeming to be the more accurate method for that specific scenario. The code shows the mathematical elements behind how features in video games works, especially collision detection.

Bibliography

- [1] H. Serrano, "Algorithms in Game Engine Development," 15 December 2016. [Online]. Available: Algorithms in Game Engine Development — Harold Serrano. [Accessed 8 November 2020].
- [2] J. & B. L. Van Verth, Essential Mathematics for Games and Interactive Applications, Boca Raton: Taylor and Francis Group, 2016.
- [3] J. Dickinson, "Numerical Integration in Games Development," 22 January 2015. [Online]. Available: <https://jdickinsongames.wordpress.com/2015/01/22/numerical-integration-in-games-development-2/>. [Accessed 8 November 2020].
- [4] N. Pronost, "Game Physics - Game and Media Technology - Utrecht University," [Online]. Available: <https://perso.liris.cnrs.fr/nicolas.pronost/UUCourses/GamePhysics/lectures/lecture%205%20Numerical%20Integration.pdf>. [Accessed 8 November 2020].
- [5] J. Rodriguez, "Numerical Methods for Physics Integration in Video Games," 9 July 2016. [Online]. Available: <http://vinoisnotouzo.com/blog/10.htm>. [Accessed 8 November 2020].
- [6] K. Stroud, Advanced Engineering Mathematics, New York: Palgrave Macmillan, 2003.

GAM Assessment 1a Writeup

GRADEMARK REPORT

FINAL GRADE

GENERAL COMMENTS

Instructor

79/100

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

GRADING FORM: SMU GENERIC_GJ

RHYS DAVID WHATTY

78.50

RESEARCH-INFORMED LITERATURE



Some good work here. Well done. A good review of methods required for the assignment.

7

KNOWLEDGE AND UNDERSTANDING



Good understanding of the concepts involved and they are applied well to the problem.

17

ANALYSIS



Good arguments for integrator choice provided and graph of velocity plotted for Verlet and Euler's methods. Missed opportunity to also plot position and acceleration. Test design suits the problem and code.

13

PRACTICAL APPLICATION AND DEP.



Application covers the requisite tasks well. Could be better documented in places though.

38

SKILLS FOR PROFESSIONAL PRAC.



Good referencing. Rodriguez's looks useful here. Structured report..

3.5