Assignment 2: Physics simulation: bouncing spheres

1 Introduction

1.1 Assignment Goals

This problem involves predicting the movement and collisions of a large number of objects moving in a three-dimensional space.

The problem is relevant, for example, to air-traffic control, self-driving cars, or game simulations (asteroids, curling, pool)

The objects undergo perfect elastic collision.

1.2 Group Size

For this assignment, the maximum group size is 3.

1.3 Due Date

This assignment is due Apr 24 2021 at 11:59pm.

This assignment has a sliding late window of partial credit for late assignments. The late window ends at precisely 25/04/2021 **23:59**. If you submit after the due date but before the close of the late window, your assignment grade will be scaled from 100% to **90%** on a linear time scale from the due time to the window end.

1.4. Identifying authorship

Each author should be identified on a separate comment line with the following format:

// Copyright year person1name email1@sdu.edu
// Copyright year person2name email2@sdu.edu

where personXname is replaced by your name (in any format).
The line must start with // and must end with your full SDU email.

2 Background: Modeling data with Classes

The idea of a class is useful not only as a way of structuring code, but as a way of thinking about:

- modeling
- abstraction
- defining specifications
- defining expectations
- defining requirements
- organization
- relationships between parts of a larger system

More concretely, we can used classes to

- organize and collect related information
- define methods that operate on the object
- allow for integration into the language using operators.

In the C++ standard library most code is provided in the form of families of related classes.

2.1 Classes and Objects

A class is like a type.

An object is like a variable.

Please, cover C++ Classes topic before the start of the assignment.

2.2 Operator Overloading

In C++, we can make operators to work for user defined classes. This means C++ has the ability to provide the operators with a special meaning for a data type, this ability is known as operator overloading.

For example, we can overload an operator '+' in a class like String so that we can concatenate two strings by just using +.

Other example classes where arithmetic operators may be overloaded are Complex Number, Fractional Number, Big Integer, etc.

Since we are going to work with vectors, it would be nice to overload standard mathematical operations behaviour to avoid repetition of the code.

You can create a new Vector class:

```
class Vector3d {
public:
   double x, y, z;

   Vector3d(double x, double y, double z) {
      this -> x = x;
      this -> y = y;
      this -> z = z;
   }

   Vector3d() {
      this -> x = 0;
      this -> y = 0;
      this -> z = 0;
   };
```

Let's initialize two vectors and third vector *c* as a sum of vectors *a* and *b*:

```
Vector3d a(1,2,5);
Vector3d b(1,2,5);
Vector3d c(a.x + b.x, a.y + b.y, a.z + b.z);
```

We can overload an operator '+' in a class *Vector3d* so that we can concatenate two vectors by just using +

```
Vector3d operator+(Vector3d pos) {
      return Vector3d(x + pos.x, y + pos.y, z + pos.z);
   }
```

You can find example code here.

3 Bouncing Spheres

Your task is to determine collisions between moving objects given a starting scenario. The program will be called "spheres.cpp"

3.1 Starting scenario

The objects are specified by entering the following information: the mass, radius, x/y/z position, x/y/z velocity and name of each sphere.

All the values are in SI units: kg, m, and m/s.

The location information (x/y/z coordinates in meters) and velocity (x/y/z coordinates, in meters/second) is the initial values for time t = 0.

3.2 Motion model

The objects travel in three dimensional space. Each object travels in a straight line at a constant velocity. Each object will continue travelling in this direction forever, unless it collides with another object.

We ignore any gravitational or frictional forces.

3.2.1 Limits of space

In this simulation, the spheres are all contained within a spherical container, called “the universe”.

When objects hit the edge of the universe, they bounce back into the universe without loss of energy as if they are bouncing off of a wall with effectively infinite mass.

The radius of the containing sphere is specified on the command line, and the center of the containing sphere is the origin, i.e. (0,0,0).

3.2.2 Spheres disappear

Each sphere can only experience a finite number of collisions (either with each other or with the container wall).

The number of collisions is specified on the command line.

When a colliding sphere reaches its collide limit, it undergoes that elastic collision and then vanishes. Its energy and momentum are therefore removed from the universe.

3.3 Collision model

Two objects collide at the moment the distance between them becomes equal to the collision distance. For this problem, the collision distance will be the sum of the radii of the colliding spheres.

3.3.1 Elastic Collisions

When two objects collide, they undergo elastic collision.

If two objects with positions r1 and r2 collide, the new velocities will be

$$\mathbf{v}_1' = \mathbf{v}_1 - \frac{2m_2}{m_1 + m_2} \frac{(\mathbf{v}_1 - \mathbf{v}_2) \cdot (\mathbf{r}_1 - \mathbf{r}_2)}{(\mathbf{r}_1 - \mathbf{r}_2) \cdot (\mathbf{r}_1 - \mathbf{r}_2)} (\mathbf{r}_1 - \mathbf{r}_2)$$

$$\mathbf{v}_2' = \mathbf{v}_2 - \frac{2m_1}{m_1 + m_2} \frac{(\mathbf{v}_2 - \mathbf{v}_1) \cdot (\mathbf{r}_2 - \mathbf{r}_1)}{(\mathbf{r}_2 - \mathbf{r}_1) \cdot (\mathbf{r}_2 - \mathbf{r}_1)} (\mathbf{r}_2 - \mathbf{r}_1)$$

In the above equation $\cdot$ represents the dot product. If u = (a, b) and v = (c, d) then

$$\mathbf{u} \cdot \mathbf{v} = ac + bd$$

These equations are from https://en.wikipedia.org/wiki/Elastic_collision in section "Two-dimensional collision with two moving objects".

They apply equally well to three dimensions.

3.3.2 Collision prediction

Let the position of object i at time t be denoted as $r_i(t)$ and its radius R_i.

For any pair of objects i and j, they collide at the moment that

$$|\mathbf{r}_i(t) - \mathbf{r}_j(t)| = R_i + R_j$$

This is most easily calculated by squaring, so we get

$$|\mathbf{r}_i(t) - \mathbf{r}_j(t)|^2 = (\mathbf{r}_i(t) - \mathbf{r}_j(t)) \cdot (\mathbf{r}_i(t) - \mathbf{r}_j(t)) = (R_i + R_j)^2$$

Since the velocities are constant, you can write

$$\mathbf{r}_i(t) = \mathbf{p}_i + t\mathbf{v}_i$$

where p_i is the position of the center of object i at time $t = 0$ and v_i is the velocity of object i.

The equation can be solved for t, and it turns out that it is of the form

$$at^2 + bt + c = 0$$

This is the quadratic equation, and it has 0, 1, or 2 real solutions.

It is an exercise for you to determine what these mean as applied to the physics of the situation. For the collision between i and j to be a real collision, it turns out that

$$(\mathbf{r}_i - \mathbf{r}_j) \cdot (\mathbf{v}_i - \mathbf{v}_j) < 0$$

is a necessary condition. This means “these objects are approaching each other.”

Time did not exist before$t = 0$, so if two objects would have collided in the past, we ignore this event. It never happened.

3.3.3 Collision Modeling

We assume that the object collisions occur instantly. If multiple collisions occur simultaneously, they should be processed pair-wise until no more collisions are pending.

These idealized assumption leads to possibly unrealistic behaviors. Your program should model this behavior correctly.

4 Program requirements

4.1 Input format

The input will be from stdin

Here is an example input:
```
20 1    0 0 0        0 0 1  one
2   5    0 1 100     0 0 0  two
```

Sphere one has mass 20, radius 1. Its initial position is (0,0,0) and its initial velocity is (0,0,1)

Sphere two has mass 2, radius 5. Its initial position is (0,1,100) and its initial velocity is (0,0,0)

When the program encounters end-of-file (EOF, or ctrl-D), it means that all objects have been entered and the simulation should begin.

The number of spheres can be 0, 1, 2, or more (there is no limit).

4.2 Command Line arguments

The program must be run with two command line arguments, like this:

./spheres 120 3

which means the container is 120 m radius and the maximum collisions is 3.

4.2.1 Numerical ranges

All mass, radius, positions can be arbitrary floating point numbers. The collision limit is a positive integer.

4.3 Program behavior

4.3.1 Input Phase

The program starts off by prompting for the starting scenario:

Please enter the mass, radius, x/y/z position, x/y/z velocity
and name of each sphere
When complete, use EOF / Ctrl-D to stop entering

4.3.2 Output phase

The program, once it knows about the initial conditions, will first print out the initial situation, including the universe's radius, the max collisions, and the status of each sphere.

It then reports on

- each collision (sphere to sphere)
- each reflection (sphere to container)
- each sphere disappearance

After each collision or reflection, the condition of all spheres is reported, including the new velocities and the number of bounces each sphere has undergone.

If the collision or reflection results in a sphere disappearance, this is also reported on.

When there are no more events, the program should exit with return code 0.

Each floating point number should be printed using the :g format specifier, as in f"{x:g}"

4.3.3 Input and Output Example

Consider this scenario with three spheres:

20 1 0 0 0 0 0 1 one
2 5 0 1 100 0 0 0 two
3 1 2 -1 -2 0 0 0 three

The output of

./spheres 120 2

when this is the input will be:

Here are the initial conditions.
universe radius 120.0
max collisions 2
one m=20 R=1 p=(0,0,0) v=(0,0,1) bounces=0
two m=2 R=5 p=(0,1,100) v=(0,0,0) bounces=0
three m=3 R=1 p=(2,-1,-2) v=(0,0,0) bounces=0
energy: 10
momentum: (0,0,20)

Here are the events.

time of event: 94.0839
colliding one two

one m=20 R=1 p=(0,0,94.0839) v=(0,-0.0298792,0.823232) bounces=1
two m=2 R=5 p=(0,1,100) v=(0,0.298792,1.76768) bounces=1
three m=3 R=1 p=(2,-1,-2) v=(0,0,0) bounces=0
energy: 10
momentum: (0,0,20)

time of event: 102.539
reflecting two
one m=20 R=1 p=(0,-0.252632,101.044) v=(0,-0.0298792,0.823232) bounces=1
two m=2 R=5 p=(0,3.52632,114.946) v=(0,0.189874,-1.78267) bounces=2
three m=3 R=1 p=(2,-1,-2) v=(0,0,0) bounces=0
energy: 10
momentum: (0,-0.217836,12.8993)

disappear two

time of event: 124.346
reflecting one
one m=20 R=1 p=(0,-0.904204,118.997) v=(0,-0.0173657,-0.823591) bounces=2
three m=3 R=1 p=(2,-1,-2) v=(0,0,0) bounces=0
energy: 6.78604
momentum: (0,-0.347314,-16.4718)

disappear one

4.4 Notes

The ID is a single contiguous string in any format.

The objects are never in an initially overlapping or colliding state. Therefore, you may assume that every object is at least the collision distance away at time t = 0 (i.e. touching is ok but not overlapping), and the first collision occurs at t > 0.

4.5 Error handling

If the input format has any problems (too many fields on one line, invalid numbers, etc) the program should exit with return value 1.

As long as the correct return code is produced, any output to stderr is allowable, but nothing should be printed to stdout (except for the initial instructions.)

5 Files for Download

The following ZIP file contains a suite of input files and output files.

examples_spheres.zip

If the input scenario is “three.txt”, then

./spheres 120 2 < three.txt > three_120_2_myid.txt

creates the output “answer” file “three_120_2_myid.txt”

You can use provided files as an example of how the program should work.

The sample structure of the program without implementation of two methods:
Spheres_template.cpp
However, do not treat the sample code as a complete implementation of the assignment.
The sample provided only as a reference.