Wiegold Stannard D (FCES)

17736382

Object oriented programming

Assignment 1

Contents

[Introduction 2](#_Toc119706196)

[Background 2](#_Toc119706197)

[Polymorphism 2](#_Toc119706198)

[Inheritance 3](#_Toc119706199)

[Collision Detection 3](#_Toc119706200)

[Collision Resolve 4](#_Toc119706201)

[UML 4](#_Toc119706202)

[Code 5](#_Toc119706203)

[Testing 12](#_Toc119706204)

[Evaluation 14](#_Toc119706205)

[References 15](#_Toc119706206)

[Table of Figures 15](#_Toc119706207)

# Introduction

The study of particle behaviour and collision-detection is used in a wide range of applications, not only in simple computer games but is also vitally important in the research and understanding of fluids and gasses in the petrochemical and many other industries.

(Tamrakar, Devarampally and Ramachandran, 2018)

This assignment is to study and simulate the behaviour of particles within a regular two-dimensional world, each particle has several attributes such as size, velocity, drag, acceleration, mass and location, which if used correctly will show realistic simulation within a graphical 2d frame.

The graphic platform used for the visualisation for the assignment is OpenGL, OpenGL is an API for both 2D and 3D vector graphics and is available for different platforms. (‘Praise for OpenGL R Programming Guide, Eighth Edition’, no date)

# Background

## Polymorphism

Polymorphism in computer programming is where the same method or object can behave in different ways depending on the scenario it’s working with, this is mostly a result of a method being overloaded.(‘Object-Oriented Programming in C++, Fourth Edition’, no date)

Method or function overloading is where there are more than one different declarations of a function with the same name, each declaration will have a different template of parameters being passed to or returned from the function. This can also apply to a common function/method name being declared in both a base class and a derived class, however, the method or function will only be polymorphic in the derived class in this case.

Below are a few (of many) examples of polymorphism created by function overloading within the particle class:

*void setVelocity(const Vector2 &velocity);*

*void setVelocity(const float x, const float y);*

*Vector2 getVelocity() const;*

*void getVelocity(Vector2 \*velocity) const;*

*void setPosition(const float x, const float y);*

*void setPosition(const Vector2 &position);*

*Vector2 getPosition() const;*

*void getPosition(Vector2 \*position) const;*

*Vector2()*  is overloaded several times within the “coremath” class so is polymorphic.

## Inheritance

This where a derived class inherits the attributes and methods from a base class, in this case the derived class becomes bigger than itself by keeping its own properties and functionality along with those inherited from the base class, however, the base class is still just the base class.

Within the code and indicated by the UML class diagram it’s clear that the ParticleCollision class inherits attributes and methods from the ParticleContactGenerator class. (Figure 1 - UML Diagram)

# Collision Detection

Particles travel within the 2D space until they collide with either part of the frame or another particle, once a collision is detected then the behaviour of the particle will change (direction and speed) depending on several of the factors mentioned above.

Collisions:

For both this assignment and real-world study, the detection of collisions is essential, the detection of a collision in this simulation is computationally quite expensive in the initial ‘brute-force’ manner. However, the study of spatial partitioning methods such as quadtrees along with various grid methods show how efficiency of collision detection can be improve considerably, the benefits may be negligible with a low number of particles, but considerable for larger particle counts.

When using the simple brute-force method for n blobs there would be n \* (n-1) /2 tests for a collision, this because there wouldn’t be a test for a blob colliding with itself (n-1), and of course if blob1 and blob2 have been tested then blob2 and blob1 (/2) would not need to be tested.

The use of data structures such as quad trees (not yet implemented in this code) can help improve this situation, it does this by subdividing the box into four other boxes and basically only test for blobs that are close enough to actually collide. This process can be recursive and each level further reducing the number of tests (still brute force) required by a factor of four.

Throughout this application there are different types of blob collisions, each type of collision is detected and responded to in different ways.

Each blob is an instance of the Particle class and the actual location of a particular blob is its x,y coordinates of the centre of the blob within the 2D world, in this case blobs are circular two-dimensional objects. As a blob is circular one of its attributes will of course be radius, this radius is a factor in the calculation (added or subtracted to the actual x,y) when testing for a collision.

Particle to platform collisions.

Particle to particle collisions:

This is currently done in quite a long-winded (brute force) way, this is where for every time interval each blob is checked to see if any of the other blobs are making contact with it.

The actual test for a blob-on-blob collision is if the distance between the two blobs being tested is less than/equal to the sum of the radii of both blobs then a collision has occurred, this allows some flexibility to cope with a situation where a collision between blobs of different sizes could be detected.

Limitations of the system used:

The application is based on regular time slots where the ‘glutTimerFunc’ in the main section of the program is passed a numeric value and a function name, after each period of time the function will be called. If collision detection is not complete due to either inefficient methods like the simple ‘brute-force’ method in the predetermined time interval or very large particle counts, then problems will arise. More efficient method will allow processing of larger particle groups within the same time slot.

Penetration:

This is when the centre of a particle is closer to a boundary than its radius, or in the case of particle to particle collisions where the centre of the particles is less than the sum of their radii, this is due to particle movement within a time slot and it being tested, increasing the time interval would probably worsen this situation.

(Ericson, 2004)

# Collision Resolve

Collision resolve is the change of behaviour in a particle after a collision has been detected, a change in behaviour depends on most attributes of the particle and its movement before the collision. One can imagine that a particle colliding head on with a flat surface would literally reverse its direction, things like mass, velocity and restitution(the bounciness) will all affect its velocity after the collision. However, a particle colliding with another particle where both particles are spherical then glancing collision more imaginable and the behaviour is very different from the previous example. A real-world visualization of the application of the physics involved in this simulation can be seen on any pool table everyday.

(Lengyel *et al.*, 2012)

# UML

Diagram

Description automatically generated

Figure - UML Diagram

# Code

When developing the particle simulation, several variables needed to be declared to facilitate the graphical weighing scale simulator. This was done by declaring integers for left and right sides, if there was an imbalance between these two values then the scale will tip left or right according to there weights. Purely for visual effect lines were drawn to and from each particle, again for visual effect only points would be drawn to highlight the start and end point of the platform, variables to support these features can be seen in the (Figure 2 - Libraries and Variables) below.

Text

Description automatically generated

Figure - Libraries and Variables

The count variable will hold the platform index, which is used to identify each specific platform which is used when collision testing and looking for a specific platform for the weighing scale to be activate. (Figure 3 - Count Variable Platform)

Text

Description automatically generated

Figure - Count Variable Platform

When the specific platform for the scale has been collided with and the weight value for the side of the platform is less then 25 increase the weight of that side of the platform if both of the weight values are greater then 25 the reset both weight values to 0. This code is repeated three times for each section of the platform. (Figure 4 - Calculation for the Scale)

Text

Description automatically generated

Figure - Calculation for the Scale

From seen in the figure below a new function was implemented in the BlobDemo class for the scale. (Figure 5 - Creating a Function)

Text

Description automatically generated

Figure - Creating a Function

The code below shows the creation of the four platforms that form the box around the 2d world, also shown are parameters for the 2d world (Figure 6 - Creating the Box)

Text

Description automatically generated

Figure - Creating the Box

The code below for the creation of platforms show how the weighing scale created. (Figure 7 - Weighing Scale Platforms)

Text

Description automatically generated

Figure - Weighing Scale Platforms

The code below shows how the hexagon was created. (Figure 8- Hexagon)

Text

Description automatically generated

Figure - Hexagon

The platforms get passed the index variable i to the count variable. (Figure 9 - Index Count)

Text

Description automatically generated

Figure - Index Count

The code below shows the breaking up of particle count into smaller groups contained in the width of the 2d world, this allows more particles to be released into the world at any one time. A minor addition to this code provides a simple random radius for the particles between 3 and 1 (Figure 10 - blob[j]setPosition)

Text

Description automatically generated

Figure - blob[j]setPosition

The movement of scale is with a function that depending on the imbalance between the left and the right weight will cause appropriate movement of the scale. (Figure 11 - Scale Part 1)(Figure 12 - Scale Part 2)

Text

Description automatically generated

Figure - Scale Part 1

Text

Description automatically generated

Figure - Scale Part 2

## Testing

The figures below show the start point, end point and middle section collision test with platform to particle. The Images will show that all particles start life as one colour change at the point of collision.(Figure 13 - Start Point Test)(Figure 15 - Platform Middle Test)(Figure 14 - End Point Test)

Graphical user interface

Description automatically generated with medium confidence

Figure - Start Point Test

Graphical user interface

Description automatically generated

Figure - End Point Test

A picture containing graphical user interface

Description automatically generated

Figure - Platform Middle Test

This shows the function of the scale when unbalanced. (Figure 16 - Weight Values)

A picture containing text, electronics

Description automatically generated

Figure - Weight Values

# Evaluation

This study although challenging was very interesting, further development and understanding of many of the concepts learnt could be applied to 3d simulation at some point in the future. On reflection there are many aspects of the simulation that could be later improved and met targets fulfilled.

# References

Ericson, C. (2004) ‘Real-Time Collision Detection’, *Real-Time Collision Detection*, pp. 1–595. Available at: https://doi.org/10.1201/b14581.

Lengyel, E. *et al.* (2012) ‘Mathematics for 3D Game Programming and Computer Graphics Third Edition A part of Cengage Learning’.

‘Object-Oriented Programming in C++, Fourth Edition’ (no date).

‘Praise for OpenGL R Programming Guide, Eighth Edition’ (no date).

Tamrakar, A., Devarampally, D.R. and Ramachandran, R. (2018) ‘Advanced multiphase hybrid model development of fluidized bed wet granulation processes’, *Computer Aided Chemical Engineering*, 41, pp. 159–187. Available at: https://doi.org/10.1016/B978-0-444-63963-9.00007-5.

# Table of Figures

[Figure 1 - UML Diagram 4](#_Toc119705613)

[Figure 2 - Libraries and Variables 5](#_Toc119705614)

[Figure 3 - Count Variable Platform 5](#_Toc119705615)

[Figure 4 - Calculation for the Scale 6](#_Toc119705616)

[Figure 5 - Creating a Function 6](#_Toc119705617)

[Figure 6 - Creating the Box 7](#_Toc119705618)

[Figure 7 - Weighing Scale Platforms 8](#_Toc119705619)

[Figure 8- Hexagon 9](#_Toc119705620)

[Figure 9 - Index Count 9](#_Toc119705621)

[Figure 10 - blob[j]setPosition 10](#_Toc119705622)

[Figure 11 - Scale Part 1 11](#_Toc119705623)

[Figure 12 - Scale Part 2 12](#_Toc119705624)

[Figure 13 - Start Point Test 13](#_Toc119705625)

[Figure 14 - End Point Test 13](#_Toc119705626)

[Figure 15 - Platform Middle Test 14](#_Toc119705627)

[Figure 16 - Weight Values 14](#_Toc119705628)