Collusion Detection & Resolution

Introduction

In this project, we usually implement an algorithm that will detect collisions of particles with each other and resolve those collisions. According to (Zhang & Kim, 2007) "the goal of collision detection is to determine whether one or more geometric objects overlap in space and, if they do, detect the overlapping features, also known as collision witness features". From this line, we can simply understand that detecting digital things that touch in a virtual world helps us understand how they interact with each other. We will also look at the points, surfaces, and areas where they meet to figure out how they're connected. This helps make things in the virtual world behave realistically and accurately. Once we figure out the point where those particles are meeting with each other, we will simply determine the angle, velocity, and other properties of those particles that help us understand what features they have, and based on that, we shall resolve their collusion with each other in this virtual work of those particles. After detecting the particles collusion, we can simply solve those relations by changing those particles properties and the angle of contact so that those Pericles will respond according to their properties and move at different angles based on their angle of collusion.

Understand the Collusions

_When particles in real or virtual worlds collide because of motion, gravity, or digital interactions, this is known as a particle collision.
That means that collusion may happen because of the masses, forces, and velocity parameters that we set during the simulation of our virtual particle world. So when can we say that two particles are colliding with each other? From (Kulon's, 2023) lecture slide of week three, we can visualise that the Euclidean distance between two particles is less than the total distance between those two particles. Then we can say that those two particles are colliding with each other. So this is how we can detect particles collusion.

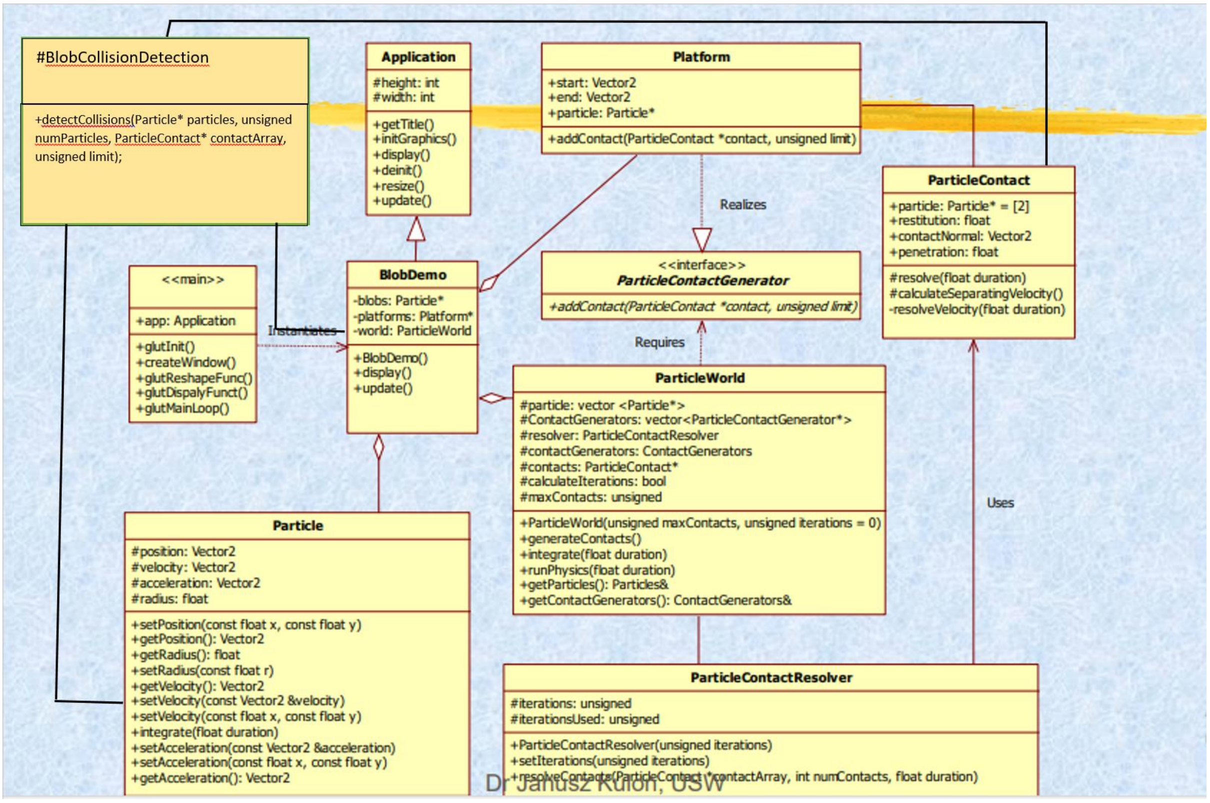
Collision resolving technique

We can simply resolve the particle collisions by detecting their angle when they collide with each other. A simple solution would be like two particles travelling in the same direction, and if they collide, we can add a new positive velocity as their angle of collision was positive. If two particles were coming from opposite directions and if they collided, then we can simply put negative velocity as the angle of collusion is greater than 90, which means after the collision the particles will move in the negative direction of their velocity.

Contact generation algorithm, penetration resolution

By brute force, finding collisions with one object with every other is a very slow process, as we need to find out if any particles touch or overlap with each other. When it finds a hit, it tries to fix it by moving the objects apart or adjusting how they move. If there are lots of problems and a lot of checks, the performance will drop more and more. Instead, more efficient ways, like grouping objects or organising them in a smart way, are used to speed things up, and still a brute force way can find collisions accurately. ? From (Kulon's, 2023)lecture slide of week four, we could see that the particle contact class was created where a particle contact array was used to store all the particle lists that collided with each other. Then there is a contact resolver class that resolves all the perticles of contact, which are stored in the contact array.

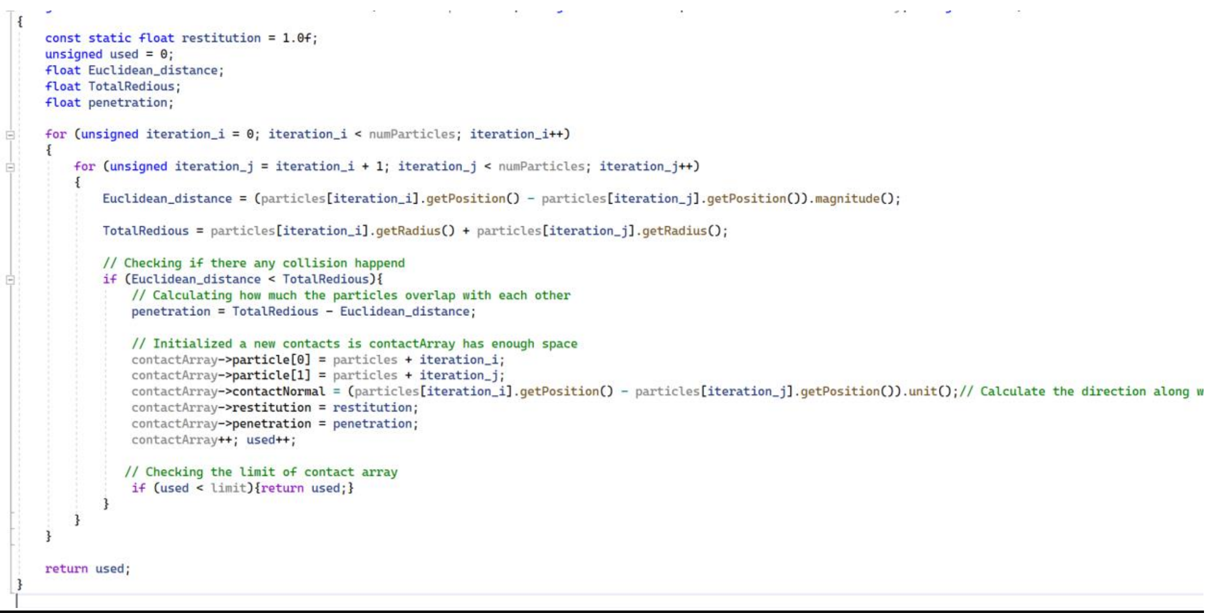
UML Class Diagram



#BlobCollisionDetection
+detectCollisions(Particle* particles, unsigned numParticles, ParticleContact* contactArray, unsigned limit);
Application
#height: int
#width: int
+getTitle()
+initGraphics()
+display()
+deinit()
+resize()
+update()
Platform
+start: Vector2
+end: Vector2
+particle: Particle*
+addContact(ParticleContact *contact, unsigned limit)
Realizes
ParticleContact
+particle: Particle* = [2]
+restitution: float
+contactNormal: Vector2
+penetration: float
#resolve(float duration)
#calculateSeparatingVelocity()
-resolveVelocity(float duration)
<<main>>
+app: Application
+glutInit()
+createWindow()
+glutReshapeFunc()
+glutDispalyFunct()
+glutMainLoop()
Instantiates
BlobDemo
-blobs: Particle*
-platforms: Platform*
-world: ParticleWorld
+BlobDemo()
+display()
+update()
<<interface>>
ParticleContactGenerator
+addContact(ParticleContact *contact, unsigned limit)
Requires
ParticleWorld
#particle: vector <Particle*>
#ContactGenerators: vector<ParticleContactGenerator*>
#resolver: ParticleContactResolver
#contactGenerators: ContactGenerators
#contacts: ParticleContact*
#calculateIterations: bool
#maxContacts: unsigned
+ParticleWorld(unsigned maxContacts, unsigned iterations = 0)
+generateContacts()
+integrate(float duration)
+runPhysics(float duration)
+getParticles(): Particles&
+getContactGenerators(): ContactGenerators&
Uses
Particle
#position: Vector2
#velocity: Vector2
#acceleration: Vector2
#radius: float
+setPosition(const float x, const float y)
+getPosition(): Vector2
+getRadius(): float
+setRadius(const float r)
+getVelocity(): Vector2
+setVelocity(const Vector2 &velocity)
+setVelocity(const float x, const float y)
+integrate(float duration)
+setAcceleration(const Vector2 &acceleration)
+setAcceleration(const float x, const float y)
+getAcceleration(): Vector2
ParticleContactResolver
#iterations: unsigned
#iterationsUsed: unsigned
+ParticleContactResolver(unsigned iterations)
+setIterations(unsigned iterations)
+resolveContacts(ParticleContact *contactArray, int numContacts, float duration)
Dr Janusz Kulon, USW

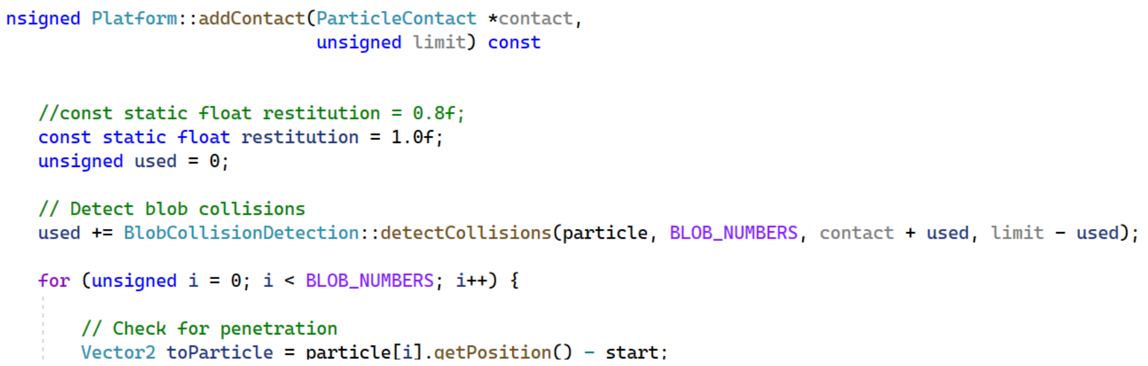
Here I added a class named blobcollisiondetection that will resolve the collision of two particles and resolve it.

Screenshort Of Collision detection and resolution code



```cpp
{
    const static float restitution = 1.0f;
    unsigned used = 0;
    float Euclidean_distance;
    float TotalRedious;
    float penetration;

    for (unsigned iteration_i = 0; iteration_i < numParticles; iteration_i++)
    {
        for (unsigned iteration_j = iteration_i + 1; iteration_j < numParticles; iteration_j++)
        {
            Euclidean_distance = (particles[iteration_i].getPosition() - particles[iteration_j].getPosition()).magnitude();

            TotalRedious = particles[iteration_i].getRadius() + particles[iteration_j].getRadius();

            // Checking if there any collision happend
            if (Euclidean_distance < TotalRedious){
                // Calculating how much the particles overlap with each other
                penetration = TotalRedious - Euclidean_distance;

                // Initialized a new contacts is contactArray has enough space
                contactArray->particle[0] = particles + iteration_i;
                contactArray->particle[1] = particles + iteration_j;
                contactArray->contactNormal = (particles[iteration_i].getPosition() - particles[iteration_j].getPosition()).unit();// Calculate the direction along w
                contactArray->restitution = restitution;
                contactArray->penetration = penetration;
                contactArray++; used++;

               // Checking the limit of contact array
                if (used < limit){return used;}
            }
        }
    }

    return used;
}
```

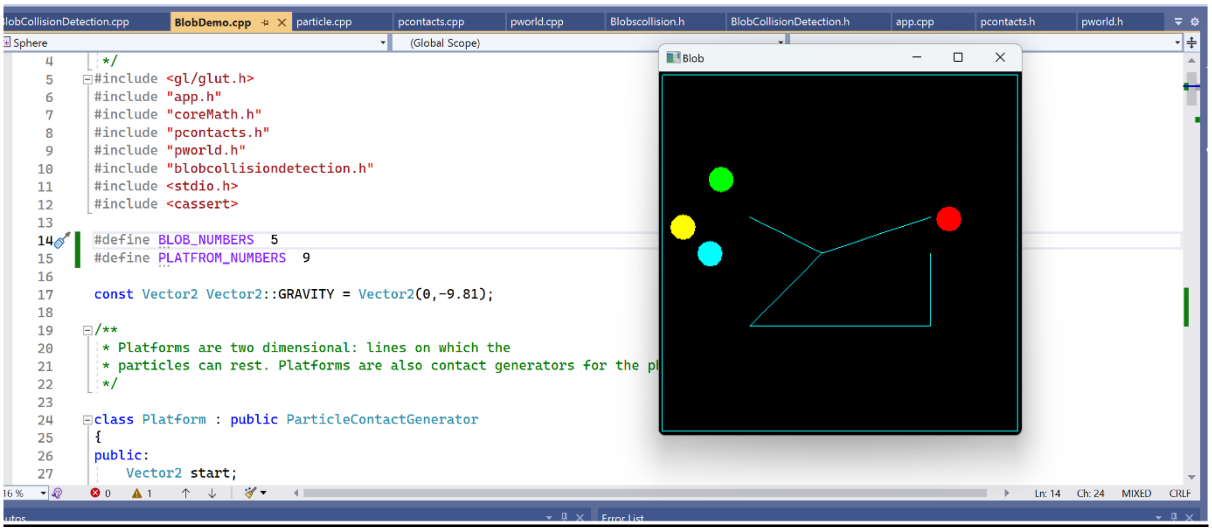
Here in this code, I declare all the necessary variables for our nested for loop. Here, a loop is used to iterate over all the particles once and compare one particle with the rest of the particles. First, we find out the Euclidean distance of two particles, then find the total distance of those particles. Next, check if the distance is less than the total red, because if the distance is less than the total red, then there will be a collision. Then we find out how much collision happens with two particles and store those two collided particles into a contact array for resolving their collision. Next, increase the used value and contactArray index value and check if the used index of the contact array is less than the contact array limit, then return the used value; if not, further checking will continue. At least we return the used value.



```cpp
nsigned Platform::addContact(ParticleContact *contact,
                              unsigned limit) const


    //const static float restitution = 0.8f;
    const static float restitution = 1.0f;
    unsigned used = 0;

    // Detect blob collisions
    used += BlobCollisionDetection::detectCollisions(particle, BLOB_NUMBERS, contact + used, limit - used);

    for (unsigned i = 0; i < BLOB_NUMBERS; i++) {

        // Check for penetration
        Vector2 toParticle = particle[i].getPosition() - start;
```

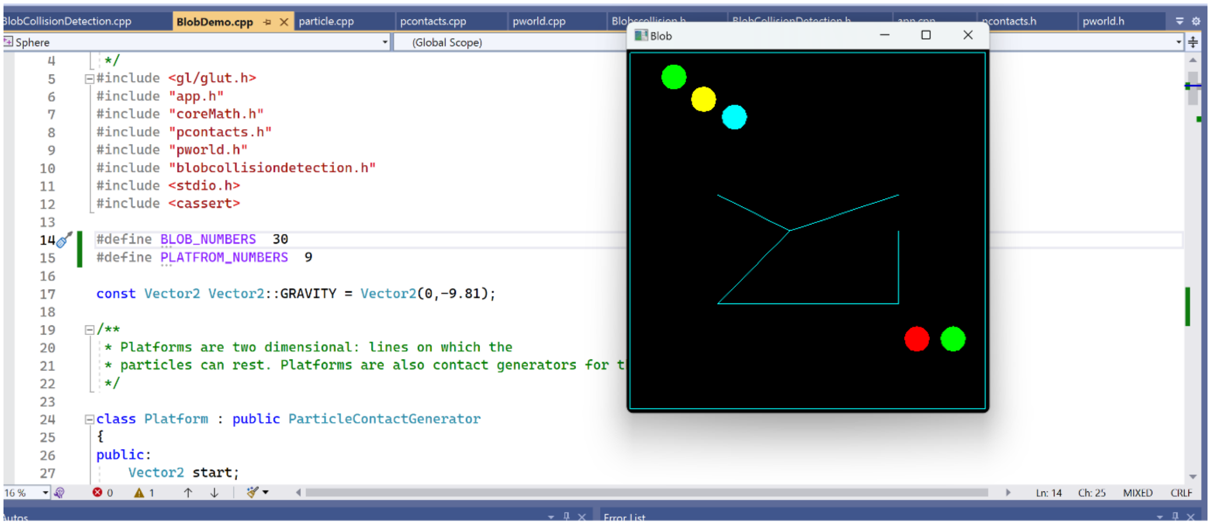
Also in blobDemo class, we need to add the used values with the return value so that it may keep track of the contactArray limit.

Testing and avalution



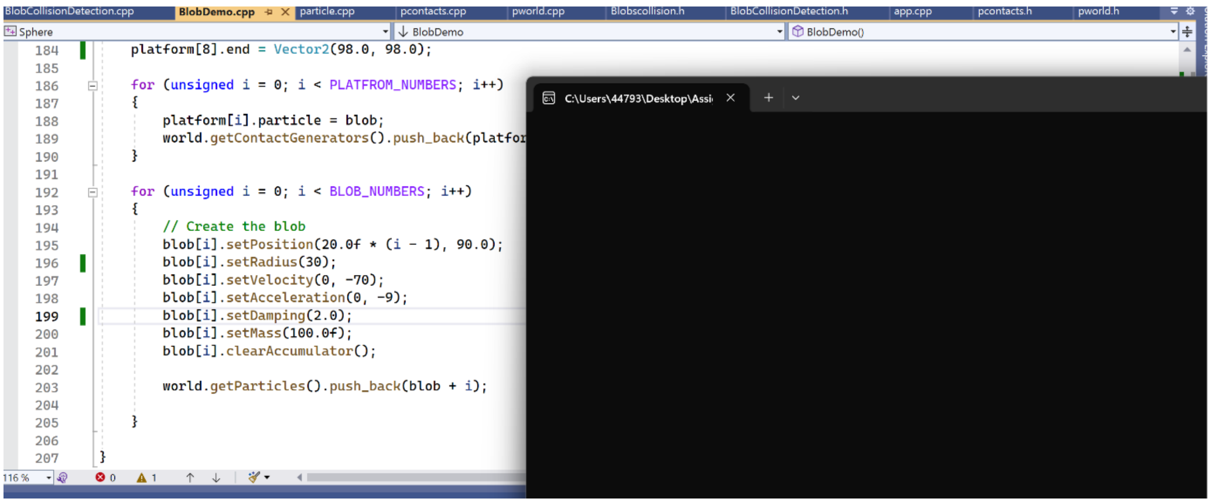
```
 */
#include <gl/glut.h>
#include "app.h"
#include "coreMath.h"
#include "pcontacts.h"
#include "pworld.h"
#include "blobcollisiondetection.h"
#include <stdio.h>
#include <cassert>

#define BLOB_NUMBERS  5
#define PLATFROM_NUMBERS  9

const Vector2 Vector2::GRAVITY = Vector2(0,-9.81);

/**
 * Platforms are two dimensional: lines on which the
 * particles can rest. Platforms are also contact generators for the pl
 */

class Platform : public ParticleContactGenerator
{
public:
    Vector2 start;
```

For generating 5 ball work fine



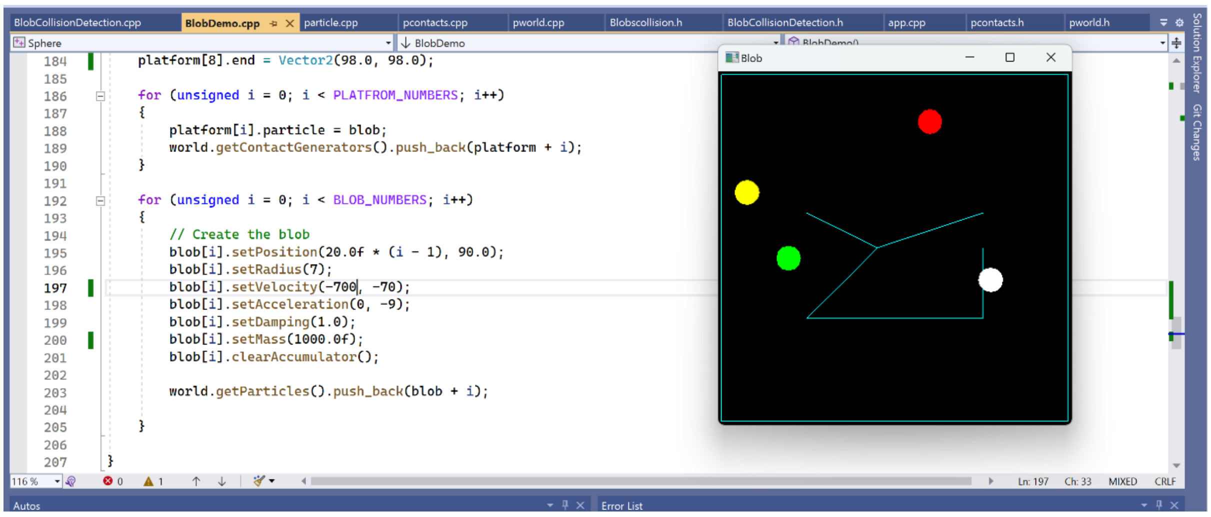
```
 */
#include <gl/glut.h>
#include "app.h"
#include "coreMath.h"
#include "pcontacts.h"
#include "pworld.h"
#include "blobcollisiondetection.h"
#include <stdio.h>
#include <cassert>

#define BLOB_NUMBERS  30
#define PLATFROM_NUMBERS  9

const Vector2 Vector2::GRAVITY = Vector2(0,-9.81);

/**
 * Platforms are two dimensional: lines on which the
 * particles can rest. Platforms are also contact generators for t
 */

class Platform : public ParticleContactGenerator
{
public:
    Vector2 start;
```

Generate 30 blob but show only few. Because some ball generating position out of the box.



```
    platform[8].end = Vector2(98.0, 98.0);

    for (unsigned i = 0; i < PLATFROM_NUMBERS; i++)
    {
        platform[i].particle = blob;
        world.getContactGenerators().push_back(platfor
    }

    for (unsigned i = 0; i < BLOB_NUMBERS; i++)
    {
        // Create the blob
        blob[i].setPosition(20.0f * (i - 1), 90.0);
        blob[i].setRadius(30);
        blob[i].setVelocity(0, -70);
        blob[i].setAcceleration(0, -9);
        blob[i].setDamping(2.0);
        blob[i].setMass(100.0f);
        blob[i].clearAccumulator();

        world.getParticles().push_back(blob + i);

    }
}
```

Changing radius to 30 and damping 2, program crushed.



```cpp
    platform[8].end = Vector2(98.0, 98.0);

    for (unsigned i = 0; i < PLATFROM_NUMBERS; i++)
    {
        platform[i].particle = blob;
        world.getContactGenerators().push_back(platform + i);
    }

    for (unsigned i = 0; i < BLOB_NUMBERS; i++)
    {
        // Create the blob
        blob[i].setPosition(20.0f * (i - 1), 90.0);
        blob[i].setRadius(7);
        blob[i].setVelocity(-700, -70);
        blob[i].setAcceleration(0, -9);
        blob[i].setDamping(1.0);
        blob[i].setMass(1000.0f);
        blob[i].clearAccumulator();

        world.getParticles().push_back(blob + i);

    }
}
```

Set velocity -700,-70 and mass 1000, Blob move firstly inside box. Few minute later some blob go out of box.

Conclusion

With this brute-force algorithm, for a few particles, our collision detection worked fine. But if the particle number grows, then this algorithm doesn't perform well, and sometimes the programme gets crushed. Also, this algorithm is more time-consuming. So for certain number of particles this algorithm this algorithm worked well but for large number of particles and their properties changes it didn't perform very well. But we can say that this algorithm do resolve the collision of particles.

References

Kulon's, D. J., 2023. *Object Oriendted Programming with Data Structure ,* Treforest, Pontypprid: University of south wales.

Kulon's, D. J., 2023. *Object Oriendted Programming with Data Structure, Week Four class lecture, P.27,* Treforest, Pntypprid: University of south wales .

Zhang, X. & Kim, Y. J., 2007. Interactive Collision Detection for Deformable Models Using Streaming AABBs. *IEEE,* Issue 2.