# Collision detection and response algorithms

## Introduction

Collision detection algorithms seek to calculate when, where and how moving objects come into contact with each other. Collision response algorithms seek to resolve these detected collisions according to the laws of physics. Ericson (2005) identifies and discusses a variety of fields where collision detection and response algorithms are utilised, including computer games, physical / engineering simulations and robotics.

Collision detection and response algorithms are used in computer games in order to maintain the illusion that the world is composed of objects which interact with each other according to the laws of physics. Without collision detection, it would not be possible to build realistic environments, since objects would freely pass through each other whenever they come into contact.

In a similar manner, collision detection and response algorithms are used in simulations of physical events, or simulations in engineering. Here, they are used to model interactions between objects with greater realism than in computer games. For example, they might be used to simulate the deformation of a car upon crashing into a solid object such as a wall. In robotics, collision detection and resolution are used for finding suitable paths for the robot to take, and for predicting and avoiding real-world collisions.

These different fields tend to have different performance requirements for a collision detection and response algorithm (Ericson, 2005). In the case of computer games, performance takes priority over precision since collisions must be resolved in real time. So long as performance is adequate, and there are no noticeable visual flaws in the implementation, this is sufficient. Physical / engineering simulations are not calculated in real time, and are more likely to require completely realistic results. As such, precision would be more important than performance.

In robotics, both performance and precision are likely to be important. It may be possible to pre-calculate potential collisions where a robot’s field of movement is restricted i.e. on a factory line. However, if the robot must dynamically calculate a path, then it will need to predict and resolve collisions in real time.

There are two main types of collision detection. These are discrete (a posteriori) and continuous (a priori) collision detection. In discrete collision detection, the simulation is updated step by step, with collision detection taking place once per step. Due to this, it is possible for inter-penetration to occur depending on the size of the particles and their speed of movement. This requires that inter-penetration be resolved before attempting to resolve a collision in the normal way. Our simulation is based on discrete collision detection.

The other type of collision detection is continuous, where future collisions are calculated based on particles’ current velocity etc. In continuous collision detection, inter-penetration does not occur. It is more accurate and reliable than discrete collision detection, but also more complex to implement.

Collision detection and resolution is a common problem, and there are many different methods of implementing it, each with different advantages and disadvantages. One such method is the separating axis theorem, developed by Gottschalk et al. (1996). It is an efficient method of collision detection between convex polygons. The method involves the construction of a hierarchy of oriented bounding boxes (OBB) to encapsulate polygons and groups of polygons. An oriented bounding box is a box that fits and contains a convex polygon. The edges of the bounding box are calculated based on the furthest extremes of the polygon it contains. An OBB differs from an axis-aligned bounding box in that it can be rotated so that it better fits its contents. This gives it more flexibility.

Once a hierarchy of OBBs has been established, the separating axis theorem can be applied. This theorem states that if it is possible to draw a line between two objects (OBBs, in this case), then those objects are not in contact. Gottschalk et al. (1996) apply the separating axis theorem to test whether OBBs have collided. They project the two OBBs in question onto an arbitrary axis and calculate the half-widths of the two OBBs relative to this axis. If the first half-width plus the second half-width is less than or equal to the distance between the OBB centre positions along this axis, then they may be colliding, and it is necessary to run more tests along different axes in order to verify if the OBBs are in contact. Otherwise, it is guaranteed that the OBBs are not in contact.

In general, the separating axis theorem is fast. It tends to be efficient when the polygons it is testing are not in contact. It is also accurate. However, the number of tests required by SAT depends on the specific geometry of the convex polygons. Complex polygons with many edges may require many tests before it can be confirmed whether they are in contact.

Polygons that are in contact will also necessitate more tests, since SAT must check every one of the polygon’s normals in order to confirm that there is in fact a collision. The performance of SAT deteriorates if many complex polygons are colliding. Also, it only works with convex polygons, not concave polygons. Another issue is that it cannot tell which edges of two polygons are touching, only how far they overlap, and in what direction. This can make it unsuitable for more sophisticated forms of collision resolution that require this information.

An alternative method of collision detection is the Minkowski difference / GJK algorithm, developed by Gilbert, Johnson and Keerthi (1988). The GJK algorithm is an optimisation of the basic Minkowski difference method of collision detection. With the basic Minkowski difference method, a new polygon is created by subtracting all of the vertices of one object from all of the vertices of the other. The values on the furthest extremes of the Minkowski difference polygon become the boundaries of the Minkowski difference. If this Minkowski difference polygon contains the origin, then this means that the two original objects have collided. Otherwise, the objects are not in contact. This is the method used in this simulation.

Repeatedly recalculating the Minkowski difference and checking whether the resulting polygon contains the origin is a costly operation, so the GJK algorithm seeks to avoid explicit calculation of every vertex of the Minkowski difference. Instead, it seeks to construct a triangle within the Minkowski difference region that contains the origin and has two of its edges on the borders of the Minkowski difference. This triangle is referred to as the simplex. Constructing this simplex is only possible with convex shapes if these shapes intersect.

In order to calculate the simplex, a support function is used which can determine the furthest point of a polygon in a given direction. Using this, the furthest point in a given direction is determined for the first shape, and the furthest point in the reverse direction is determined for the second shape. When one point is subtracted from the other, this will give a point on the border of the Minkowski difference. This process is carried out three times, each time with a direction that is at right angles to the previous directions. The directions used will be ones that set the simplex as close to the origin as possible. Once the simplex has been calculated, it is determined whether or not it contains the origin. This is done by checking if the point added last was past the origin in that direction. If it is, then the Minkowski difference contains the origin and therefore there is a collision.

One benefit of the GJK method over SAT is its speed, since it avoids explicit calculation of the Minkowski difference and requires only a limited number of operations in all cases. Another benefit is that it does not require much information about the shape. It only needs a set of vertices to run – it does not need to know about the edges of the shape. In addition, it does not depend on the use of oriented bounding boxes like SAT does. This means that it can produce more accurate results.

However, a disadvantage of GJK is that it is harder to determine the level of inter-penetration than it is for SAT. To determine the amount of inter-penetration, it is necessary to find the closest point on the border of the Minkowski difference to the origin, which is tricky. Like SAT, GJK cannot handle concave shapes without these shapes first being broken down into convex shapes.

## Code listing

### coreMath.h

#include <math.h>

/\*\*

\* @file

\*

\* The core contains utility functions, helpers and a basic set of

\* mathematical types.

\*/

#ifndef CORE\_MATH

#define CORE\_MATH

class Vector2

{

public:

/\*\* Holds the value along the x axis. \*/

float x;

/\*\* Holds the value along the y axis. \*/

float y;

public:

/\*\* The default constructor creates a zero vector. \*/

Vector2() : x(0), y(0) {}

/\*\*

\* The explicit constructor creates a vector with the given

\* components.

\*/

Vector2(const float x, const float y)

: x(x), y(y) {}

const static Vector2 GRAVITY;

const static Vector2 UP;

float operator[](unsigned i) const

{

if (i == 0) return x;

return y;

}

float& operator[](unsigned i)

{

if (i == 0) return x;

return y;

}

/\*\* Adds the given vector to this. \*/

void operator+=(const Vector2& v)

{

x += v.x;

y += v.y;

}

/\*\*

\* Returns the value of the given vector added to this.

\*/

Vector2 operator+(const Vector2& v) const

{

return Vector2(x + v.x, y + v.y);

}

/\*\* Subtracts the given vector from this. \*/

void operator-=(const Vector2& v)

{

x -= v.x;

y -= v.y;

}

/\*\*

\* Returns the value of the given vector subtracted from this.

\*/

Vector2 operator-(const Vector2& v) const

{

return Vector2(x - v.x, y - v.y);

}

/\*\* Multiplies this vector by the given scalar. \*/

void operator\*=(const float value)

{

x \*= value;

y \*= value;

}

/\*\* Returns a copy of this vector scaled the given value. \*/

Vector2 operator\*(const float value) const

{

return Vector2(x\*value, y\*value);

}

/\*

\* Multiplies each component of the vector by the inverse of the value

\* and returns the result

\*/

Vector2 operator/(const float value) const

{

return Vector2(x\*(1.0f / value), y\*(1.0f / value));

}

/\*\*

\* Calculates and returns a component-wise product of this

\* vector with the given vector.

\*/

Vector2 componentProduct(const Vector2 &vector) const

{

return Vector2(x \* vector.x, y \* vector.y);

}

/\*\*

\* Performs a component-wise product with the given vector and

\* sets this vector to its result.

\*/

void componentProductUpdate(const Vector2 &vector)

{

x \*= vector.x;

y \*= vector.y;

}

/\*\*

\* Calculates and returns the scalar product of this vector

\* with the given vector.

\*/

float scalarProduct(const Vector2 &vector) const

{

return x\*vector.x + y\*vector.y;

}

/\*\*

\* Calculates and returns the scalar product of this vector

\* with the given vector.

\*/

float operator \*(const Vector2 &vector) const

{

return x\*vector.x + y\*vector.y;

}

/\*\*

\* Adds the given vector to this, scaled by the given amount.

\*/

void addScaledVector(const Vector2& vector, float scale)

{

x += vector.x \* scale;

y += vector.y \* scale;

}

/\*\* Gets the magnitude of this vector. \*/

float magnitude() const

{

return sqrt(x\*x + y\*y);

}

/\*\* Gets the squared magnitude of this vector. \*/

float squareMagnitude() const

{

return x\*x + y\*y;

}

/\*\* Limits the size of the vector to the given maximum. \*/

void trim(float size)

{

if (squareMagnitude() > size\*size)

{

normalise();

x \*= size;

y \*= size;

}

}

/\*\* Turns a non-zero vector into a vector of unit length. \*/

void normalise()

{

float l = magnitude();

if (l > 0)

{

(\*this) \*= ((float)1) / l;

}

}

/\*\* Returns the normalised version of a vector. \*/

Vector2 unit() const

{

Vector2 result = \*this;

result.normalise();

return result;

}

/\*\* Checks if the two vectors have identical components. \*/

bool operator==(const Vector2& other) const

{

return x == other.x &&

y == other.y;

}

/\*\* Checks if the two vectors have non-identical components. \*/

bool operator!=(const Vector2& other) const

{

return !(\*this == other);

}

/\*\*

\* Checks if this vector is component-by-component less than

\* the other.

\*

\* @note This does not behave like a single-value comparison:

\* !(a < b) does not imply (b >= a).

\*/

bool operator<(const Vector2& other) const

{

return x < other.x && y < other.y;

}

/\*\*

\* Checks if this vector is component-by-component less than

\* the other.

\*

\* @note This does not behave like a single-value comparison:

\* !(a < b) does not imply (b >= a).

\*/

bool operator>(const Vector2& other) const

{

return x > other.x && y > other.y;

}

/\*\*

\* Checks if this vector is component-by-component less than

\* the other.

\*

\* @note This does not behave like a single-value comparison:

\* !(a <= b) does not imply (b > a).

\*/

bool operator<=(const Vector2& other) const

{

return x <= other.x && y <= other.y;

}

/\*\*

\* Checks if this vector is component-by-component less than

\* the other.

\*

\* @note This does not behave like a single-value comparison:

\* !(a <= b) does not imply (b > a).

\*/

bool operator>=(const Vector2& other) const

{

return x >= other.x && y >= other.y;

}

/\*\* Zero all the components of the vector. \*/

void clear()

{

x = y = 0;

}

/\*\* Flips all the components of the vector. \*/

void invert()

{

x = -x;

y = -y;

}

};

#endif // CORE\_H

### app.h

class Application

{

protected:

int height;

int width;

float nRange;

float timeinterval;

public:

virtual void initGraphics();

virtual void display();

virtual void update();

virtual void resize(int width, int height);

int getheight();

int getwidth();

float getTimeinterval();

void setTimeinterval(float timeint);

};

### app.cpp

#include <gl/glut.h>

#include "app.h"

int Application::getwidth()

{

return width;

}

int Application::getheight()

{

return height;

}

float Application::getTimeinterval()

{

return timeinterval;

}

void Application::setTimeinterval(float timeinterval)

{

Application::timeinterval = timeinterval;

}

void Application::initGraphics()

{

glClearColor(0.0f, 0.0f, 0.0f, 1.0f);

}

void Application::display()

{

glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT);

glMatrixMode(GL\_MODELVIEW);

glLoadIdentity();

}

void Application::update()

{

glutPostRedisplay();

}

void Application::resize(int width, int height)

{

//nRange = 100.0f;

GLfloat aspectRatio = (GLfloat)width / (GLfloat)height;

// Prevent a divide by zero

if (height == 0) height = 1;

// Set Viewport to window dimensions

glViewport(0, 0, width, height);

// Reset coordinate system

glMatrixMode(GL\_PROJECTION);

glLoadIdentity();

//Establish clipping volume (left, right, bottom, top, near, far)

if (width <= height)

{

Application::width = nRange;

Application::height = nRange / aspectRatio;

glOrtho(-nRange, nRange, -nRange / aspectRatio, nRange / aspectRatio, -nRange\*2.0f, nRange\*2.0f);

}

else

{

Application::width = nRange\*aspectRatio;

Application::height = nRange;

glOrtho(-nRange\*aspectRatio, nRange\*aspectRatio, -nRange, nRange, -nRange\*2.0f, nRange\*2.0f);

}

// Reset the modelview matrix

glMatrixMode(GL\_MODELVIEW);

glLoadIdentity();

}

### particle.h

/\*\*

\* A particle is the simplest object that can be simulated in the

\* physics system.

\*/

#ifndef PARTICLE\_H

#define PARTICLE\_H

#include "coreMath.h"

#include <vector>

class Particle

{

protected:

float inverseMass;

float radius;

//True if particle is a circle, false if it is a convex polygon

bool circle = true;

//Stores vertices, width and height of the particle (unless it is a circle, in which case no vertices are stored)

std::vector<Vector2> vertices;

float width;

float height;

Vector2 position;

Vector2 velocity;

Vector2 forceAccum;

Vector2 acceleration;

public:

void integrate(float duration);

void setMass(const float mass);

float getMass() const;

void setInverseMass(const float inverseMass);

float getInverseMass() const;

bool hasFiniteMass() const;

//Allows vertices of shape to be set and retrieved.

//Particle defaults to circle shape, but the user can set vertices to create a convex polygon instead

void setVertices(std::vector<Vector2>& vertices);

std::vector<Vector2>& getVertices();

//Allows shape's width and height to be set and retrieved

void setWidthAndHeight(float w, float h);

float getWidth();

float getHeight();

//Tells caller whether this particle is a circle or not

bool isCircle();

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

void setPosition(const Vector2 &position);

Vector2 getPosition() const;

void getPosition(Vector2 \*position) const;

void setRadius(const float r);

float getRadius() const;

void setVelocity(const Vector2 &velocity);

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

Vector2 getVelocity() const;

void getVelocity(Vector2 \*velocity) const;

void setAcceleration(const Vector2 &acceleration);

void setAcceleration(const float x, const float y);

Vector2 getAcceleration() const;

void clearAccumulator();

void addForce(const Vector2 &force);

};

#endif

### particle.cpp

#include "particle.h"

#include <math.h>

#include <assert.h>

#include <float.h>

void Particle::integrate(float duration)

{

// We don't integrate things with zero mass.

if (inverseMass <= 0.0f)

return;

assert(duration > 0.0);

position.addScaledVector(velocity, duration);

// Work out the acceleration from the force

Vector2 resultingAcc = acceleration;

resultingAcc.addScaledVector(forceAccum, inverseMass);

// Update linear velocity from the acceleration.

velocity.addScaledVector(resultingAcc, duration);

//velocity \*= pow(0.9f, duration);

//velocity \*= 0.95;

// Clear the forces.

clearAccumulator();

}

//Set particle to be a different shape (instead of the default circle shape).

//Takes a reference to a vector containing the desired vertices

void Particle::setVertices(std::vector<Vector2>& vertices)

{

circle = false;

Particle::vertices.resize(vertices.size());

for (int i = 0; i < vertices.size(); i++)

{

Particle::vertices[i].x = vertices[i].x;

Particle::vertices[i].y = vertices[i].y;

}

}

//Returns the vertices of the shape

//If shape is a circle, an empty vector will be returned

std::vector<Vector2>& Particle::getVertices()

{

if (!circle)

return vertices;

else

return std::vector<Vector2>();

}

void Particle::setWidthAndHeight(float w, float h)

{

width = w;

height = h;

}

float Particle::getWidth()

{

return width;

}

float Particle::getHeight()

{

return height;

}

//Tells the caller if the shape is a circle or not

bool Particle::isCircle()

{

return circle;

}

void Particle::setMass(const float mass)

{

assert(mass != 0);

Particle::inverseMass = ((float)1.0) / mass;

}

float Particle::getMass() const

{

if (inverseMass == 0) {

return DBL\_MAX;

}

else {

return ((float)1.0) / inverseMass;

}

}

void Particle::setInverseMass(const float inverseMass)

{

Particle::inverseMass = inverseMass;

}

float Particle::getInverseMass() const

{

return inverseMass;

}

bool Particle::hasFiniteMass() const

{

return inverseMass >= 0.0f;

}

void Particle::setPosition(const float x, const float y)

{

position.x = x;

position.y = y;

}

void Particle::setPosition(const Vector2 &position)

{

Particle::position = position;

}

Vector2 Particle::getPosition() const

{

return position;

}

void Particle::getPosition(Vector2 \*position) const

{

\*position = Particle::position;

}

void Particle::setRadius(const float r)

{

radius = r;

}

float Particle::getRadius() const

{

return radius;

}

void Particle::setVelocity(const float x, const float y)

{

velocity.x = x;

velocity.y = y;

}

void Particle::setVelocity(const Vector2 &velocity)

{

Particle::velocity = velocity;

}

Vector2 Particle::getVelocity() const

{

return velocity;

}

void Particle::getVelocity(Vector2 \*velocity) const

{

\*velocity = Particle::velocity;

}

void Particle::setAcceleration(const Vector2 &acceleration)

{

Particle::acceleration = acceleration;

}

void Particle::setAcceleration(const float x, const float y)

{

acceleration.x = x;

acceleration.y = y;

}

Vector2 Particle::getAcceleration() const

{

return acceleration;

}

void Particle::clearAccumulator()

{

forceAccum.clear();

}

void Particle::addForce(const Vector2 &force)

{

forceAccum += force;

}

### ParticleCollision.h

#pragma once

#include "pcontacts.h"

#include "particle.h"

#include <vector>

using namespace std;

class ParticleCollision : public ParticleContactGenerator

{

private:

//Total number of particles in array

const int NUM\_PARTICLES;

//Pointer to the first element of the particle array (saves passing in an entire array to ParticleCollision object)

Particle\* particles;

//Default restitution value

float restitution = 0.9;

//Far borders of the convex polygon created by the Minkowski difference.

//Recalculated for every collision where at least one particle is not a circle.

vector<Vector2> MDVertices;

public:

//When instantiating particle collision object, tell it how many other particles there are to collide with

//and give it a pointer to first particle in the array.

ParticleCollision(int numParticles, Particle\* arrayPtr);

void setRestitution(float restitution) { this->restitution = restitution; }

//Add all of the particle's current contact data to the relevant ParticleContact objects

unsigned addContact(ParticleContact \*contact, unsigned limit);

//Determine if one particle and another are touching

bool checkCollision(Particle& particle1, Particle& particle2, float distance);

//Calculate vertices for Minkowski difference of two particles and return it as a vector of Vector2 objects

vector<Vector2> calcMinkowskiDifferenceVertices(Particle& particle1, Particle& particle2);

//Calculates 8 of a circle particle's vertices (this gives an imperfect representation of the shape, but

//is close enough for our purposes)

vector<Vector2> estimateCircleVertices(Particle& particle) const;

//Determines whether the origin lies within the bounds of the calculated Minkowski difference

bool polygonContainsOrigin(vector<Vector2>& vertices);

};

### ParticleCollision.cpp

#include "pcontacts.h"

#include "ParticleCollision.h"

#include <gl/glut.h>

using namespace std;

ParticleCollision::ParticleCollision(int numParticles, Particle\* arrayPtr) : NUM\_PARTICLES(numParticles)

{

particles = arrayPtr;

}

unsigned ParticleCollision::addContact(ParticleContact \*contact, unsigned limit)

{

//const static float restitution = 1.0f;

unsigned used = 0;

for (int i = 0; i < NUM\_PARTICLES; i++)

{

Vector2 pos1 = particles[i].getPosition();

float radius1 = particles[i].getRadius();

Vector2 velocity1 = particles[i].getVelocity();

float mass1 = particles[i].getMass();

for (int j = 0; j < NUM\_PARTICLES; j++)

{

//Particle cannot collide with itself

if (i == j)

continue;

Vector2 pos2 = (particles[j]).getPosition();

Vector2 velocity2 = (particles[j]).getVelocity();

float radius2 = (particles[j]).getRadius();

float mass2 = (particles[j]).getMass();

//Distance from circle 2 to circle 1

Vector2 circleDistanceVec = pos1 - pos2;

float distance = circleDistanceVec.magnitude();

if (checkCollision(particles[i], particles[j], distance))

{

// We have a collision

contact->contactNormal = circleDistanceVec.unit();

contact->restitution = restitution;

contact->particle[0] = &particles[i];

contact->particle[1] = &particles[j];

if (particles[i].isCircle() && particles[j].isCircle())

contact->penetration = (radius1 + radius2) - distance;

else if (!particles[i].isCircle() || !particles[j].isCircle())

{

Vector2 closestPoint = MDVertices[0];

float interPenetrationDist = 100;

//Find closest point of Minkowski difference to origin and calculate inter-penetration from it

for (int i = 1; i < MDVertices.size(); i++)

{

float temp = (Vector2(0, 0) - MDVertices[i]).magnitude();

if (temp < interPenetrationDist)

interPenetrationDist = temp;

}

contact->penetration = interPenetrationDist;

}

used++;

contact++;

}

}

}

return used;

}

bool ParticleCollision::checkCollision(Particle& particle1, Particle& particle2, float distance)

{

//Simple check for collision between two circles

if (particle1.isCircle() && particle2.isCircle())

{

//If length of circleDistanceVec <= (radius1 + radius2), then circles have collided or inter-penetrated

if (distance <= particle1.getRadius() + particle2.getRadius())

return true;

else

return false;

}

//One of the shapes is a convex polygon, so check for collision using Minkowski difference

else

{

//Calculate vertices of Minkowski difference between two particles

MDVertices = calcMinkowskiDifferenceVertices(particle1, particle2);

//Check if Minkowski difference contains the origin or not

bool collision = polygonContainsOrigin(MDVertices);

return collision;

}

}

/\*

Code used in this function to detect whether the Minkowski difference contains origin is based on

code written by W Randolph Franklin. See https://wrf.ecse.rpi.edu//Research/Short\_Notes/pnpoly.html.

Method is based on projecting a ray through the polygon from the origin and working out

whether it is inside or outside the polygon by the number of edges it intersects.

\*/

bool ParticleCollision::polygonContainsOrigin(vector<Vector2>& vertices)

{

bool collision = false;

for (int i = 0, j = vertices.size() - 1; i < vertices.size(); j = i++)

{

if (((vertices[i].y>0) != (vertices[j].y>0)) &&

(0 < (vertices[j].x - vertices[i].x) \* (0 - vertices[i].y) / (vertices[j].y - vertices[i].y) + vertices[i].x))

collision = !collision;

}

return collision;

}

vector<Vector2> ParticleCollision::calcMinkowskiDifferenceVertices(Particle& particle1, Particle& particle2)

{

vector<Vector2> particle1Vertices;

vector<Vector2> particle2Vertices;

vector<Vector2> minkowskiDiffVertices;

//Populate particle1Vertices vector. If particle1 == circle, estimate vertices. Else, retrieve particle1's stored vertices

if (particle1.isCircle())

particle1Vertices = estimateCircleVertices(particle1);

else

particle1Vertices = particle1.getVertices();

//Populate particle2Vertices vector. If particle2 == circle, estimate vertices. Else, retrieve particle2's stored vertices

if (particle2.isCircle())

particle2Vertices = estimateCircleVertices(particle2);

else

particle2Vertices = particle2.getVertices();

//Populate vector of Minkowski difference vertices with particle 1 vertices - particle 2 vertices

for (int i = 0; i < particle1Vertices.size(); i++)

for (int j = 0; j < particle2Vertices.size(); j++)

minkowskiDiffVertices.push\_back((particle1Vertices[i] + particle1.getPosition()) - (particle2Vertices[j] + particle2.getPosition()));

return minkowskiDiffVertices;

}

//calculate 8 of the vertices of a circle particle (cannot calculate all, since a circle has infinite vertices)

vector<Vector2> ParticleCollision::estimateCircleVertices(Particle& particle) const

{

vector<Vector2> vertices;

float radius = particle.getRadius();

vertices.push\_back(Vector2(-radius, 0));

vertices.push\_back(Vector2(radius, 0));

vertices.push\_back(Vector2(0, -radius));

vertices.push\_back(Vector2(0, -radius));

vertices.push\_back(Vector2(1, 1).unit() \* radius);

vertices.push\_back(Vector2(-1, 1).unit() \* radius);

vertices.push\_back(Vector2(-1, -1).unit() \* radius);

vertices.push\_back(Vector2(1, -1).unit() \* radius);

return vertices;

}

### pcontacts.h

/\*

\* Interface file for the contact resolution system for particles.

\*

\*/

#ifndef PCONTACTS\_H

#define PCONTACTS\_H

#include "particle.h"

class ParticleContactResolver;

/\*\*

\* A Contact represents two objects in contact (in this case

\* ParticleContact representing two Particles).

\*/

class ParticleContact

{

/\*\*

\* The contact resolver object needs access into the contacts to

\* set and effect the contact.

\*/

friend ParticleContactResolver;

public:

/\*\*

\* Holds the particles that are involved in the contact. The

\* second of these can be NULL, for contacts with the scenery.

\*/

Particle\* particle[2];

/\*\*

\* Holds the normal restitution coefficient at the contact.

\*/

float restitution;

/\*\*

\* Holds the direction of the contact in world coordinates.

\*/

Vector2 contactNormal;

/\*\*

\* Holds the depth of penetration at the contact.

\*/

float penetration;

protected:

/\*\*

\* Resolves this contact, for both velocity and interpenetration.

\*/

void resolve(float duration);

/\*\*

\* Calculates the separating velocity at this contact.

\*/

float calculateSeparatingVelocity() const;

private:

/\*\*

\* Handles the impulse calculations for this collision.

\*/

void resolveVelocity(float duration);

};

/\*\*

\* The contact resolution routine for particle contacts. One

\* resolver instance can be shared for the whole simulation.

\*/

class ParticleContactResolver

{

protected:

/\*\*

\* Holds the number of iterations allowed.

\*/

unsigned iterations;

/\*\*

\* This is a performance tracking value - we keep a record

\* of the actual number of iterations used.

\*/

unsigned iterationsUsed;

public:

/\*\*

\* Creates a new contact resolver.

\*/

ParticleContactResolver(unsigned iterations);

/\*\*

\* Sets the number of iterations that can be used.

\*/

void setIterations(unsigned iterations);

/\*\*

\* Resolves a set of particle contacts for both penetration

\* and velocity.

\*

\*/

void resolveContacts(ParticleContact \*contactArray,

unsigned numContacts,

float duration);

};

/\*\*

\* This is the basic polymorphic interface for contact generators

\* applying to particles.

\*/

class ParticleContactGenerator

{

public:

/\*\*

\* Fills the given contact structure with the generated

\* contact.

\*/

virtual unsigned addContact(ParticleContact \*contact,

unsigned limit) = 0;

};

#endif // CONTACTS\_H

### pcontacts.cpp

#include <float.h>

#include <pcontacts.h>

// Contact implementation

void ParticleContact::resolve(float duration)

{

resolveVelocity(duration);

}

float ParticleContact::calculateSeparatingVelocity() const

{

Vector2 relativeVelocity = particle[0]->getVelocity();

if (particle[1]) relativeVelocity -= particle[1]->getVelocity();

return relativeVelocity \* contactNormal;

}

void ParticleContact::resolveVelocity(float duration)

{

// Find the velocity in the direction of the contact

float separatingVelocity = calculateSeparatingVelocity();

// Check if it needs to be resolved

if (separatingVelocity > 0.05)

{

// The contact is either separating, or stationary - there's

// no impulse required.

return;

}

// Calculate the new separating velocity

float newSepVelocity = -separatingVelocity \* restitution;

float deltaVelocity = newSepVelocity - separatingVelocity;

// We apply the change in velocity to each object in proportion to

// their inverse mass (i.e. those with lower inverse mass [higher

// actual mass] get less change in velocity)..

float totalInverseMass = particle[0]->getInverseMass();

if (particle[1]) totalInverseMass += particle[1]->getInverseMass();

// If all particles have infinite mass, then impulses have no effect

if (totalInverseMass <= 0) return;

// Calculate the impulse to apply

float impulse = deltaVelocity / totalInverseMass;

// Find the amount of impulse per unit of inverse mass

Vector2 impulsePerIMass = contactNormal \* impulse;

// Apply impulses: they are applied in the direction of the contact,

// and are proportional to the inverse mass.

particle[0]->setVelocity(particle[0]->getVelocity() +

impulsePerIMass \* particle[0]->getInverseMass()

);

if (particle[1])

{

// Particle 1 goes in the opposite direction

particle[1]->setVelocity(particle[1]->getVelocity() +

impulsePerIMass \* -particle[1]->getInverseMass()

);

}

//resolve inter-penetration by moving two circles apart from each other by the distance they have inter-penetrated

if (particle[0] && particle[1])

{

if (penetration > 0)

{

Vector2 interpenetrationVec = particle[0]->getPosition() - particle[1]->getPosition();

interpenetrationVec.normalise();

interpenetrationVec \*= penetration;

particle[0]->setPosition(particle[0]->getPosition() + interpenetrationVec \* 0.5);

particle[1]->setPosition(particle[1]->getPosition() - interpenetrationVec \* 0.5);

}

}

//resolve inter-penetration

}

ParticleContactResolver::ParticleContactResolver(unsigned iterations)

:

iterations(iterations)

{

}

void ParticleContactResolver::setIterations(unsigned iterations)

{

ParticleContactResolver::iterations = iterations;

}

void ParticleContactResolver::resolveContacts(ParticleContact \*contactArray,

unsigned numContacts,

float duration)

{

unsigned i;

iterationsUsed = 0;

while (iterationsUsed < iterations)

{

// Find the contact with the largest closing velocity;

float max = DBL\_MAX;

unsigned maxIndex = numContacts;

for (i = 0; i < numContacts; i++)

{

float sepVel = contactArray[i].calculateSeparatingVelocity();

if (sepVel < max && (sepVel < 0 || contactArray[i].penetration > 0))

{

max = sepVel;

maxIndex = i;

}

}

//Do we have anything worth resolving?

if (maxIndex == numContacts) break;

// Resolve this contact

contactArray[maxIndex].resolve(duration);

iterationsUsed++;

}

}

### pworld.h

/\*

\* Interface file for the particle

\*

\*/

#ifndef PWORLD\_H

#define PWORLD\_H

#include <vector>

#include "pcontacts.h"

class ParticleWorld

{

public:

typedef std::vector<Particle\*> Particles;

typedef std::vector<ParticleContactGenerator\*> ContactGenerators;

protected:

/\*\*

\* Holds the particles

\*/

Particles particles;

/\*\*

\* True if the world should calculate the number of iterations

\* to give the contact resolver at each frame.

\*/

bool calculateIterations;

/\*\*

\* Holds the resolver for contacts.

\*/

ParticleContactResolver resolver;

/\*\*

\* Contact generators.

\*/

ContactGenerators platformContactGenerators;

ContactGenerators particleContactGenerator;

/\*\*

\* Holds the list of contacts.

\*/

ParticleContact \*contacts;

/\*\*

\* Holds the maximum number of contacts allowed (i.e. the

\* size of the contacts array).

\*/

unsigned maxContacts;

public:

/\*\*

\* Creates a new particle simulator that can handle up to the

\* given number of contacts per frame.

\*/

ParticleWorld(unsigned maxContacts, unsigned iterations = 0);

/\*\*

\* Deletes the simulator.

\*/

~ParticleWorld();

/\*\*

\* Calls each of the registered contact generators to report

\* their contacts. Returns the number of generated contacts.

\*/

unsigned generateContacts();

/\*\*

\* Integrates all the particles in this world forward in time

\* by the given duration.

\*/

void integrate(float duration);

/\*\*

\* Processes all the physics for the particle world.

\*/

void runPhysics(float duration);

/\*\*

\* Returns the list of particles.

\*/

Particles& getParticles();

/\*\*

\* Returns the list of contact generators.

\*/

ContactGenerators& getPlatformContactGenerators();

ContactGenerators& getParticleContactGenerator();

};

#endif // PWORLD\_H

### pworld.cpp

#include <cstdlib>

#include <pworld.h>

ParticleWorld::ParticleWorld(unsigned maxContacts, unsigned iterations)

:

resolver(iterations),

maxContacts(maxContacts)

{

contacts = new ParticleContact[maxContacts];

calculateIterations = (iterations == 0);

}

ParticleWorld::~ParticleWorld()

{

delete[] contacts;

}

unsigned ParticleWorld::generateContacts()

{

int limit = maxContacts;

ParticleContact \*nextContact = contacts;

//generate contacts for particles

unsigned used = particleContactGenerator[0]->addContact(nextContact, limit);

limit -= used;

nextContact += used;

//If the max number of contacts has been exceeded, do not attempt to generate contacts for platforms,

//and return the number of contacts used (all of them)

if (limit <= 0)

return maxContacts;

//generate contacts for platforms

for (ContactGenerators::iterator g = platformContactGenerators.begin(); g != platformContactGenerators.end(); g++)

{

used = (\*g)->addContact(nextContact, limit);

limit -= used;

nextContact += used;

// We've run out of contacts to fill. This means we're missing

// contacts.

if (limit <= 0)

break;

}

// Return the number of contacts used.

return maxContacts - limit;

}

void ParticleWorld::integrate(float duration)

{

for (Particles::iterator p = particles.begin(); p != particles.end(); p++)

{

// Remove all forces from the accumulator

(\*p)->integrate(duration);

}

}

void ParticleWorld::runPhysics(float duration)

{

//Apply drag force to every particle

for (Particles::iterator p = particles.begin(); p != particles.end(); p++)

{

float radius = (\*p)->getRadius();

Vector2 velocity = (\*p)->getVelocity();

float magnitude = velocity.magnitude();

float k2 = 0.1 \* radius \* radius;

float dragCoeff = k2 \* magnitude \* magnitude;

Vector2 dragForce = velocity;

dragForce \*= -dragCoeff;

dragForce.normalise();

dragForce \*= 50; //normalised drag force is too small to produce much effect, so scale it up

(\*p)->addForce(dragForce);

}

// Then integrate the objects

integrate(duration);

// Generate contacts

unsigned usedContacts = generateContacts();

// And process them

if (usedContacts)

{

if (calculateIterations) resolver.setIterations(usedContacts \* 2);

resolver.resolveContacts(contacts, usedContacts, duration);

}

}

ParticleWorld::Particles& ParticleWorld::getParticles()

{

return particles;

}

ParticleWorld::ContactGenerators& ParticleWorld::getPlatformContactGenerators()

{

return platformContactGenerators;

}

ParticleWorld::ContactGenerators& ParticleWorld::getParticleContactGenerator()

{

return particleContactGenerator;

}

### BlobDemo.cpp

/\*

\* The Blob demo.

\*

\*/

#include <gl/glut.h>

#include "app.h"

#include "coreMath.h"

#include "pcontacts.h"

#include "pworld.h"

#include <stdio.h>

#include <cassert>

#include "ParticleCollision.h"

#include <iostream>

#include <vector>

using namespace std;

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

const int NUM\_CIRCLES = 10; //Number of circles in simulation

const int NUM\_QUADS = 1; //Number of quads in simulation (these will all be placed after circles in the particle array)

const int NUM\_TRIANGLES = 1; //Number of triangles in simulation (these will all be placed after quads in the particle array)

const int NUM\_PARTICLES = NUM\_CIRCLES + NUM\_QUADS + NUM\_TRIANGLES; //Total number of particles of all kinds in simulation

const int NUM\_PLATFORMS = 1;

const int BASE\_CIRCLE\_RADIUS = 5; //Minimum radius of a circle

const int BASE\_CIRCLE\_MASS = 5; //Minimum mass of a circle

/\*\*

\* Platforms are two dimensional: lines on which the

\* particles can rest. Platforms are also contact generators for the physics.

\*/

class Platform : public ParticleContactGenerator

{

public:

Vector2 start;

Vector2 end;

/\*\*

\* Holds a pointer to the particles we're checking for collisions with.

\*/

Particle\* particle[NUM\_PARTICLES];

//default restitution value

float restitution = 0.8;

void setRestitution(float restitution) { this->restitution = restitution; }

virtual unsigned addContact(ParticleContact \*contact, unsigned limit);

};

unsigned Platform::addContact(ParticleContact \*contact, unsigned limit)

{

int used = 0;

for (int i = 0; i < NUM\_PARTICLES; i++)

{

// Check for penetration

Vector2 toParticle = particle[i]->getPosition() - start;

Vector2 lineDirection = end - start;

float projected = toParticle \* lineDirection;

float platformSqLength = lineDirection.squareMagnitude();

float squareRadius = particle[i]->getRadius()\*particle[i]->getRadius();;

//Calculate whether non-circle objects have made contact with platform

if (!particle[i]->isCircle())

{

Vector2 pos = particle[i]->getPosition();

vector<Vector2> vertices = particle[i]->getVertices();

//check if particle has an x-coordinate that allows it to touch the platform

if (pos.x + particle[i]->getWidth() / 2.0f > start.x && pos.x - particle[i]->getWidth() / 2.0f < end.x)

{

float slope = (end.y - start.y) / (end.x - start.x);

float yIntercept = end.y - slope \* end.x;

float platformYVal = slope \* pos.x + yIntercept;

float distanceToPlatform = toParticle.squareMagnitude() - projected\*projected / platformSqLength;

//check if the particle is touching the line

if (pos.y - particle[i]->getHeight() / 2.0f <= platformYVal && pos.y + particle[i]->getHeight() / 2.0f >= platformYVal)

{

// We have a collision

Vector2 closestPoint = start + lineDirection\*(projected / platformSqLength);

contact->contactNormal = (particle[i]->getPosition() - closestPoint).unit();

contact->restitution = restitution;

contact->particle[0] = particle[i];

contact->particle[1] = 0;

contact->penetration = particle[i]->getHeight() \* 0.5f - (pos.y - platformYVal);//particle[i]->getRadius() - sqrt(distanceToPlatform);

used++;

contact++;

}

}

}

else if (projected <= 0)

{

// The blob is nearest to the start point

if (toParticle.squareMagnitude() < squareRadius)

{

// We have a collision

contact->contactNormal = toParticle.unit();

contact->restitution = restitution;

contact->particle[0] = particle[i];

contact->particle[1] = 0;

contact->penetration = particle[i]->getRadius() - toParticle.magnitude();

used++;

contact++;

}

}

else if (projected >= platformSqLength)

{

// The blob is nearest to the end point

toParticle = particle[0]->getPosition() - end;

if (toParticle.squareMagnitude() < squareRadius)

{

// We have a collision

contact->contactNormal = toParticle.unit();

contact->restitution = restitution;

contact->particle[0] = particle[i];

contact->particle[1] = 0;

contact->penetration = particle[i]->getRadius() - toParticle.magnitude();

used++;

contact++;

}

}

else

{

// the blob is nearest to the middle.

float distanceToPlatform = toParticle.squareMagnitude() - projected\*projected / platformSqLength;

if (distanceToPlatform < squareRadius)

{

// We have a collision

Vector2 closestPoint = start + lineDirection\*(projected / platformSqLength);

contact->contactNormal = (particle[i]->getPosition() - closestPoint).unit();

contact->restitution = restitution;

contact->particle[0] = particle[i];

contact->particle[1] = 0;

contact->penetration = particle[i]->getRadius() - sqrt(distanceToPlatform);

used++;

contact++;

}

}

}

return used;

}

class BlobDemo : public Application

{

Particle\* blob;

ParticleCollision\* particleCollision;

Platform\* platform[NUM\_PLATFORMS];

ParticleWorld world;

public:

/\*\* Creates a new demo object. \*/

BlobDemo();

virtual ~BlobDemo();

/\*\* Returns the window title for the demo. \*/

virtual const char\* getTitle();

/\*\* Display the particles. \*/

virtual void display();

/\*\* Update the particle positions. \*/

virtual void update();

void BlobDemo::boxCollisionResolve(Particle\* particle);

bool BlobDemo::outOfBoxTest(Particle\* particle);

void BlobDemo::outOfBoxResolve(Particle\* particle);

};

// Method definitions

BlobDemo::BlobDemo() : world((NUM\_PARTICLES + NUM\_PLATFORMS) \* (NUM\_PARTICLES + NUM\_PLATFORMS - 1), 5)

{

width = 400; height = 400;

nRange = 100.0;

// Create the blob storage

blob = new Particle[NUM\_PARTICLES];

//Create a new particle collision object, and tell it how many other particles there are to watch for collisions with.

//Also, give it a pointer to the array of particles, and a pointer to the specific particle it is associated with

particleCollision = new ParticleCollision(NUM\_PARTICLES, blob);

// Create the platform

platform[0] = new Platform;

platform[0]->setRestitution(0.6);

platform[0]->start = Vector2(-50.0, 20.0);

platform[0]->end = Vector2(45.0, 15.0);

// Make sure the platform knows which particle it should collide with.

for (int i = 0; i < NUM\_PLATFORMS; i++)

for (int j = 0; j < NUM\_PARTICLES; j++)

platform[i]->particle[j] = blob + j;

//Add platforms to world object's vector of platform contact generators

for (int i = 0; i < NUM\_PLATFORMS; i++)

world.getPlatformContactGenerators().push\_back(platform[i]);

//Add particle collision object to world object's vector of particle contact generators

world.getParticleContactGenerator().push\_back(particleCollision);

// Initialise circle particles

for (int i = 0; i < NUM\_CIRCLES; i++)

{

(blob + i)->setPosition(i \* 10, 80);

(blob + i)->setRadius(BASE\_CIRCLE\_RADIUS + (i % 10));

(blob + i)->setMass(BASE\_CIRCLE\_MASS + (i % 10));

(blob + i)->setVelocity(0, -1);

(blob + i)->setAcceleration(Vector2::GRAVITY \* 20.0f);

(blob + i)->clearAccumulator();

world.getParticles().push\_back(blob + i);

}

//Initialise quad particles

for (int i = NUM\_CIRCLES; i < NUM\_CIRCLES + NUM\_QUADS; i++)

{

(blob + i)->setPosition(i \* 10, 80);

(blob + i)->setRadius(20);

(blob + i)->setMass(20);

(blob + i)->setVelocity(0, -2);

(blob + i)->setAcceleration(Vector2::GRAVITY \* 20.0f);

(blob + i)->clearAccumulator();

float radius = (blob + i)->getRadius();

//Set up vertices for non-circle particles

vector<Vector2> vertices = {

Vector2(1, 1).unit() \* radius,

Vector2(1, -1).unit() \* radius,

Vector2(-1, -1).unit() \* radius,

Vector2(-1, 1).unit() \* radius

};

(blob + i)->setVertices(vertices);

float width = vertices[0].x - vertices[3].x;

float height = vertices[0].y - vertices[1].y;

(blob + i)->setWidthAndHeight(width, height);

world.getParticles().push\_back(blob + i);

}

//Initialise triangle particles

for (int i = NUM\_CIRCLES + NUM\_QUADS; i < NUM\_PARTICLES; i++)

{

(blob + i)->setPosition(i \* 10, 80);

(blob + i)->setRadius(20);

(blob + i)->setMass(15);

(blob + i)->setVelocity(0, -2);

(blob + i)->setAcceleration(Vector2::GRAVITY \* 20.0f);

(blob + i)->clearAccumulator();

float radius = (blob + i)->getRadius();

//Set up vertices for non-circle particles

vector<Vector2> vertices = {

Vector2(0, 1).unit() \* radius,

Vector2(1.1, -0.9).unit() \* radius,

Vector2(-1.1, -0.9).unit() \* radius

};

(blob + i)->setVertices(vertices);

float width = vertices[1].x - vertices[2].x;

float height = vertices[0].y - vertices[1].y;

(blob + i)->setWidthAndHeight(width, height);

world.getParticles().push\_back(blob + i);

}

}

BlobDemo::~BlobDemo()

{

// Release the blob storage

delete[] blob;

}

void BlobDemo::display()

{

Application::display();

//Render platforms

for (int i = 0; i < NUM\_PLATFORMS; i++)

{

const Vector2 &p0 = platform[i]->start;

const Vector2 &p1 = platform[i]->end;

glBegin(GL\_LINES);

glColor3f(0, 1, 1);

glVertex2f(p0.x, p0.y);

glVertex2f(p1.x, p1.y);

glEnd();

}

//Render platforms

//Render circle particles

float r = 0.0, g = 0.0, b = 1;

for (int i = 0; i < NUM\_CIRCLES; i++, r += 0.1, g += 0.1)

{

if (r > 0.9 && g > 0.9)

r = g = 0;

glColor3f(r, g, b);

Vector2 &p = (blob + i)->getPosition();

glPushMatrix();

glTranslatef(p.x, p.y, 0);

glutSolidCircle((blob + i)->getRadius(), 12, 12);

glPopMatrix();

}

//Render circle particles

//Render quads

r = 1, g = 0, b = 0;

for (int i = NUM\_CIRCLES; i < NUM\_CIRCLES + NUM\_QUADS; i++, g += 0.2, b += 0.2)

{

vector<Vector2> vertices = (blob + i)->getVertices();

int numVertices = vertices.size();

Vector2 position = (blob + i)->getPosition();

glPushMatrix();

glTranslatef(position.x, position.y, 0);

if (g > 0.9 && b > 0.9)

g = b = 0;

glColor3f(r, g, b);

glBegin(GL\_QUADS);

for (int i = 0; i < numVertices; i++)

glVertex2f(vertices[i].x, vertices[i].y);

glEnd();

glPopMatrix();

}

//Render quads

//Render triangles

r = 0, g = 1, b = 0;

for (int i = NUM\_CIRCLES + NUM\_QUADS; i < NUM\_PARTICLES; i++, r += 0.2, b += 0.2)

{

vector<Vector2> vertices = (blob + i)->getVertices();

int numVertices = vertices.size();

Vector2 position = (blob + i)->getPosition();

glPushMatrix();

glTranslatef(position.x, position.y, 0);

if (r > 0.9 && b > 0.9)

r = b = 0;

glColor3f(r, g, b);

glBegin(GL\_TRIANGLES);

for (int i = 0; i < numVertices; i++)

glVertex2f(vertices[i].x, vertices[i].y);

glEnd();

glPopMatrix();

}

//Render triangles

glutSwapBuffers();

}

void BlobDemo::update()

{

// Recenter the axes

float duration = timeinterval / 1000;

// Run the simulation

world.runPhysics(duration);

//Boundary collision detection and resolution

for (int i = 0; i < NUM\_PARTICLES; i++)

{

boxCollisionResolve(blob + i);

if (outOfBoxTest(blob + i))

outOfBoxResolve(blob + i);

}

Application::update();

}

// detect if the particle colided with the box and produce a response

void BlobDemo::boxCollisionResolve(Particle\* particle)

{

Vector2 position = particle->getPosition();

Vector2 velocity = particle->getVelocity();

float radius = particle->getRadius();

float w = Application::width;

float h = Application::height;

if (particle->isCircle())

{

// Reverse direction when you reach left or right edge

if (position.x > w - radius || position.x < -w + radius)

particle->setVelocity(-velocity.x, velocity.y);

// Reverse direction when you reach top or bottom edge

if (position.y > h - radius || position.y < -h + radius)

particle->setVelocity(velocity.x, -velocity.y);

}

else

{

float particleWidth = particle->getWidth();

float particleHeight = particle->getHeight();

// Reverse direction when you reach left or right edge

if (position.x - 0.5f \* particleWidth < -w || position.x + 0.5f \* particleWidth > w)

particle->setVelocity(-velocity.x, velocity.y);

// Reverse direction when you reach top or bottom edge

if (position.y - 0.5f \* particleHeight < -h || position.y + 0.5f \* particleHeight > h)

particle->setVelocity(velocity.x, -velocity.y);

}

}

// Check bounds. This is in case the window is made

// smaller while the circle is bouncing and the

// circle suddenly finds itself outside the new

// clipping volume

bool BlobDemo::outOfBoxTest(Particle\* particle)

{

Vector2 position = particle->getPosition();

Vector2 velocity = particle->getVelocity();

float radius = particle->getRadius();

if (particle->isCircle())

{

if ((position.x > Application::width - radius) || (position.x < -Application::width + radius)) return true;

if ((position.y > Application::height - radius) || (position.y < -Application::height + radius)) return true;

}

else

{

if (position.x - 0.5f \* particle->getWidth() < -Application::width || position.x + 0.5f \* particle->getWidth() > Application::width)

return true;

if (position.y - 0.5f \* particle->getHeight() < -Application::height || position.y + 0.5f \* particle->getHeight() > Application::height)

return true;

}

return false;

}

// Check bounds. This is in case the window is made

// smaller while the circle is bouncing and the

// circle suddenly finds itself outside the new

// clipping volume

void BlobDemo::outOfBoxResolve(Particle\* particle)

{

Vector2 position = particle->getPosition();

Vector2 velocity = particle->getVelocity();

float radius = particle->getRadius();

if (particle->isCircle())

{

if (position.x > Application::width - radius) position.x = Application::width - radius;

else if (position.x < -Application::width + radius) position.x = -Application::width + radius;

if (position.y > Application::height - radius) position.y = Application::height - radius;

else if (position.y < -Application::height + radius) position.y = -Application::height + radius;

}

else

{

if (position.x - 0.5f \* particle->getWidth() < -Application::width)

position.x = -Application::width + 0.5f \* particle->getWidth();

else if (position.x + 0.5f \* particle->getWidth() > Application::width)

position.x = Application::width - 0.5f \* particle->getWidth();

if (position.y - 0.5f \* particle->getHeight() < -Application::height)

position.y = -Application::height + 0.5f \* particle->getHeight();

else if (position.y + 0.5f \* particle->getHeight() > Application::height)

position.y = Application::height - 0.5f \* particle->getHeight();

}

particle->setPosition(position.x, position.y);

}

const char\* BlobDemo::getTitle()

{

return "Blob Demo";

}

/\*\*

\* Called by the common demo framework to create an application

\* object (with new) and return a pointer.

\*/

Application\* getApplication()

{

return new BlobDemo();

}

### main.cpp

#include <gl/glut.h>

#include "app.h"

extern Application\* getApplication();

Application\* app;

void display(void)

{

app->display();

}

void createWindow(const char\* title, int h, int w)

{

glutInitDisplayMode(GLUT\_DOUBLE | GLUT\_RGB | GLUT\_DEPTH);

glutInitWindowSize(w, h);

glutCreateWindow(title);

}

void TimerFunc(int value)

{

app->update();

float timeinterval = app->getTimeinterval();

glutTimerFunc(timeinterval, TimerFunc, 1);

}

void resize(int width, int height)

{

app->resize(width, height);

}

int main(int argc, char\* argv[])

{

glutInit(&argc, argv);

app = getApplication();

float timeinterval = 10;

app->setTimeinterval(timeinterval);

createWindow("Blob", app->getheight(), app->getwidth());

glutReshapeFunc(resize);

glutDisplayFunc(display);

glutTimerFunc(timeinterval, TimerFunc, 1);

app->initGraphics();

glutMainLoop();

delete app;

return 0;

}

## C:\Users\User\Documents\Advanced Data Structures\Janusz\Coursework2\ClassDiagram1.pngDesign and implementation of collision detection and response algorithms

The UML class diagram for the collision detection and resolution program is shown above. The main() function is responsible for initialising GLUT, using it to set up OpenGL, and registering callback functions e.g. glutTimerFunc(), glutDisplayFunc(). The main() function also instantiates a BlobDemo object.

The BlobDemo object inherits from the Application class, which has functions to handle the application window and display. The BlobDemo object contains an array of Particle objects, which are the particles to perform collision detection and resolution on. It also contains an array of Platform objects, which the particles can collide with. BlobDemo also owns a ParticleCollision object that is used to determine whether particles are colliding or not, and register the relevant contacts. BlobDemo holds a ParticleWorld object. BlobDemo also has functions for ensuring all particles remain within the boundaries of the window.

A Particle object has properties such as position, velocity, acceleration, mass, along with the appropriate getters and setters. It also has a Boolean variable denoting whether the particle in question is a sphere or not. Particle objects have a vector of vertices, along with a width and height for the particle. The vertices, width and height of a particle are only stored if the particle is not a sphere. Only the radius needs to be stored for sphere particles, since all of the other variables can be calculated from that. When setVertices() is called, the Boolean variable is set to false to show that the particle is not a sphere.

The ParticleWorld object is used to run the physics of the simulation. It holds a vector of particle object pointers, a vector of Platform pointers, and a pointer to the ParticleCollision object. It also has an array of ParticleContact pointers, and a ParticleContactResolver object. All of these objects are used to register and resolve collisions. The array of ParticleContact objects stores all detected contacts between particles and the ParticleContactResolver object is responsible for going through this array and resolving the collisions for each pair of particles in contact.

The Platform and ParticleCollision classes both inherit from an abstract class called ParticleContactGenerator. As this name suggests, both Platform and ParticleCollision objects are responsible for generating particle contacts and registering them with ParticleContact objects. Platform objects are responsible for detecting contacts between platforms and particles. The ParticleCollision object is responsible for detecting contacts between particles and other particles.

The ParticleCollision object holds a constant denoting the number of particles in the simulation and an array of pointers to these particles. It also contains a vector of vertices used to store the Minkowski difference between two particles. Upon instantiation, the ParticleCollision object takes the number of particles and a pointer to the first particle in the array. When addContact() is called, the program will compare each particle to each other particle in a loop.

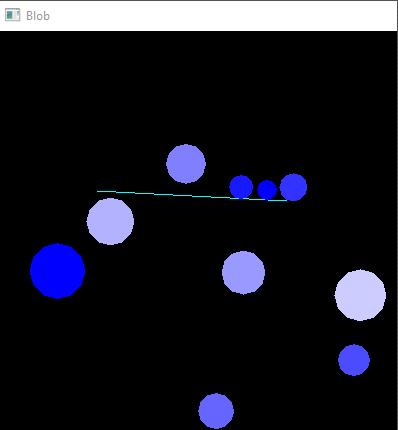
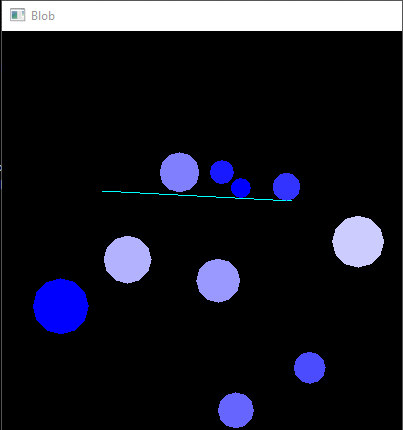
To check for collisions, addContact() calls checkCollision() to compare two given particles. If both particles are circles, then a collision is checked by comparing the total radii of the circles with the distance between them. If one particle is not a circle, then the calcMinkowskiDifferenceVertices() function is called to work out the Minkowski difference of the two particles. The vertices of the Minkowski difference are passed to the polygonContainsOrigin() function, which determines whether or not the Minkowski difference contains the origin. If the Minkowski difference contains the origin, then the two particles have collided. Otherwise, there is no collision.

## Testing

In order to verify that the developed program works correctly, a variety of tests will be conducted. Each test will include a screenshot. Some tests will require “before and after” screenshots. The tests will include the following:

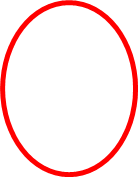
1. Test that circles interact with platform correctly
2. Test that collision detection and resolution works with circles
3. Test that drag force works (i.e. particles slow down and lose energy over time). Test that particles stay within the boundaries of the window
4. Increase number of circles to observe effect
5. Test with different masses and radii (one small, light particle and one large, heavy one)
6. Test that collision detection and resolution works between circles and boxes
7. Test that collision detection and resolution works between circles and triangles
8. Test that collision detection and resolution works between boxes and triangles
9. Increase number of boxes and triangles to observe effect

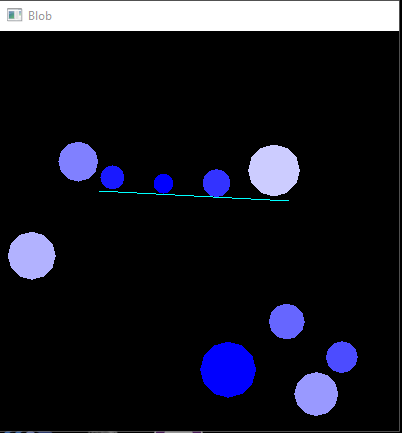
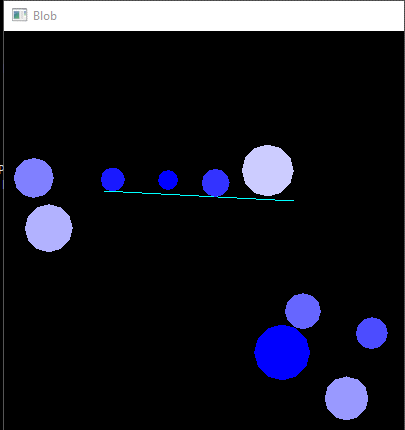
### Test 1: Circle interaction with platforms



These screenshots have been taken a moment apart. The one on the left is the earlier screenshot, and the one on the right is the later screenshot. It can be seen that the circle particles do not penetrate the platform. Instead, they bounce and roll off it, in the direction of the platform’s downwards slope. The coefficient of restitution for the platform is 0.6, so the particles bouncing off the platform lose most of their momentum.

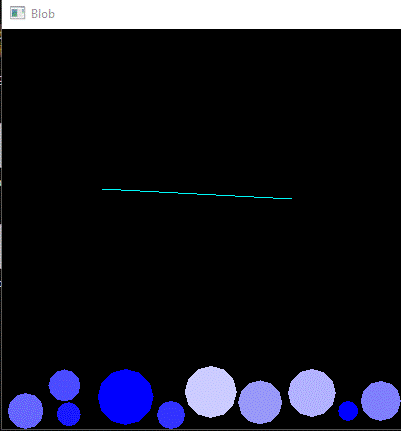
### Test 2: Collision detection and resolution with circles





As can be seen, there are two pairs of particles that collide during the time between these two screenshots. Both collisions are detected when they have occurred, and are resolved according to the masses and velocities of the particles in question.

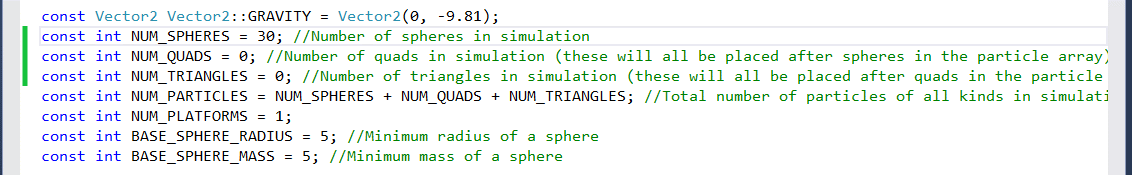
### Test 3: Test that drag force works. Test that particles stay within the boundaries of the window

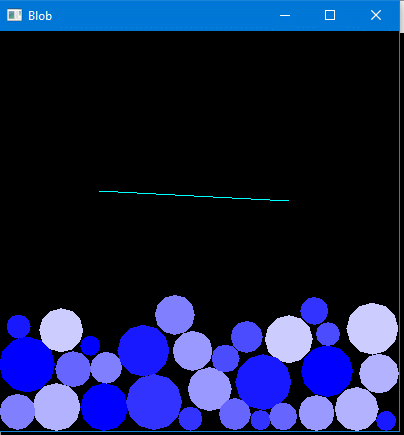
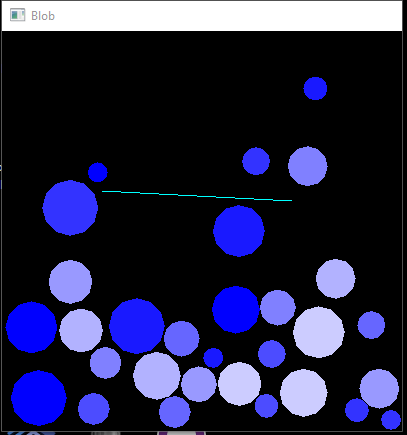


All of the particles eventually settle along the bottom of the screen and remain there, although they continue to bounce around slightly. This is due to the drag force, which decreases their velocity over time, along with the downwards acceleration due to gravity, which causes them to gather at the bottom of the screen.

This also allows us to see that the particles are prevented from leaving the boundaries of the window. This is done by reversing the relevant component of their velocity when they touch a horizontal or vertical window boundary.

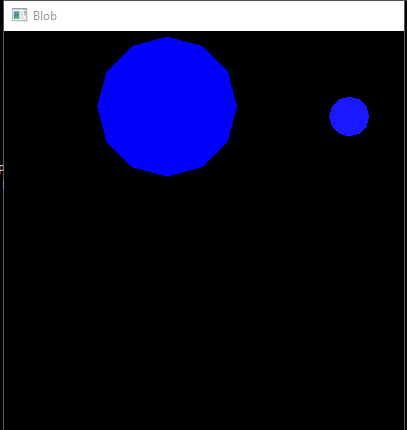
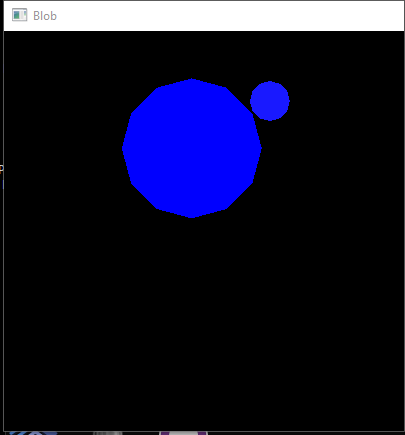
### Test 4: Increase number of circles to observe effect





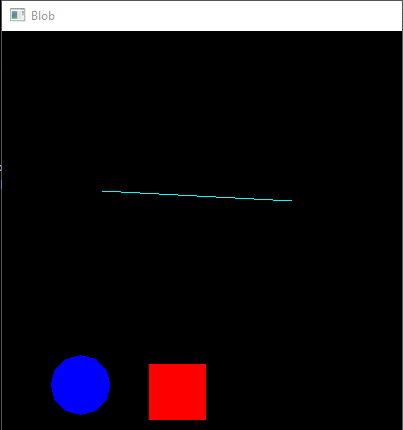
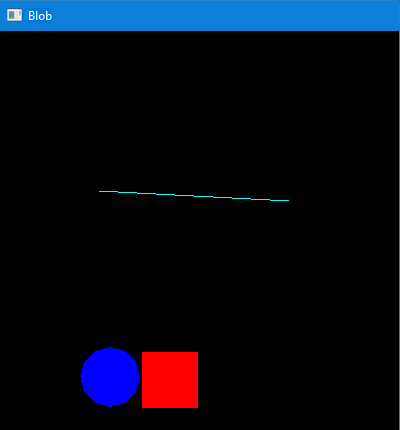
Upon increasing the number of circles to 30, it can be seen that they show the same behaviour, colliding together and slowly losing momentum until they gather at the bottom of the screen. The inter-penetration resolution between the circles works very well, as none of the circles have slipped past each other’s borders. The collision detection and resolution algorithm scales fairly well here, since the check for collision between circles is simple and efficient.

### Test 5: Test with different masses and radii (one small, light particle and one large, heavy one)



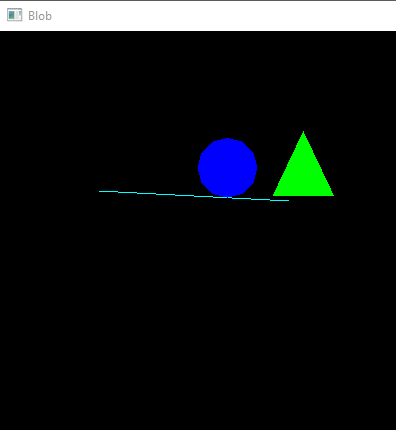
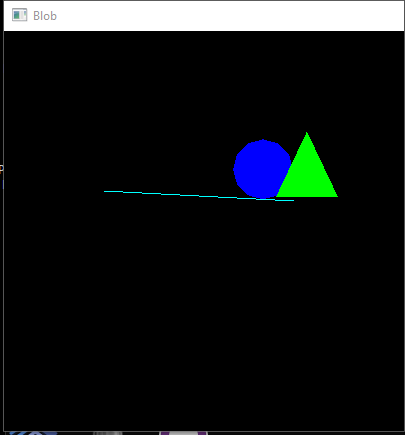
The smaller particle has been given a small radius and mass, and the large particle has been assigned a large radius and mass. Upon collision, it can be seen that the small and light particle has been affected more by the collision than the large circle. It has bounced to the edge of the window, while the large circle has not moved much.

### Test 6: Test that collision detection and resolution works between circles and boxes

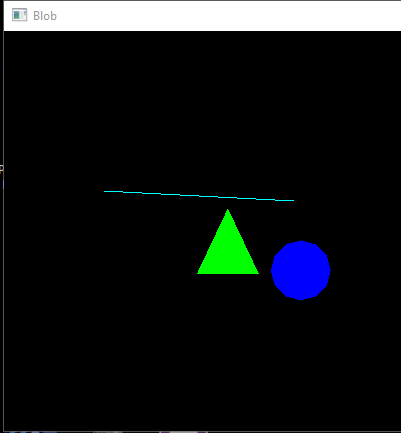
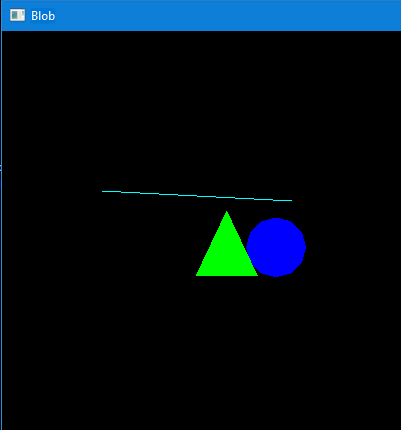


In the pair of screenshots above, it can be seen that the collision is detected when the circle comes into contact with the left edge of the square, and the two particles bounce away from each other correctly.

### Test 7: Test that collision detection and resolution works between circles and triangles

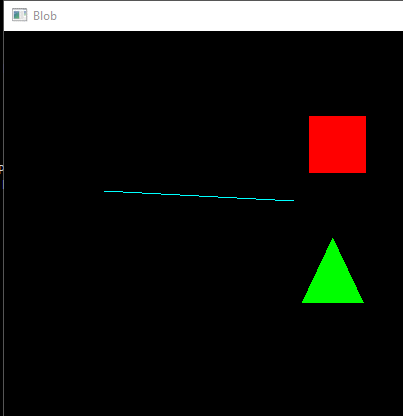
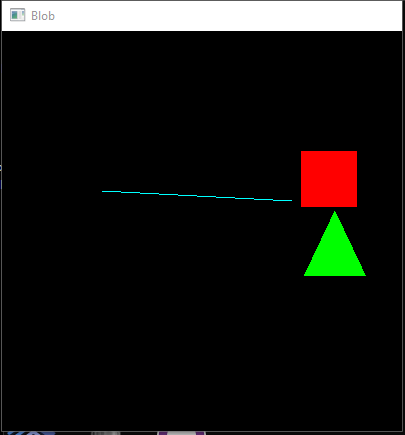


Here, the collision between the circle and the left edge of the triangle is detected and responded to correctly. However, there is some interpenetration between the particles before the collision is resolved.

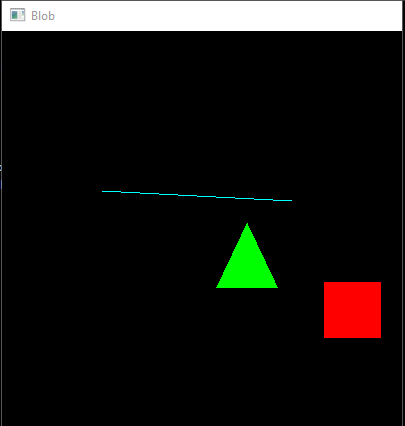
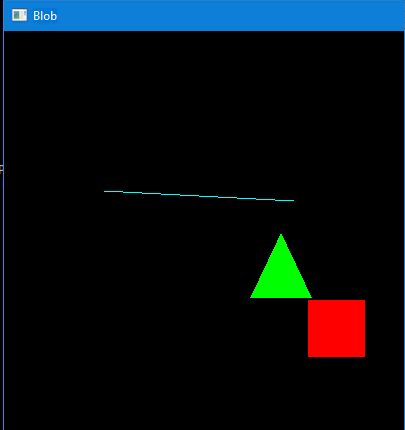


In this collision, there is no issue with inter-penetration when the circle collides with the right edge of the triangle and bounces away. The collision is detected and responded to correctly.

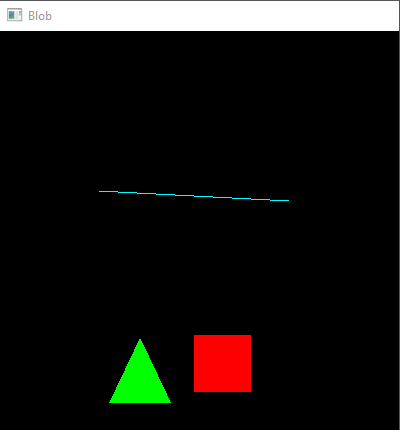
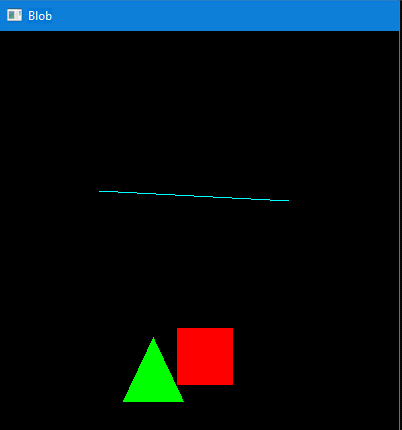
### Test 8: Test that collision detection and resolution works between boxes and triangles



In this test, the collision between the bottom edge of the square and the top point of the triangle is detected and resolved correctly, with the particles bouncing away from each other vertically.

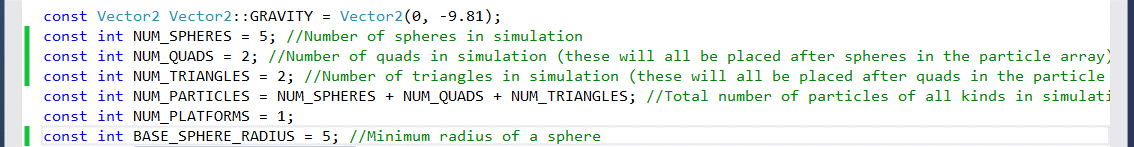


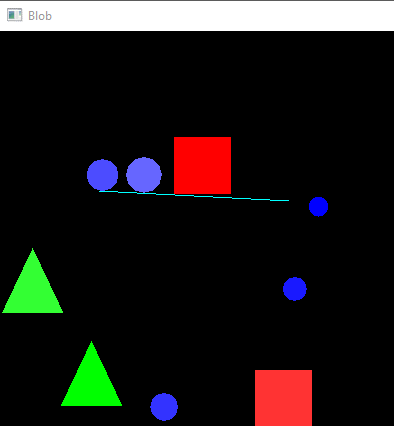
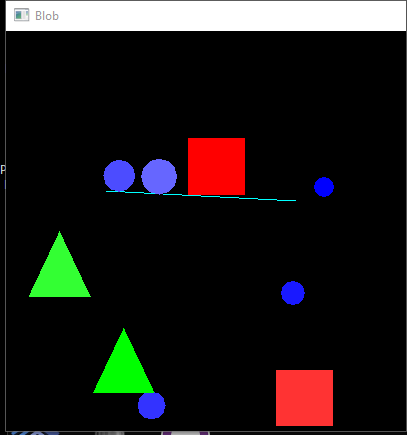
In the pair of screenshots above, it is shown that the collision detection also works between the corners of the square and triangle particles.



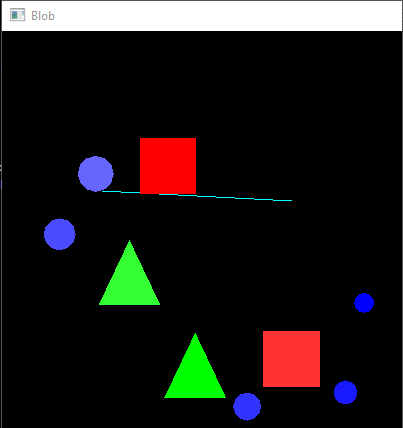
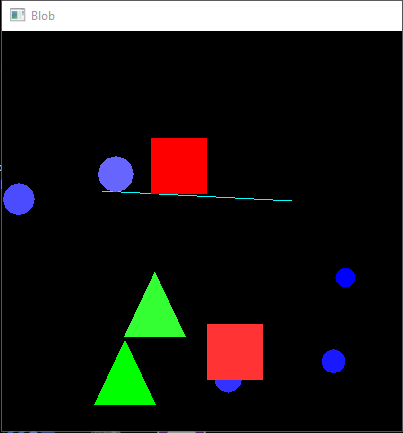
This last pair of screenshots shows the corner of the square making contact with the right edge of the triangle. There does not seem to be any inter-penetration, and the collision is resolved correctly.

### Test 9: Increase number of boxes and triangles to observe effect





In the pair of screenshots above, there are 5 circles, 2 squares, and 2 triangles. The triangle and circle highlighted have collided and bounced apart without any errors.



There are two sets of collisions in the screenshots above. The corners of the two triangles collide, and the collision is resolved correctly. However, when one of the squares collides with one of the circles, a large amount of interpenetration is visible before the collision is resolved. In general, there do seem to be some bugs or incorrect values for inter-penetration. This is a common difficulty when using the Minkowski difference for collision detection. However, the majority of collisions are detected and resolved correctly, with little to no inter-penetration.

It should be noted that due to the implementation, which explicitly calculates the Minkowski difference between every circle and non-circle particle combination, the collision detection and resolution algorithm performance is O(n2). As such, it does not scale well as additional particles are added. If more particles needed to be added, then an optimisation such as the Gilbert-Johnson-Keerthi distance algorithm would be required. This would avoid the need to calculate the entire Minkowski difference for each combination of particles.

## References

Ericson, C. (2005) *Real-Time Collision Detection.* San Francisco : Morgan Kaufmann

Gilbert, E., Johnson, D. and Keerthi, S. (1988). ‘A fast procedure for computing the distance between complex objects in three-dimensional space’, *IEEE Journal on Robotics and Automation*, 4(2), pp. 193-203.

Gottschalk, S., Lin, M. and Manocha D. (1996) ‘OBBTree: A Hierarchical Structure for

Rapid Interference Detection’, *SIGGRAPH ’96*: *23rd Annual Conference on Computer Graphics and Interactive Techniques,* New Orleans, Louisiana, 4-9 August. New York: ACM [Online]. Available at: http://gamma.cs.unc.edu/SSV/obb.pdf (Accessed: 12 January 2018).