Collision Detection Report

# Background Information

Collision detection is the problem of determining if, when and where two objects encounter each other. According to Ericson (Ericson, 2005), the if, when and where are increasingly more complex queries; “if” being a Boolean result, “when” determining the time in which a collision occurs and “where” establishing how objects meet each other. Information about when and where collisions occur is often called “collision determination”. Additionally, intersection detection and interference detection are synonyms.

Schwarzl (Schwarzl, 2012), also states that collision detection is boiled down to a Boolean yes/no for the intersection of shapes and the how and where of how an intersection occurs. He goes further to state that within a 2D environment, Boolean checks for collision are sufficient for much of the problem solving within the context of collision detection.

Within the context of computer graphics, particularly computer games, collision detection is used to help maintain the illusion of a solid world composed of physical objects that can be bumped into. Without this detection, models and entities within a game will simply clip through other models and entities, greatly breaking immersion.

There are many factors that influence the implementation of a collision detection algorithm. These include the geometrical representation used for a scene, different query types on collision, the performance of a system, the robustness of a system and the ease of implementation and use. Developing collision detection is a careful balancing act between the sophistication of a system and its ability to stay within performance limits. For example, a system that models many thousands of particles striking each other every second may be somewhat realistic compared to a system that models only a few dozen but the computational power required increases exponentially, particularly if the system is not optimised.

Aiqing (Aiqing, 2012) identifies two broad categories of collision detection within virtual reality, the surface model and the body model. The boundary surface model is used in representing objects at the surface level with the body model uses voxels and may deform upon impact with another object. The use of the common AABB (aligned axis bounding box) method is used to simulate collisions between objects in VR for most generic objects. While the author does go into detail on different methods of implementing collision detection, the poor grammar within the article suggests that the article was either poorly translated or written by a non-native speaker of English. Instead I have found that Xing et al. (Xing, Liu and Xu, 2010) are a better source on AABB collision detection. They state that:

“The AABB bounding box of an object is defined as the smallest hexahedron which contains the collision object and the edges parallel to the axis. Therefore, there is only a need for six float values to describe an AABB. When constructing the AABBs, it needs to go along the axial (X, Y, Z) of the object’s local coordinate system. Therefore, there is only a need for six float values to describe an AABB. Because its construction difficulty is low and its storage space is small enough, the AABBs algorithm is widely used.”

Modelling collisions can be very expensive in computational power. According to Hwan Sul (Hwan Sul, 2010), pre-computing collision matrices can be an extremely effective tool for complex collision detection involving cloth or clothing items rendered within a graphical environment. Cloth is particularly difficult to render accurately because very small and light collisions between the material and objects in the environment is highly complex. As one example, a single sheet of cloth, draped over a table like a tablecloth with items upon it could have as many polygons as all the object placed upon it and the table underneath. Simulating the collision of so many polygons is particularly computationally expensive a reason why it is not often modelled in an any detail in video games especially when it involves a realistic level of physics.

# Introduction

This report and code will be to implement a collision detection system between various types of shapes in a C++ object oriented environment. OpenGL will be used as a graphical engine to visualise shapes and show collision detection between shapes. The development will test different types of collision such as between particles and the boundaries of the window, with differing and varied forces acting on all such particles within the system.

# Program Listing

### App.h

//Application header file

//Self documenting

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

//Main system files

#include <gl/glut.h>

#include "app.h"

//Various basic getters/setters

int Application::getwidth() { return width; }

int Application::getheight() { return height; }

float Application::getTimeinterval() { return timeinterval; }

void Application::setTimeinterval(float timeinterval) { Application::timeinterval = timeinterval; }

//Sets the background colour of the screen to black, 1.0f is for the alpha channel which is entirely opaque

void Application::initGraphics() { glClearColor(0.0f, 0.0f, 0.0f, 1.0f ); }

//Display function that clears the graphical state

void Application::display()

{

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

glMatrixMode(GL\_MODELVIEW);

glLoadIdentity();

}

//Glut's refresh function, used to update the screen when needed

void Application::update() { glutPostRedisplay(); }

//Resize function that transforms the size of the screen depending on how the user has moved the window

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

{

// Prevent a divide by zero

if (height == 0) height = 1;

//nRange = 100.0f; //Obsolete due to width and height being passed in, legacy

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

// 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, clears matrix by loading identity

glMatrixMode(GL\_MODELVIEW);

glLoadIdentity();

}

### BlobDemo.cpp

//List of includes, system files and user created files

#include <gl/glut.h>

#include "app.h"

#include "coreMath.h"

#include "pcontacts.h"

#include "pworld.h"

#include "collision.h"

#include <stdio.h>

#include <cassert>

#include <random>

#include <time.h>

//Number of Particles, main system control of the amount of particles generated.

//Advised to keep to 200 or less unless mass controls in the constructor are changed

const int NoOfParticles = 20;

//Gravity set to standard level

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 physics.

\*/

class Platform : public ParticleContactGenerator

{

public:

Vector2 start;

Vector2 end;

/\*\*

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

\*/

Particle \*particle;

virtual unsigned addContact(

ParticleContact \*contact,

unsigned limit

) const;

};

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

{

//const static float restitution = 0.8f;

const static float restitution = 1.0f;

unsigned used = 0;

// Check for penetration

Vector2 toParticle = particle->getPosition() - start;

Vector2 lineDirection = end - start;

float projected = toParticle \* lineDirection;

float platformSqLength = lineDirection.squareMagnitude();

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

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;

contact->particle[1] = 0;

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

used ++;

contact ++;

}

}

else if (projected >= platformSqLength)

{

// The blob is nearest to the end point

toParticle = particle->getPosition() - end;

if (toParticle.squareMagnitude() < squareRadius)

{

// We have a collision

contact->contactNormal = toParticle.unit();

contact->restitution = restitution;

contact->particle[0] = particle;

contact->particle[1] = 0;

contact->penetration = particle->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->getPosition()-closestPoint).unit();

contact->restitution = restitution;

contact->particle[0] = particle;

contact->particle[1] = 0;

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

used ++;

contact ++;

}

}

return used;

}

//Main class for application, overrides application class

class BlobDemo : public Application

{

//Array of particles, control of array size at top of program

Particle \*blob[NoOfParticles];

Platform \*platform;

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();

//Collision detection methods

//Checks to see if particles are close enough for a collision with the program window

//Bounces particles off the sides of the window

void box\_collision\_resolve(Particle &particle);

//Checks to see if particles are outside of the window

bool out\_of\_box\_test(Particle particle);

//Moves particles back in window if needed

void out\_of\_box\_resolve(Particle &particle);

//Checks particles for validity for collision and then processes collisions by sending events to the collision class

void particle\_collision\_test(Particle &particle);

};

// Method definitions

BlobDemo::BlobDemo():world(2, 1)

{

//Global control for the window width and height

width = 800; height = 800;

//Range in units for the window, both positive and negative

nRange = 100.0; //Default = 100.0, giving range of 200 by 200 (-100 to 100)

//Seeds random numbers at this point, needed to randomly geenrate particles

srand(time(NULL));

// Create the blobs for the program, core constructor loop for blob details

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

{

//Create new particle

blob[i] = new Particle;

//Initiate random position of a particle within the nRange bounds specified further above

//Changes to this area should be accomplished via changes to the nRange variable

float randomX = -nRange + static\_cast <float> (rand()) / (static\_cast <float> (RAND\_MAX / (nRange - -nRange)));

float randomY = -nRange + static\_cast <float> (rand()) / (static\_cast <float> (RAND\_MAX / (nRange - -nRange)));

//Setup initial position and velocity of the particles

blob[i]->setPosition(randomX, randomY);

blob[i]->setVelocity(10, 10);

//Set up damping for the particles or in other words the amount of energy particles lose on contact with other particles or the sides of the box

//1.0 is default, which equates to no damping, lower numbers give more damping effects

//blob[i]->setDamping(0.9);

blob[i]->setDamping(1.0);

//Acceleration force is initially set to that of gravity to pull particles down

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

//Main control for the upper and lower limits of mass

//Changing these values will effect the colour of particles, the radius of particles and the mass of particles

//Default is 1.0f to 10.0f, which gives blue as heavy larger particles and red as smaller faster particles

float lowerMass = 1.0f;

float upperMass = 10.0f;

float randomMass = lowerMass + static\_cast <float> (rand()) / (static\_cast <float> (RAND\_MAX / (upperMass - lowerMass)));

blob[i]->setMass(randomMass);

blob[i]->setRed(1 / randomMass);

blob[i]->setBlue(randomMass / upperMass); //inverse calculation used to promote high blue in heavier objects

blob[i]->setRadius(randomMass / 2);

//Clear forces before program start

blob[i]->clearAccumulator();

//Set ID equal to loop counter

//This is used in order to help speed up calculation checks between particles

blob[i]->setID(i);

//Collision status is set to false so that the particle defaults to not being in a collision state on program launch

//variable is used for preventing multiple collisions by one particle in a single frame

blob[i]->setCollisionStatus(false);

//Particle world is assigned each particle in turn, currently unused

world.getParticles().push\_back(blob[i]);

}

// Create and initialise the platform

platform = new Platform;

//platform->start = Vector2 ( -50.0, 0.0 );

//platform->end = Vector2 ( 50.0, 0.0 );

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

// platform->particle = blob;

//world.getContactGenerators().push\_back(platform);

}

//Destructor for the demo

BlobDemo::~BlobDemo()

{

delete blob;

}

//Main display function, used to render objects onto the screen

void BlobDemo::display()

{

//Call main display function from the application class

Application::display();

//Defines the beginning and end of the platform

const Vector2 &p0 = platform->start;

const Vector2 &p1 = platform->end;

//Renders the platform onto screen

glBegin(GL\_LINES);

glColor3f(0,1,1);

glVertex2f(p0.x, p0.y);

glVertex2f(p1.x, p1.y);

glEnd();

//Begin main render loop for the particles

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

{

//Sets the colour of the particles depending on the random values assigned in the constructor

glColor3f(blob[i]->getRed(), blob[i]->getGreen(), blob[i]->getBlue());

//Code block that renders the particles to the position they are in

const Vector2 &p = blob[i]->getPosition();

glPushMatrix();

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

glutSolidSphere(blob[i]->getRadius(), 12, 12); //Radius, slices of the circle, stacks of the circle

glPopMatrix();

//Push/Pop matrix is used to place a new matrix onto the stack and then removes it, in effect allows the translation of objects

}

//Presents the back buffer to the screen

glutSwapBuffers();

}

//Main program update step

void BlobDemo::update()

{

// Recenter the axes

float duration = timeinterval/1000;

// Run the simulation

world.runPhysics(duration);

//Main loop for all collisions

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

{

//Checks to see if out of bounds or particle hits the edge of the box

box\_collision\_resolve(\*blob[i]);

if (out\_of\_box\_test(\*blob[i])) out\_of\_box\_resolve(\*blob[i]);

//Checks to see if particles collide with each other, if so feeds into the main algorithm for collision detection

particle\_collision\_test(\*blob[i]);

}

//Reset loop for reseting the state of collisions to false at the end of the frame update

//This is needed because particles if they collide with each other have their status set to true so multiple collisions do not happen with the same particle in the same frame

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

{

//Flicks bool to false

blob[i]->setCollisionStatus(false);

}

//Run main application update step (does little)

Application::update();

}

//Sets title for the program

const char\* BlobDemo::getTitle()

{

return "Collision Detection";

}

/\*\*

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

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

\*/

Application\* getApplication()

{

return new BlobDemo();

}

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

void BlobDemo::box\_collision\_resolve(Particle &particle)

{

//Gets core variables for calculations

Vector2 position = particle.getPosition();

Vector2 velocity = particle.getVelocity();

float radius = particle.getRadius();

float w = Application::width;

float h = Application::height;

// Reverse direction when you reach left or right edge

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

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

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

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

}

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

// smaller while the sphere is bouncing and the

// sphere suddenly finds itself outside the new

// clipping volume

bool BlobDemo::out\_of\_box\_test(Particle particle)

{

//Gets variables for testing if out of bounds

Vector2 position = particle.getPosition();

Vector2 velocity = particle.getVelocity();

float radius = particle.getRadius();

//Tests if out of bounds, returns true if so

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;

//false indicates particle within bounds

return false;

}

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

// smaller while the sphere is bouncing and the

// sphere suddenly finds itself outside the new

// clipping volume

void BlobDemo::out\_of\_box\_resolve(Particle &particle)

{

//Get variables for the resolution

Vector2 position = particle.getPosition();

Vector2 velocity = particle.getVelocity();

float radius = particle.getRadius();

//Sets up variables for moving a particle back to the box

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;

//Moves the particle to within the box if needed

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

}

//Main particle checking method, feeds into the collision.cpp files

void BlobDemo::particle\_collision\_test(Particle &particle)

{

//Main loop for the particle array

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

{

//Check to see if particle being checked against is the same as the particle we're checking

//Also checks to see if a collision with each particle examined has happened this frame, if so then ignore said particle

if (particle.getID() == blob[i]->getID() || particle.getCollisionStatus() == true || blob[i]->getCollisionStatus() == true)

{

return;

//Do nothing if same particle

//or if particle has been collided this frame

}

else

{

//Setup new collision

Collision \*collision = new Collision(&particle, blob[i]);

//Check for collision

//If collision is checked as true, then resolve the collision

if(collision->checkForCollision()) collision->resolveCollision();

//clean up

delete collision;

}

}

}

### Collision.h

#pragma once

#include "particle.h"

//Collision class declaration

//This class deals with the collisions between particles

class Collision

{

//Particle variables stored so that collision detection can be calculated between the two

//Private by default

Particle \*particle1;

Particle \*particle2;

public:

//Declaration of methods for the collision class

//Default constructor and checking/resolving methods

Collision(Particle \*p1, Particle \*p2);

bool checkForCollision();

void resolveCollision();

};

### Collision.cpp

#include "collision.h"

//Default constructor the collision, assigns two particles for a collision to the object's variables for future calculations

Collision::Collision(Particle \*p1, Particle \*p2)

{

particle1 = p1;

particle2 = p2;

}

//Method that checks for a collision, if a collision is found it returns true

bool Collision::checkForCollision()

{

//Distance calculations and checks for determining if a collision is possible

Vector2 vecD = particle1->getPosition() - particle2->getPosition();

float distance = sqrt(pow(vecD.x, 2) + pow(vecD.y, 2));

float sumRadius = particle1->getRadius() + particle2->getRadius();

//If the distance is smaller than the sum of both particles radius's then a collision will occur

if (distance <= sumRadius)

{

//Fluffing code

//Moves particle slightly off centre from the collision if particles overlap

//In essence this section deals with the issue of particles wrapping around each other or circling each other

Vector2 interpenetrationVec = particle1->getPosition() - particle2->getPosition();

float penetration = (particle1->getRadius() + particle2->getRadius()) - interpenetrationVec.magnitude();

interpenetrationVec.normalise();

interpenetrationVec \*= penetration;

particle1->setPosition(particle1->getPosition() + interpenetrationVec \* 0.5);

particle2->setPosition(particle2->getPosition() - interpenetrationVec \* 0.5);

//Without this section of code, particles will circle each other in death spirals every so often

//since the distance is closer than the sum of radius's, a collision will occur and the method will return true

return true;

}

//if distance is greater then no collision will occur

return false;

}

//Particle collision method that performs all calculations and variable changes for a collision

void Collision::resolveCollision()

{

//Initial vector of the difference in position is taken

Vector2 x = particle1->getPosition() - particle2->getPosition();

//Vector is normalised to find common plane between particles

x.normalise();

//Sphere1 Calculations

Vector2 v1 = particle1->getVelocity();

float x1 = x.scalarProduct(v1);

Vector2 v1x = x \* x1;

Vector2 v1y = v1 - v1x;

float m1 = particle1->getMass();

//Sphere2 Calculations

x = x \* -1;

Vector2 v2 = particle2->getVelocity();

float x2 = x.scalarProduct(v2);

Vector2 v2x = x \* x2;

Vector2 v2y = v2 - v2x;

float m2 = particle2->getMass();

//Calculate final velocities

//This is based on newtonian physics

//Worth stating I don't understand all the mathematics in this section of code, both above and below

//But I do know how to use and implement it

particle1->setVelocity(v1x \* ((m1 - m2) / (m1 + m2)) + v2x \* ((2 \* m2) / (m1 + m2)) + v1y);

particle2->setVelocity(v1x \* ((2 \* m1) / (m1 + m2)) + v2x \* ((m2 - m1) / (m1 + m2)) + v2y);

//Sets the collision status of particles to true for this frame

//This is used to prevent multiple collisions occuring for one particle in a single frame of the application

particle1->setCollisionStatus(true);

particle2->setCollisionStatus(true);

}

### 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);

}

/\*\*

\* 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

### 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"

class Particle

{

protected:

//Particle variables used to control all aspects of a particle

//Self documenting

float inverseMass;

float damping;

float radius;

Vector2 position;

Vector2 velocity;

Vector2 forceAccum;

Vector2 acceleration;

int ID;

bool collisionStatus;

float red, green, blue;

public:

//A long list of accessor and mutator methods, I do not need to comment on each of these

//Code here is in my opinion, self documenting

void integrate(float duration);

void setMass(const float mass);

float getMass() const;

void setInverseMass(const float inverseMass);

float getInverseMass() const;

bool hasFiniteMass() const;

void setDamping(const float damping);

float getDamping() const;

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);

int getID();

void setID(int i);

bool getCollisionStatus();

void setCollisionStatus(bool c);

float getRed();

float getGreen();

float getBlue();

void setRed(float r);

void setGreen(float g);

void setBlue(float b);

};

#endif //

### Particle.cpp

#include "particle.h"

#include <math.h>

#include <assert.h>

#include <float.h>

//Main force application section

//Deals with forces applied to particles

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);

// Impose drag.

velocity \*= pow(damping, duration);

// Clear the forces.

clearAccumulator();

}

//Getters and Setters for the Particle class

//I am not commenting on every single one as its a waste of time for you and me combined

//This kind of code is entirely self documenting due to its extreme simplicity

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::setDamping(const float damping) { Particle::damping = damping; }

float Particle::getDamping() const { return damping; }

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; }

int Particle::getID() { return ID; }

void Particle::setID(int i) { ID = i; }

bool Particle::getCollisionStatus() { return collisionStatus; }

void Particle::setCollisionStatus(bool c) { collisionStatus = c; }

float Particle::getRed() { return red; }

float Particle::getGreen() { return green; }

float Particle::getBlue() { return blue; }

void Particle::setRed(float r) { red = r; }

void Particle::setGreen(float g) { green = g; }

void Particle::setBlue(float b) { blue = b; }

### 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) const = 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)

{

// 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()

);

}

}

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 contactGenerators;

/\*\*

\* 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& getContactGenerators();

};

#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()

{

unsigned limit = maxContacts;

ParticleContact \*nextContact = contacts;

for (ContactGenerators::iterator g = contactGenerators.begin();

g != contactGenerators.end();

g++)

{

unsigned 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)

{

// 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::getContactGenerators()

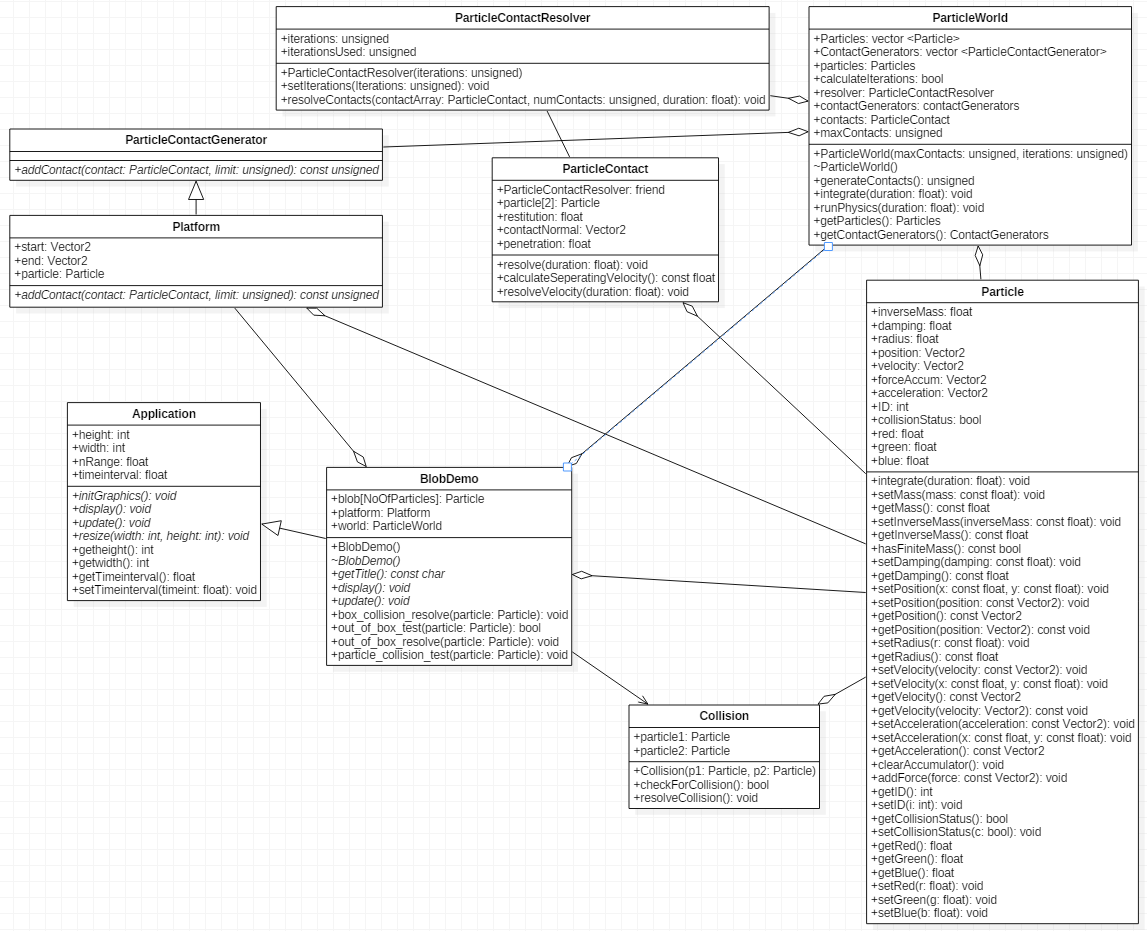
{

return contactGenerators;

}

# Collision Detection and Response Algorithms

## UML Design



## Design Process and Implementation

The collision detection implementation was broadly divided into two stages; implementation of collision detection with the window edges and implementation of collision detection with other particles. The first thing I did was to implement many particles and automate the random generation of particles into the application.

The detection of the window edges, including out of bounds checking was something I easily pulled from provided files. Since they did the job, it was a basic process of integrating them into the program and then testing their effectiveness.

Implementation of the collision detection with other particles was a much more drawn out process. I found an online resource (Studiofreya.com, 2018), that contained basic information on sphere collision detection. A few calculations were required to obtain a collision. Testing of collisions was done simply by using breakpoints within code that was within the completed conditions for collisions. True verification would have to wait until the completion of the collision response algorithm however.

The collision response for particles was a more difficult route given the amount of calculations required. Again I used an online resource (Studiofreya.com, 2018) as a basis for the calculations needed. Initially I tried to abstract away from the guide provided but found many errors during the implementation. Eventually I rewrote the code to be more in line with the guide I was using, ironing out bugs in my code. However even after completing the code to a perfect degree of accuracy, I found that many collisions were not working. Indeed approximately 1/10 collisions were working, with much of the particles simply passing through each other. Even with testing of distance levels, I could not find a reason for collisions to appear to be random. All particles collided at various points but did not collide on a consistent basis.

An idea I had previously came back to mind, that of adding in a Boolean value to confirm that a particle had collided which would lock a particle in a state of being unable to undergo a collision. I initially used this method as a way of trying to avoid problems with particles circling around each other in tight proximity. Upon implementation of this locking mechanism which was by design impossible to deadlock due to automatic release, collisions happened on a consistent basis by all particles. Why this technique worked, I do not know. I previously had particles which did not collide at all with each other suddenly colliding. A plausible but unconfirmed explanation would perhaps be that the particles were indeed colliding previously but were doing so repeatedly in a way that caused them to appear to carry on without effect. Yet given no evidence of this during my testing, I don’t believe this to be the most likely scenario.

The next issue I addressed was particles clumping up together into bunches or circling around each other, as if attached to one another. I reasoned that the most effective way of dealing with this after some talk with colleagues was to address interpenetration of particles with one another through shifting them slightly outside of each other’s range. The implementation of this was simple enough and should it have proved insufficient in one frame, another few frames would move the particles away from each other sufficiently enough, so they could carry on moving.

With the major issues dealt with, at this point I had gotten the particles to work to a level in which the application modelled collision detection and response seamlessly and even with large amounts of particles. The next step was to implement some method of testing and changing the masses and sizes of particles involved to work out if the mathematics under the hood so to speak were working correctly. I used random number generation again to automatically select appropriate mass and radius levels and tied the colour of the particles into this too. With this done, particles would be a deep blue if they were heavier and bigger and red if they were smaller and lighter. An addition result due to the physics of the application was that the smallest particles would be shot at great speed by collision with larger objects, due to the transfer of relatively large amounts of kinetic energy. With this completed, I decided to halt implementation there.

# Testing

Note: While screenshots are requested, I don’t see a need in screen shotting every test, when it is not visible that I have proven that such a test works. Instead, I will post screenshots of the final development showcasing the capabilities of the system as final test which will prove the system working.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Name of Test | Purpose of Test | Initial State | Expected Result | Outcome |
| Gravity Test | To test that gravity works | Single particle at top of screen | Particle falls downwards towards the bottom of the screen | Particle falls downwards towards the bottom of the screen |
| Border collision test (bottom) | To test that the particle collides with the ground | Single particle at top of screen, gravity present | Particle falls and hits the bottom, bouncing back up | Particle falls and hits the bottom, bouncing back up |
| Border collision test (left) | To test that the particle collides with the left side of the window | Single particle at top of screen, initial velocity set to negative x | Particle hits left side and bounces back | Particle hits left side and bounces back |
| Border collision test (top) | To test that the particle collides with the top of the window | Single particle at top of screen, initial velocity set to positive y | Particle hits top of window and bounces back | Particle hits top of window and bounces back |
| Border collision test (right) | To test that the particle collides with the right side of the window | Single particle at top of screen, initial velocity set to positive x | Particle hits right side and bounces back | Particle hits right side and bounces back |
| Damping Test | To test that damping works | Single particle at the top of the screen, gravity present | Particle falls and repeatedly bounces with a smaller bounce each time | Particle falls and repeatedly bounces with a smaller bounce each time |
| Collision of particles | To test the collision of multiple particles together | 10 randomly generated particles on screen | Collisions occur between particles when they collide | Collisions occur between particles when they collide |
| Collision differing based on angle | To test that particles realistically deflect based on angle of impact | 10 randomly generated particles on screen | Collisions occur realistically | Collisions occur realistically |
| Collision differing based on mass difference | To test different weights of impact | 10 randomly generated particles on screen with different masses and sizes | Collision of light and heavy objects gives proportionate change in velocity | Collision of light and heavy objects gives proportionate change in velocity |

### Example screenshot

# 

<https://gyazo.com/384891c3c991435440d02ce199e62be7>

Video link to animation of above screenshot, demonstrating most of tests performed in action.

### With Dampening:

<https://gyazo.com/c526a67e0e865ad71a056af84219be7a>

# References

Ericson, C. (2005). *Real-time collision detection*. Amsterdam: Elsevier.

Schwarzl, T. (2012). *2D game collision detection*. [Scotts Valley, Calif.: CreateSpace].

Studiofreya.com. (2018). *Simple Sphere-Sphere Collision Detection and Collision Response | Studio Freya*. [online] Available at: https://studiofreya.com/3d-math-and-physics/simple-sphere-sphere-collision-detection-and-collision-response/ [Accessed 12 Jan. 2018].

Hwan Sul, I. (2010). Fast cloth collision detection using collision matrix. *International Journal of Clothing Science and Technology*, 22(2/3), pp.145-160.

Aiqing, H. (2012). The Research on Collision Detection in Virtual Reality. *Emerging Computation and Information technologies for Education*, 146, pp.pp 71-76.

Xing, Y., Liu, X. and Xu, S. (2010). Efficient collision detection based on AABB trees and sort algorithm. *IEEE ICCA 2010*.