University of Hull

# 08024 ACW Report

Word Count:2189

Yuk L Ip : 201308286

5-15-2017

## Contents

## Introduction

The objectives of this ACW is to research, design and implement a distributed, physically-based modelling demonstrator for a series of gravity wells. The world would consist a large plane that is surrounded by a circular wall. A large number of spheres are placed on surface. The physics simulation would be run across a number of peer as a peer to peer network. Each peer would control an individual gravity well and display a consistent view of the world based on the position of the gravity well. Users could be applied forces to spheres via the gravity well from different peers.

## System Architecture

The UML class diagram of the project is shown in Appendix A to establish a brief overview how classes are developed to provide different functions. Based on their functionality, Classes are mainly divided to 5 components: Framework, Visualisation, Networking, Simulation and Entity Management.

### Class Description

#### Framework

Framework classes provide a foundation for the software to work on.

##### *WinMain*

Software entry point for Window

##### *System Class*

System Class manage the window framework and maintain the lifecycle of all other classes. When the system initialise different classes, new threads would be created and assigned to specific processors. Detail of threading and process affinity would be discussed in detail later.

##### *Config Class*

Config Class would keep hold of all the configuration data until the life cycle of the application finished. While it is being initialised, a configuration file would be read in order to obtain the variables to set up the simulation scene. In Appendix B, a template of configuration file has been shown that allow users to set starting values for different variables, such as Screen size, radius of the ground surface, Ports for network connections etc..

Config class has also been used for storing run time changeable values including target updating frequency for graphics, physics and network classes, elasticity, frictional forces and timescale.

##### *StepTimer Class*

The class provides elapsed time information and setting target elapsed seconds between each update. The timer would run and call the update in order to meet the expected fixed update time as close as possible. For physics, graphic and simulation class, they would initialise their own timer with target elapsed time from the Config class. As a result, users would be able to control different updating frequency on rendering, simulating and networking.

#### Visualisation

##### *Graphic Class*

The Graphic Class is developed to handle the visualisation of the physics modelling. To visualise objects on the screen, the Graphic Class manage the Direct3D Class and Camera Class and monitor user input at each frame.

##### *D3D Class*

This is a class to handle all the Direct3D functions. It maintains the ID3D11Device and Device Context and set up the characteristics of the swap chain in order to create a swap chain instance.

##### *Camera Class*

The Camera Class handles the view matrix and the location of camera. If the camera is updated to a new location or rotation, a new view matrix will be calculated.

## Networking

### *Network Class*

Network class handle all the connections to the peer to peer network, including communicate with other peers with TCP and UDP protocol, Data packing and unpacking.

While a peer connect to the network, it will scan the host from the set UDP port. If no incoming connection, this peer will host the network, waiting for incoming connection via TCP and broadcast its IP on UDP port. As Client will use the broadcasting IP to connect to the host via TCP and exchange data with the host

## Simulation

### *Simulation Class*

Simulation Class handle physics simulating including applying gravity, applying forces from gravity wells, collision detection and generate contact Manifold.

### *Contact Manifold Class*

This class is designed to resolve the contact Manifold generated by the Simulation Class. The detail of Collision detection and response in this application would be discussed in later session.

## Entity Management

### *Ball Manager*

This class is designed to store the data of ball in the simulating world.

### *Gravity Manager*

This class is designed to store the data of gravity well including other peer’s gravity well.

## Threading

In order to increase the performance of the software, major components would be needed to operate asynchronously. Therefore multithreading has been introduced to the system.

Firstly visualisation, simulation and networking have been separated into different threads after the system has finished the initialisation. As shown in Appendix C&D, the threads would run until user shutdown the program. Besides separate the main components, more threading have been used for small task in order to reduce the duration for each update.

As illustrated in Appendix C, threads have been created to handle communication with other peers in network Class:

- UDP Broadcasting
- UDP Listening
- TCP sending
- TCP receiving

The usage of threads in Simulation class has also been shown in Appendix D:

- Collision Detection:
    - Ground-Ball
    - Wall-Ball
    - Ball-Ball
- Collision Resolving

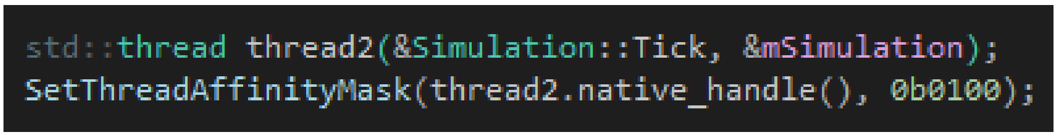
### Race condition

Race conditions would occur when two threads access a shared variable at the same time. To prevent race condition in the simulating system, mutex and lock_guard have been implemented in the Manager Class and Config Class. With lock_guard, resources could not be accessed while the mutex is owned by other threads.

In Collision detection, three different collision would be detect at the same time and add new contact point to the manifold and result in race condition. Therefore they would be setting up a temporary local manifold to store collision detected. Then three temporary manifold would be added to the main manifold in one. Similar approach has been used in networking for updating ball information from other peer after each simulation loop.

### Process Affinity

As required from the specification, threads for specific application have been allocated to specific processors by using SetThreadAffinityMask as shown in below diagram.

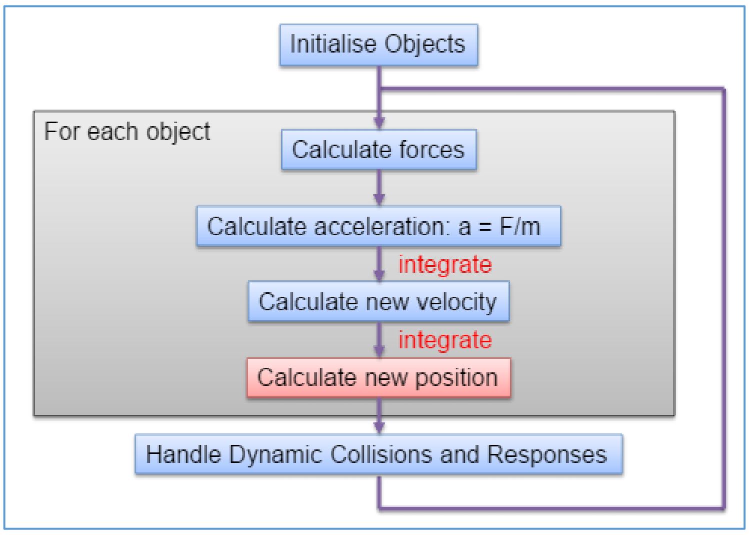


```
std::thread thread2(&Simulation::Tick, &mSimulation);
SetThreadAffinityMask(thread2.native_handle(), 0b0100);
```

Therefore Processor 1 has been set for Visualization application. Core 2 has been assigned for networking usage. All other remaining cores have been used for Simulation. 3 cores has been assigned to perform 3 different collision detection: Core 4 for the collisions between the ground and balls. Core 5 for the collisions between balls and Core 6 for the collisions between Wall and balls.

## Physics implementations

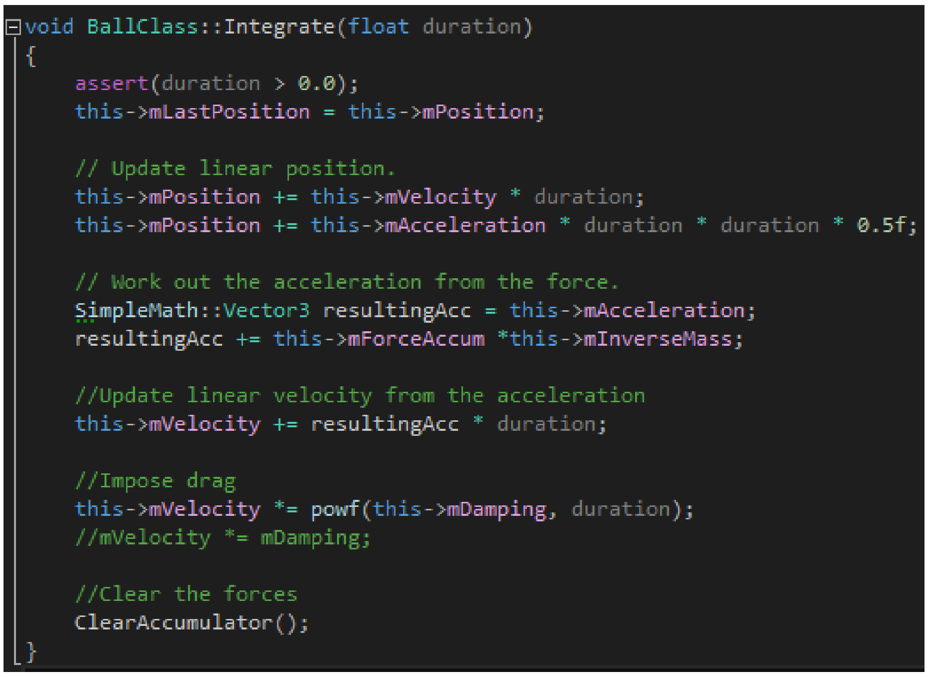
The basic process of the physics simulation loop is shown as below



For each simulation loop, position of objects would be updated according to their velocity, acceleration, forces that being applied on the object and the elapsed time from last simulation loop.

### Balls motion implementation

Below diagram has shown the first implemenation of the ball montion.



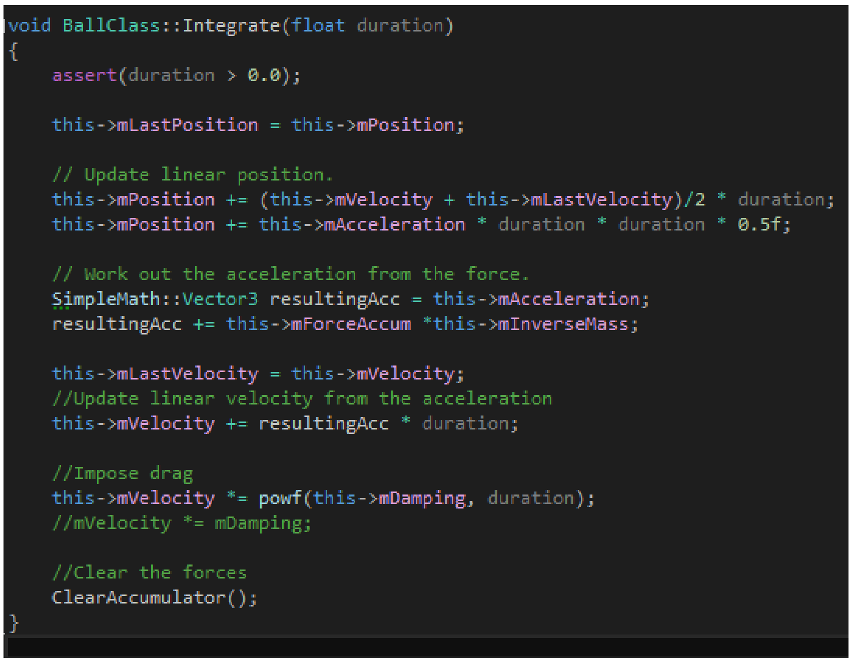
```
void BallClass::Integrate(float duration)
{
    assert(duration > 0.0);
    this->mLastPosition = this->mPosition;

    // Update linear position.
    this->mPosition += this->mVelocity * duration;
    this->mPosition += this->mAcceleration * duration * duration * 0.5f;

    // Work out the acceleration from the force.
    SimpleMath::Vector3 resultingAcc = this->mAcceleration;
    resultingAcc += this->mForceAccum *this->mInverseMass;

    //Update linear velocity from the acceleration
    this->mVelocity += resultingAcc * duration;

    //Impose drag
    this->mVelocity *= powf(this->mDamping, duration);
    //mVelocity *= mDamping;

    //Clear the forces
    ClearAccumulator();
}
```

While focusing on performance, Euler Integration has been chosen for simulating Ball motion. At the beginning of each simulation loop, the position would be updated by using the following equation:

$$p' = p + \dot{p}t + \frac{1}{2}\ddot{p}t^2$$

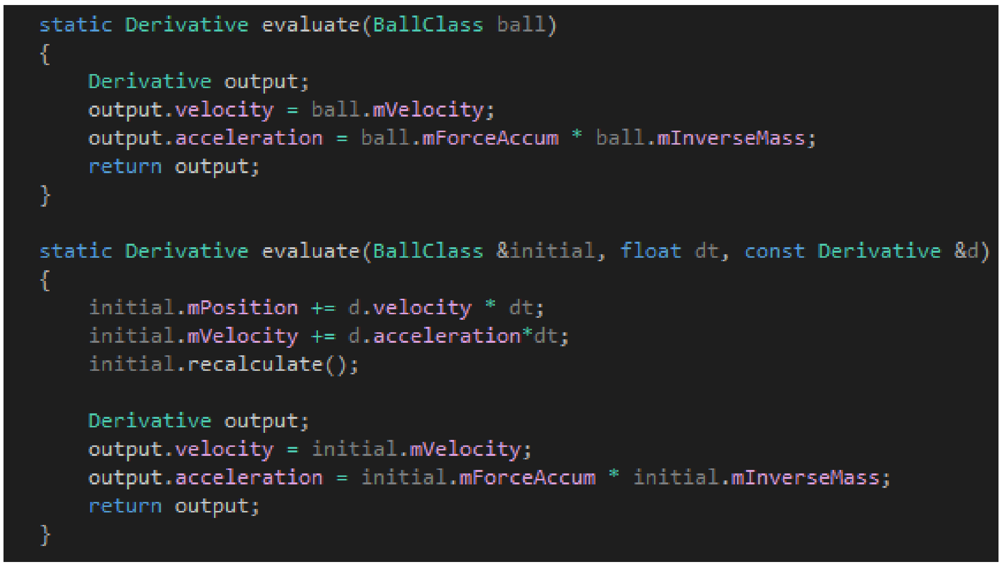
After implement feature on changing timescale of the world, the problem of increasing energy have been found frequently. Therefore Improved Euler Integration and Runge-Kutta Integration have been implemented for having a better physics integration in comparison.

Below diagram has shown implementation of Improved Euler Integration:

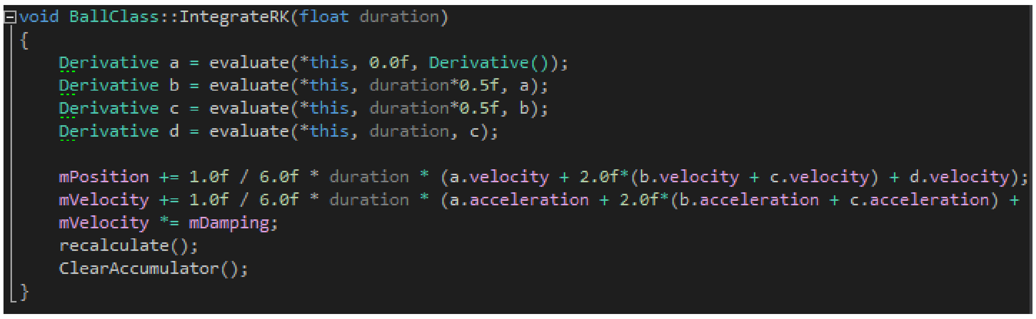


```cpp
void BallClass::Integrate(float duration)
{
    assert(duration > 0.0);

    this->mLastPosition = this->mPosition;

    // Update linear position.
    this->mPosition += (this->mVelocity + this->mLastVelocity)/2 * duration;
    this->mPosition += this->mAcceleration * duration * duration * 0.5f;

    // Work out the acceleration from the force.
    SimpleMath::Vector3 resultingAcc = this->mAcceleration;
    resultingAcc += this->mForceAccum *this->mInverseMass;

    this->mLastVelocity = this->mVelocity;
    //Update linear velocity from the acceleration
    this->mVelocity += resultingAcc * duration;

    //Impose drag
    this->mVelocity *= powf(this->mDamping, duration);
    //mVelocity *= mDamping;

    //Clear the forces
    ClearAccumulator();
}
```

Below diagram has shown implementation of Runge-Kutta Integration:



```cpp
static Derivative evaluate(BallClass ball)
{
    Derivative output;
    output.velocity = ball.mVelocity;
    output.acceleration = ball.mForceAccum * ball.mInverseMass;
    return output;
}

static Derivative evaluate(BallClass &initial, float dt, const Derivative &d)
{
    initial.mPosition += d.velocity * dt;
    initial.mVelocity += d.acceleration*dt;
    initial.recalculate();

    Derivative output;
    output.velocity = initial.mVelocity;
    output.acceleration = initial.mForceAccum * initial.mInverseMass;
    return output;
}
```



```cpp
void BallClass::IntegrateRK(float duration)
{
    Derivative a = evaluate(*this, 0.0f, Derivative());
    Derivative b = evaluate(*this, duration*0.5f, a);
    Derivative c = evaluate(*this, duration*0.5f, b);
    Derivative d = evaluate(*this, duration, c);

    mPosition += 1.0f / 6.0f * duration * (a.velocity + 2.0f*(b.velocity + c.velocity) + d.velocity);
    mVelocity += 1.0f / 6.0f * duration * (a.acceleration + 2.0f*(b.acceleration + c.acceleration) +
    mVelocity *= mDamping;
    recalculate();
    ClearAccumulator();
}
```

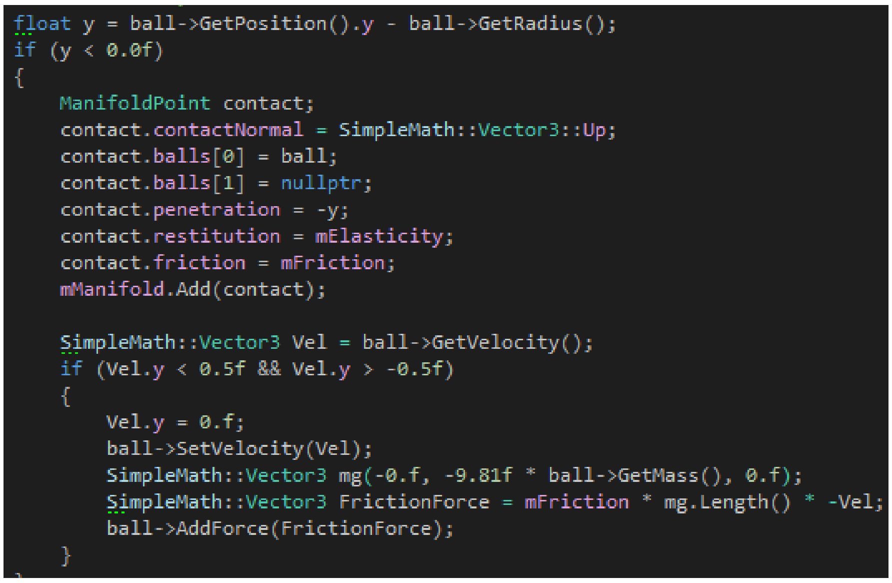
Both Integration have demonstrate a more stable simulation. However considering the system would be used for simulating large amount of balls at te same time, Improved Euler Integration would be a more suitable solution because of its faster calculateion process. Runge-Kutta Integration would need to calculate four times of Integration for each motion update. The simulation loop could be very slow when the system has 10000 balls to simulate. Therefore, Improved Euler Integration has been chosen.

## Collision Detection

The collision detection has been divided into 3 parts: Ground-Ball Collision, Wall-Ball Collision and Ball-Ball Collision. When a collision has been found, the collision would be added to Collision Manifold to be resolved later. By using Collision Manifold, all collision could be handled at the same elapsed time. This could benefit on resolving interpenetration between objects during collision response.

### Ground-Ball Collision Detection

By taking the y position of the ball minus the radius of the ball, the ball is colliding with the ground if the result is equal or less zero. If the result is less than zero, there would be a collision and a penetration between the ball and the ground. After the collision has been detected, the collision point would be added to the manifold as shown below:

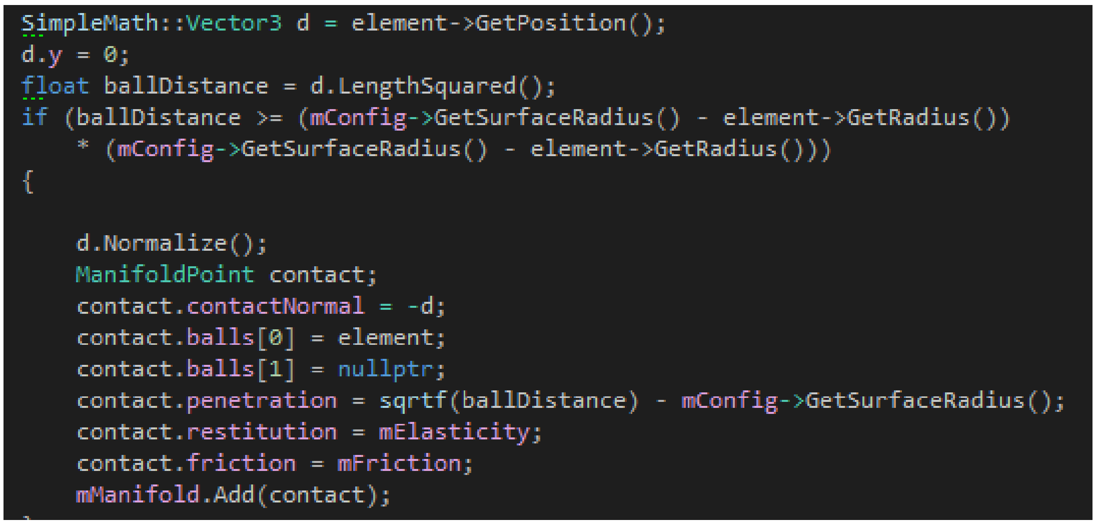


```
float y = ball->GetPosition().y - ball->GetRadius();
if (y < 0.0f)
{
    ManifoldPoint contact;
    contact.contactNormal = SimpleMath::Vector3::Up;
    contact.balls[0] = ball;
    contact.balls[1] = nullptr;
    contact.penetration = -y;
    contact.restitution = mElasticity;
    contact.friction = mFriction;
    mManifold.Add(contact);

    SimpleMath::Vector3 Vel = ball->GetVelocity();
    if (Vel.y < 0.5f && Vel.y > -0.5f)
    {
        Vel.y = 0.f;
        ball->SetVelocity(Vel);
        SimpleMath::Vector3 mg(-0.f, -9.81f * ball->GetMass(), 0.f);
        SimpleMath::Vector3 FrictionForce = mFriction * mg.Length() * -Vel;
        ball->AddForce(FrictionForce);
    }
```

The contact normal would always be upward. The first collision object would be the colliding ball. Second colliding object would be empty to notify the ball is colliding with a scenery object. The penetration between would also been recorded for being resolved later. In addition, if the ball is colliding with the plane and rolling on the surface, Frictional force would also be applied to the ball as shown in the above diagram.

## Wall-Ball Collision Detection

The collision detections would be calculated as shown in diagram below:



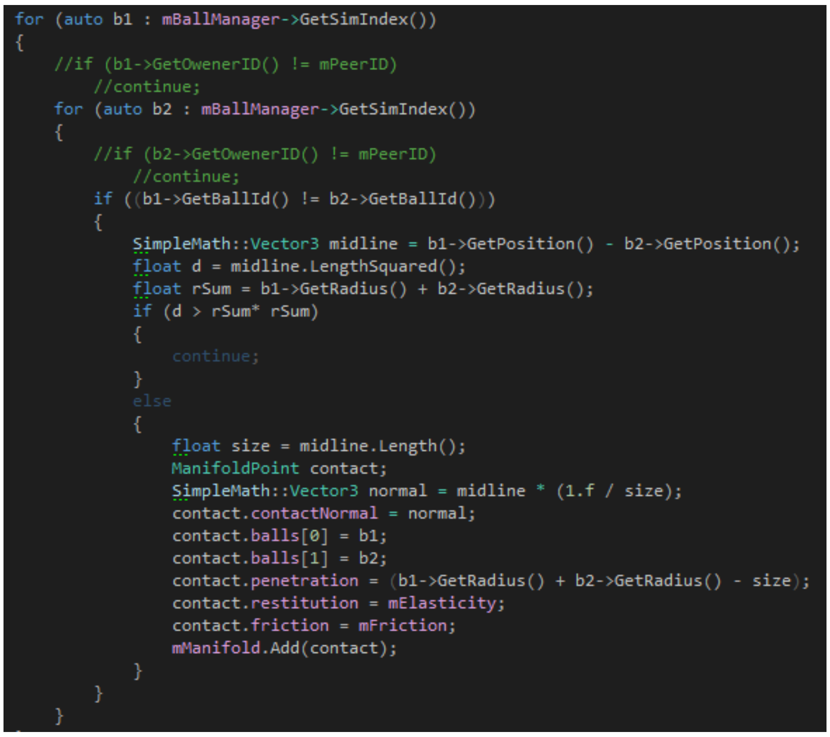
```
SimpleMath::Vector3 d = element->GetPosition();
d.y = 0;
float ballDistance = d.LengthSquared();
if (ballDistance >= (mConfig->GetSurfaceRadius() - element->GetRadius())
    * (mConfig->GetSurfaceRadius() - element->GetRadius()))
{

    d.Normalize();
    ManifoldPoint contact;
    contact.contactNormal = -d;
    contact.balls[0] = element;
    contact.balls[1] = nullptr;
    contact.penetration = sqrtf(ballDistance) - mConfig->GetSurfaceRadius();
    contact.restitution = mElasticity;
    contact.friction = mFriction;
    mManifold.Add(contact);
```

1. Get the position of the ball and set the y value to zero for comparing the distance on x-z plane in later calculation.
2. Obtain the squared magnitude of the modified ball position which is the squared length from the centre point of the surface.
3. Comparing the squared length from step 2 and the squared result of surface radius minus ball radius
4. If the squared length from step 2 is larger or equal to the other side, there is a collision and collision point would be added to the manifold as shown in the diagram.

By using the squared magnitude in step 2 and 3, calculation of squared root could be avoided and result in a faster process since squared root calculation are very slow comparing to making an variables multiply to itself.

## Ball-Ball Collision Detection

The collision detections would be calculated as shown in diagram below:



```
for (auto b1 : mBallManager->GetSimIndex())
{
    //if (b1->GetOwenerID() != mPeerID)
        //continue;
    for (auto b2 : mBallManager->GetSimIndex())
    {
        //if (b2->GetOwenerID() != mPeerID)
            //continue;
        if ((b1->GetBallId() != b2->GetBallId()))
        {
            SimpleMath::Vector3 midline = b1->GetPosition() - b2->GetPosition();
            float d = midline.LengthSquared();
            float rSum = b1->GetRadius() + b2->GetRadius();
            if (d > rSum* rSum)
            {
                continue;
            }
            else
            {
                float size = midline.Length();
                ManifoldPoint contact;
                SimpleMath::Vector3 normal = midline * (1.f / size);
                contact.contactNormal = normal;
                contact.balls[0] = b1;
                contact.balls[1] = b2;
                contact.penetration = (b1->GetRadius() + b2->GetRadius() - size);
                contact.restitution = mElasticity;
                contact.friction = mFriction;
                mManifold.Add(contact);
            }
        }
    }
```

1. Check IDs of both balls to ensure they are not the same ball
2. Calculate the squared distance between the positions
3. Calculate the sum of the radius
4. Comparing the squared distance from step 2 and the squared of the sum from step 3
5. If the squared distance is smaller or equal to the squared of the sum from step 3, a collision has been detected and collision point would be added to the Manifold.

## Collision Response

After Collision Detection, the Collision Manifold would be filled with different contact points. In order to resolve them correctly, the contact point with most negative separating velocity would be resolve first. This would avoid unnecessary work when an object has two contacts points with another same object. If one of the collision has been processed, the separating velocity would be position and not need any further processing.

The contact point resolution order would be:

1. Check the separating velocity of each contact point in the Manifold and look for the contact with most negative value.
2. If the most negative value is greater than or equal to zero, it indicated that no more contacts need to be resolved. Otherwise proceed to Step 3.
3. Process the contact point with the most negative value.
4. Repeat to Step 1 if there is available iterations left.

Available iterations has been used to prevent spending too many time on processing collisions for one simulating loop. If the iteration has run out, Collision Response will halt and progress to next simulation loop.

## Resolve Contact point

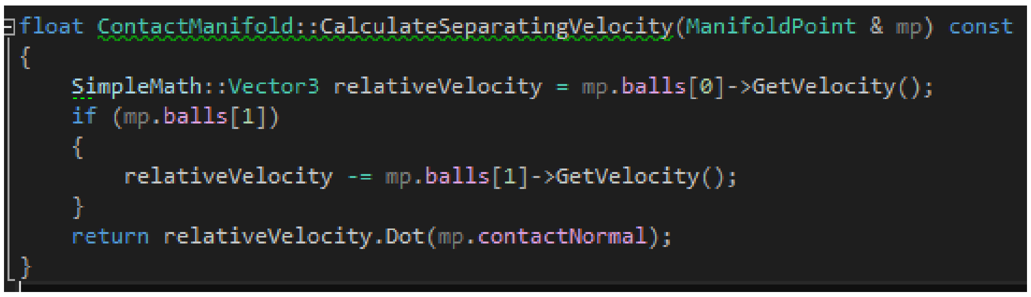
Contact point would be resolved as following order:

1. Resolve Velocity
2. Resolve Interpenetration

### *Resolve Velocity*

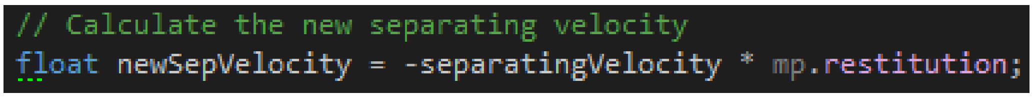
To resolve the velocity, following step would be implemented:

1. Calculating the separating velocity in the direction of contact as below



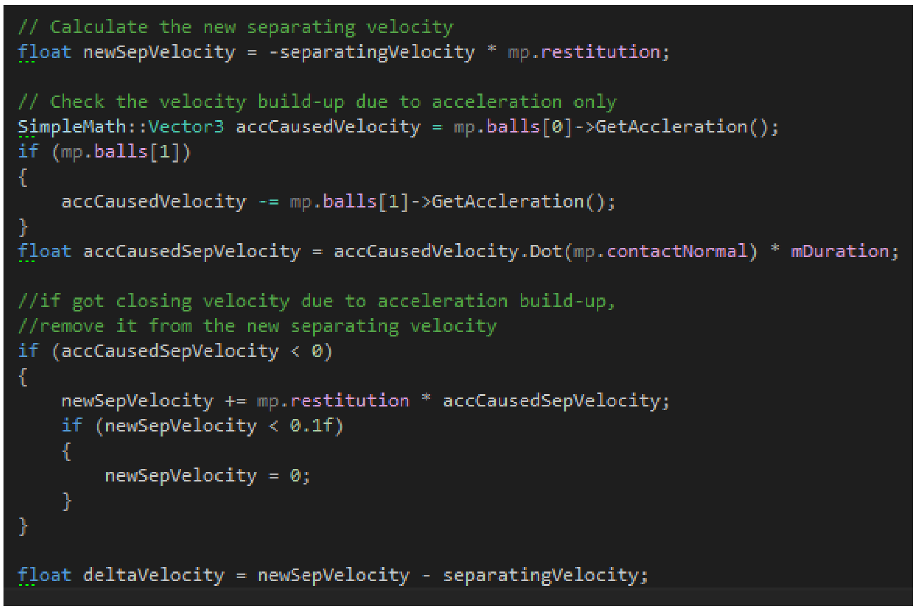
```cpp
float ContactManifold::CalculateSeparatingVelocity(ManifoldPoint & mp) const
{
    SimpleMath::Vector3 relativeVelocity = mp.balls[0]->GetVelocity();
    if (mp.balls[1])
    {
        relativeVelocity -= mp.balls[1]->GetVelocity();
    }
    return relativeVelocity.Dot(mp.contactNormal);
}
```

2. If the separating velocity is larger than zero, the contact could be stationary or separating. Therefore no impulse is required to apply on the colliding objects. Otherwise proceed to next step.
3. Calculating the new separating velocity with elasticity



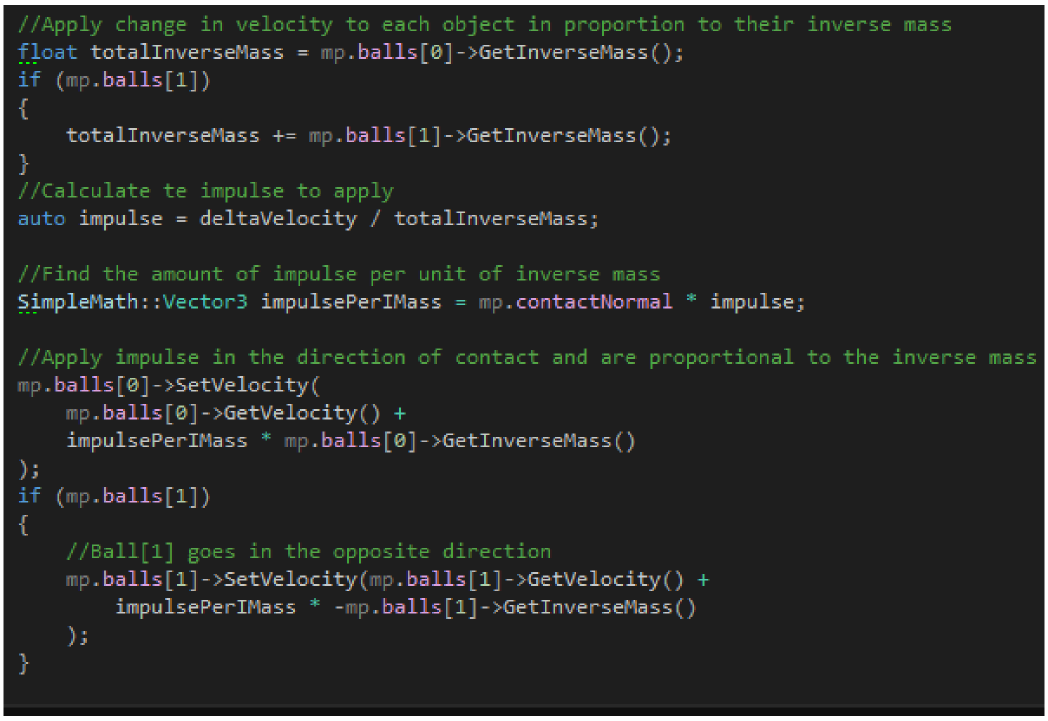
```cpp
// Calculate the new separating velocity
float newSepVelocity = -separatingVelocity * mp.restitution;
```

4. Check if the velocity is built up from acceleration only. If true, remove it from the new separate velocity. This can help to reduce the vibration caused from constant gravity acceleration.



```cpp
// Calculate the new separating velocity
float newSepVelocity = -separatingVelocity * mp.restitution;

// Check the velocity build-up due to acceleration only
SimpleMath::Vector3 accCausedVelocity = mp.balls[0]->GetAccleration();
if (mp.balls[1])
{
    accCausedVelocity -= mp.balls[1]->GetAccleration();
}
float accCausedSepVelocity = accCausedVelocity.Dot(mp.contactNormal) * mDuration;

//if got closing velocity due to acceleration build-up,
//remove it from the new separating velocity
if (accCausedSepVelocity < 0)
{
    newSepVelocity += mp.restitution * accCausedSepVelocity;
    if (newSepVelocity < 0.1f)
    {
        newSepVelocity = 0;
    }
}

float deltaVelocity = newSepVelocity - separatingVelocity;
```

5. Apply change in velocity to each object in proportion to their inverse mass.



```
//Apply change in velocity to each object in proportion to their inverse mass
float totalInverseMass = mp.balls[0]->GetInverseMass();
if (mp.balls[1])
{
    totalInverseMass += mp.balls[1]->GetInverseMass();
}
//Calculate te impulse to apply
auto impulse = deltaVelocity / totalInverseMass;

//Find the amount of impulse per unit of inverse mass
SimpleMath::Vector3 impulsePerIMass = mp.contactNormal * impulse;

//Apply impulse in the direction of contact and are proportional to the inverse mass
mp.balls[0]->SetVelocity(
    mp.balls[0]->GetVelocity() +
    impulsePerIMass * mp.balls[0]->GetInverseMass()
);
if (mp.balls[1])
{
    //Ball[1] goes in the opposite direction
    mp.balls[1]->SetVelocity(mp.balls[1]->GetVelocity() +
        impulsePerIMass * -mp.balls[1]->GetInverseMass()
    );
}
```

### *Resolve Interpenetration*

To resolve the Interpenetration, following step would be implemented:

1. Check if the penetration is larger than zero. If yes, proceed to next step.
2. Calculate the sum of Inverse Mass of both object
3. Calculate amount of penetration resolution per unit of inverse mass
4. Calculate the amount of movement need to made base on each object inverse mass.
5. Adjust the position of the objects by using results from step 4

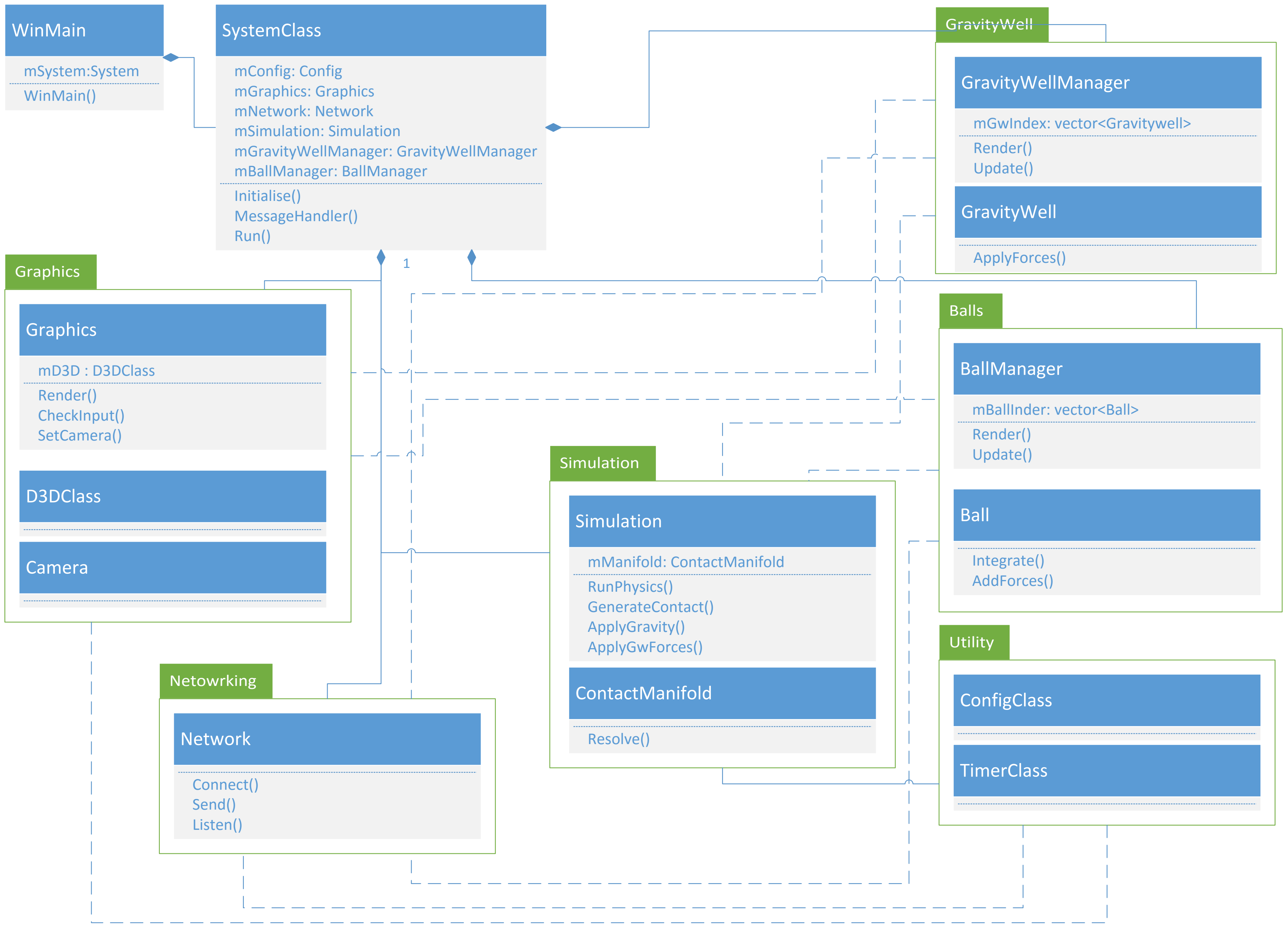
The adjusting would modify the position of the balls at the manifold. Therefore the position would be updated for the rest of contact points as well. This could prevent moving object and cause unhandled penetrations.

## Bibliography

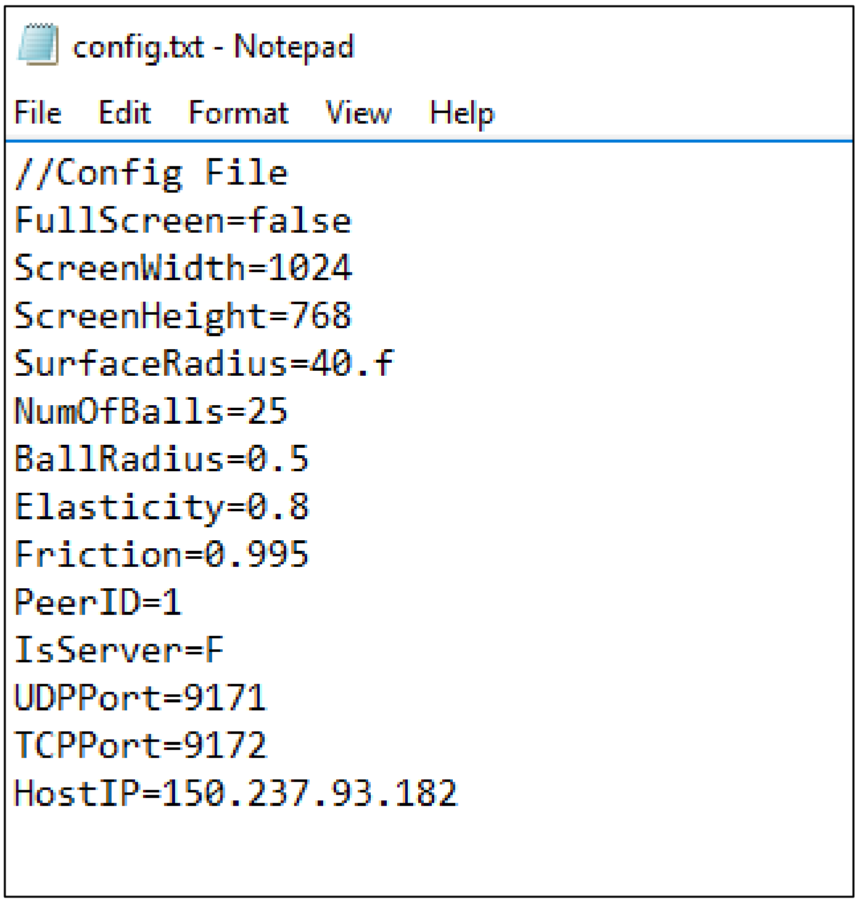
Hall, B. ". (n.d.). *Beej's Guide to Network Programming*. Retrieved from http://beej.us/guide/bgnet/output/html/singlepage/bgnet.html#select

Millington, I. (2007). *Game Physics Engine Development.* Elsevier Inc.

## Appendix A: UML Class Diagram

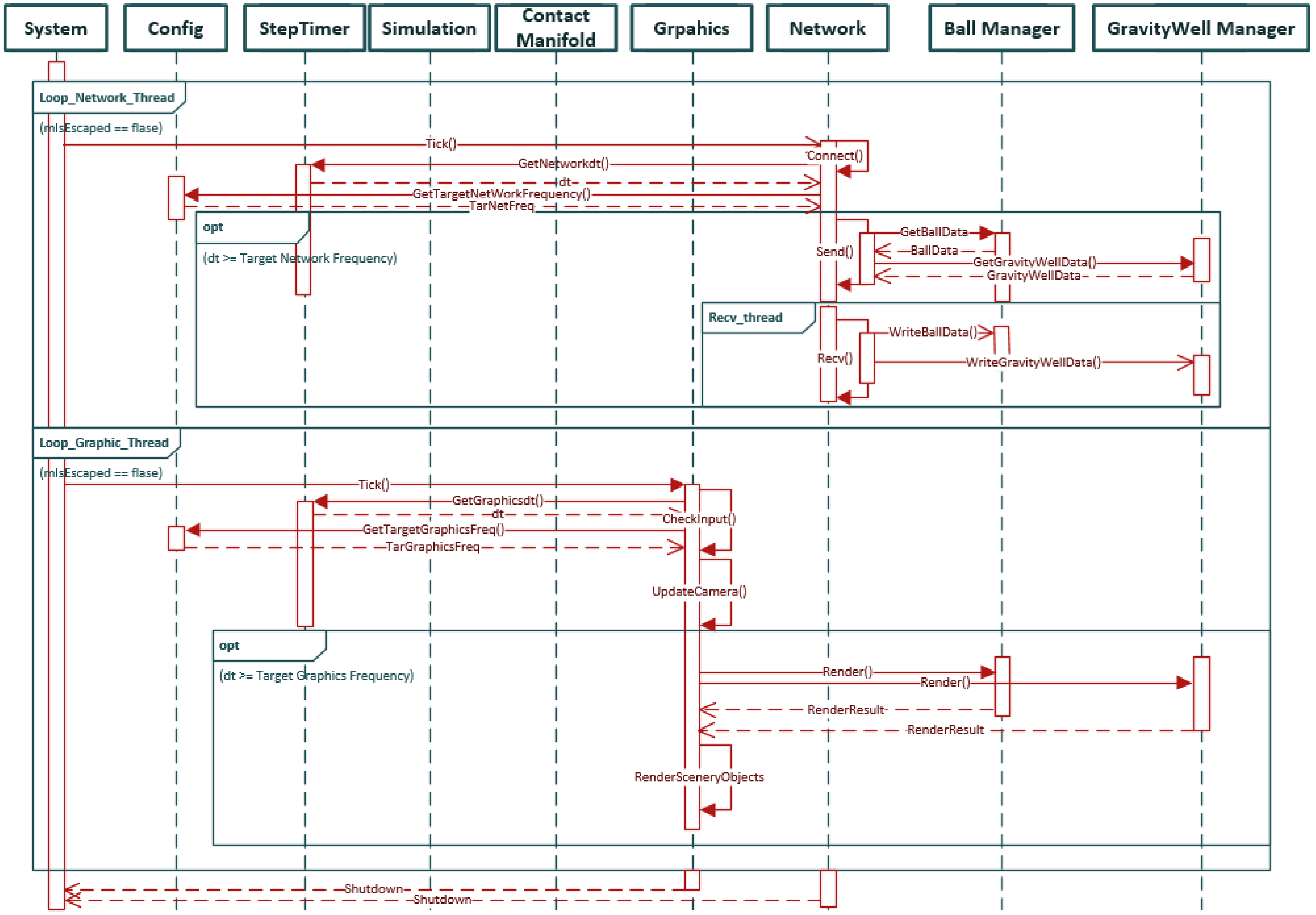


## Appendix B: Configuration file Template

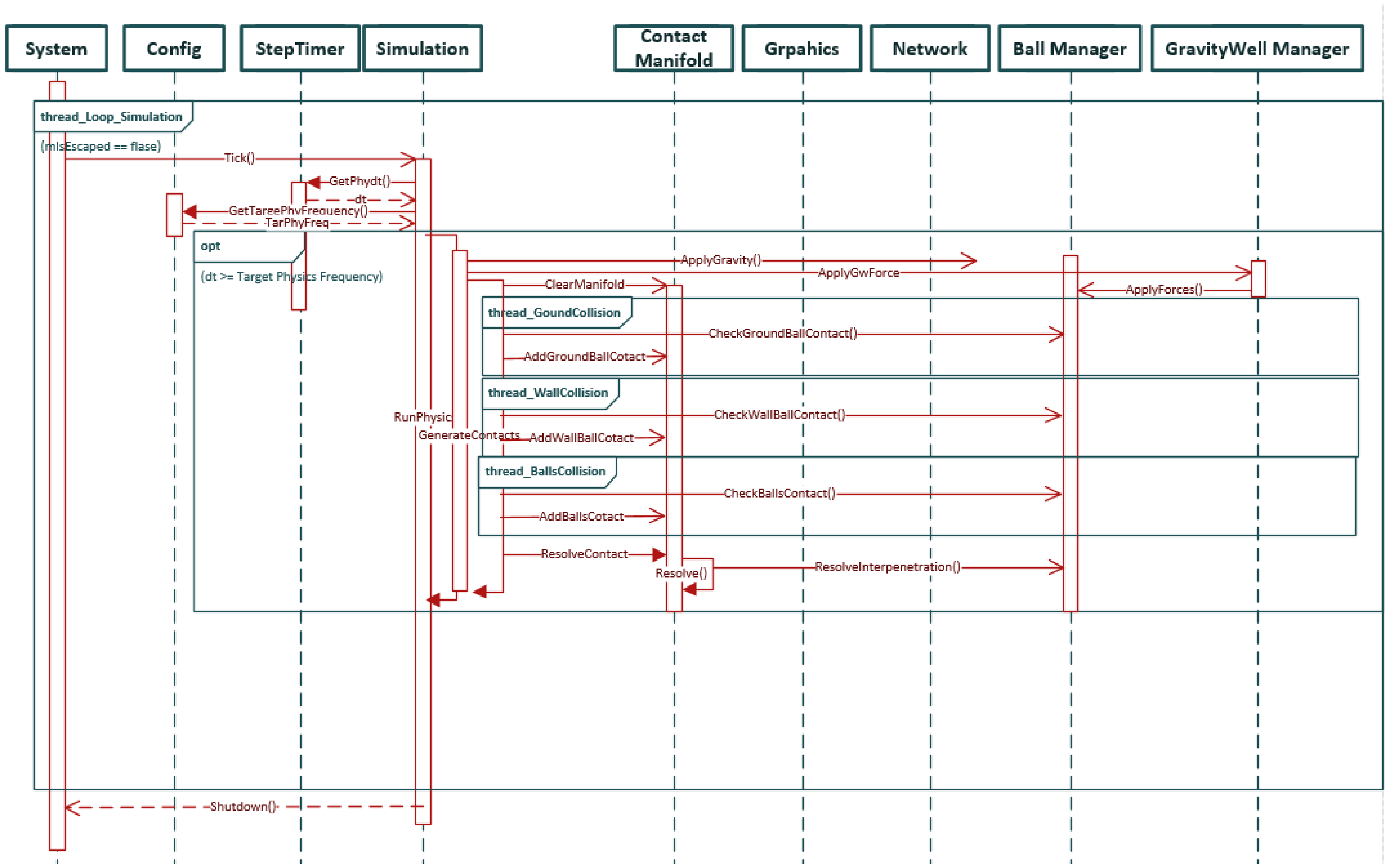


```
//Config File
FullScreen=false
ScreenWidth=1024
ScreenHeight=768
SurfaceRadius=40.f
NumOfBalls=25
BallRadius=0.5
Elasticity=0.8
Friction=0.995
PeerID=1
IsServer=F
UDPPort=9171
TCPPort=9172
HostIP=150.237.93.182
```

## Appendix C: Threading Sequence Diagram – Network & Graphics Class



## Appendix D: Threading Sequence Diagram – Simulations Class


System
Config
StepTimer
Simulation
Contact Manifold
Grpahics
Network
Ball Manager
GravityWell Manager
thread_Loop_Simulation
(mIsEscaped == flase)
Tick()
GetPhydt()
dt
GetTargePhyFrequency()
TarPhyFreq
opt
(dt >= Target Physics Frequency)
ApplyGravity()
ApplyGwForce
ApplyForces()
ClearManifold
thread_GoundCollision
CheckGroundBallContact()
AddGroundBallCotact
thread_WallCollision
CheckWallBallContact()
RunPhysic
GenerateContacts
AddWallBallCotact
thread_BallsCollision
CheckBallsContact()
AddBallsCotact
ResolveContact
Resolve()
ResolveInterpenetration()
Shutdown()