Investigation and application of physics in cue sports

**Jason Luong**

Date: [27th](mailto:quochung8@hotmail.co.uk) April 2010

Supervisor: Dr. Vasos Pavlika

School: Electronics and Computer Science

Keywords: Billiards, Pool, Physics, Mechanics, Simulation, C++, OpenGL

This report is submitted in partial fulfillment of the requirements for the BSc(Hons) Games Design Degree at the University of Westminster.

Abstract

This paper discusses a method of implementing a physically accurate simulation of a game of billiards. In order to do so, an understanding of the type of physics involved in cue sports is required. This paper can therefore be seen as the research and implementation of physics in cue sports. Some of the topics discussed were; linear motion, angular motion, conservation of linear momentum, conservation of angular momentum, collision detection, and collision response, including oblique collision response techniques, which were used to determine the collision response between two colliding balls. The implementation phase involves using the physics researched in order to create a small, physically accurate simulation of billiards programmed using the C++ programming language in conjunction with the OpenGL graphics API.

**Table of Contents**

[*Abstract*](#h.gjdgxs)

[Chapter 1 Introduction](#h.30j0zll)

[Aims](#h.1fob9te)

[Objectives](#h.3znysh7)

[Games physics](#h.2et92p0)

[Simulating Billiards](#h.tyjcwt)

[Project overview](#h.3dy6vkm)

[Chapter 2 Research and investigation, physics in billiards](#h.1t3h5sf)

[Fundamental principles](#h.4d34og8)

[Newton’s laws of motion](#h.2s8eyo1)

[Impulse](#h.17dp8vu)

[Momentum](#h.3rdcrjn)

[Energy](#h.26in1rg)

[Collisions](#h.lnxbz9)

[Collision detection](#h.35nkun2)

[Finding the point of contact](#h.1ksv4uv)

[Penetration distance](#h.44sinio)

[Collision response](#h.2jxsxqh)

[Elastic and inelastic collisions](#h.z337ya)

[Coefficient of restitution](#h.3j2qqm3)

[Elastic collisions in one-dimension](#h.1y810tw)

[Elastic collisions in two-dimensions](#h.4i7ojhp)

[Two-dimensional elastic collisions without trigonometry](#h.2xcytpi)

[Chapter 3 Further Research: Angular motion in billiards](#h.1ci93xb)

[Angular displacement](#h.3whwml4)

[Angular velocity](#h.2bn6wsx)

[Angular acceleration](#h.qsh70q)

[Angular kinetic energy](#h.3as4poj)

[Moment of inertia of a system of particles](#h.1pxezwc)

[Torque](#h.49x2ik5)

[Torque and angular acceleration](#h.2p2csry)

[Determining initial angular velocity of a billiard ball](#h.147n2zr)

[Rolling motion of a rigid object](#h.3o7alnk)

[Angular momentum of a particle](#h.23ckvvd)

[Angular momentum of a system of particles](#h.ihv636)

[Conservation of angular momentum](#h.32hioqz)

[Conservation of angular momentum in billiards](#h.1hmsyys)

[Chapter 4 Implementation](#h.41mghml)

[The ColourOGL struct](#h.2grqrue)

[The Vector3 class](#h.vx1227)

[Inline functions](#h.3fwokq0)

[Operator overloading](#h.1v1yuxt)

[The Ball class](#h.4f1mdlm)

[CreateBall method](#h.2u6wntf)

[WorkOutVelocity method](#h.19c6y18)

[UpdateMovingBalls method](#h.3tbugp1)

[BallMoving method](#h.28h4qwu)

[RailCollision method](#h.nmf14n)

[BallCollision method](#h.37m2jsg)

[BallPocketCollision method](#h.1mrcu09)

[BallCollisionResponse method](#h.46r0co2)

[The FSM class](#h.2lwamvv)

[CheckState method](#h.111kx3o)

[SetState and GetState method](#h.3l18frh)

[SetNextBallColour method](#h.206ipza)

[RenderBitmapString method](#h.4k668n3)

[The GameObjects class](#h.2zbgiuw)

[Macros](#h.1egqt2p)

[Enumerators](#h.3ygebqi)

[The SpawnBalls method](#h.2dlolyb)

[The CreatePockets method](#h.sqyw64)

[The Table method](#h.3cqmetx)

[The AimBall and NextBall methods](#h.1rvwp1q)

[The AimDot method](#h.4bvk7pj)

[The AimLine and PowerGauge methods](#h.2r0uhxc)

[The InitObjects method](#h.1664s55)

[The GetCircumPoints method](#h.3q5sasy)

[The WorkOutVel method](#h.25b2l0r)

[The BallMoving , UpdateMovingBalls, RailCollision and PocketCollision methods](#h.kgcv8k)

[The BallStationary method](#h.34g0dwd)

[The BallBallCollision method](#h.1jlao46)

[The BallBallResponse method](#h.43ky6rz)

[The UseFSM method](#h.2iq8gzs)

[Organising code files](#h.xvir7l)

[Problem 1: The program cannot compile as it can no longer find the functions](#h.3hv69ve)

[Problem 2: Multiple definitions where a class is included twice in a source file](#h.1x0gk37)

[Problem 3: The code compiled fine, but there are multiple instances of variables](#h.4h042r0)

[Using a build log](#h.2w5ecyt)

[Using a flowchart](#h.1baon6m)

[The main loop](#h.3vac5uf)

[Error checking](#h.2afmg28)

[Create Window](#h.pkwqa1)

[InitGL and InitObjects](#h.39kk8xu)

[The Game Loop](#h.1opuj5n)

[DrawGLScene](#h.48pi1tg)

[Use Finite State Machine](#h.2nusc19)

[Create Game Objects](#h.1302m92)

[Update Ball Velocities](#h.3mzq4wv)

[Collision Detection/Response](#h.2250f4o)

[KillGLWindow](#h.haapch)

[Conclusions](#h.319y80a)

[References](#h.1gf8i83)

[Appendices](#h.40ew0vw)

[Finding a point on a sphere using spherical coordinates](#h.2fk6b3p)

[Polar coordinate system](#h.upglbi)

[Spherical coordinates](#h.3ep43zb)

[Implementation of spherical coordinates](#h.1tuee74)

[Change log](#h.4du1wux)

[Program source code](#h.2szc72q)

[global.h](#h.184mhaj)

[ColourOGL.h](#h.3s49zyc)

[Vector3.h](#h.279ka65)

[Balls.h](#h.meukdy)

[Balls.cpp](#h.36ei31r)

[FSM.h](#h.1ljsd9k)

[FSM.cpp](#h.45jfvxd)

[GameObjects.h](#h.2koq656)

[GameObjects.cpp](#h.zu0gcz)

[OGL.cpp](#h.3jtnz0s)

[WinMain.cpp](#h.1yyy98l)

# Introduction

## Aims

This project aims to demonstrate an understanding of physics and mathematics in a games environment, to do this, the project aims to implement a physically accurate simulation of a game of nine ball pool. As such, the body of work will focus predominantly on the physics involved in the various collisions that occur between cue, balls, and table, as well as the motion of the balls on the table. Although the cue sport implemented is nine ball pool, the physics in nine ball does not differ from that of other cue sports, as such, the game can be changed to potentially simulate any cue sport.

The project was conducted in the following manner:

* Investigate the type of physics underpinning cue sports
* Implement the physics learnt in a game of billiards
* Produce a small, physically accurate, 2D billiards game

The initial investigation stage of the project involved researching and understanding the physics and mathematical concepts involved in cue sports.

From this initial investigation phase, formulae were to be derived to simulate various events in the game such as; striking the cue ball, collisions with other balls, and collisions with the table rails. Following on from the project proposal, research was conducted into the angular effects of the billiard balls, and again formulae were derived to simulate these effects. However, the research gathered for this was not implemented. Instead it contributed as further research into the subject.

The final stage of the project was to apply what was learnt from the research phase in order to produce a simple 2D simulation of a game of billiards. It is hoped that by applying the research in this implementation stage, I will be able to fully demonstrate my understanding of the subject. Testing will be conducted throughout this final implementation phase to ensure any bugs or glitches will be discovered and dealt with as soon as possible.

## Objectives

The outcome of the initial research is not just to identify what concepts in physics are needed to produce an accurate simulation, but to also understand the fundamental concepts required in producing an accurate simulation. As such, to achieve this aim each concept must be investigated equally and fully.

The next objective is to successfully implement what was learnt from the initial research stage. From the initial research into the physics of billiards it has been determined that the game can be broken down to four physics simulations, they are:

1. Investigate the physics involved in striking the cue ball. Here it is important to understand the concept of impulse as well as Newton’s 2nd law of motion.
2. Investigate the physics involved whilst the ball is in motion. Using Newton’s 2nd law once again, it would be possible to calculate the ball’s velocity under the effects of friction.
3. Investigate the mathematics involved in detecting collisions between other balls or the table rail. By modelling the balls as spheres it will be possible to determine whether they have collided by calculating the distance between their centres.
4. Investigate the physics involved in calculating the outcome of any collisions detected. The most difficult aspect and the part that requires the most math and physics knowledge. Physics and mathematical concepts such as conservation of momentum and energy, vector addition, subtraction and dot products will be required to determine the resulting velocities of objects after collision.

## Games physics

Physics form a fundamental cornerstone in computer games. As games become increasingly realistic the need to accurately simulate the real world increases also. As a reflection of the need to implement physics in games, consider a consequence of one with no physics, objects will be able to pass straight through each other as there are no collision detection algorithms. Indeed, a game with no collision detection would be extremely dull and although this is only one basic scenario, the fact that even simple things such as collision detection requires some use of physics concepts, perhaps underlines the importance of the subject in a games environment more than any other.

Implementing physics in a game not only increases the realism the game provides, but is also, from a performance perspective, a much more efficient way to simulate real world effects. It is much less computationally expensive, for example, to execute an algorithm that performs fluid simulation to recreate curtains blowing in the wind, than it is to motion capture the effect of curtains blowing in the wind. Having this effect simulated using physics has an added flexibility benefit that allows the curtain to adapt to the wind blowing in different directions in a realistic manner that would not be possible if one were to use other techniques such as motion capture.

## Simulating Billiards

Having understood the importance of physics in games, selecting a project where physics played an important part became apparent. Although not original in the slightest, simulating billiards is an ideal way to demonstrate ones grasp of the subject. Billiards involves many physical principles including; conservation of momentum and energy, friction, elastic and inelastic collisions, translational and rotation equations of motion, billiard simulations also require a solid understanding of Newtonian mechanics. In a field as broad as physics, simulating billiards utilises subjects that can be directly applied to many game programming scenarios and, in particular, game physics engines, where Newtonian physics is used extensively to simulate the natural environment.

In the *Théorie Mathématique des Effets du Jeu de Billard* *(Mathematical Theory of Spin, Friction, and Collision in the Game of Billiards)* (Coriolis, 1835) presented an early investigation into the physics of billiards. His work was followed up with more recent work by (Marlow, 1995), (Shepard, 1997) who extended aspects of Marlow’s work, (Stronge, 2000), and (Alciatore, 2008) who extended many aspects of the works of his priors. His work is perhaps the most comprehensive in the field of billiard physics to date, comprising of video clips to demonstrate various principles, as well as many technical proofs to support billiards principles.

## Project overview

The main body of work focuses on various topics in the field of physics.

Beginning with the first chapter, the reader will find an introduction to the fundamental principles underpinning billiards. Concepts such as conservation of momentum and energy and elastic and inelastic collisions are explored. Collision detection and collision response techniques are also discussed in this chapter.

The second chapter comprises of further research performed but which were not implemented in the final build of the program. Angular effects in billiards such as conservation of angular momentum as well as physical concepts such as torque and the moment of inertia are discussed here.

The implementation chapter comprises of the steps taken to apply the physics learnt to produce a realistic simulation. The coding process is explained here and includes the problems encountered during the coding process along with the solutions used to overcome the problems. The coding process is accompanied by flowcharts to demonstrate the program’s execution cycle.

# Research and investigation, physics in billiards

## Fundamental principles

To understand the physics required it is important to begin with the fundamental principles used. In this respect, Newton’s Laws of Motion provides a good starting point as they are very important in the study of mechanics in general.

## Newton’s laws of motion

On July 5th 1687, Sir Isaac Newton put forth his philosophies on mechanics in his *Philosophiæ**Natralis Principia Mathematica.* In this work, Newton stated his three laws of motion which are surmised here:

* *In the absence of a force, a body either is at rest or moves in a straight line at constant speed.*
* *A body experiencing a force F experiences an acceleration a related to F by F=ma, where m is the mass of the body. Alternatively, force is proportional to the time derivative of momentum.*
* *Whenever a first body exerts a force F on a second body, the second body exerts a force –F on the first body. F and –F are equal in magnitude and opposite in direction.*

These laws form the basis for much of the analysis in the field of mechanics. Of particular relevance to this project is Newton’s 2nd Law, which will be examined in greater detail as it is referred to over the course of this document.

## Impulse

Generally speaking, impulse can be defined as an average force applied through time. Specifically it is a vector quantity equal to the change in momentum.

To calculate the impulse of an object given its mass, final speed and the amount of time which an external force is applied is as follows:

**Equation 2.1**

It will be possible to determine the initial speed of an object, such as a billiard ball, given the object’s mass and the external force applied.

For instance, a cue ball of mass 0.15kg is struck by the cue with a force of 10N. The force is applied over a period of 0.2 seconds. The initial speed of the ball is calculated as follows:

**Equation 2.2**

**Equation 2.3**

**Equation 2.4**

**Equation 2.5**

Using this formula, it will be possible to determine the initial speed of the cue ball after it has been stuck. It will not be possible however, to determine the balls direction of travel, i.e. its velocity, as the position from which the cue strikes the ball is not known.

## Momentum

In linear motion, momentum is defined as the product of mass and velocity:

**Equation 2.6**

Where = linear momentum, = mass, = velocity

An important principle of momentum states that the total momentum of a closed system of objects is constant. This law is known as the conservation of momentum. It is therefore true to say that when two objects collide in a closed system, the sum of the momenta of the objects before collision will equal the sum of the momenta of the objects after collision:

**Equation 2.7**

Where are the masses of the objects, are the initial velocities of the objects, are the final velocities of the objects.

## Energy

Energy is a scalar physical quantity that describes the amount of work that can be performed by a force. Like momentum, energy is subject to a conservation law. It is true then to state that the total energy in a closed system remains constant over time, consequently, energy cannot be created nor destroyed, rather it is transformed from one form to another.

There are many forms of energy – kinetic, electromagnetic, gravitational, and thermal – of these different forms; kinetic energy is of the most relevance for this project.

Kinetic energy is a form of energy associated with moving bodies. It is defined as the work needed to accelerate a body of given mass from rest to its current velocity, which is also equal to the energy required to bring the moving body to rest. The formula for linear kinetic energy is:

**Equation 2.8**

Where = linear kinetic energy, = mass, = velocity

Taking into account the conservation of energy, the formula for the conservation of linear kinetic energy in a collision of two bodies is as follows:

**Equation 2.9**

Where = mass, = initial velocity, = final velocity

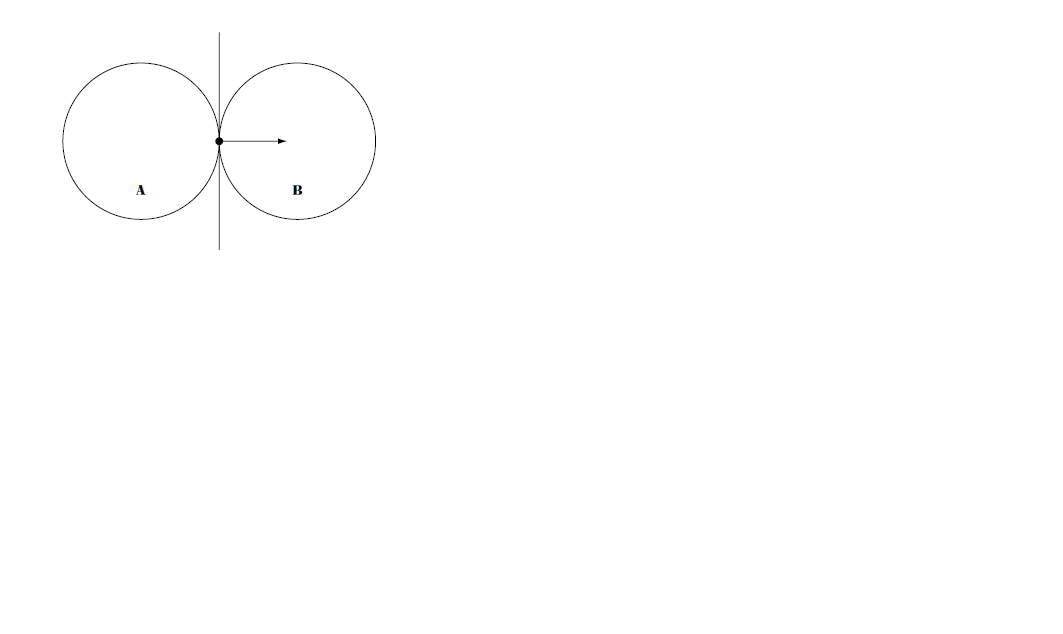
## Collisions

Much of the game’s physics focuses on collisions and collision detection. In billiards, striking the cue ball, a ball rebounding against the table rails and a ball hitting another ball will involve some form of collision and collision response, thus being able to accurately simulate these collisions is paramount.

For the purpose of this project it will be assumed the collisions between billiard balls are nearly perfectly elastic, coefficient of restitution = 0.9. It will also be assumed that the balls are rigid bodies and collisions between them will be treated as such.

### Collision detection

For the purposes of this project, to perform collision response properly, two pieces of information are needed. The first is the point of contact between the two objects A and B. At this point of contact, there is a tangent plane that passes between the two, which also intersects at that point, as shown below. The second is the normal to the tangent plane at the point of contact between A and B.



**Figure 2-1**

The biggest issue in calculating the point of contact is that a simulation such as the one in this project works in time steps (time stepping simulation). This means that in one time step, two objects may be completely separate; in the next they are overlapping. In most cases by the time collision is detected, the objects have already overlapped. Because of this, there is no single point of collision.

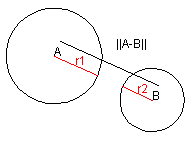
### Finding the point of contact

One possibility for finding the exact point when initial collision occurs is to do a binary search within the time interval. It begins by running the simulation and then checking for collisions. If a collision is detected, and the two objects involved are overlapping, then step the entire simulation back half a time step and check again. If they are still overlapping, then step the entire simulation back a quarter of the original time step, otherwise step the entire simulation forward a quarter of the original time step. Repeat this process until either an exact point of collision is found, which is very unlikely, or until a certain level of iteration is reached at which point a value reasonably close to the point of collision is found.

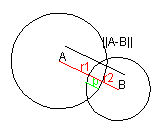
This technique has a few flaws. The most important ones are that it is slow, computationally expensive and rarely finds the exact point of collision between objects.

An alternative and more desirable method is to calculate the overlapping distance and correct the objects position so that it does not overlap. Consider two circles with their centres defined as A and B and their radii as . They collide when the distance between the two centres is less than the sum of their radii.

**Equation 2.10**



**Figure 2-2**



**Figure 2-3**

The distance between the two balls can be found using Pythagoras theorem. The value p in the second image represents the penetration distance.

### Penetration distance

This is an important measure in a time stepping simulation. It is important to know how much two shapes have penetrated each other so that a correction can be made.

In order to calculate p, the sum of the radii must be subtracted from the distance between the two centres.

**Equation 2.11**

The two circles penetrate by a distance p when . In order to correct this, the penetration vector must be found. This is the vector that moves both circles to the point where they just touch. Importantly, it is not just a vector that does this; it is the only vector which corrects the penetration by moving the minimum amount.

This is done by first normalising the distance vector between the centres of the circles and then multiplying the unit direction by the penetration distance.

**Equation 2.12**

**Equation 2.13**

Collision detection between ball and rail is done in a similar fashion.

This technique does have some flaws. The biggest is that if objects are moving fast enough the objects may pass right through each other in-between frames, if this is the case; the collisions will not be detected. Another is that if objects are moving fast enough and most of the first object has penetrated the second, when the penetration distance is calculated the contact point will be on the other side of where it should be. These flaws are mainly due to the time stepping technique used in the project, which shows the limits of this technique.

## Collision response

### Elastic and inelastic collisions

An elastic collision is a collision in which the total kinetic energy of the colliding bodies after a collision is equal total the total kinetic energy before collision. Collisions between billiard balls can be considered near perfectly elastic for the purpose of this project, as kinetic energy lost in the collision is negligible.

An inelastic collision is a collision in which kinetic energy is lost, thus it is the opposite of an elastic collision.

### Coefficient of restitution

In reality collisions are never perfectly elastic nor perfectly inelastic. It is necessary then to identify a relationship to quantify the degree of elasticity of a collision. The coefficient of restitution is the result of the relationship concerned and is given as a fractional value of an object representing the ratio of velocities after and before a collision:

**Equation 2.14**

Where = coefficient of restitution, = initial velocity, = final velocity

For perfectly elastic collisions = 1, for perfectly inelastic collisions = 0.

### Elastic collisions in one-dimension

Consider two objects of equal mass m, with each object having a velocity u before collision and velocity v after collision.

As kinetic energy and momentum is conserved in an elastic collision, it is possible to write the following:

**Equation 2.15**

**Equation 2.16**

To find the velocity of the objects after collision (), one must solve the equations simultaneously. After which one gets the following solutions:

**Equation 2.17**

**Equation 2.18**

The equations for collisions between elastic particles can be modified to use the coefficient of restitution, thus becoming applicable to inelastic collisions as well, and every possibility in-between.

**Equation 2.19**

**Equation 2.20**

It is possible now to determine the final velocities of two objects after one-dimensional elastic collision has taken place. If the two objects of equal mass were billiard balls of mass 0.15kg, with the cue ball having an initial velocity of 10 and the object ball having an initial velocity of 0, the final velocities of the balls can be determined as follows:

**Equation 2.21**

**Equation 2.22**

**Equation 2.23**

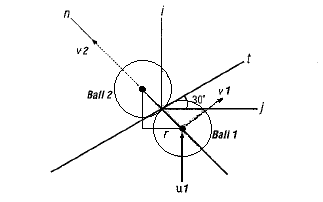
**Equation 2.24**

**Equation 2.25**

**Equation 2.26**

### Elastic collisions in two-dimensions

Although one-dimensional collisions do occur in billiards, a much more likely collision is the two-dimensional collision. Here, rather than colliding straight on, collisions occur at angles. (Bourg, 2002) suggested the following method to determine elastic collisions in two-dimensions. Consider the following collision:



**Figure 2-4**

Both balls are 6.1cm in diameter and weigh 156g. Assume the collisions are perfectly elastic. If the velocity of ball 1 when it hits ball 2 is in the x-direction, we can calculate the velocity of each ball after collision, assuming this is a frictionless collision.

Using the conservation of energy and conservation of momentum law we will break down the two-dimensional collision into two simpler one-dimensional collisions. Because it is assumed the balls are rigid bodies in a frictionless environment, the only transfer of momentum occurs along the single point of contact which also happens to be the line connecting the balls centres.

**Equation 2.27**

**Equation 2.28**

**Equation 2.29**

Where n is the unit normal vector, r is the ball radius and & represent unit vectors in the x & y axis respectively. The next step is to calculate the relative normal velocity between the balls at the instant of collision.

**Equation 2.30**

**Equation 2.31**

**Equation 2.32**

As ball 2 is initially at rest, = 0. Applying the principle of conservation of momentum in the normal direction.

**Equation 2.33**

Since we can write:

**Equation 2.34**

As ball 2 is initially at rest,

**Equation 2.35**

**Equation 2.36**

To solve for these velocities, use the equation for the coefficient of restitution discussed earlier:

**Equation 2.37**

Where

**Equation 2.38**

**Equation 2.39**

**Equation 2.40**

**Equation 2.41**

**Equation 2.42**

**Equation 2.43**

**Equation 2.44**

**Equation 2.45**

**Equation 2.46**

Since the collision is frictionless, there is no impulse acting in the tangential direction. This means that momentum is conserved in that direction too and that the final tangential speed of ball one is equal to its initial tangential speed, which in this case is 5 (10). Since ball two had no initial tangential speed, its velocity after impact lies solely in the normal direction. Converting these results back to x-y coordinates instead of normal and tangential coordinates gives the following velocities after impact for each ball.

**Equation 2.47**

**Equation 2.48**

While the solution above is perfectly adequate for the problem given, there are several issues which make it less than desirable for the purpose of this project. Firstly, the angle of the collision is explicitly given. This simplifies the solution a great deal, however in a real time simulation of two-dimensional elastic collisions the angle will not be known and will have to be calculated.

The angle of collision can be calculated using trigonometry and thus a solution can be found without explicitly knowing the angle. The trigonometry required however, is complicated, tedious and is computationally expensive.

An alternative method exists to calculate two-dimensional elastic collisions without the use of complex trigonometry; this method uses the mathematical concepts of vectors to solve the collisions.

## Two-dimensional elastic collisions without trigonometry

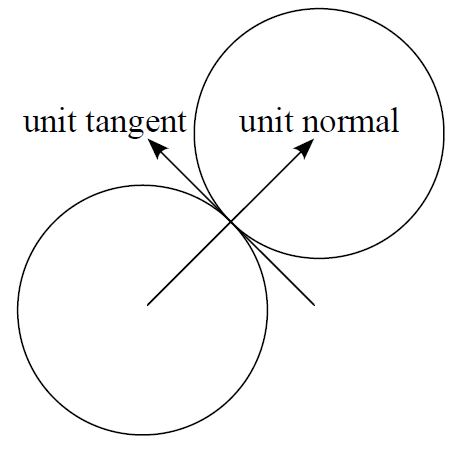
In mathematics, a vector is a geometric object that has both a magnitude and direction. Vectors play an important role in physics, the velocity and acceleration of a moving object and forces acting on a body are all described by vectors.

Vectors are commonly represented in two ways. One way is to specify a magnitude and an angle. The other is to specify the components (typically in the x, y and z directions). The two forms represent the same information, but for some tasks one form may be more convenient than the other.

The process of solving the elastic collision using component vectors is essentially the same as the method used above with trigonometry; the difference is that instead of using trigonometry to find the resultant velocities, mathematical concepts such as dot product and vector subtraction are used in its place. This reduces the actual computations to addition, subtraction, multiplication, division and square roots. This is significantly less computationally expensive then trigonometry, and much more desirable for a computer program.

The goal of the process is to project the velocity vectors of the two objects onto the vectors which are normal and tangent to the surface of the spheres at the point of collision. This gives a normal component and a tangential component for each velocity. As mentioned earlier, the tangential components of the velocities are not changed by the collision because there are no forces along the line tangent to the collision surface. The normal components of the velocities undergo a one-dimensional collision, which can be calculated using one-dimensional collision formulas presented above. Next the unit normal vector is multiplied by the scalar normal velocity after the collision to get a vector which has a direction normal to the collision surface and a magnitude which is the normal component of the velocity after the collision. The same is done with the unit tangent vector and the tangential velocity component. Finally the new velocity vectors are found by adding the normal velocity and tangential velocity vectors for each object.

The first step is to calculate the unit normal and unit tangent vectors. The unit normal vector is a vector which has a magnitude of 1 and a direction that is normal to the surfaces of the objects at the point of collision. The unit tangent vector is a vector with a magnitude of 1 which is tangent to the circles’ surfaces at the point of collision.



**Figure 2-5**

Firstly, find the normal vector. Since the billiard balls are modelled as circles, the normal vector lies along the line connecting the two centres of the circles, thus the normal vector is simply the vector whose components are the difference between the coordinates of the centres of the circles.

Let represent the x and y coordinates of the centres of the circles. The normal vector then is calculated as follows:

**Equation 2.49**

Next, find the unit normal vector,, this is done by dividing by the magnitude of .

**Equation 2.50**

Once the unit normal vector has been calculated, the unit tangent vector () is simply the vector whose x component is equal to the negative y component of the unit normal and whose y component is equal to the x component of the unit normal.

**Equation 2.51**

As the tangential component of the velocity remains unchanged after collision, and the normal velocities can be calculated using the one-dimensional collision formulae. It is necessary then to resolve the velocity vectors into normal and tangential components.

To do this, take the dot product of the unit normal and unit tangent vectors along with the velocity vector.

The dot product is an algebraic operation that takes two vectors and returns a scalar value by multiplying the vectors corresponding values and adding up those products.

Let & be the scalar velocity in the normal and tangent direction respectively of ball one. Similarly let & be the scalar velocity in the normal and tangent direction respectively of ball two. These values are found by calculating the dot product as stated earlier.

**Equation 2.52**

**Equation 2.53**

**Equation 2.54**

**Equation 2.55**

Next, find the tangential velocities after collision. The tangential velocities after collision are the same as the tangential velocities before collision.

**Equation 2.56**

**Equation 2.57**

The next step is to calculate the normal velocities after collision using the one-dimensional collision formulae.

**Equation 2.58**

**Equation 2.59**

The penultimate step is to convert the scalar normal and scalar tangent velocities into vectors. This is done by multiplying the unit normal vector by the scalar normal velocity. This will produce a vector whose magnitude is equal to the normal velocities and whose direction is normal to the point of collision. The process is the same for the tangent velocity vector.

**Equation 2.60**

**Equation 2.61**

**Equation 2.62**

**Equation 2.63**

The final step is to find the final velocity of each ball by adding the normal and tangential components for each object.

**Equation 2.64**

**Equation 2.65**

Thus the final velocity of each ball after collision is given as a vector

.

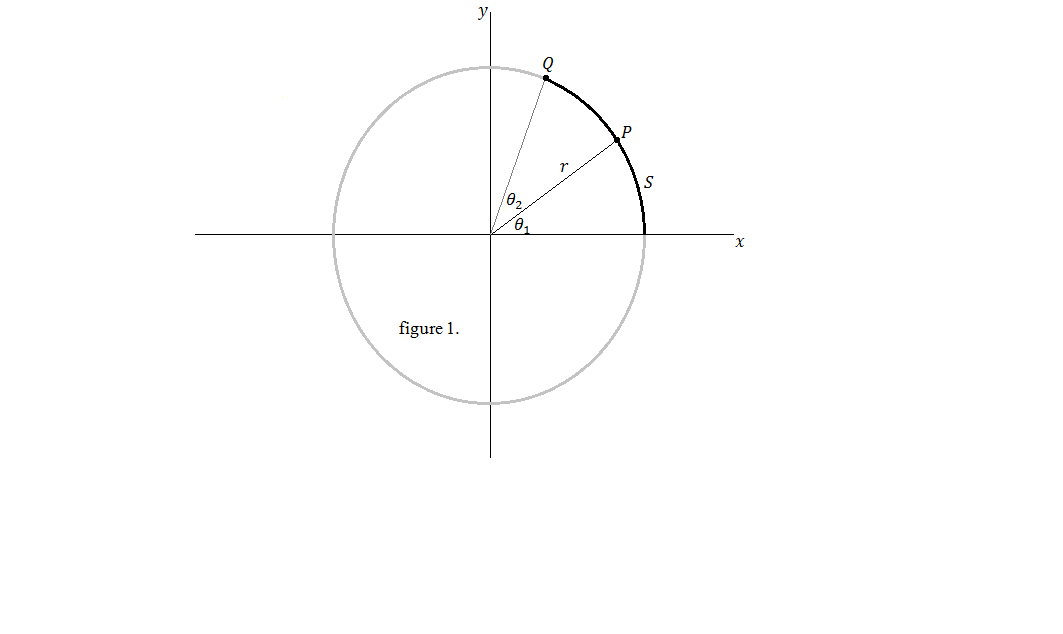
# Further Research: Angular motion in billiards

Angular motion can be a hard subject to comprehend. Understanding terms such as torque and moment of inertia, along with measurements in angles can be difficult. Analogously comparing concepts in linear motion to those in angular motion then can be an excellent way of improving ones understanding. For instance, radians are the preferred units of measurement in angular motion, with its equivalent in linear motion being metres.

If one is to use concepts in linear motion to better understand those in angular motion then it will be useful to know the definitions of the terms being used, rather than going straight into things such as rotational kinetic energy and conservation of angular momentum.

## Angular displacement

Displacement is the shortest direct distance between any two points. It is a vector quantity not to be confused with distance. To better understand angular displacement, consider a billiard ball spinning. A single particle of the ball P is at a fixed distance r from the ball’s centre of mass as shown in figure 3-1.



**Figure 3-1**

The particle P rotates about the CoM in a circle of radius r. The position of P can be represented using its polar coordinates where r is the distance from the CoM and is the angle measured counter clockwise from the positive x-axis. As mentioned earlier, the distance r is fixed, therefore the only coordinate that changes in time is . As P moves from , i.e. from the positive x-axis, to its current position, it moves through an arc of length S, which is related to the angular position through the relationship:

**Equation 3.1**

**Equation 3.2**

Because is the ratio of an arc length S and radius r of a circle, it has no units, however it is common to give the artificial unit radians (rad).

As the particle P travels from P to Q in time t, the change in () is the angular displacement of the particle.

## Angular velocity

In linear motion, average velocity is defined as the rate of change of linear displacement. Likewise average angular velocity is given as the rate of change of angular displacement, and is given by the equation:

**Equation 3.3**

In linear motion, deriving a particle’s position with respect to time will give the particle’s speed at that instant, (Hecker, 1996). Similarly, deriving of particle P with respect to time will give P’s angular speed at that instant.

**Equation 3.4**

## Angular acceleration

Average linear acceleration is defined as the rate of change of linear velocity. Average angular acceleration is defined as the rate of change of angular acceleration and is given by the equation:

**Equation 3.5**

In a similar fashion to instantaneous angular speed, the angular acceleration of particle P at any instant is the derivative of angular speed with respect to time, which is also the second derivative of angular displacement.

**Equation 3.6**

When rotating about a fixed axis, every particle on a rigid body rotates through the same angle and has the same angular velocity and the same angular acceleration. (Halliday et al., 2008). Thus, the rotational properties of particle P describe the rotational properties of the entire rigid object, which in this case, is the rotational property of the billiard ball.

## Angular kinetic energy

The total kinetic energy of a rotating rigid object is the sum of the kinetic energies of its individual particles; this is called its rotational kinetic energy. (Halliday et al., 2008)

Consider a rigid object as a collection of particles, each with a mass and linear velocity. Assume that the billiard ball mentioned earlier is a rigid body consisting of N particles that rotate about the CoM with an angular velocity . Each particle has a linear kinetic energy determined by its mass and linear velocity. If the mass of the particle is and its linear velocity is , its linear kinetic energy is:

**Equation 3.7**

Although every particle in the rigid body has the same angular velocity , the individual linear velocity is dependent on the distance r from the CoM.

The linear velocity of the particle can be determined with the expression . (Halliday et al., 2008).

Substituting this into the equation for linear kinetic energy and taking the sum of all particles that make up the rigid body gives an expression for the rotational kinetic energy:

**Equation 3.8**

As is constant in every particle the expression can be rewritten as:

**Equation 3.9**

## Moment of inertia of a system of particles

If one were to look at the equation for rotational kinetic energy and compare it with the equation for linear kinetic energy, one would realise that they are the same. Linear velocity v has been replaced with angular velocity and mass has been replaced with its angular equivalent , also known as the moment of inertia. The moment of inertia therefore, can analogously be seen as the rotational equivalent of mass.

Mass, specifically inertial mass, is the property of a body that resists changes to its state of linear motion. In the same manner, moment of inertia is the property of a body that resists changes to its state of angular motion.

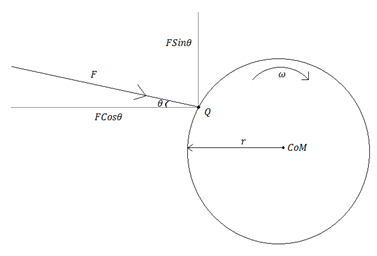
The moment of inertia of a body differs dependant on the shape and density of it. For the purposes of this project, the moment of inertia of a solid sphere is required to represent the billiard balls, and is given as:

**Equation 3.10**

*Where M is the mass and R is the radius.*

## Torque

Torque is defined as the moment of force. It can be seen as the rotational equivalent of force in linear motion. To better understand torque, consider another billiard ball example. A stationary billiard ball is struck by a cue with a force F at a point on the ball Q, figure 3-2 illustrates this.



**Figure 3-2**

The magnitude of the torque associated with the force can be described using the expression:

**Equation 3.11**

Where F is the force applied by the cue, and r is the displacement from the balls CoM to the point where the cue tip touches the ball, Q. The point Q can be determined using the balls spherical coordinate system, see appendix.

The vector quantity of the torque, , is given as the cross product of the vectors and :

**Equation 3.12**

This is true because of the definition of the cross product .

## Torque and angular acceleration

The net torque of a rigid body can be defined as the rate of change of angular momentum. (Halliday et al., 2008)

**Equation 3.13**

Where L is the angular momentum vector. For rotation about a fixed axis, the angular momentum vector can be determined by:

**Equation 3.14**

If angular velocity is the rate of change of angular acceleration, as mentioned earlier, it follows that:

**Equation 3.15**

That is to say that the torque acting on a particle is proportional to its angular acceleration. (Halliday et al., 2008).

In terms of linear and angular motion, can be thought of as the rotational equivalent of Newton’s 2nd Law.

## Determining initial angular velocity of a billiard ball

There is enough information on angular motion now to be able to determine the angular velocity of a billiard ball once it has been struck by the cue. Using the rotational equivalent of Newton’s 2nd Law:

**Equation 3.16**

can be calculated using the expression:

**Equation 3.17**

The vector is the displacement from the balls centre of mass to the point where the cue touches the ball. The vector is the force applied to the ball by the cue. The expressions can be rewritten as:

**Equation 3.18**

Noting that is defined as the rate of change of angular velocity, , the equation above can be rewritten as:

**Equation 3.19**

Rearranging the equation to make the subject gives the following expression:

**Equation 3.20**

Since the cue ball always starts from rest, the initial angular velocity is therefore:

**Equation 3.21**

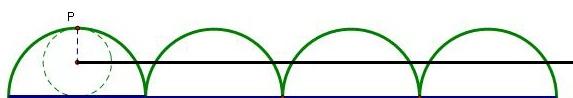
Upon closer inspection, one would notice that the equation above is the same equation as that used for linear impulse, with the linear quantities replaced by their angular equivalents. This makes sense as, like with linear motion, when the cue strikes the ball it imparts an angular and linear impulse to give the ball its initial linear and angular velocities.

## Rolling motion of a rigid object

In general, treating motion of a rigid object rotating about a moving axis is very complex. Fortunately, for the purposes of this project, several factors are true or can be assumed that simplify matters. By restricting the rigid object to a shape with a high degree of symmetry, such as a spherical billiard ball, as well as assuming the rigid object undergoes its rolling motion along a flat surface, such as the surface of a billiards table, simulating rolling motion of a rigid object becomes much less complex.

If an object such as a sphere rolls without slipping on the surface, a simple relationship exists between its rotational and translational motions. (Halliday et al., 2008).

Consider a billiard ball rolling along the table in a straight path as shown below



**Figure 3-3**

The centre of mass of the ball moves in a straight line, the point P on the rim moves in a path known as a cycloid. This shows that the axis of rotation remains parallel to its initial orientation in space.

In pure rolling motion (motion where an object rolls without slipping on the surface) as the ball rotates by angle θ, its translational motion (i.e. its centre of mass) S, moves a linear distance Rθ. This was discussed earlier as the angular displacement.

Therefore, if speed is defined as the rate of change of distance, the linear speed of the centre of mass for pure rolling motion is given by the expression:

**Equation 3.22**

The magnitude of the linear acceleration of the CoM is therefore given by the expression:

**Equation 3.23**

Acceleration is defined as the rate of change of velocity. In this case it is the angular velocity of the ball. This is also known as the angular acceleration (rate of change of angular velocity)

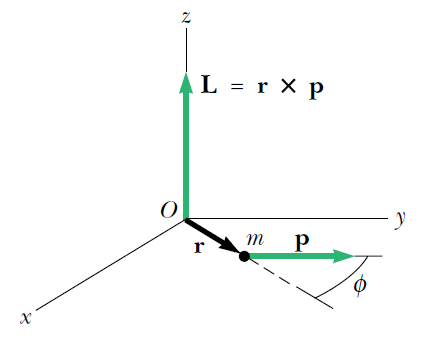
## Angular momentum of a particle

Total kinetic energy of a rolling object is the sum of the rotational kinetic energy about the centre of mass and the translational kinetic energy of the centre of mass (Halliday et al., 2008).

In translational motion, the idea of linear momentum helps us analyse translational motion. Angular momentum serves much the same purpose for angular motion, and can be viewed as the rotational analogue to linear momentum.

Consider a single particle *m* located at a distance from the CoM denoted by the vector . The particle moves with a linear velocity *v*.

The instantaneous angular momentum of the particle relative to the origin *O* is defined as the cross product of the particle’s instantaneous position vector and its instantaneous linear momentum (Halliday et al., 2008).



**Figure 3-4**

That is to say that the angular momentum of a particle, , is given by. Where is the position vector from the origin and is the linear momentum of the particle. Because momentum the magnitude of here is:

**Equation 3.24**

## Angular momentum of a system of particles

The total angular momentum of a system of particles about some point is defined as the vector sum of the angular momenta of the individual particles (Halliday et al., 2008).

**Equation 3.25**

Consider the particle in the figure above as part of a larger rigid object, a billiard ball spinning about the z-axis in this case. As stated earlier, the angular speed of any particle in the ball is ω. The magnitude of the angular momentum, , of a particle with mass about the CoM is given as. Recall also that the linear velocity,, of the *ith* particle can be determined with the expression , thus the magnitude of the angular momentum of a particle can be rewritten as:

**Equation 3.26**

This expression represents the magnitude of the angular momentum of a single particle . Finding the angular momentum of the ball requires finding the sum of the angular momenta of its particles.

**Equation 3.27**

As is constant:

**Equation 3.28**

Recall that the expression is the moment of inertia , therefore the angular momentum of a rigid body is given as:

**Equation 3.29**

## Conservation of angular momentum

The conservation of linear momentum states that the linear momentum of an object remains constant if there are no external forces acting on it (closed system). Similarly, the conservation of angular momentum states that the total angular momentum of a system is constant in both magnitude and direction if the resultant torque acting on the system is zero.

Taking the definition of the net torque on a rigid body as discussed earlier:

**Equation 3.30**

Therefore:

**Equation 3.31**

This result, called the law of conservation of angular momentum, can also be written as:

**Equation 3.32**

*(In a closed system)*

Where is the initial angular momentum and is the final angular momentum.

In the case of a billiards ball, the ball can be assumed to rotate about a fixed axis. If the ball were to rotate about the z-axis, then the following can be written:

**Equation 3.33**

Where is the component of along the relevant axis of rotation and is the moment of inertia about the axis. In this case, the conservation of momentum can be expressed as:

**Equation 3.34**

Where is the initial component of along the relevant axis of rotation and is the final component of along the relevant axis of rotation.

## Conservation of angular momentum in billiards

(Townsend, 2003) Suggested the following method of determining the transfer of angular velocity from one ball to another following a collision between two balls using the concepts of angular momentum and angular velocity.

Angular momentum is defined below:

**Equation 3.35**

Similarly the net torque of a rigid object is defined as:

**Equation 3.36**

In an ideal situation the net torque acting on the ball will be zero and so the angular momentum is conserved and the initial angular velocity before the collision will equal the final angular velocity after collision. In a real world situation however, this isn’t the case. In a billiards simulation a frictional force exists between the balls and so the net torque does not equal zero. The net torque has to be amended therefore to account for this and this is done by calculating the torque applied by this frictional force.

Calculating this torque requires calculating the perimeter velocities of each of the balls at the point of collision; recall that the linear velocity of the particle can be determined with the expression. The direction of the perimeter velocities must also be determined, and is found by subtracting the perimeter velocity of one ball with respect to the other. For example, the perimeter velocity of ball 2 with respect to ball 1 would be given by the equation:

**Equation 3.37**

Where is the linear velocity of the balls at the point of contact.

Now that the perimeter velocities at the point of contact have been determined, the frictional force at the point must be calculated. This is done by using a variation of Newton’s 2nd law of motion, , the equation .

assumes that the only component of the normal force is the force due to gravity, where the normal force . In this case, the magnitude of the frictional force is the product of the mass, acceleration due to gravity and the coefficient of friction. However, in this case, the friction occurs at the point of contact between the two balls, therefore the normal force applied to ball 2 by ball 1 must be determined. This normal force is proportional to the change in momentum of ball 1 (Townsend, 2003).

Finally, the direction of the frictional force applied to ball 2 by ball 1 must be determined. It is found by adding the relative perimeter velocity of ball 2 to its tangential velocity. As the direction is all that is needed, the unit vector of this will be determined by dividing by its magnitude. Thus the frictional force applied to ball 2 can be determined using the equation:

**Equation 3.38**

The value µ represents the coefficient of friction between the two balls and can be arbitrarily assigned a value on the order of 0.1. It will be better however to experimentally determine the value.

The resultant angular velocity of ball 2 can be determined using the rotational equivalent of Newton’s 2nd law, along with :

**Equation 3.39**

**Equation 3.40**

**Equation 3.41**

**Equation 3.42**

**Equation 3.43**

# Implementation

## The ColourOGL struct

OpenGL implements colour using the glColor3f() function. The function takes RGB values in an unfamiliar 0.0-1.0 format, rather than the more familiar 1-255 colour range. The ColourOGL struct is a simple struct that allows the user to enter RGB values in the more familiar 1-255 colour range. This versatile struct was implemented for convenience and will accept both the traditional OpenGL 0.0-1.0 range as well as the 1-255 range.

## The Vector3 class

In the game world, every object has an x, y, and z coordinate. While it is possible to have separate variables to hold each of these positions, it is much more convenient to use mathematical vectors to store this information. The benefits of creating a user defined vector variable include; storing the three position variables of a game object in one easy to maintain variable, vector operations such as dot products and cross products can be performed easily on a vector, operator overloading allows the user to perform common operations, such as vector addition, using familiar syntax that otherwise could not be possible.

### Inline functions

The vector class will be the most heavily used class in the program. Normally during run time, when a vector method is called in the program, it will go into the vector class and execute the method and then return to the point where the method was called to continue its execution cycle. Considering the fact that vectors are used throughout the program, this results in an added time overhead while the program constantly goes back and forth between its current position in the program and the vector class. To overcome this problem, it is beneficial to use inline functions. This is done by having the ‘inline’ prefix before the function, as can be seen in the Vector3 class. What inline does is tell the compiler to perform inline expansion on a particular function. In other words, the compiler will insert the entire body of a function in every place in the code where the function is called. This increases performance of the program but at the cost of having a larger program size, therefore inline functions should only be used on functions that are frequently called, such as the methods in the Vector3 class. It should also be noted that in C++, an inline function must be defined, that is to say that you cannot inline the function prototype and provide the definition elsewhere. This results in the Vector3 class being entirely contained in its header file.

### Operator overloading

If one were to try to add two vectors together using the standard ‘+’ operator, one would find that the compiler will produce an error. This is because the ‘+’ operator only works on a pre defined number of variables, namely the float, int, and double variables. When the compiler encounters an operation such as *Vector1 = Vector2 + Vector3;* it will not know ‘how’ to add a Vector and so it cannot perform the operation. One possible alternative to this is to create a method to perform Vector addition. The method, VecAdd(Vector &a, Vector &b), will take two vectors as arguments, perform the vector addition and then return a result. While this is perfectly adequate, the syntax will be unfamiliar to anyone but the person who created it. A better, more user friendly way to perform this operation is to use operation overloading. Operator overloading is a specific case of polymorphism in which operators such as +, -, =, \*, have different implementations depending on the types of their arguments. By overloading the ‘+’ operator so that compiler knows ‘how’ to add two vectors, any programmer can simply type *Vector1 = Vector2 + Vector3;* and the compiler will recognise the argument type and perform the correct operation.

Because now the position of game objects are stored as mathematical vectors, performing vector specific operations on them is now possible. It is important to be able to find the magnitude or to be able to normalise a vector as well as be able to find the cross and dot product of two vectors. The importance of which will be discussed later.

## The Ball class

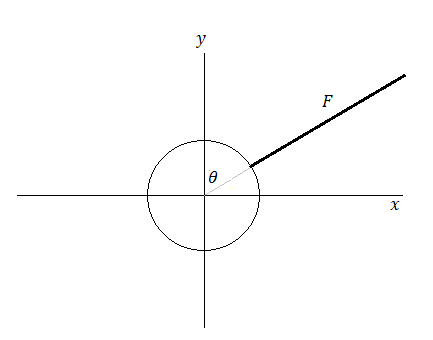
The ball class serves to encapsulate all variables and methods that are associated with a ball in the game. As a ball may have a position and colour, the ColourOGL struct and the Vector3 class are both composed within the Ball class.

### CreateBall method

The CreateBall method takes a ColourOGL colour as a parameter. It works by first creating the ball using the glutSolidSphere() function, it is then assigned a colour using the glColor3f () function. Once this is done, the ball’s position is then continually updated by adding the ball’s displacement to its position. Finally the position of the ball is given by using the glTranslated() function.

### WorkOutVelocity method

The WorkOutVelocity method calculates the ball velocity using the concept of impulse. Firstly, the impulse equation is used to calculate the balls translational speed; here the time that the force is applied is arbitrarily assigned a reasonable value. The next step is to find the vector components of the speed to find the translational velocity. To do this, the angle at which the cue strikes the ball is needed. This value is determined from player input (aiming the cue), the angle is taken from the positive y axis. In order to calculate the translational velocity of the cue ball after it has been struck by the cue, consider the following diagram.



**Figure 4-1**

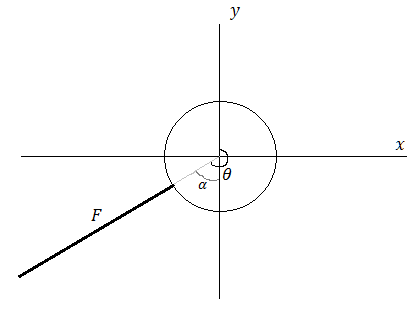
The cue strikes the ball with a force F. The magnitude of F varies dependant on the player input. Using the impulse equation, one may be able to determine the magnitude of F. The components of the vector F can be used to find the balls translational velocity:

**Equation 4.1**

**Equation 4.2**

The translational velocity of the cue ball after it has been struck by the cue is thus .

Should the angle be greater than as is the case below:



**Figure 4-2**

then the angle. Once this angle has been determined then using trigonometry, the x and y components of F can be determined. In this case, the components of F will be given as:

**Equation 4.3**

**Equation 4.4**

Finally, it will be assumed that there will be no loss of energy between the collision between the cue and cue ball. Thus, the velocity at which the cue strikes the ball with is transferred to the ball. This is done by assigning the ball’s velocity to the cue velocity.

### UpdateMovingBalls method

In accordance with Newton’s 1st law, whilst the ball is in motion it will experience a frictional force from the felt of the table, as well as a resistive force from the air, that causes it to slow down and eventually come to rest. It will be assumed that the air resistance will be negligible, and so it may be ignored. The frictional force will act in a direction opposite to the balls direction of travel. The magnitude of this frictional force is given as a variation of Newton’s 2nd law by the equation:

**Equation 4.5**

Where = coefficient of friction, = mass, = acceleration due to gravity

is arbitrarily assigned a reasonable value. The direction of the frictional force is calculated using vectors as follows:

If the direction of the frictional force is opposite the ball’s velocity , then:

**Equation 4.6**

This will give the direction vector of but with a magnitude of . must therefore be normalised so that it has a magnitude of 1:

**Equation 4.7**

The frictional force can then be applied to to find the vector of the frictional force, :

**Equation 4.8**

Once the magnitude and direction of the frictional force has been determined, the velocity of the ball must be updated by applying the friction to the ball velocity.

A final check is needed here, as the ball velocity will approach 0 but rarely ever become 0, thus a check is needed to stop the ball once its velocity is close enough to 0 to be able to consider that the ball has stopped.

### BallMoving method

This simple method sets the balls moving flag to true if the ball’s speed is not 0, otherwise the flag will be set to false.

### RailCollision method

This method determines when the ball has collided with any of the table cushions. If it has, then the method will then correct any overlap by the ball into the rail, this method of correcting the overlap was discussed earlier under the heading *Collisions.* Once any overlap has been corrected then the ball is reflected back off the rail with a loss of some of its velocity. The percentage loss of energy due to the collision has been arbitrarily assigned a reasonable value; however it would be better to empirically determine this energy loss. Finally a safety check is performed in case the ball passes through the rails via the pocket, but does not get potted. This check prevents the ball continuing on past the table and instead instantly pots the ball, as this is what should happen.

### BallCollision method

This method detects collisions between balls and uses the method of collision detection outlined earlier under the heading *Collisions.* The method takes a ball variable as an argument in its parameter and uses this to compare the position of one ball with another. Firstly, the method calculates the distance between the two balls, then corrects the positions if they overlap. Finally, the balls collide if the distance between them is less than the sum of their radii, in this case the method returns true. If there is no collision then the method returns false.

### BallPocketCollision method

Since pockets are also spheres, collision detecting between balls and pockets is handled the same way as collision detection between balls. If a ball collides with a pocket, then the ball’s potted flag is set to true, any operations associated with that flag will then execute.

### BallCollisionResponse method

The BallCollisionResponse method is the most physics intense method in the project. It makes use of most of the topics researched and is the culmination of the research into translational motion. The process of determining the collision response was outlined earlier under the heading *two-dimensional elastic collisions without trigonometry.* The method takes a ball variable as an argument in its parameter. Determining the collision response is done through the following steps:

* The first step to calculating the collision response is to determine the unit normal and unit tangent vectors of the balls that have collided.
* Once this is found the next step is to calculate the normal and tangential components of velocity.
* Once these are found the tangential velocity after collision can be determined, as the tangential component of the velocity remains unchanged after collision.
* The next step is to calculate the normal component of the velocity after collision. This is done by using the one-dimensional elastic collisions equation adapted to use the coefficient of restitution.
* Finally, once the normal and tangential velocities have been found for each ball, their resultant velocities can be determined by adding their respective normal and tangential velocities.

After a collision has occurred, the balls are no longer in contact and so their respective collision flags must be updated.

## The FSM class

While the physics will handle collision detection and collision response it will not be able to handle non-physics aspects of a game of billiards. Aspects such as the rules of the game govern the types of shots that are deemed legal or illegal, winning conditions, as well as player turns. Implementing these rules should be considered just as important as implementing the physics of the game.

One method of implementing these rules is to hard code them into the program. Using a series of if statements, each rule can be checked and an appropriate action can be taken if, say, a rule is broken. Here there is nothing to distinguish the game from one game state to another. The if statements will go through every single rule and apply the rule to the current game state. Although this may work, this method is clearly not ideal.

An alternative is to use a technique which lends itself nicely to this problem. This alternate method is known as a finite state machine (FSM). A FSM is a model of behaviour composed of a finite number of states, transitions between those states, and actions. A state in the context of this project can be something like whose turn is it to shoot, whether there has been a foul, and whether the game has been won. The current game state can be determined through the use of transitions. A transition is what links one state to another, an example of a transition is if a player hits the wrong ball then the state should change to foul. Transitions are implemented in this project as a series of if statements. Finally, actions are code that is implemented once the game is in a game state. For example, the foul game state causes the player turn to change as well as display a foul message.

### CheckState method

The checkstate method uses a switch statement to determine the current game state. Once the game state has been set, the action associated with the state is performed.

### SetState and GetState method

The SetState method takes a state as an argument in its parameter. This state is then assigned to the gamestate variable within the class. The GetState method returns the gamestate variable within the class.

### SetNextBallColour method

This method determines the colour of the next ball game object displayed at the bottom of the screen. It does this by taking the lowest value ball that has not been potted; logically this ball will be the next ball to be potted.

### RenderBitmapString method

This method renders a bitmap font onto the screen. A bitmap is an image file format, so essentially the font being displayed is in the form of a 2D image. The fonts will have no thickness and cannot be rotated or scaled, only translated. The fonts will also always face the viewer. Although this can be seen as a potential disadvantage, for the purposes of this project, it is actually an advantage, as one would not have to worry about orientating the font to face the viewer. This method uses the GLUT function glutBitmapCharacter to display text. Its syntax is as follows:

void glutBitmapCharacter(void \*font, int character);

Parameters:

* Font – the name of the font to use. These are predefined fonts taken from GLUT.
* Character – what to render, a letter, symbol, number, etc.

For example, to render the character ‘a’ one would write:

glutBitmapCharater(GLUT\_HELVETICA\_18, ‘a’);

The next thing to do is to set the x, y and z coordinates of the font itself. This is done by using the OpenGL function glRasterPos3. Its syntax is as follows:

void glRasterPos3(float x, float y, float z);

The parameters indicate the local coordinates for the text.

By combining these two functions it is possible to render text as a string of characters. The function glutBitmapCharacter renders a character at the specified position and then advances the position by the width of a character. Therefore, to render a string of characters, successive calls to glutButmapCharacter will be needed. This is what the function RenderBitmapString does.

## The GameObjects class

The GameObjects class is the largest class in the project. It serves the purpose of handling all actions associated with objects within the game and, as such, composition is used extensively here to help encapsulate all the data required. The methods in this class are divided up into methods that deal with the game objects themselves, spawning and creating objects, and methods that deal with the game logic, such as the physics, FSM, etc.

### Macros

Macros are useful features in C++. They allow the user to create symbolic names for constants. Doing so is good programming practice, and will allow other programmers to quickly identify a constant. Macros are used in this class to define several constants.

### Enumerators

An enumerated type is a data type consisting of a set of named values called elements. Enumeration is a useful tool and can be used to set up collections of named integer constants. Again, use of enumerators is considered good programming practice and so enums are used in this class to define the ball colours and pocket positions. They are used here in place of macros as they are more convenient, user friendly and result in much neater code.

### The SpawnBalls method

This method spawns the balls into the game world using the CreateBall method from the Ball class. As the CreateBall method requires a colour in its parameter, the first step in this method is to set a colour using the ColourOGL struct. Next the ball is created by calling the CreateBall method and passing the colour into its parameter. Finally, should any of the balls be potted, they should respawn to a new position outside the playing area to indicate that they are no longer in play; the appropriate ball status should also be updated in this situation.

### The CreatePockets method

This method is required to set the colour of the pockets. It does this the same way as in the SpawnBalls method.

### The Table method

This method creates the table and the table rails. The table is created using the GLUT function glutSolidCube, in conjunction with the OpenGL function glScaled in order to scale the cube. The table rails are created using the GL\_QUADS function in OpenGL. GL\_QUADS is used to create four sided shapes. In order to do this four points must be specified for OpenGL to draw the quad.

### The AimBall and NextBall methods

These methods create the aiming ball used to indicate topspin and backspin and the next ball that needs to be hit. The balls are created the same way, by using glutSolidSphere in conjunction with glTranslated and glColor to set its position and colour respectively. The next ball differs however; as the colour of the next ball will have to be updated once the current next ball has been potted. To do this the FSM, which is composed within the GameObject class, is used.

### The AimDot method

This method creates the dot seen on the aiming ball and is required to indicate the amount of backspin or topspin applied to the cue ball. The dot is created using the GL\_POINT function along with the glPointSize function to indicate the size of the dot and the glVertex3f function to indicate the position of the dot. The position of the aim dot will vary dependant on player input.

### The AimLine and PowerGauge methods

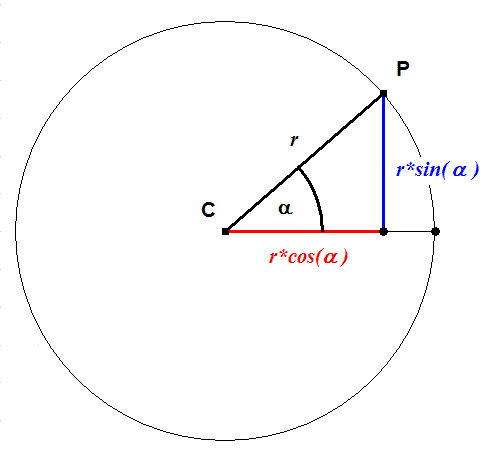
Both these methods use the GL\_LINES function to create lines. GL\_LINES requires several pieces of information, the first is the line width, this is specified by using the glLineWidth function. The second is colour, this is specified by using the glColor3f function. Finally the start and end point of the line is required. The AimLine method draws the aiming line seen on the cue ball, therefore the start point of the aim line must be equal to the location of the cue ball. The end point of the aim line is determined by using the GetCircumPoints method, this method will be discussed later. The power gauge indicates to the player how much force the cue ball will be struck with. The end point of this line is determined by the force the player inputs. Thus, the greater the force, the longer the line.

### The InitObjects method

This method initialises all the game objects at the start of the game. The balls are initialised first, with relevant ball statuses initialised as well as their positions on the table. Next, the positions of the pockets are initialised. Finally, the aim dot position and the coefficient of restitution are initialised.

### The GetCircumPoints method

This method finds a point on the circumference of a circle given its radius, origin and angle. This is done through the use of trigonometry as illustrated below:



**Figure 4-3**

Given the angle α, the coordinates of point P may be determined using the equations:

**Equation 4.9**

**Equation 4.10**

This technique is implemented in the GetCircumPoints method.

### The WorkOutVel method

This method calls the WorkOutVelocity method in the ball class to determine the velocity of the cue ball using the concept of impulse. The method was explained earlier under the *Ball Class* heading.

### The BallMoving , UpdateMovingBalls, RailCollision and PocketCollision methods

The BallMoving, UpdateMovingBalls, RailCollision and PocketCollision methods in the ball class are only able to apply the method to the ball it is associated with. It is required therefore, to have these calls in the GameObjects class so that the methods are called for all balls in the game. This is achieved by iterating through each ball and calling their respective BallMoving, UpdateMovingBalls, RailCollisoin and PocketCollision methods.

### The BallStationary method

This method checks to see when all the balls are stationary, it does this by looping through each ball and checking its moving flag. If all balls are stationary then this method returns true, otherwise it will return false. This is an important method for the FSM and is used extensively to determine when to update game states and apply transitions.

### The BallBallCollision method

This method checks to see when a ball has collided with another ball. To do this the BallCollision method from the Ball class is used. To use the BallCollision method effectively however, one must specify a ball for the method to compare with. One technique to achieve this is to manually compare each ball with another, in other words, one would call the BallCollision method for one ball and compare it with every other ball and then move onto the next ball and repeat the process. Clearly this method is not ideal and is not an efficient way of programming. What needs to be done then is to produce an algorithm which performs this task, but only takes a few lines to code.

The algorithm produced works by iterating through the array of balls and selecting two balls to use the BallCollision method on. By using a nested for loop, one can select one ball to compare with all others before moving onto the next ball to repeat the process. To avoid comparing a ball with itself, a simple check is introduced before the BallCollision method is executed. As a result of using this algorithm, the method of detecting ball-ball collisions was significantly shortened.

If a collision has occurred then the method will return true and a transition in the FSM, which indicates that a ball has been struck, will be updated. Otherwise the method will return false and the FSM transition will again be updated appropriately.

### The BallBallResponse method

This method resolves ball-ball collisions using the elastic collisions equation. This method works in conjunction with the BallBallCollision method. The BallBallCollision method, as stated, compares two balls using the BallCollision method in the Ball class. When the BallCollision method executes and finds that two balls have collided, their respective collision flags will be changed accordingly. It is these collision flags that the BallBallResponse method uses to identify which two balls have collided with another.

The algorithm works by getting the ball ID of the two balls that have collided. Once the ID’s have been determined, then the collision can be resolved by using the BallCollisionResponse method in the Ball class.

The first step then, is to determine the ID of the two balls. This is done by looping through the array of balls and checking their collision flags. If a ball is found to have collided, then the ID of the first ball in the collision is recorded. The loop must continue as, logically, if one ball has collided then it must have collided with another ball further into the array. Once the second ball has been found, the ID of that ball is also recorded. When two balls have been found, the loop must exit so that the collision can be resolved.

In build 3.4.6 of the program, a bug was introduced which occasionally caused incorrect collision response between two balls. The cause of this bug was found in a later build. It was caused when the speed of ballRep1 was less than that of ballRep2. Once the cause was identified, a check was introduced to recognise the problem and resolve it by swapping the ballRep values so that ballRep1 was never less than ballRep2.

At this stage, one could call the BallCollisionResponse method to resolve the collision and finish the method. However, this should not be done yet as an important transition regarding the FSM can be performed before the BallCollisionResponse method is called. The transition applies one of the game rules, which states the first ball the cue ball hits MUST be the ball indicated by the next ball game object. This transition, as with all other FSM transitions, is implemented using if-statements and executes only when the first ball has not been hit yet. Another check must be implemented to ensure that only the first collision involving the cue ball is checked and not subsequent collisions involving the cue ball.

Once this transition has been applied, the BallCollisionResponse method can then be called to resolve the collision between the two balls. Finally, once the collision has occurred then it is assumed that no balls are in contact, and so the collision flag of each ball is changed appropriately.

### The UseFSM method

This method implements the transitions for the finite state machine class to determine the game state. While the states and actions are determined in the FSM class, the transitions between states must occur in the game objects class.

Before the transitions are implemented, there are a couple of issues still to be dealt with. Firstly, the next ball the cue ball must hit first has to be determined and secondly, the FSM must know how many balls have been potted so that it can decide winning conditions and player turns.

The first issue is to determine which ball must be hit first by the cue ball. To achieve this, a simple for loop is used to iterate through the balls, making sure to ignore the cue ball itself. The next ball will be the lowest value ball that has not been potted. Once this ball has been found, the SetNextBallColour method from the FSM class is called to display the next ball.

The second issue is to determine how many balls are left on the table. The solution is a relatively simple one. The for loop iterates through the balls and finds how many balls have been potted. It then updates the current balls variable in the FSM class, which holds a value to indicate the amount of balls left on the table. Once these issues are dealt with, then implementing the transitions can begin.

The transitions must occur logically so that the states do not flip flop between each other. Flip flopping occurs when the action performed by the current state causes a transition to the previous state, which in turn causes a transition back to the current state. Essentially the game gets stuck ‘flip-flopping’ between the two states. By organising the transitions in a logical order this problem can be avoided. The transitions should check for fouling conditions firstly, and then move onto player turns, finally the transitions must check for winning/losing conditions. Once the order has been specified, the FSM must update the game state to reflect any transitions that have occurred during the last shot.

There are three ways to foul in the game:

* If the white ball has been potted then a foul has occurred.
* If the cue ball hit the wrong ball first then a foul has again been committed.
* A foul is committed when the cue ball fails to hit any ball.

Grouping the transitions in a series of if-else statements, each of these foul conditions is checked. If any of these statements are true, then the game state is updated by calling the SetState method in the FSM class.

The next set of transitions to check are the player turns transitions, again they are grouped in a series of if-else statements that are independent from the if-else statements that checks for fouls. The transition checks if the previous play committed a foul, if they did then the players change turns. The transition must also check whether the player keeps their turn. Players keep their turn by potting a ball in a legal manner. It is here that determining how many balls are left on the table is used. If there are fewer balls on the table after a shot is played, then it is clear that the previous player has potted a ball. If the ball was potted in a legal manner, then the previous player keeps their turn, and gets another chance to pot another ball.

Finally, transitions to the game over state are checked. Game over occurs when all balls have been potted, or when the black has been potted in a legal manner. If any of these conditions are met, then the transition to game over will occur through the SetState method.

Once the transitions have been dealt with, the program must update the game state. As fouls can only occur while the balls are moving, the BallStationary method becomes useful here to indicate when the balls are moving. If there is a foul, the game state will reflect this. The update game state will then perform the action associated with the foul, change players and display a foul message. All other transitions occur when the ball is stationary so the BallStationary method is used again to check for this. Once the balls are stationary a call to the CheckState method in the FSM class will perform any actions associated with the current game state.

## Organising code files

While it may fine to code in one source file in small, simple programs, coding in larger projects will require multiple source files. In order to maintain manageable code, it is important to split up code in one source file into multiple smaller, more user friendly source and header files. Aside from keeping code manageable there are several other benefits to using multiple source files:

* It speeds up compilation – considering the fact that most compilers work on one file at a time. If all code is in one file, with potentially over 10,000 lines of code, changing one line of code in that file will cause the compiler to recompile every single line of code in that file. Compare this with having multiple files, where there can be 100’s of lines of code, changing one line of code in one of these files will cause the compiler to recompile far less lines of code.
* Better organisation – placing code into relevant files will make it easier for oneself and any other person to find functions, variables, and other declarations. For example, by having a file called WinMain.cpp and a file called OGL.cpp, it would be clear that all Windows related functions will be contained within WinMain.cpp and all OpenGL related functions will be contained within OGL.cpp.
* Enables code reuse – provided that the code in the files are able to operate independently of each other, such as the case in some of the classes and structs, this lets the code in those files, which may perform specific tasks or enable the user to use user defined variables, be reused in other projects.
* Split tasks among programmers – for particularly large projects, rather than having a single programmer, a team of programmers is usually employed. Each programmer will have specific coding responsibilities and in this situation it is not practical for more than one person to be making changes to a single file at any given time. Therefore multiple files are used so that each programmer can be working on a separate part of the program without affecting other programmers. However, there will still have to be checks done to ensure that programmers are not trying to alter the same file.

In order to implement multiple files, several issues must be addressed.

### Problem 1: The program cannot compile as it can no longer find the functions

The program cannot compile as it can no longer find the functions. This problem is akin to using a function before it is declared, in which case the solution would be to provide a function prototype at the start of the program. Similarly, the solution to this problem is to provide function prototypes in a header file that will be included in any source file that requires those functions. The global header file serves this purpose.

### Problem 2: Multiple definitions where a class is included twice in a source file

This problem arises when multiple header files are included in a source file. If those header files include a common header file, then that common header file will be defined twice, causing a compiler error. The pseudo code below demonstrates this:

|  |  |  |
| --- | --- | --- |
|  | // Header1.h  #include "Header3.h"  class ClassOne { ... };  // Header2.h  #include "Header3.h"  class ClassTwo { ... };  // File1.cpp  #include "Header1.h"  #include "Header2.h" |  |

In this example, Header1 and Header2 both include Header3. Once the compiler reaches File1 it will attempt to include Header1, which in turn includes Header3, once it successfully includes Header1 it will attempt to include Header2, which in turn also include Header3, once the compiler sees this an error will arise.

To overcome this problem, the compiler must be able to check if a file has already been defined, and if it has, the compiler must ignore any subsequent definitions of that file. This is achieved by using inclusion guards. Using the example above once again to demonstrate use of inclusion guards:

|  |  |  |
| --- | --- | --- |
|  | // Header1.h  ***#ifndef header1\_h***  ***#define header1\_h***  #include "Header3.h"  class ClassOne { ... };  ***#endif***  // Header2.h  ***#ifndef header2\_h***  ***#define header2\_h***  #include "Header3.h"  class ClassTwo { ... };  ***#endif***  // Header3.h  ***#ifndef header3\_h***  ***#define header3\_h***  class ClassThree { ... };  ***#endif***  // File1.cpp  #include "Header1.h"  #include "Header2.h" |  |

Here the inclusion guards state that “if header3\_h has not been defined, then define header\_3, else skip to the #endif statement”. This ensures that the definition of header3 will only ever occur once, thus avoiding the multiple definitions error.

### Problem 3: The code compiled fine, but there are multiple instances of variables

This is a linking error and can often be difficult to understand. Doing so requires some knowledge of the linking process which will not be discussed here. Essentially the problem arises when the linker finds two or more instances of a variable or in the object files. The linker then does not know which one is the ‘correct’ one to use. To resolve this issue, it is necessary to declare a variable with the keyword ‘extern’ preceding the declaration. Doing so will allow the variable to be used in multiple source files.

## Using a build log

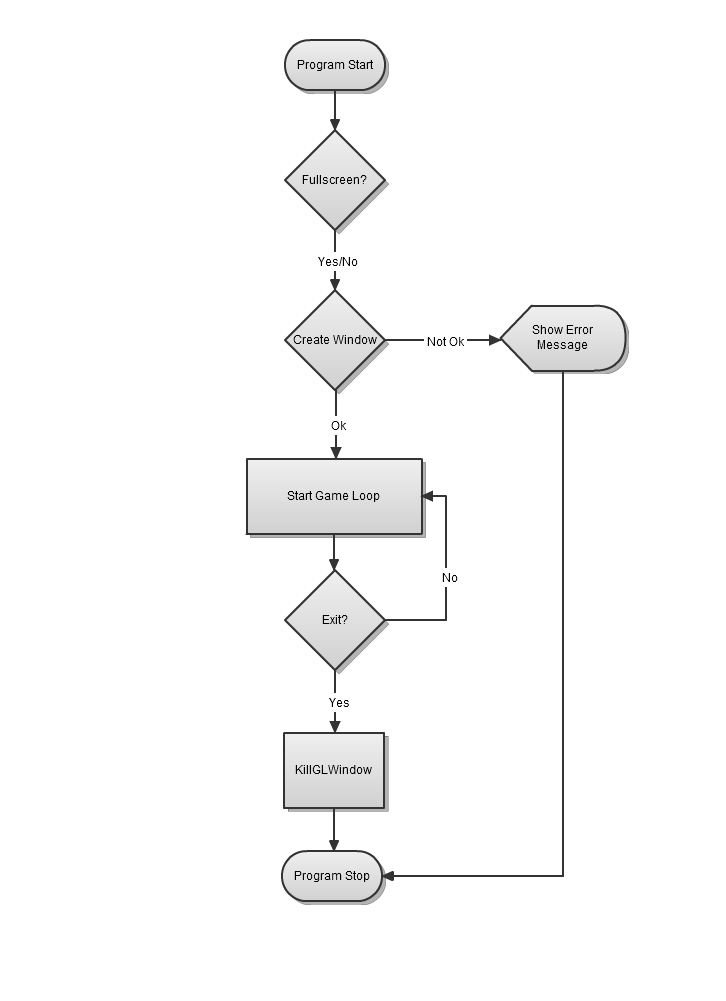
As stated in the aims section, the project was to be tested throughout its development cycle. In order to effectively test for bugs, a change log had to be kept which kept a record of several things:

* Features that were added in the last build
* Features that are bug free
* Features that contained bugs
* Bugs that were fixed in the last build
* Work to be done in the next build

The change log serves as evidence of the testing process and is included in the final build of the program as well as in the appendix.

## Using a flowchart

A flowchart is a common type of diagram, which represents an algorithm or process. By using a flowchart to represent the program that has been implemented, it is hoped that the program becomes clearer and easier to follow.



**Figure 4-4 The main loop**

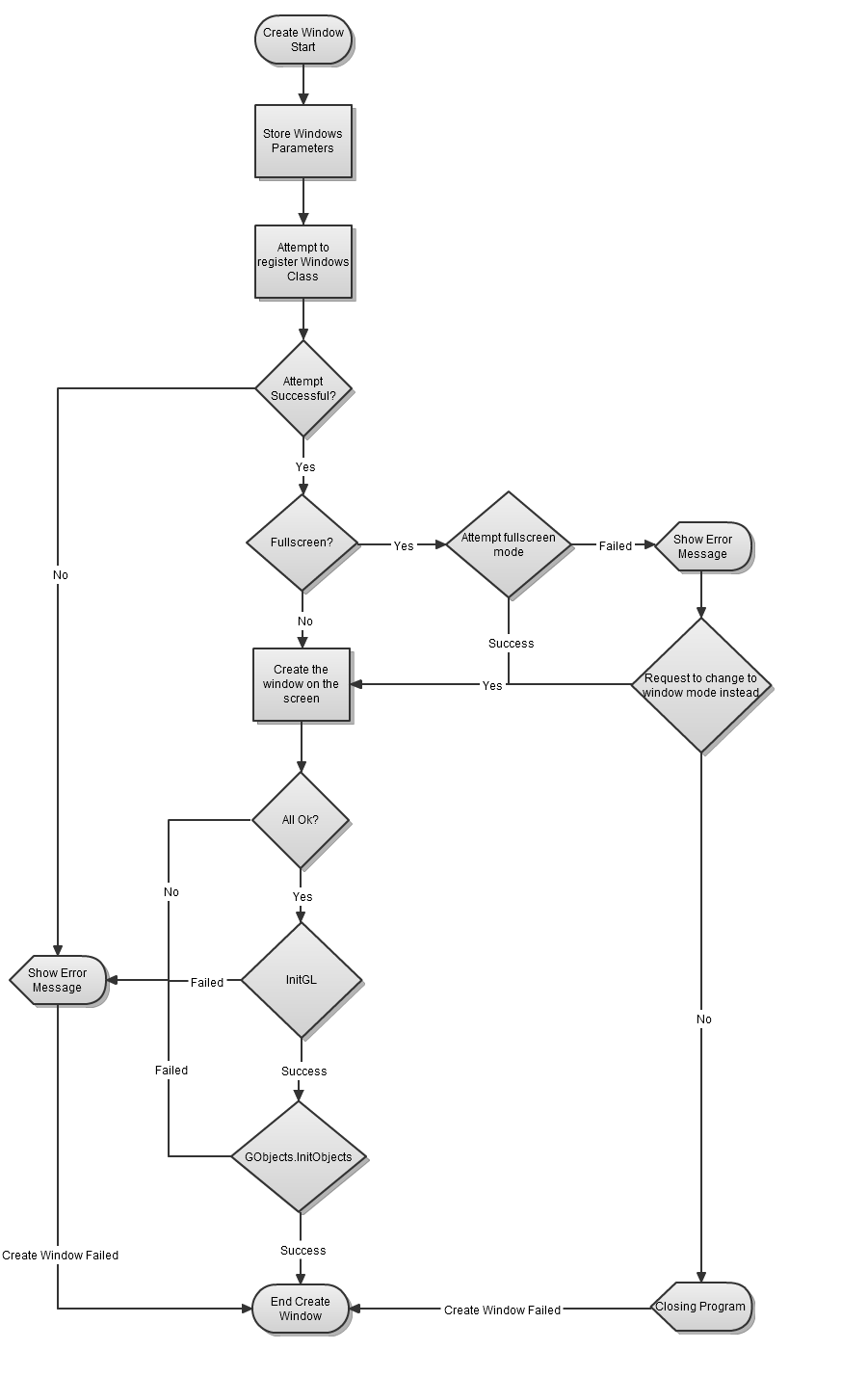
## The main loop

The program starts in WinMain.cpp with the WINAPI WinMain() function. Upon starting the user is prompted to start in full screen or windowed mode. In the next step the program attempts to create the game window. If, for any reason, this attempt was unsuccessful, then an appropriate error message will be displayed and the program will exit.

### Error checking

In large projects, it is often good practice to implement some form of error handling within the code. This allows oneself as well as fellow programmers to quickly discern the general code location of the error. In general it is good to have some kind of error handling in processes that can fail, in which case the program should terminate and produce some kind of error message or error log to allow the user to locate the error. Not having an error handler could lead to unexpected crashes in the program, which the programmer must manually debug to find.

Once the window has been successfully created, the main game loop then runs until an exit message is received, after which the program will destroy the window then terminate.



**Figure 4-5 Create window**

## Create Window

In order to create a window, Windows must first set up several Windows parameters. The first step in doing this is to set up the variables which will be used.

extern HDC hDC;

The first variable to set up is the Device Context, hDC. A DC connects the window to the GDI (Graphics Device Interface) and is required in order to create a window, it is declared here with ‘extern’ as a prefix, and this allows the variable to be used in other source files as discussed earlier.

HGLRC hRC=NULL;

The second variable to set up is the Rendering Context, hRC. A RC is what connects OpenGL to the Device Context.

HWND hWnd=NULL;

The third variable to set up is the handle to a window, hWnd. Handles are integers that Windows assigns to identify an object. In this case hWnd identifies a window.

HINSTANCE hInstance;

The next variable to set up is the handle to an instance, hInstance. An instance is a copy of an application. Due to multitasking and the ability to run multiple copies of a program, Windows needs a way to know which program is which. It does this by giving each program a hInstance (an integer to distinguise between programs). When the program starts, Windows assigns this variable a unique number.

extern bool fullscreen;

The last variable to set up is the fullscreen flag. This bool determines whether to run the program in windowed or fullscreen mode. Once again ‘extern’ is used here.

Once the initial variables have been initialised the next step is to set the Windows Class structure parameters.

WNDCLASS wc;

The Windows Class structure, wc, is a struct containing information for the Windows Class. Once the parameters have been set, the program will attempt to register the wc. As this is an operation that can potentially fail, it would be wise to include some error handling here. Error handling here is provided by a simple if statement which returns an error message and then exits if the program failed to register the wc. The error message is in the form of a pop up box containing the relevant information. Should the program fail here, the pop up box will allow the programmer to quickly discern the location of the error.

The processes following can all potentially fail and so error checking is used throughout. If for any reason the program cannot create a fullscreen window, then the user is prompted to start in windowed mode or quit the program.

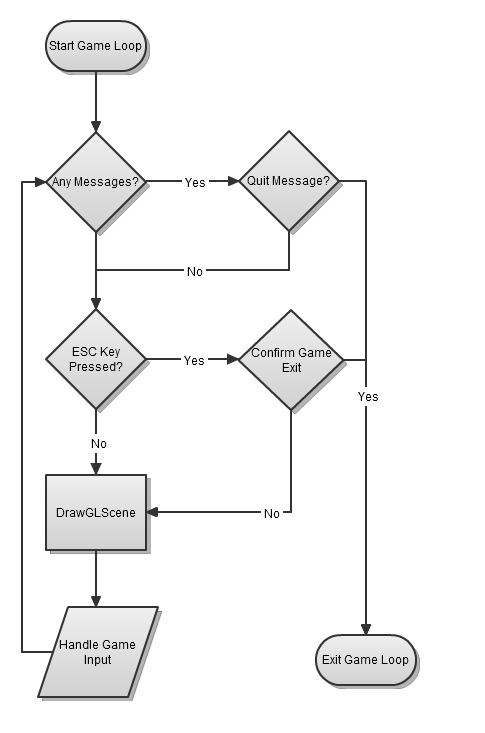
Once this is done, all the information required to create a window has been gathered. The next step then would be to attempt to create the window on the screen. This is done using the CreateWindowEx() function. Once the relavent information has been passed in the parameters, the program checks to see if the window was created properly. Note here that the creation of the window and the error check all occur on one line of code, this speeds up compilation and is a more productive method of programming.

Once this is done, the pixel format is chosen and further error checks are performed. If all error checks are passed, the program will attempt to initalise OpenGL where the lighting, textures and anything else that needs to be set up can occur. Finally, the program will attempt to initialise all game objects.

|  |  |
| --- | --- |
| **Figure 4-6 InitGL** | **Figure 4-7 InitObjects** |

## InitGL and InitObjects

The process of initialising OpenGL is relatively short and straightforward. The steps required are explained well enough in the flowchart. The process of initialising the game objects was explained earlier, under the heading *The InitObjects Method*.



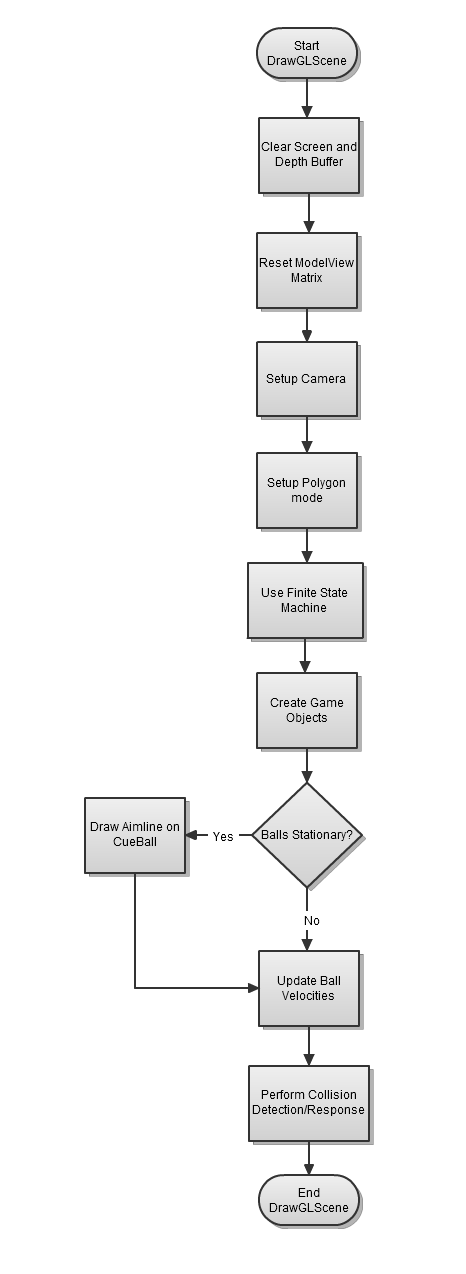
**Figure 4-8 Game Loop**

## The Game Loop

The key component of any game, from a programming standpoint, is the game loop. The game loop allows the game to run smoothly regardless of a user’s input or lack thereof.

Most traditional software programs respond to user input and do nothing without it. A word processor for example will idle and do nothing while there is no input from the end user. Games on the other hand, must continue to operate regardless of a users input, and it is the game loop that allows this.

The game loop in this program follows a common game loop structure. While there are no exit messages from the user, the game loop will continue running. The DrawGLScene function performs all game related actions, from rendering the graphics to performing all game logic.



**Figure 4-9 DrawGLScene**

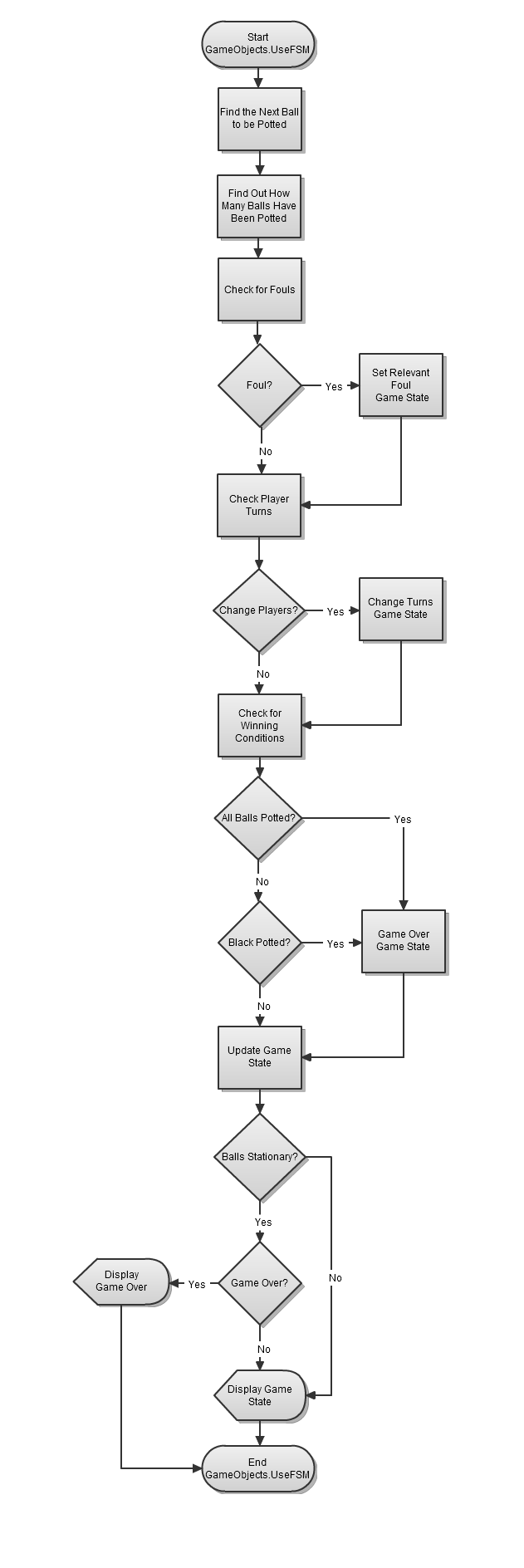
## DrawGLScene

DrawGLScene begins by initialising the scene. The glClear function is used to clear the screen and depth buffer, and then the glLoadIdentity function is then used to reset the modelview matrix. The gluLookAt function is used to set up a camera and finally, glPolygonMode is used to set the polygon mode.

DrawGLScene then handles all game logic, including handling collisions, using the FSM to handle game states, and creating and updating all objects within the game. Once the game logic has been taken care of, DrawGLScene will exit.

### Use Finite State Machine

The process of using the FSM was explained earlier in the chapter under the headings *The FSM Class* and *The UseFSM method*.

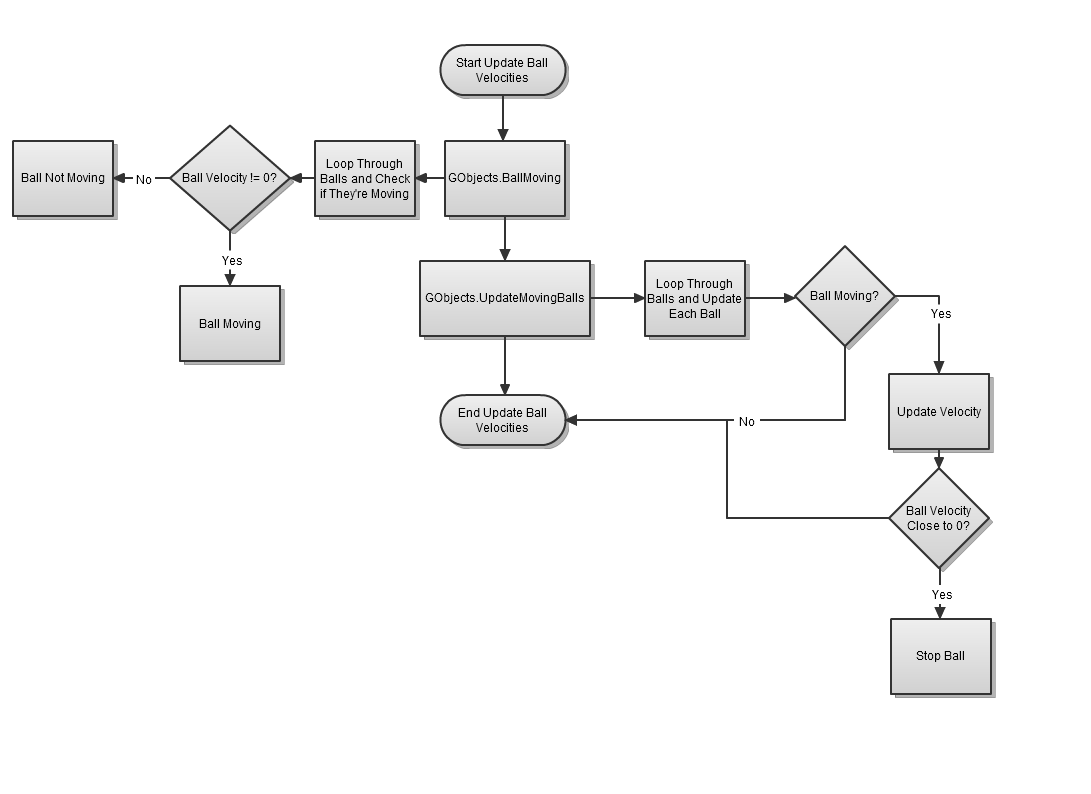


**Figure 4-10 UseFSM**

### Create Game Objects

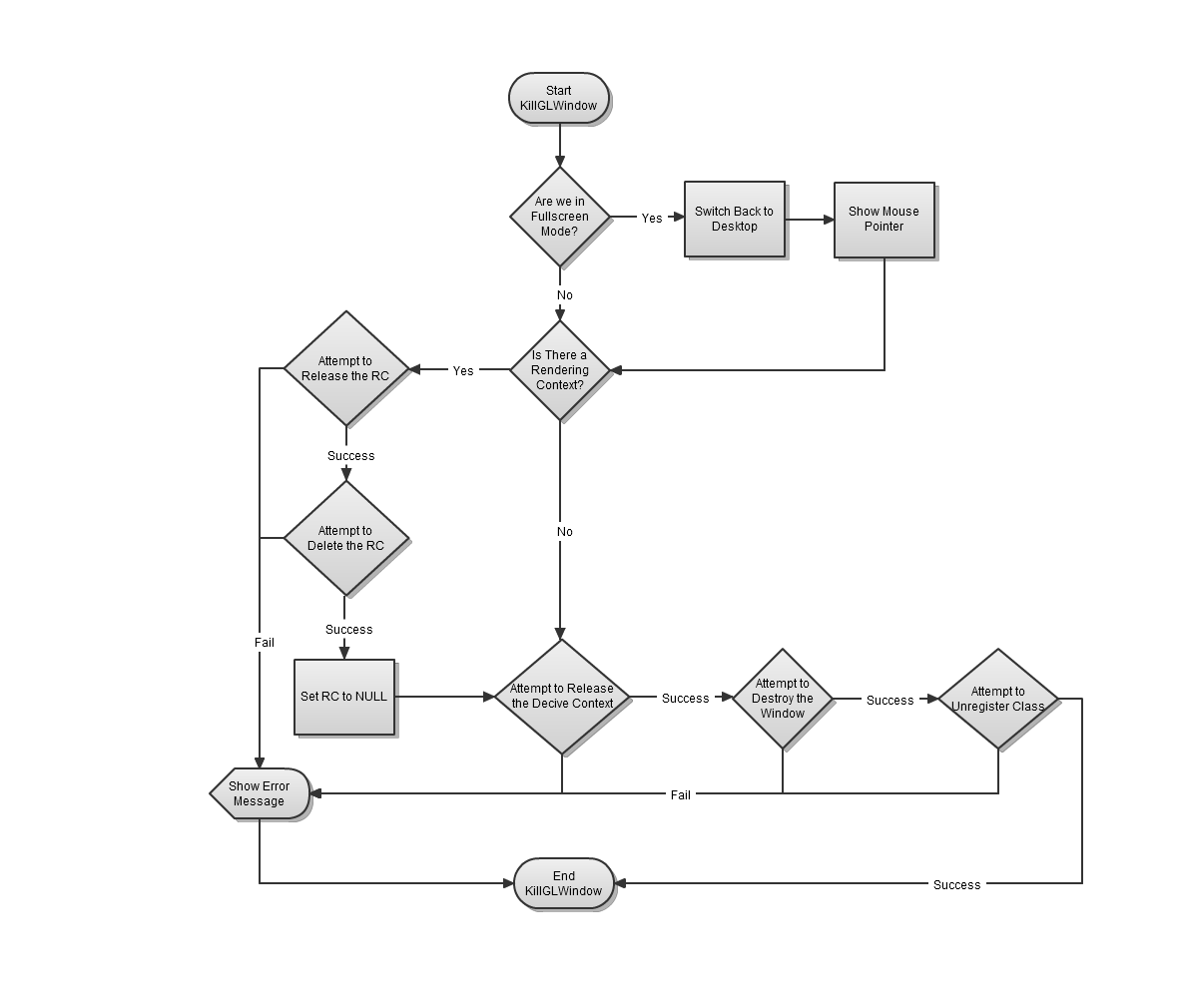
### 

### Update Ball Velocities



### Collision Detection/Response

### 



## KillGLWindow

Once an exit message is received the program must perform a cleanup process before it can terminate. If the program was in full screen mode then it will switch back to the desktop. The device context and rendering context which links the program to the GDI will have to be deleted and set to NULL. Finally, the window class will have to be deleted and unregistered. All this is required to free up the memory the program used whilst it was running. It is important here to have error checking, as, if any of these cleanup processes fail, the operating system could lose memory.

# Conclusions

This project has presented a method of applying the physics in the game of nine ball pool to produce a working simulation of the game. Principles such as linear motion, elastic and inelastic collisions, conservation of momentum and energy have all been implemented. In addition to implementing the physics, a simple game engine was also created. The game engine ran a simple game loop, contained a working finite state machine to determine game states and was able to handle player input.

This dissertation also discussed further research done regarding angular motion in cue sports, and it is hoped that in future revisions of this program this research can be implemented to further increase the sense of realism provided. Further research topics may also include the use of the Separating Axis Theorem, which may be used to handle collisions between convex shapes. For objects lying in a [plane](http://en.wikipedia.org/wiki/Plane_%28mathematics%29), the SAT states that, given two [convex](http://en.wikipedia.org/wiki/Convex_set) shapes, there exists a line onto which their projections will be separate [if and only if](http://en.wikipedia.org/wiki/If_and_only_if) they are not intersecting. The SAT could be used to handle collisions between the ball and table rails as it provides a more efficient method of achieving this.

Although the program worked effectively to simulate nine ball pool, as stated in the research and investigation chapter, the use of a time-stepping mechanism proved to be detrimental. Collisions occurring at high velocity run the risk of being miscalculated if a time-stepping technique is used. To overcome this issue in future revisions, the program will run on a time-based system where frames run on a timer. By introducing a numerical integrator to the program along with the time-based system one will be able to discern the exact time a collision occurs, and so one may be able to calculate the collision response accurately no matter what velocities the balls travel at.

Finally, although it was not the original intention to produce a “pretty looking game”, should the physics and glitches be fixed, then the next logical aspect to work on will be the aesthetics. Future revisions of the program will contain sound, textures and make use of OpenGL’s graphics capabilities to utilise its lighting effects. A quaternion based camera will also be implemented, which can be used to move around the table to give the player better angles to aim shots.

# References

Alciatore, D., 2008. *Pool and Billiards Physics Principles by Coriolis and Others*. Fort Collins, CO: Colorado State University.

Berchek, C., 2009. *2-Dimensional Elastic Collisions without Trigonometry*. [Online] Available at: <http://www.vobarian.com/collisions/> [Accessed 17 November 2009].

Bourg, D.M., 2002. *Physics for game developers*. USA: O'Reilly & Associates.

Coriolis, G.G., 1835. *Theorie mathematique des effets du jeu de billard (Mathematical Theory of the Game of Billiards)*. Paris: Journal de l'Ecole polytechnique École Polytechnique.

Halliday, D., Resnick, R. & Walker, J., 2008. *Fundamentals of Physics Extended*. 8th ed. John Wiley & Sons, Inc.

Hecker, C., 1996. *Physics, The Next Frontier*. [Online] Available at: <http://chrishecker.com/images/d/df/Gdmphys1.pdf> [Accessed 20 November 2009].

Leckie, W. & Greenspan, M., 2006. *Pool physics simulation by event prediction 1: Motion transitions*. Kingston, Ontario, Canada: ICGA Journal Queen's University.

Leckie, W. & Greenspan, M., 2006. *Pool physics simulation by event prediction 2: Collisions*. Kingston, Ontario, Canada: ICGA Journal Queen's University.

Marlow, W., 1995. *The Physics of Pocket Billiards*. Marlow Advanced Systems Technologies (MAST).

Shepard, R., 1997. *Amateur Physics for the Amateur Pool Player*.

Stronge, W.J., 2000. Impact Mechanics. *Meccanica*, 35(5), pp.470-71.

Townsend, B., 2003. *The math and physics of billiards*. [Online] Available at: <http://archive.ncsa.illinois.edu/Classes/MATH198/townsend/math.html#COLLISION> [Accessed 04 November 2009].

Tremblay, C., 2004. *Mathematics for game developers*. Boston, MA: Thomson Course Technology.

Van Verth, J.M. & Bishop, L.M., 2008. *Essential Mathematics for games & interactive applications*. 2nd ed. Oxford, England: Morgan Kaufmann Publishers.

# Appendices

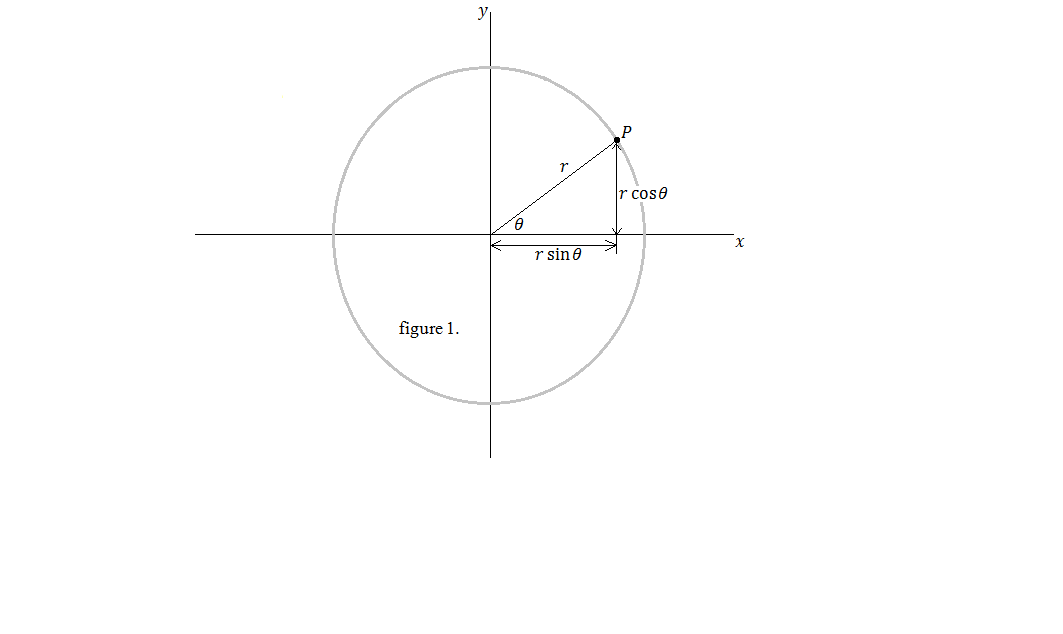
## Finding a point on a sphere using spherical coordinates

In order to calculate the initial angular velocity of a billiard ball the point at which the cue is in contact with the cue ball must be determined, to this end, using the cue ball’s local spherical coordinate system is an ideal method for this.

### Polar coordinate system

Since the three dimensional spherical coordinate system is an extension to the two dimensional polar coordinate system, it would be useful to know how to determine this as well as how to convert from polar coordinates to Cartesian coordinates.

Consider a circle of radius r, the point P is located in a position within the circle, , as illustrated in figure 1:



The polar coordinates of point P is given by its angle, which is measured counter clockwise from the positive x-axis, and the distance r from the origin.

The polar coordinate system can be converted to the Cartesian coordinate system using trigonometry as follows:

Where is the angle measured counter clockwise from the positive x-axis.

### Spherical coordinates

Calculating spherical coordinates introduces an extra value to the polar coordinate system, .

and represent the same information as they do in polar coordinates, the new value represents an angle taken from the z-axis to the point in question.

Consider a point with Cartesian coordinates (x,y,z) its distance from the origin is denoted by r, as illustrated below.

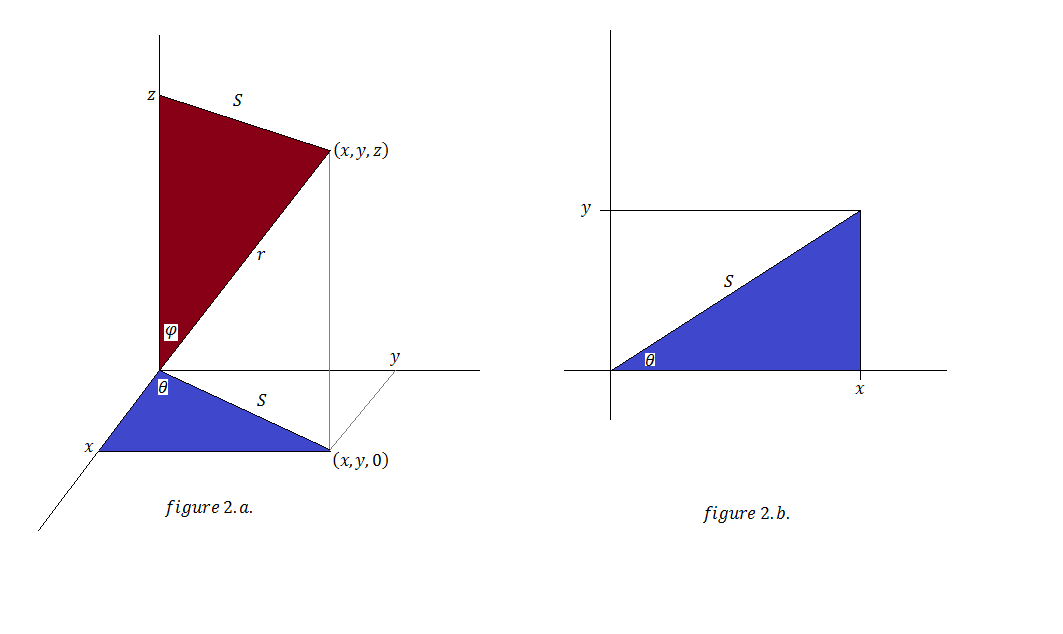


Figure 2.b. shows the xy-plane from figure 2.a. Using Pythagoras theorem:

Thus, converting from spherical to Cartesian coordinates is as follows:

### Implementation of spherical coordinates

Spherical coordinates are implemented in the project as follows:

//find points on the surface of a sphere, given its radius, origin, longtitude and latitude

Vector getSpherePoints(float radius, Vector pos, float radial, float elevation)

{

float i = (sin(elevation\*fPI/180) \* sin(radial\*fPI/180));

float j = cos(elevation\*fPI/180);

float k = (sin(elevation\*fPI/180) \* cos(radial\*fPI/180));

Vector a;

a.x = pos.x + radius \* i;

a.y = pos.y + radius \* j;

a.z = pos.z + radius \* k;

return a;

}

The function getSpherePoints returns a vector that contains the Cartesian coordinates of the point on the sphere. The function parameter takes four arguments:

* Radius: The distance from the origin of the sphere, r.
* Radial: The angle taken counter clockwise from the positive x-axis,.
* Elevation: The angle taken from the z-axis to the point in question, .
* Pos: A vector containing the world coordinates of the sphere.

The local coordinates of a point on the sphere can be determined by subtracting the world coordinates of the point by the world coordinates of the sphere’s origin.

## Change log

Billiards V3.7 [FINAL] last updated 15/04/10

Scrapped

-Rotational motion of spheres (couldn't fully understand how to implement) [RESEARCH HAS BEEN DONE]

Working

-FSM to determine gamestates

-Font class to display text

-Nine ball poolimplemented

-psuedo topspin and backspin using the coefficient of restitution

-OO programming implemented

-Perfectly elastic oblique collisions

-Collision detection between spheres

-Muliple oblique collisions

-Psuedo friction where the velocity components are continuously subtracted by a value

-Change from psuedo friction to REAL friction using F = umg formula

-Ability to aim cue ball and determine force in the shot

-Aiming and shooting the ball involves the use of the impulse concept

Buggy [CANNOT BE FIXED WITH CURRENT TIMESTEPPING METHOD]

-Glancing collision at high velocity causes the object ball to move with wrong velocity CAUSE FOUND

Collision detection between spheres does not take into account the exact time a collision occurs, therefore the

sphere can 'warp' past another ball between frames.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Billiards V3.6 last updated 10/04/10

To do

-Fix bugs

Added

-FSM to determine gamestates

-Font class to display text

Scrapped

-Rotational motion of spheres (couldn't fully understand how to implement)

Working

-Nine ball poolimplemented

-psuedo topspin and backspin using the coefficient of restitution

-OO programming implemented

-Perfectly elastic oblique collisions

-Collision detection between spheres

-Muliple oblique collisions

-Psuedo friction where the velocity components are continuously subtracted by a value

-Change from psuedo friction to REAL friction using F = umg formula

-Ability to aim cue ball and determine force in the shot

-Aiming and shooting the ball involves the use of the impulse concept

Buggy

-Glancing collision at high velocity causes the object ball to move with wrong velocity CAUSE FOUND

-Collision detection between spheres does not take into account the exact time a collision occurs, therefore the

sphere can 'warp' past another ball between frames.

-Incorrect collision response when a higher numbered ball collides with a lower numbered ball. (FIXED)

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Billiards V3.5 last updated 19/03/10

To do

-Fix bugs

-Implement FSM to determine gamestates

-Add font class to display text

Added

-Nine ball poolimplemented

-psuedo topspin and backspin using the coefficient of restitution

Scrapped

-Rotational motion of spheres (couldn't fully understand how to implement)

Working

-OO programming implemented

-Perfectly elastic oblique collisions

-Collision detection between spheres

-Muliple oblique collisions

-Psuedo friction where the velocity components are continuously subtracted by a value

-Change from psuedo friction to REAL friction using F = umg formula

-Ability to aim cue ball and determine force in the shot

-Aiming and shooting the ball involves the use of the impulse concept

Buggy

-Glancing collision at high velocity causes the object ball to move with wrong velocity CAUSE FOUND

-Collision detection between spheres does not take into account the exact time a collision occurs, therefore the

sphere can 'warp' past another ball between frames.

-Incorrect collision response when a higher numbered ball collides with a lower numbered ball. (FIXED)

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Billiards V3.4.6 last updated 13/03/10

To do

-Fix bugs

Added

-Creating classes to make the code OO

Scrapped

-Rotational motion of spheres (couldn't fully understand how to implement)

Working

-Perfectly elastic oblique collisions

-Collision detection between spheres

-Muliple oblique collisions

-Psuedo friction where the velocity components are continuously subtracted by a value

-Ability to aim cue ball and determine force in the shot

-Change from psuedo friction to REAL friction using F = umg formula

-Aiming and shooting the ball involves the use of the impulse concept

Buggy

-Glancing collision at high velocity causes the object ball to move with wrong velocity CAUSE FOUND

-Collision detection between spheres does not take into account the exact time a collision occurs, therefore the

sphere can 'warp' past another ball between frames.

-Incorrect collision response when a higher numbered ball collides with a lower numbered ball.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Billiards V3.4.3 last updated 18/12/09

To do

-Rotational motion of spheres (IN PROGRESS)

-Fix bugs

Added

-Point of contact between cue and cue ball has been calculated using sperical coordinate system, this will be used to calculate the angular velocity of cue ball

-Extra cue ball added on the right called aimBall, this ball has a aiming dot allowing users to apply spin on cue ball

Working

-Perfectly elastic oblique collisions

-Collision detection between spheres

-Muliple oblique collisions

-Psuedo friction where the velocity components are continuously subtracted by a value

-Ability to aim cue ball and determine force in the shot

-Change from psuedo friction to REAL friction using F = umg formula

-Aiming and shooting the ball involves the use of the impulse concept

Buggy

-Glancing collision at high velocity causes the object ball to move with wrong velocity CAUSE FOUND

-Collision detection between spheres does not take into account the exact time a collision occurs, therefore the

sphere can 'warp' past another ball between frames.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Billiards V3.4.2 last updated 18/12/09

fix'd

-Collisions between spheres now take the coefficient of restitution into consideration

-Collisions between sphere and rail now causes the sphere to lose some of its velocity

To do

-Rotational motion of spheres

-Fix bugs

Added

-Aiming and shooting the ball now involves the use of the impulse concept

Working

-Perfectly elastic oblique collisions

-Collision detection between spheres

-Muliple oblique collisions

-Psuedo friction where the velocity components are continuously subtracted by a value

-Ability to aim cue ball and determine force in the shot

-Change from psuedo friction to REAL friction using F = umg formula

Buggy

-Glancing collision at high velocity causes the object ball to move with wrong velocity CAUSE FOUND

-Collision detection between spheres does not take into account the exact time a collision occurs, therefore the

sphere can 'warp' past another ball between frames.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Billiards V3.4.1 last updated 18/12/09

fix'd

-Collision between bottom and left rails(-ve x/z axis)

To do

-Rotational motion of spheres

-Fix bugs

Added

-Aiming and shooting the ball now involves the use of the impulse concept

Working

-Perfectly elastic oblique collisions

-Collision detection between spheres

-Muliple oblique collisions

-Psuedo friction where the velocity components are continuously subtracted by a value

-Ability to aim cue ball and determine force in the shot

-Change from psuedo friction to REAL friction using F = umg formula

Buggy

-Glancing collision at high velocity causes the object ball to move with wrong velocity

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Billiards V3.4 last updated 18/12/09

To do

-Change from psuedo friction to REAL friction using F = umg formula

Added

-Aiming and shooting the ball now involves the use of the impulse concept

Working

-Perfectly elastic oblique collisions

-Collision detection between spheres

-Muliple oblique collisions

-Psuedo friction where the velocity components are continuously subtracted by a value

-Ability to aim cue ball and determine force in the shot

Buggy

-Collision between bottom and left rails(-ve x/z axis)

-Glancing collision at high velocity causes the object ball to move with wrong velocity

Not done

-Rotational motion of spheres

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Billiards V3.3 last updated 16/12/09

Added

-Psuedo friction where the velocity components are continuously subtracted by a value

-Ability to aim cue ball and determine force in the shot

Working

-Perfectly elastic oblique collisions

-Collision detection between spheres

-Muliple oblique collisions

Buggy

-Collision between bottom and left rails(-ve x/z axis)

Not done

-Rotational motion of spheres

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Billiards V3.2 last updated 13/12/09

fix'd

-Collision detection between spheres and rails

-Got 3rd ball to show itself

Working

-Perfectly elastic oblique collisions

-Collision detection between spheres

-Muliple oblique collisions

Buggy

-Collision between bottom and left rails(-ve x/z axis)

Not done

-Rotational motion of spheres

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Billiards V3.1 last updated 7/12/09

fix'd

-Collision detection between spheres and rails

-Got 3rd ball to show itself

Working

-Perfectly elastic oblique collisions

-Collision detection between spheres

Not done

-Mulitple oblique collisions

-Rotational motion of spheres

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Billiards V3 last updated 6/12/09

fix'd

-Collision detection between spheres and rails

Working

-Perfectly elastic oblique collisions

-Collision detection between spheres

Buggy

-Can't get the 3rd ball to show itself!

Not done

-Mulitple oblique collisions

-Rotational motion of spheres

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Billiards V2 last updated 6/12/09

Working

-Perfectly elastic oblique collisions

-Collision detection between spheres

Buggy

-Collision detection between spheres and rails, penetration distance must be correctted as with sphere sphere collisions

Not done

-Mulitple oblique collisions

-Rotational motion of spheres

## Program source code

### global.h

#ifndef GLOBAL\_H

#define GLOBAL\_H

#include <windows.h> // Header File For Windows

#include <gl\gl.h> // Header File For The OpenGL32 Library

#include <gl\glu.h> // Header File For The GLu32 Library

#include <gl\glaux.h> // Header File For The Glaux Library

#include "Balls.h"

#include "Vector3.h"

#include "ColourOGL.h"

#include "GameObjects.h"

//=============== Function prototypes ====================

//WinMain.cpp

LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM); // Declaration For WndProc

int WINAPI WinMain( HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nCmdShow);

//OGL.cpp

GLvoid ReSizeGLScene(GLsizei width, GLsizei height);

int InitGL(GLvoid);

int DrawGLScene(GLvoid);

GLvoid KillGLWindow(GLvoid);

BOOL CreateGLWindow(char\* title, int width, int height, int bits, bool fullscreenflag);

#endif //GLOBAL\_H

### ColourOGL.h

#ifndef COLOUROGL\_H

#define COLOUROGL\_H

//holds a colour (0.0 - 1.0) used by OpenGL

struct ColourOGL {

float fRed;

float fGreen;

float fBlue;

void Set(float Red, float Green, float Blue) {

fRed = Red;

fGreen = Green;

fBlue = Blue;

}

void Set(int Red, int Green, int Blue) {

if (Red < 0) {

fRed = 0.0f;

}

else if (Red > 255) {

fRed = 1.0f;

}

else {

fRed = Red/255.0f;

}

if (Green < 0) {

fGreen = 0.0f;

}

else if (Green > 255) {

fGreen = 1.0f;

}

else {

fGreen = Green/255.0f;

}

if (Blue < 0) {

fBlue = 0.0f;

}

else if (Blue > 255) {

fBlue = 1.0f;

}

else {

fBlue = Blue/255.0f;

} //end if-else

} //end Set()

}; //ColourOGL

#endif COLOUROGL\_H //COLOUROGL\_H

### Vector3.h

#ifndef VECTOR3\_H

#define VECTOR3\_H

#include <cmath>

//=================== Includes Above ====================|==================== Classes Below ========================

class Vector3

{

public:

//---------------------------------------------------------------------------------------------------------------

// Name: Vector3()

// Desc: default constructor, sets vector to 0, 0, 0

//

inline Vector3() {

m\_fX = 0.0f;

m\_fY = 0.0f;

m\_fZ = 0.0f;

} //Vector3

//---------------------------------------------------------------------------------------------------------------

// Name: Vector3()

// Desc: default contructor, sets vector to disired value

//

inline Vector3(float x, float y, float z) {

m\_fX = x;

m\_fY = y;

m\_fZ = z;

} //Vector3

//---------------------------------------------------------------------------------------------------------------

// Name: ~Vector3()

// Desc: default destructor

//

virtual ~Vector3() {}

//---------------------------------------------------------------------------------------------------------------

// Name: operator\*()

// Desc: scale a vector by a scalar value using operator overloading: ie Vector \* Scalar

//

inline Vector3 operator\*(const float &fValue) const {

return Vector3(m\_fX \* fValue, m\_fY \* fValue, m\_fX \* fValue);

} //operator\*

//---------------------------------------------------------------------------------------------------------------

// Name: operator\*()

// Desc: scale a vector by a scalar value using operator overloading: ie Scalar \* Vector

//

inline friend Vector3 operator\*(const float &fValue, const Vector3 v) {

return Vector3(fValue \* v.m\_fX, fValue \* v.m\_fY, fValue \* v.m\_fZ);

} //operator\*

//---------------------------------------------------------------------------------------------------------------

// Name: operator+()

// Desc: add two vectors using operator overloading: ie Vector1 + Vector2

//

inline Vector3 operator+(const Vector3 &v) const {

return Vector3(m\_fX + v.m\_fX, m\_fY + v.m\_fY, m\_fZ + v.m\_fZ);

} //operator+

//---------------------------------------------------------------------------------------------------------------

// Name: operator-()

// Desc: subtract two vectors using operator overloading: ie Vector1 + Vector2

//

inline Vector3 operator-(const Vector3 &v) const {

return Vector3(m\_fX - v.m\_fX, m\_fY - v.m\_fY, m\_fZ - v.m\_fZ);

} //operator-

//---------------------------------------------------------------------------------------------------------------

// Name: operator+=()

// Desc: assign a vector by addition using operator overloading: ie Vector1 += Vector2

//

inline Vector3& operator+=(const Vector3 &vec3) {

m\_fX += vec3.m\_fX;

m\_fY += vec3.m\_fY;

m\_fZ += vec3.m\_fZ;

return \*this;

} //operator+=

//---------------------------------------------------------------------------------------------------------------

// Name: operator-=()

// Desc: assign a vector by subtraction using operator overloading: ie Vector1 -= Vector2

//

inline Vector3& operator-=(const Vector3 &vec3) {

m\_fX -= vec3.m\_fX;

m\_fY -= vec3.m\_fY;

m\_fZ -= vec3.m\_fZ;

return \*this;

} //operator-=

//---------------------------------------------------------------------------------------------------------------

// Name: Cross()

// Desc: find the cross product of two vectors

//

inline Vector3 Cross(Vector3 v1) {

Vector3 result;

result.m\_fX = m\_fY \* v1.m\_fZ - v1.m\_fY \* m\_fZ;

result.m\_fY = m\_fZ \* v1.m\_fX - v1.m\_fZ \* m\_fX;

result.m\_fZ = m\_fX \* v1.m\_fY - v1.m\_fX \* m\_fY;

return result;

} //Cross

//---------------------------------------------------------------------------------------------------------------

// Name: Dot()

// Desc: find the dot product of two vectors

//

inline float Dot(Vector3 v1) {

return m\_fX \* v1.m\_fX + m\_fY \* v1.m\_fY + m\_fZ \* v1.m\_fZ;

} //Cross

//---------------------------------------------------------------------------------------------------------------

// Name: Magnitude()

// Desc: find the magnitude of a vector

//

inline float Magnitude() {

return sqrt(m\_fX \* m\_fX + m\_fY \* m\_fY + m\_fZ \* m\_fZ);

} //Magnitude

//---------------------------------------------------------------------------------------------------------------

// Name: Normalise()

// Desc: normalise a vector

//

inline Vector3 Normalise() {

float fMagnitude = sqrt(m\_fX \* m\_fX + m\_fY \* m\_fY + m\_fZ \* m\_fZ);

m\_fX = m\_fX \* (1.0f/fMagnitude);

m\_fY = m\_fY \* (1.0f/fMagnitude);

m\_fZ = m\_fZ \* (1.0f/fMagnitude);

Vector3 result(m\_fX, m\_fY, m\_fZ);

return result;

} //Normalise

//public member variables

float m\_fX;

float m\_fY;

float m\_fZ;

};

#endif //VECTOR3\_H

### Balls.h

#ifndef BALLS\_H

#define BALLS\_H

#include "ColourOGL.h"

#include "Vector3.h"

#include "glut.h"

//================== Includes Above ====================|================ Classes Below ================

class Balls

{

public:

Balls();

virtual ~Balls();

void CreateBall(ColourOGL colour);

void WorkOutVelocity(float fForceCue, float fAngle);

void UpdateMovingBalls(float fCoFr);

void BallMoving();

void RailCollision();

bool BallCollision(Balls &ball);

void BallPocketCollision(Balls &pocket);

void BallCollisionResponse(Balls &ball);

//============== Class Methods Above ==============|============== Member Variables Below ==============

Vector3 m\_vecPosition;

Vector3 m\_vecVelocity;

Vector3 m\_vecFriction;

float m\_fRadius;

float m\_fCoRest; //coefficient of restitution

bool m\_bMoving;

bool m\_bCollided;

bool m\_bPocketed;

bool m\_bOutOfPlay; //is the ball out of play? i.e. it has been potted

int m\_iID; //ID of ball

private:

float m\_fMass;

float m\_fPI;

float m\_fGrav; //gravity

};

#endif BALLS\_H //BALLS\_H

### Balls.cpp

#include "Balls.h"

//---------------------------------------------------------------------------------------------------------------

// Name: Balls()

// Desc: default constructor

//

Balls::Balls() {

m\_vecVelocity = Vector3(0.0f,0.0f,0.0f);

m\_vecFriction = Vector3(0.0f,0.0f,0.0f);

m\_fMass = 0.16f;

m\_fRadius = 0.70f;

m\_iID = 0;

m\_fPI = 3.14159265358979323846264338327950288419716939937510f;

m\_fGrav = 9.81f;

m\_fCoRest = 0.90f;

m\_bMoving = false;

m\_bCollided = false;

m\_bPocketed = false;

m\_bOutOfPlay= false;

} //Balls()

//---------------------------------------------------------------------------------------------------------------

// Name: ~Balls()

// Desc: default destructor

//

Balls::~Balls() {

} //~Balls()

//---------------------------------------------------------------------------------------------------------------

// Name: CreateBall()

// Desc: creates the ball object

//

void Balls::CreateBall(ColourOGL colour) {

glPushMatrix();

glTranslated(m\_vecPosition.m\_fX, m\_vecPosition.m\_fY, m\_vecPosition.m\_fZ);

m\_vecPosition.m\_fX += m\_vecVelocity.m\_fX;

m\_vecPosition.m\_fZ += m\_vecVelocity.m\_fZ;

glColor3f(colour.fRed, colour.fGreen, colour.fBlue);

glutSolidSphere(m\_fRadius, 20.0,20.0);

glPopMatrix();

} //CreateBall()

//---------------------------------------------------------------------------------------------------------------

// Name: WorkOutVelocity()

// Desc: calculate velocity of the cue ball using the concept of impulse

//

void Balls::WorkOutVelocity(float fForceCue, float fAngle) {

//use of the impulse equation to find translational SPEED

float fVelocity = (fForceCue \* 0.2f)/m\_fMass;

float fTheta;

Vector3 vecVelocity;

//Vector3 vecForceCue;

//finding the vector components of the SPEED to find the translational VECLOCITY

if(fAngle <= 90.0f) {

vecVelocity = Vector3(fVelocity \* sin(fAngle\*m\_fPI/180), 0.0f, fVelocity \* cos(fAngle\*m\_fPI/180)); //the x (ISin(angle)) and z (ICos(angle)) component of the speed

//vecForceCue = Vector3(fVelocity \* cos(fAngle\*m\_fPI/180), 0.0f, fVelocity \* sin(fAngle\*m\_fPI/180));

m\_vecVelocity = vecVelocity; //assign the cue velocity to the ball

}

//if the angle is greater than 90, subtract the correct amount to make it less than 90

else if(fAngle > 90.0f && fAngle <= 180.0f) {

fTheta = fAngle - 90.0f; //subtract the correct amount

vecVelocity = Vector3(fVelocity \* cos(fTheta\*m\_fPI/180), 0.0f, (fVelocity \* sin(fTheta\*m\_fPI/180)) \* -1);

//vecForceCue = Vector3((fVelocity \* sin(fTheta\*m\_fPI/180))\*-1, 0.0f, fVelocity \* cos(fTheta\*m\_fPI/180));

m\_vecVelocity = vecVelocity;

}

else if(fAngle > 180.0f && fAngle <= 270.0f) {

fTheta = fAngle - 180.0f;

vecVelocity = Vector3((fVelocity \* sin(fTheta\*m\_fPI/180)) \* -1, 0.0f, (fVelocity \* cos(fTheta\*m\_fPI/180)) \* -1);

//vecForceCue = Vector3((fVelocity \* cos(fTheta\*m\_fPI/180)) \* -1, 0.0f, (fVelocity \* sin(fTheta\*m\_fPI/180)) \* -1);

m\_vecVelocity = vecVelocity;

}

else {

fTheta = fAngle - 270.0f;

vecVelocity = Vector3((fVelocity \* cos(fTheta\*m\_fPI/180)) \* -1, 0.0f, fVelocity \* sin(fTheta\*m\_fPI/180));

//vecForceCue = Vector3((fVelocity \* cos(fTheta\*m\_fPI/180)) \* -1, 0.0f, fVelocity \* sin(fTheta\*m\_fPI/180));

m\_vecVelocity = vecVelocity;

} //end if-else

} //WorkOutVelocity()

//---------------------------------------------------------------------------------------------------------------

// Name: BallMoving()

// Desc: Checks if the ball is moving or not

//

void Balls::BallMoving() {

if(m\_vecVelocity.Magnitude() != 0.0f) {

m\_bMoving = true;

}

else {

m\_bMoving = false;

}

} //BallMoving()

//---------------------------------------------------------------------------------------------------------------

// Name: UpdateMovingBalls()

// Desc: updates the balls velocities

//

void Balls::UpdateMovingBalls(float fCoFr) {

if(m\_bMoving) {

//frictional force = umg

float fFriction = (fCoFr \* m\_fMass) \* m\_fGrav;

Vector3 frDirection, normFrDirection;

frDirection = -1.0f \* m\_vecVelocity; //frictional force acts opposite motion of balls

normFrDirection = frDirection.Normalise(); //normalising value of force opposite motion

m\_vecFriction = fFriction \* normFrDirection; //applying the frictional force to balls

//update the velocity

m\_vecVelocity.m\_fX += m\_vecFriction.m\_fX;

m\_vecVelocity.m\_fZ += m\_vecFriction.m\_fZ;

} //end if

if(m\_vecVelocity.m\_fX < 0.001 && m\_vecVelocity.m\_fX > -0.001 && m\_vecVelocity.m\_fZ < 0.001 && m\_vecVelocity.m\_fZ > -0.001) {

m\_vecVelocity = Vector3(0.0f,0.0f,0.0f);

m\_bMoving = false;

} //end if

} //UpdateMovingBalls()

//---------------------------------------------------------------------------------------------------------------

// Name: RailCollision()

// Desc: checks for collisions between the ball and rail

//

void Balls::RailCollision() {

//collsion with right rail

if(m\_vecPosition.m\_fZ < 17.10 && m\_vecPosition.m\_fZ > -17.10) {

if(m\_vecPosition.m\_fX + m\_fRadius >= 20) { //ball-rail overlap?

float fOverlap = 20 - (m\_vecPosition.m\_fX + m\_fRadius);

if(fOverlap < 0)

m\_vecPosition.m\_fX += fOverlap; //correct overlap

m\_vecVelocity.m\_fX \*= -1; //reflect the ball back off the rail

m\_vecVelocity.m\_fX \*= 0.70f; //energy loss colliding with the rail. the ball loses 30% of its momentum

m\_fCoRest = 0.90f;

}

}

//safety check

else if(m\_vecPosition.m\_fX > 19.0) //stops the ball continuing on through the rails and ensures that it is pocketed

m\_bPocketed = true;

//collision with left rail

if(m\_vecPosition.m\_fZ < 17.10 && m\_vecPosition.m\_fZ > -17.10) {

if(m\_vecPosition.m\_fX - m\_fRadius <= -20) {

float fOverlap = -20 - (m\_vecPosition.m\_fX - m\_fRadius);

if(fOverlap > 0)

m\_vecPosition.m\_fX += fOverlap;

m\_vecVelocity.m\_fX \*= -1;

m\_vecVelocity.m\_fX \*= 0.70f;

m\_fCoRest = 0.90f;

}

}

else if(m\_vecPosition.m\_fX < -19.0)

m\_bPocketed = true;

//collision with top rail

if((m\_vecPosition.m\_fX > -18.80 && m\_vecPosition.m\_fX < -1.18) || (m\_vecPosition.m\_fX > 1.18 && m\_vecPosition.m\_fX < 18.80)) {

if(m\_vecPosition.m\_fZ + m\_fRadius >= 18 && !m\_bPocketed) { //!m\_bPockected ensures the balls are translated to the corrected position once they've been pocketed

float fOverlap = 18 - (m\_vecPosition.m\_fZ + m\_fRadius);

if(fOverlap < 0)

m\_vecPosition.m\_fZ += fOverlap;

m\_vecVelocity.m\_fZ \*= -1;

m\_vecVelocity.m\_fZ \*= 0.70f;

m\_fCoRest = 0.90f;

}

}

else if(m\_vecPosition.m\_fZ > 18)

m\_bPocketed = true;

//collision with bottom rail

if((m\_vecPosition.m\_fX > -18.80 && m\_vecPosition.m\_fX < -1.18) || (m\_vecPosition.m\_fX > 1.18 && m\_vecPosition.m\_fX < 18.80)) {

if(m\_vecPosition.m\_fZ - m\_fRadius <= -18) {

float fOverlap = -18 - (m\_vecPosition.m\_fZ - m\_fRadius);

if(fOverlap > 0)

m\_vecPosition.m\_fZ += fOverlap;

m\_vecVelocity.m\_fZ \*= -1;

m\_vecVelocity.m\_fZ \*= 0.70f;

m\_fCoRest = 0.90f;

}

}

else if(m\_vecPosition.m\_fZ < -18)

m\_bPocketed = true;

} //RailCollisions()

//---------------------------------------------------------------------------------------------------------------

// Name: BallCollision()

// Desc: checks for collisions between the ball and ball

//

bool Balls::BallCollision(Balls &ball) {

Vector3 vecBallDist;

float fHypotinuse, fOverlap;

//calculate the distance between balls

vecBallDist = m\_vecPosition - ball.m\_vecPosition;

fHypotinuse = vecBallDist.Magnitude();

//check to see if the balls overlap

fOverlap = fHypotinuse - (m\_fRadius + ball.m\_fRadius);

//correct positions of balls if they overlap

if(fOverlap < 0) {

Vector3 fNormalisedDist = vecBallDist.Normalise();

Vector3 vecOverlap = fOverlap \* fNormalisedDist;

m\_vecPosition -= vecOverlap;

}

//collision check

if(fHypotinuse <= (m\_fRadius + ball.m\_fRadius)) {

m\_bCollided = true;

ball.m\_bCollided = true;

return true;

}

else

return false;

} //BallCollision()

//---------------------------------------------------------------------------------------------------------------

// Name: BallPocketCollision()

// Desc: checks for collisions between the ball and pocket

//

void Balls::BallPocketCollision(Balls &pocket) {

Vector3 vecBallDist;

float fHypotinuse, fOverlap;

//calculate the distance between ball & pocket

vecBallDist = m\_vecPosition - pocket.m\_vecPosition;

fHypotinuse = vecBallDist.Magnitude();

//check to see if the balls over the pocket enough

fOverlap = fHypotinuse - (m\_fRadius + pocket.m\_fRadius);

fOverlap \*= -1.0f;

if(fOverlap > m\_fRadius)

m\_bPocketed = true;

} //BallPocketCollision()

//---------------------------------------------------------------------------------------------------------------

// Name: BallCollisionResponse()

// Desc: find the resultant velocities of two balls after they have collided

//

void Balls::BallCollisionResponse(Balls &ball) {

//find the unit normal and unit tangent vectors of the balls that have collided

Vector3 vecNormalVector = ball.m\_vecPosition - m\_vecPosition;

Vector3 vecUnitNormalVector = vecNormalVector.Normalise();

Vector3 vecUnitTanVector = Vector3(-vecUnitNormalVector.m\_fZ, 0.0f, vecUnitNormalVector.m\_fX);

//calculate normal and tangential components of velocity

float fInitVelNormBall1 = vecUnitNormalVector.Dot(m\_vecVelocity);

float fInitVelNormBall2 = vecUnitNormalVector.Dot(ball.m\_vecVelocity);

float fInitVelTanBall1 = vecUnitTanVector.Dot(m\_vecVelocity);

float fInitVelTanBall2 = vecUnitTanVector.Dot(ball.m\_vecVelocity);

//calculate tangential velocity of balls after collision

float fVelTanBall1 = fInitVelTanBall1;

float fVelTanBall2 = fInitVelTanBall2;

//calculate normal velocity of balls after collision

//had to correct c++ here because it can't figure out 1D collisions using the equations properly

//the following code is the implementation of the one-dimensional elastic collisions equations adapted to use coefficient of restitution

float a = m\_fMass \* fInitVelNormBall1;

float b = ball.m\_fMass \* fInitVelNormBall2;

//final velocity in the normal direction of ball 2

float c = ball.m\_fMass \* (m\_fCoRest \* (fInitVelNormBall2 - fInitVelNormBall1));

float d = a + b + c;

//final velocity in the normal direction of ball 1

float j = m\_fMass \* (m\_fCoRest \* (fInitVelNormBall1 - fInitVelNormBall2));

float k = a + b + j;

float fTotalMass = m\_fMass + ball.m\_fMass;

float fVelNormBall1, fVelNormBall2;

//avoid division by 0

if(d == 0.0f)

fVelNormBall1 = 0.0f;

else

fVelNormBall1 = d/fTotalMass;

if(k == 0.0f)

fVelNormBall2 = 0.0f;

else

fVelNormBall2 = k/fTotalMass;

//calculate resultant velocity of balls after collision

Vector3 vecResNormVelBall1 = fVelNormBall1 \* vecUnitNormalVector;

Vector3 vecResTanVelBall1 = fVelTanBall1 \* vecUnitTanVector;

Vector3 vecResNormVelBall2 = fVelNormBall2 \* vecUnitNormalVector;

Vector3 vecResTanVelBall2 = fVelTanBall2 \* vecUnitNormalVector;

m\_vecVelocity = vecResNormVelBall1 + vecResTanVelBall1;

ball.m\_vecVelocity = vecResNormVelBall2 + vecResTanVelBall2;

m\_bCollided = false;

ball.m\_bCollided= false;

} //BallCollisionResponse()

### FSM.h

#ifndef FSM\_H

#define FSM\_H

#include "Balls.h"

#include "ColourOGL.h"

//================ Includes Above ================|============== Classes Below ========================

class FSM

{

public:

FSM();

~FSM();

void CheckState();

void SetState(int state);

int GetState() {return m\_iGameState;} //returns the value of the current state

void SetNextBallColour(Balls &ball);

void RenderBitmapString(float x,float y,float z,void \*font,char \*string);

//============== Class Methods Above ==============|============== Member Variables Below ==============

bool m\_bFoul; //was the last shot a foul?

bool m\_bInstaLose; //has the black been potted illegally?

bool m\_bBallPocketed; //has the ball been potted?

bool m\_bGameOver; //is the game over?

bool m\_bHitFirstBall; //has the 1st ball been hit?

bool m\_bAnyBallHit; //has any ball been hit?

bool m\_bBallMoving; //are the balls moving?

bool m\_bHitWrongBall; //has the wrong ball been hit?

int m\_iNextBall; //the next ball to be hit

int m\_iPlayer; //who's turn is it

int m\_iGameState; //the current gamestate

int m\_iBallsLeft; //how many balls were on the table in the last shot

int m\_iCurrentBalls; //how many balls are currently on the table

enum { P1SHOOT = 0, P2SHOOT, GAMEOVER, POCKETWHITEFOUL, HITWRONGBALLFOUL, NOBALLHITFOUL, HELP }; //Gamestates

ColourOGL m\_colour;

};

#endif FSM\_H

### FSM.cpp

#include "FSM.h"

//---------------------------------------------------------------------------------------------------------------

// Name: FSM()

// Desc: default constructor

//

FSM::FSM() {

m\_bFoul = false;

m\_bInstaLose = false;

m\_bBallPocketed = false;

m\_bGameOver = false;

m\_bHitFirstBall = false;

m\_bBallMoving = false;

m\_bHitWrongBall = false;

m\_iPlayer = 2;

m\_iNextBall = 1;

m\_iGameState = P1SHOOT;

m\_iBallsLeft = 9;

m\_iCurrentBalls = 9;

} //FSM

//---------------------------------------------------------------------------------------------------------------

// Name: ~FSM()

// Desc: default destructor

//

FSM::~FSM() {

} //~FSM

//---------------------------------------------------------------------------------------------------------------

// Name: CheckState()

// Desc: Checks the current gamestate

//

void FSM::CheckState() {

switch(m\_iGameState) {

case P1SHOOT: //output whose turn it is to shoot

RenderBitmapString(-10.5,0.0,-23.0,GLUT\_BITMAP\_TIMES\_ROMAN\_24,"Player 1 must hit: ");

RenderBitmapString(10.0,0.0,-23.0,GLUT\_BITMAP\_TIMES\_ROMAN\_24,"Press H for help");

m\_bFoul = false;

break;

case P2SHOOT:

RenderBitmapString(-10.5,0.0,-23.0,GLUT\_BITMAP\_TIMES\_ROMAN\_24,"Player 2 must hit: ");

RenderBitmapString(10.0,0.0,-23.0,GLUT\_BITMAP\_TIMES\_ROMAN\_24,"Press H for help");

m\_bFoul = false;

break;

case POCKETWHITEFOUL: //if a foul is incurred then change the player turns and set the foul flag to true

if(m\_iPlayer == 1)

m\_iPlayer = 2;

else

m\_iPlayer = 1;

m\_bFoul = true;

break;

case HITWRONGBALLFOUL:

if(m\_iPlayer == 1)

m\_iPlayer = 2;

else

m\_iPlayer = 1;

m\_bFoul = true;

break;

case NOBALLHITFOUL:

if(m\_iPlayer == 1)

m\_iPlayer = 2;

else

m\_iPlayer = 1;

m\_bFoul = true;

break;

case GAMEOVER:

if(m\_bInstaLose) { //this bool returns true when a foul has been committed, it resets to false when all balls have stopped

if(m\_iPlayer == 1) { //if a foul has been committed when in this state then the player has potted the black illegally, thus they must forfeit the match

RenderBitmapString(-10.0,-5.0,0.0,GLUT\_BITMAP\_TIMES\_ROMAN\_24,"BLACK POTTED ILLEGALLY P2 WINS");

RenderBitmapString(-10.0,-5.0,-2.0,GLUT\_BITMAP\_TIMES\_ROMAN\_24,"PRESS ESC TO QUIT");

}

else {

RenderBitmapString(-10.0,-5.0,0.0,GLUT\_BITMAP\_TIMES\_ROMAN\_24,"BLACK POTTED ILLEGALLY P1 WINS");

RenderBitmapString(-10.0,-5.0,-2.0,GLUT\_BITMAP\_TIMES\_ROMAN\_24,"PRESS ESC TO QUIT");

}

}

else if(m\_iPlayer == 1) { //if the player has potted the black legally then that player wins

RenderBitmapString(-5.0,-5.0,0.0,GLUT\_BITMAP\_TIMES\_ROMAN\_24,"GAME OVER P1 WINS");

RenderBitmapString(-5.0,-5.0,-2.0,GLUT\_BITMAP\_TIMES\_ROMAN\_24,"PRESS ESC TO QUIT");

}

else {

RenderBitmapString(-5.0,-5.0,0.0,GLUT\_BITMAP\_TIMES\_ROMAN\_24,"GAME OVER P2 WINS");

RenderBitmapString(-5.0,-5.0,-2.0,GLUT\_BITMAP\_TIMES\_ROMAN\_24,"PRESS ESC TO QUIT");

}

break;

case HELP: //display help text when help is the current game state

RenderBitmapString(-15.0,-5.0,16.0,GLUT\_BITMAP\_HELVETICA\_18,"ARROW KEYS: Aim/Power");

RenderBitmapString(-15.0,-5.0,14.0,GLUT\_BITMAP\_HELVETICA\_18,"T/B: Topspin/Backspin");

RenderBitmapString(-15.0,-5.0,12.0,GLUT\_BITMAP\_HELVETICA\_18,"SPACEBAR: Shoot");

RenderBitmapString(-15.0,-5.0,10.0,GLUT\_BITMAP\_HELVETICA\_18,"H: Toggle Help");

RenderBitmapString(-15.0,-5.0,8.0,GLUT\_BITMAP\_HELVETICA\_18,"F1: Toggle Fullscreen");

RenderBitmapString(-15.0,-5.0,6.0,GLUT\_BITMAP\_HELVETICA\_18,"ESC: Quit");

RenderBitmapString(2.0,-5.0,16.0,GLUT\_BITMAP\_HELVETICA\_18,"RULES:");

RenderBitmapString(2.0,-5.0,15.0,GLUT\_BITMAP\_HELVETICA\_12,"Players take turns to pot balls.");

RenderBitmapString(2.0,-5.0,13.5,GLUT\_BITMAP\_HELVETICA\_12,"The first ball to be hit MUST be the ball shown at");

RenderBitmapString(2.0,-5.0,12.5,GLUT\_BITMAP\_HELVETICA\_12,"the bottom of the screen otherwise a foul is incurred.");

RenderBitmapString(2.0,-5.0,11.0,GLUT\_BITMAP\_HELVETICA\_12,"As long as the correct ball is hit first, balls may be");

RenderBitmapString(2.0,-5.0,10.0,GLUT\_BITMAP\_HELVETICA\_12,"potted in any order.");

RenderBitmapString(2.0,-5.0,8.5,GLUT\_BITMAP\_HELVETICA\_12,"The player who pots the black in a legal manner, wins.");

RenderBitmapString(2.0,-5.0,7.0,GLUT\_BITMAP\_HELVETICA\_12,"If the black is potted in an illegal manner (through a foul)");

RenderBitmapString(2.0,-5.0,6.0,GLUT\_BITMAP\_HELVETICA\_12,"the player who potted the black illegally, loses.");

RenderBitmapString(2.0,-1.0,-22.6,GLUT\_BITMAP\_HELVETICA\_18,"NEXT BALL");

RenderBitmapString(-24.5,-1.0,-4.0,GLUT\_BITMAP\_HELVETICA\_18,"AIM BALL");

RenderBitmapString(20.0,-5.0,-20.0,GLUT\_BITMAP\_HELVETICA\_18,"POW BAR");

break;

} //end switch

} //CheckState

//---------------------------------------------------------------------------------------------------------------

// Name: SetState()

// Desc: Sets the current gamestate

//

void FSM::SetState(int state) {

m\_iGameState = state;

} //SetState

//---------------------------------------------------------------------------------------------------------------

// Name: SetNextBallColour()

// Desc: Sets the colour of the next ball to be hit

//

void FSM::SetNextBallColour(Balls &ball) {

m\_iNextBall = ball.m\_iID;

switch(m\_iNextBall) {

case 1:

m\_colour.Set(1.0f,0.0f,0.0f);

break;

case 2:

m\_colour.Set(0.0f,0.0f,1.0f);

break;

case 3:

m\_colour.Set(1.0f,1.0f,0.0f);

break;

case 4:

m\_colour.Set(0.0f,0.5f,0.0f);

break;

case 5:

m\_colour.Set(1.0f,0.50f,0.0f);

break;

case 6:

m\_colour.Set(0.63f,0.13f,0.94f);

break;

case 7:

m\_colour.Set(0.55f,0.27f,0.07f);

break;

case 8:

m\_colour.Set(1.0f,0.75f,0.79f);

break;

case 9:

m\_colour.Set(0.0f,0.0f,0.0f);

break;

} //end switch

} //SetNextBallColour

//---------------------------------------------------------------------------------------------------------------

// Name: RenderBitmapString()

// Desc: Renders text onto the screen

//

void FSM::RenderBitmapString(float x, float y, float z, void \*font, char \*string) {

char \*c;

glRasterPos3f(x,y,z);

for (c=string; \*c != '\0'; c++)

glutBitmapCharacter(font, \*c);

} //RenderBitmapString

### GameObjects.h

#ifndef GAMEOBJECTS\_H

#define GAMEOBJECTS\_H

#include <cmath>

#include "glut.h"

#include "ColourOGL.h"

#include "Balls.h"

#include "Vector3.h"

#include "FSM.h"

//================ Includes Above ================|============== Macros Below =======================

#define NUMBALLS 10 //increase this if you add more balls

#define NUMPOCKETS 6

#define PI 3.14159265358979323846264338327950288419716939937510

//=============== Macros Above ===================|============== Enumerators Below ==================

enum { WHITE = 0, RED, BLUE, YELLOW, GREEN, ORANGE, PURPLE, BROWN, PINK, BLACK }; //the balls

enum { UPLEFT = 0, UPMID, UPRIGHT, LOWLEFT, LOWMID, LOWRIGHT }; //the pockets

//============== Enumerators Above ===============|============== Classes Below ======================

class GameObjects

{

public:

GameObjects();

virtual ~GameObjects();

void SpawnBalls();

void CreatePockets();

void Table();

void AimBall();

void NextBall();

void AimDot();

void AimLine();

void PowerGauge();

int InitObjects();

//=========== Game Object Methods Above ===========|============ Game Logic Methods Below ============

Vector3 GetCircumPoints(float radius, Vector3 pos, float radial);

Vector3 GetSpherePoints(float radius, Vector3 pos, float radial, float elevation); //this function is now redundant as angular motion is not implemented

//Ball status

void WorkoutVel();

void BallMoving();

void UpdateMovingBalls();

bool BallStationary();

//Collisions

void RailCollision();

void PocketCollision();

bool BallBallCollision();

void BallBallResponse();

//FiniteStateMachine

void UseFSM();

//============= Class Methods Above =============|============== Member Variables Below ==============

float m\_fForceCue; //initial force of the cue strike

float m\_fRads; //radius of a imaginary circle, used to determine the length of the aiming line

float m\_fAngle; //angle used to determine the aiming direction

float m\_fCOFr; //coefficient of friction converted to program units

Vector3 m\_vecPoint; //The position of the aiming dot

Vector3 m\_vecAimLine; //The position of the aiming line

Balls m\_CBalls[NUMBALLS]; //Array of balls

FSM m\_FiniteStateMachine;

private:

ColourOGL m\_colour; //Sets the colour of the object

Balls m\_CPockets[NUMPOCKETS]; //Array of pockets

};

#endif

### GameObjects.cpp

#include "GameObjects.h"

//---------------------------------------------------------------------------------------------------------------

// Name: GameObjects()

// Desc: default constructor

//

GameObjects::GameObjects() {

m\_fForceCue = 0.5f;

m\_fRads = 20.0;

m\_fAngle = 270.0f;

//m\_fCOFr = 0.0001f; //laptop

m\_fCOFr = 0.0003f; //pc

} //GameObjects()

//---------------------------------------------------------------------------------------------------------------

// Name: ~GameObjects()

// Desc: default destructor

//

GameObjects::~GameObjects() {

} //~GameObjects()

//---------------------------------------------------------------------------------------------------------------

// Name: SpawnBalls()

// Desc: spawns all the balls using the CreateBall method from the Balls class

//

void GameObjects::SpawnBalls() {

m\_colour.Set(1.0f,1.0f,1.0f); //set the colour of the ball

m\_CBalls[WHITE].CreateBall(m\_colour); //create the ball

if(m\_CBalls[WHITE].m\_bPocketed) { //what to do when the ball has been potted

if(!BallStationary()){

m\_CBalls[WHITE].m\_vecPosition = Vector3(-12.0f,-1.0f,21.0f); //move the ball off the table is ball are still moving

m\_CBalls[WHITE].m\_vecVelocity = Vector3(0.0f,0.0f,0.0f);

}

else {

m\_CBalls[WHITE].m\_vecPosition = Vector3(10.0f,-1.0f,0.0f); //move it back onto the table when all balls have stopped

m\_CBalls[WHITE].m\_bPocketed = false;

}

}

m\_colour.Set(1.0f,0.0f,0.0f); //set the colour of the ball

m\_CBalls[RED].CreateBall(m\_colour); //create the ball

if(m\_CBalls[RED].m\_bPocketed) { //what to do when the ball has been potted

m\_CBalls[RED].m\_vecPosition = Vector3(-10.0f,-1.0f,21.0f);

m\_CBalls[RED].m\_vecVelocity = Vector3(0.0f,0.0f,0.0f);

m\_CBalls[RED].m\_bOutOfPlay = true; //once the ball has been potted, it is out of play for good

}

m\_colour.Set(0.0f,0.0f,1.0f);

m\_CBalls[BLUE].CreateBall(m\_colour);

if(m\_CBalls[BLUE].m\_bPocketed) {

m\_CBalls[BLUE].m\_vecPosition = Vector3(-8.0f,-1.0f,21.0f);

m\_CBalls[BLUE].m\_vecVelocity = Vector3(0.0f,0.0f,0.0f);

m\_CBalls[BLUE].m\_bOutOfPlay = true;

}

m\_colour.Set(1.0f,1.0f,0.0f);

m\_CBalls[YELLOW].CreateBall(m\_colour);

if(m\_CBalls[YELLOW].m\_bPocketed) {

m\_CBalls[YELLOW].m\_vecPosition = Vector3(-6.0f,-1.0f,21.0f);

m\_CBalls[YELLOW].m\_vecVelocity = Vector3(0.0f,0.0f,0.0f);

m\_CBalls[YELLOW].m\_bOutOfPlay = true;

}

m\_colour.Set(0.0f,0.5f,0.0f);

m\_CBalls[GREEN].CreateBall(m\_colour);

if(m\_CBalls[GREEN].m\_bPocketed) {

m\_CBalls[GREEN].m\_vecPosition = Vector3(-4.0f,-1.0f,21.0f);

m\_CBalls[GREEN].m\_vecVelocity = Vector3(0.0f,0.0f,0.0f);

m\_CBalls[GREEN].m\_bOutOfPlay = true;

}

m\_colour.Set(1.0f,0.50f,0.0f);

m\_CBalls[ORANGE].CreateBall(m\_colour);

if(m\_CBalls[ORANGE].m\_bPocketed) {

m\_CBalls[ORANGE].m\_vecPosition = Vector3(-2.0f,-1.0f,21.0f);

m\_CBalls[ORANGE].m\_vecVelocity = Vector3(0.0f,0.0f,0.0f);

m\_CBalls[ORANGE].m\_bOutOfPlay = true;

}

m\_colour.Set(0.63f,0.13f,0.94f);

m\_CBalls[PURPLE].CreateBall(m\_colour);

if(m\_CBalls[PURPLE].m\_bPocketed) {

m\_CBalls[PURPLE].m\_vecPosition = Vector3(0.0f,-1.0f,21.0f);

m\_CBalls[PURPLE].m\_vecVelocity = Vector3(0.0f,0.0f,0.0f);

m\_CBalls[PURPLE].m\_bOutOfPlay = true;

}

m\_colour.Set(0.55f,0.27f,0.07f);

m\_CBalls[BROWN].CreateBall(m\_colour);

if(m\_CBalls[BROWN].m\_bPocketed) {

m\_CBalls[BROWN].m\_vecPosition = Vector3(2.0f,-1.0f,21.0f);

m\_CBalls[BROWN].m\_vecVelocity = Vector3(0.0f,0.0f,0.0f);

m\_CBalls[BROWN].m\_bOutOfPlay = true;

}

m\_colour.Set(1.0f,0.75f,0.79f);

m\_CBalls[PINK].CreateBall(m\_colour);

if(m\_CBalls[PINK].m\_bPocketed) {

m\_CBalls[PINK].m\_vecPosition = Vector3(4.0f,-1.0f,21.0f);

m\_CBalls[PINK].m\_vecVelocity = Vector3(0.0f,0.0f,0.0f);

m\_CBalls[PINK].m\_bOutOfPlay = true;

}

m\_colour.Set(0.0f,0.0f,0.0f);

m\_CBalls[BLACK].CreateBall(m\_colour);

if(m\_CBalls[BLACK].m\_bPocketed) {

m\_CBalls[BLACK].m\_vecPosition = Vector3(6.0f,-1.0f,21.0f);

m\_CBalls[BLACK].m\_vecVelocity = Vector3(0.0f,0.0f,0.0f);

m\_CBalls[BLACK].m\_bOutOfPlay = true;

}

} //SpawnBalls()

//---------------------------------------------------------------------------------------------------------------

// Name: CreatePockets()

// Desc: creates all the pockets on the table

//

void GameObjects::CreatePockets() {

//glPushMatrix();

m\_colour.Set(0.6f,0.6f,0.6f);

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

m\_CPockets[i].CreateBall(m\_colour);

//glPopMatrix();

} //CreatePockets()

//---------------------------------------------------------------------------------------------------------------

// Name: Table()

// Desc: creates the table

//

void GameObjects::Table() {

glPushMatrix();

glColor3f(0.0,1.0,0.5); //set the table colour to green

glScaled(1.0,0.01,0.9); //scale the table

glutSolidCube(40.0); //create the cube for the table

glPopMatrix();

//top rail

glBegin(GL\_QUADS); //create the rails using GL\_QUADS

glColor3f(0.55,0.27,0.07);

glVertex3f(-18.80,0.0,18.0); //bottom left of quad

glVertex3f(-1.18,0.0,18.0); //bottom right of quad

glVertex3f(-0.75,0.0,19.5); //top right of quad

glVertex3f(-19.20,0.0,19.5); //top left of quad

glColor3f(0.55,0.27,0.07);

glVertex3f(1.18,0.0,18.0); //bottom left of quad

glVertex3f(18.80,0.0,18.0); //bottom right of quad

glVertex3f(19.20,0.0,19.5); //top right of quad

glVertex3f(0.75,0.0,19.5); //top left of quad

glEnd();

//bottom rail

glBegin(GL\_QUADS);

glColor3f(0.55,0.27,0.07);

glVertex3f(-1.18,0.0,-18.0); //top right of quad

glVertex3f(-18.80,0.0,-18.0); //top left of quad

glVertex3f(-19.20,0.0,-19.5); //bottom left of quad

glVertex3f(-0.75,0.0,-19.5); //bottom right of quad

glColor3f(0.55,0.27,0.07);

glVertex3f(18.80,0.0,-18.0); //top right of quad

glVertex3f(1.18,0.0,-18.0); //top left of quad

glVertex3f(0.75,0.0,-19.5); //bottom left of quad

glVertex3f(19.20,0.0,-19.5); //bottom right of quad

glEnd();

//left rail

glBegin(GL\_QUADS);

glColor3f(0.55,0.27,0.07);

glVertex3f(-20.0,0.0,17.10); //top right of quad

glVertex3f(-21.3,0.0,18.40); //top left of quad

glVertex3f(-21.3,0.0,-18.40); //bottom left of quad

glVertex3f(-20.0,0.0,-17.10); //bottom right of quad

glEnd();

//right rail

glBegin(GL\_QUADS);

glColor3f(0.55,0.27,0.07);

glVertex3f(21.3,0.0,18.40); //top right of quad

glVertex3f(20.0,0.0,17.10); //top left of quad

glVertex3f(20.0,0.0,-17.10); //bottom left of quad

glVertex3f(21.3,0.0,-18.40); //bottom right of quad

glEnd();

} //Table()

//---------------------------------------------------------------------------------------------------------------

// Name: AimBall()

// Desc: creates the aiming ball used to indicate top/backspin

//

void GameObjects::AimBall() {

glPushMatrix();

glColor3f(1.0,1.0,1.0);

glTranslated(-22.5,-1.0,0.0);

glutSolidSphere(2.0, 20.0,20.0);

glPopMatrix();

} //AimBall()

//---------------------------------------------------------------------------------------------------------------

// Name: NextBall()

// Desc: creates the next ball that must be hit

//

void GameObjects::NextBall() {

glPushMatrix();

glColor3f(m\_FiniteStateMachine.m\_colour.fRed,m\_FiniteStateMachine.m\_colour.fGreen,m\_FiniteStateMachine.m\_colour.fBlue); //use the FSM to determine the colour of the next ball

glTranslated(0.0,-1.0,-22.4);

glutSolidSphere(1.5, 20.0,20.0);

glPopMatrix();

} //NextBall()

//---------------------------------------------------------------------------------------------------------------

// Name: AimDot()

// Desc: creates the aiming dot used to indicate top/backspin on the aiming ball

//

void GameObjects::AimDot() {

glPointSize(5.0);

glBegin(GL\_POINTS);

glColor3f(0.0,0.0,0.0);

glVertex3f(m\_vecPoint.m\_fX, m\_vecPoint.m\_fY, m\_vecPoint.m\_fZ); //the position of the dot varies dependant on player input

glEnd();

glFlush();

} //AimDot()

//---------------------------------------------------------------------------------------------------------------

// Name: AimLine()

// Desc: creates the aiming line used to indicate top/backspin

//

void GameObjects::AimLine() {

glLineWidth(1.0);

glBegin(GL\_LINES);

glColor3f(0.0, 0.0, 0.0);

glVertex3f(m\_CBalls[WHITE].m\_vecPosition.m\_fX,-1,m\_CBalls[WHITE].m\_vecPosition.m\_fZ); //start point is the location of the centre of the cue ball

glVertex3f(m\_vecAimLine.m\_fX,m\_vecAimLine.m\_fY-2,m\_vecAimLine.m\_fZ); //end point is determined by using the GetCircumPoints method in DrawGLScene

glEnd();

} //AimLine()

//---------------------------------------------------------------------------------------------------------------

// Name: PowerGauge()

// Desc: creates the power bar used to indicate how strong you want to hit the cue ball

//

void GameObjects::PowerGauge() {

glLineWidth(10.0);

glBegin(GL\_LINES);

glColor3f(1.0, 1.0, 0.0);

glVertex3f(22.0,0.0,-18.0); //start of the line

glVertex3f(22.0,0.0,-18.0 + (m\_fForceCue\*25)); //end point is determined by the value of m\_fForceCue which varies dependant on player input

glEnd();

} //PowerGauge()

//---------------------------------------------------------------------------------------------------------------

// Name: InitObjects()

// Desc: initialise game objects at the beggining of the game

//

int GameObjects::InitObjects() {

//initialise balls

for(int i = 0; i < NUMBALLS; i++) { //set the relevant ball statuses

m\_CBalls[i].m\_bPocketed = false; //intially no balls have been potted

m\_CBalls[i].m\_bOutOfPlay = false; //all balls are in play

m\_CBalls[i].m\_iID = i; //assigning the ID's of the balls

m\_CBalls[i].m\_vecVelocity = Vector3(0.0f,0.0f,0.0f); //all balls are stationary

}

m\_CBalls[WHITE].m\_vecPosition = Vector3(10.0f,-1.0f,0.0f); //initialise the positions of the balls

m\_CBalls[RED].m\_vecPosition = Vector3(-10.0f,-1.0f,0.0f);

m\_CBalls[BLUE].m\_vecPosition = Vector3(-11.4f,-1.0f,0.75f);

m\_CBalls[YELLOW].m\_vecPosition = Vector3(-11.4f,-1.0f,-0.75f);

m\_CBalls[GREEN].m\_vecPosition = Vector3(-12.8f,-1.0f,-1.5f);

m\_CBalls[ORANGE].m\_vecPosition = Vector3(-12.8f,-1.0f,1.5f);

m\_CBalls[PURPLE].m\_vecPosition = Vector3(-14.2f,-1.0f,0.75f);

m\_CBalls[BROWN].m\_vecPosition = Vector3(-14.2f,-1.0f,-0.75f);

m\_CBalls[PINK].m\_vecPosition = Vector3(-15.6f,-1.0f,0.0f);

m\_CBalls[BLACK].m\_vecPosition = Vector3(-12.8f,-1.0f,0.0f);

//pockets

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

m\_CPockets[i].m\_fRadius = 1.2f;

m\_CPockets[UPLEFT].m\_vecPosition = Vector3(-20.0f,0.0f,18.5f); //initialise the position of the pockets

m\_CPockets[UPMID].m\_vecPosition = Vector3(0.0f,0.0f,18.5f);

m\_CPockets[UPRIGHT].m\_vecPosition = Vector3(20.0f,0.0f,18.5f);

m\_CPockets[LOWLEFT].m\_vecPosition = Vector3(-20.0f,0.0f,-18.5f);

m\_CPockets[LOWMID].m\_vecPosition = Vector3(0.0f,0.0f,-18.5f);

m\_CPockets[LOWRIGHT].m\_vecPosition = Vector3(20.0f,0.0f,-18.5f);

//other initialisations

m\_vecPoint = Vector3(-22.5f,-5.0f,0.0f); //initialise the position of the aiming dot

m\_CBalls[WHITE].m\_fCoRest = 0.90f; //initialise/reset the coefficient of restitution

return true;

} //InitObjects()

//---------------------------------------------------------------------------------------------------------------

// Name: GetCircumPoints()

// Desc: finds points on the circumference of a circle, given its radius, origin and angle

//

Vector3 GameObjects::GetCircumPoints(float radius, Vector3 pos, float radial) {

Vector3 a;

a.m\_fX = pos.m\_fX + radius \* sin(radial\*PI/180);

a.m\_fY = pos.m\_fY;

a.m\_fZ = pos.m\_fZ + radius \* cos(radial\*PI/180);

return a;

} //GetCircumPoints()

//---------------------------------------------------------------------------------------------------------------

// Name: GetSpherePoints()

// Desc: finds points on the surface of a sphere, given its radius, origin longtitude and latitude

//

Vector3 GameObjects::GetSpherePoints(float radius, Vector3 pos, float radial, float elevation) {

float i = (sin(elevation\*PI/180) \* sin(radial\*PI/180));

float j = cos(elevation\*PI/180);

float k = (sin(elevation\*PI/180) \* cos(radial\*PI/180));

Vector3 a;

a.m\_fX = pos.m\_fX + radius \* i;

a.m\_fY = pos.m\_fY + radius \* j;

a.m\_fZ = pos.m\_fZ + radius \* k;

return a;

} //GetSpherePoints()

//---------------------------------------------------------------------------------------------------------------

// Name: WorkoutVel()

// Desc: calculate velocity of the cue ball using the concept of impulse

//

void GameObjects::WorkoutVel() {

m\_CBalls[WHITE].WorkOutVelocity(m\_fForceCue, m\_fAngle);

} //WorkoutVel()

//---------------------------------------------------------------------------------------------------------------

// Name: BallMoving()

// Desc: checks if any balls are moving

//

void GameObjects::BallMoving() {

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

m\_CBalls[i].BallMoving();

} //BallMoving()

//---------------------------------------------------------------------------------------------------------------

// Name: UpdateMovingBalls()

// Desc: checks if any balls are moving

//

void GameObjects::UpdateMovingBalls() {

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

m\_CBalls[i].UpdateMovingBalls(m\_fCOFr);

} //UpdateMovingBalls()

//---------------------------------------------------------------------------------------------------------------

// Name: BallStationary()

// Desc: checks to see when all balls are stationary

//

bool GameObjects::BallStationary() {

for(int i = 0; i < NUMBALLS; i++) {

if(m\_CBalls[i].m\_bMoving)

return false;

}

return true;

} //BallStationary()

//---------------------------------------------------------------------------------------------------------------

// Name: RailCollision()

// Desc: checks to see when a ball collides with the table rails

//

void GameObjects::RailCollision() {

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

m\_CBalls[i].RailCollision();

} //RailCollision()

//---------------------------------------------------------------------------------------------------------------

// Name: PocketCollision()

// Desc: checks to see when a ball collides with the table pockets

//

void GameObjects::PocketCollision() {

for(int i = 0; i < NUMBALLS; i++) { //loop through the balls

for(int j = 0; j < NUMPOCKETS; j++) //loop through the pockets

m\_CBalls[i].BallPocketCollision(m\_CPockets[j]); //check for collsions between a ball and all pockets, then repeat process with another ball

}

} //PocketCollision()

//---------------------------------------------------------------------------------------------------------------

// Name: BallBallCollision()

// Desc: checks to see when a ball collides with another ball

//

bool GameObjects::BallBallCollision() {

for(int i = 0; i < NUMBALLS; i++) { //loop through the balls to find the 1st ball to compare

for(int j = NUMBALLS; j > 0; j--) { //loop through the balls to find the 2nd ball to compare

if(i != j) { //if the two balls are equal skip the check. i.e. you dont want to check if a ball collides with itself

if(m\_CBalls[i].BallCollision(m\_CBalls[j])) { //check for collisions between 2 balls here

m\_FiniteStateMachine.m\_bAnyBallHit = true; //a collision has occurred so this becomes true

return true;

}

} //end if

} //end nested for

} //end for

if(!BallStationary() && !m\_FiniteStateMachine.m\_bHitFirstBall) //if the 1st ball has not been hit, then no balls have been hit

m\_FiniteStateMachine.m\_bAnyBallHit = false;

return false;

} //BallBallCollision()

//---------------------------------------------------------------------------------------------------------------

// Name: BallBallResponse()

// Desc: resolves ball ball collisions using elastic collisions equation

//

void GameObjects::BallBallResponse() {

//check to see which balls have collided

int ballRep1 = 0,ballRep2 = 0;

bool skip = false;

for(int i = 0; i < NUMBALLS; i++) { //loop through the balls to find any that have collided

if(m\_CBalls[i].m\_bCollided) { //once a ball has been found, assign it to the appropriate ballRep (ball response) flag

if(!skip) {

ballRep1 = i;

skip = true; //once the 1st ball has been assigned, skip this step and move on to assign the 2nd ball

}

else { //assign the 2nd ball here

ballRep2 = i;

skip = false; //reset the skip check for the next loop through

i = NUMBALLS; //exit the loop once 2 balls have been found, perform the collisions response with the 2 balls

} //end if-else

} //end if

} // end for

//fixes the bug found in build 3.4.6:

//-Incorrect collision response when a higher numbered ball collides with a lower numbered ball.

//CAUSE: occurs when the speed of ballRep1 is less than that of ballRep2

if(m\_CBalls[ballRep1].m\_vecVelocity.Magnitude() < m\_CBalls[ballRep2].m\_vecVelocity.Magnitude()) { //swap the values of the bellRep if this is true

int temp = ballRep1;

ballRep1 = ballRep2;

ballRep2 = temp;

}

//check if the 1st ball hit is the correct ball

if(!m\_FiniteStateMachine.m\_bHitFirstBall) { //the first ball flag is resets everytime the balls are stationary

if(ballRep1 == 0 || ballRep2 == 0) { //if the white ball was involved in a collision

if(ballRep1 == 0) { //if ballRep1 was the white ball

if(ballRep2 != m\_FiniteStateMachine.m\_iNextBall) { //if ballRep2 is not the next ball that needs to be hit, then a foul has occurred

m\_FiniteStateMachine.m\_bHitWrongBall = true; //the wrong ball was hit. this bool will reset when the appropriate gamestate has been set

m\_FiniteStateMachine.m\_bHitFirstBall = true; //the white has hit the 1st ball so this is true. will be false again when all balls are stationary

}

}

else { //if ballRep1 isn't the white, then ballRep2 is

if(ballRep1 != m\_FiniteStateMachine.m\_iNextBall) { //perform the same checks as above but for ballRep1

m\_FiniteStateMachine.m\_bHitWrongBall = true;

m\_FiniteStateMachine.m\_bHitFirstBall = true;

}

} //end if-else

m\_FiniteStateMachine.m\_bHitFirstBall = true; //the first ball has been hit if this if statement has to execute, so we make sure to set this to true no matter what

} //end nested if

} //end if

//collision response using 2-dimensional inelastic collisions here

m\_CBalls[ballRep1].BallCollisionResponse(m\_CBalls[ballRep2]);

//after the collision no balls are in contact, thus update their status

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

m\_CBalls[i].m\_bCollided = false;

} //ballBallResponse()

//---------------------------------------------------------------------------------------------------------------

// Name: UseFSM()

// Desc: use the finite state machine class to determine the gamestate

//

void GameObjects::UseFSM() {

for(int i = 1; i < NUMBALLS; i++) { //find the next ball to be pocketed

if(!m\_CBalls[i].m\_bPocketed) {

m\_FiniteStateMachine.SetNextBallColour(m\_CBalls[i]);

i = NUMBALLS; //once the next ball has been found, exit loop

}

}

for(int i = 1; i < NUMBALLS; i++) { //find out how many balls have been pocketed

if(m\_CBalls[i].m\_bPocketed && !m\_CBalls[i].m\_bOutOfPlay)//needs 2 conditions otherwise m\_iBallsLeft will always decrement

m\_FiniteStateMachine.m\_iCurrentBalls--;

}

//Check for fouls first, then player turns, then winning/losing conditions

//=============================================================== Check for fouls ===============================================================

if(m\_CBalls[WHITE].m\_bPocketed) { //foul if the white is pocketed

m\_FiniteStateMachine.SetState(m\_FiniteStateMachine.POCKETWHITEFOUL);

}

else if(m\_FiniteStateMachine.m\_bHitWrongBall) { //foul if the wrong ball has been hit

m\_FiniteStateMachine.SetState(m\_FiniteStateMachine.HITWRONGBALLFOUL);

m\_FiniteStateMachine.m\_bHitWrongBall = false; //update this bool once the gamestate has been set

}

else if(!m\_FiniteStateMachine.m\_bAnyBallHit) { //foul if no balls were hit

m\_FiniteStateMachine.SetState(m\_FiniteStateMachine.NOBALLHITFOUL);

m\_FiniteStateMachine.m\_bAnyBallHit = true; //update this bool once the gamestate has been set

}

//============================================================== Check player turns ==============================================================

if(m\_FiniteStateMachine.m\_iPlayer == 2 && m\_FiniteStateMachine.m\_bFoul) { //update player turns after a foul

m\_FiniteStateMachine.SetState(m\_FiniteStateMachine.P2SHOOT);

}

else if(m\_FiniteStateMachine.m\_iPlayer == 1 && m\_FiniteStateMachine.m\_bFoul) { //update player turns after a foul

m\_FiniteStateMachine.SetState(m\_FiniteStateMachine.P1SHOOT);

}

else if(m\_FiniteStateMachine.m\_iCurrentBalls != m\_FiniteStateMachine.m\_iBallsLeft) { //players keep their turn as long as they pot a ball legally

if(m\_FiniteStateMachine.m\_iPlayer == 1 && !m\_FiniteStateMachine.m\_bInstaLose) {

m\_FiniteStateMachine.SetState(m\_FiniteStateMachine.P1SHOOT);

m\_FiniteStateMachine.m\_iBallsLeft = m\_FiniteStateMachine.m\_iCurrentBalls;

}

else if(m\_FiniteStateMachine.m\_iPlayer == 2 && !m\_FiniteStateMachine.m\_bInstaLose) {

m\_FiniteStateMachine.SetState(m\_FiniteStateMachine.P2SHOOT);

m\_FiniteStateMachine.m\_iBallsLeft = m\_FiniteStateMachine.m\_iCurrentBalls;

}

}

//=========================================================== Check winning conditions ===========================================================

if(m\_FiniteStateMachine.m\_iCurrentBalls == 0) //if all balls are pocketed, gameover

m\_FiniteStateMachine.SetState(m\_FiniteStateMachine.GAMEOVER);

else if(m\_CBalls[BLACK].m\_bPocketed) //if the black is potted, gameover

m\_FiniteStateMachine.SetState(m\_FiniteStateMachine.GAMEOVER);

//=============================================================== Update game state ==============================================================

if(!BallStationary() && m\_FiniteStateMachine.GetState() == m\_FiniteStateMachine.POCKETWHITEFOUL) { //Display foul type

m\_FiniteStateMachine.m\_bBallMoving = true;

m\_FiniteStateMachine.m\_bInstaLose = true; //If the black is potted when a foul has occurred then the player who committed the foul loses instantly

m\_FiniteStateMachine.RenderBitmapString(-15.0,0.0,-23.0,GLUT\_BITMAP\_TIMES\_ROMAN\_24,"FOUL White Pocketed");

}

else if(!BallStationary() && m\_FiniteStateMachine.GetState() == m\_FiniteStateMachine.HITWRONGBALLFOUL) { //Display foul type

m\_FiniteStateMachine.m\_bBallMoving = true;

m\_FiniteStateMachine.m\_bInstaLose = true;

m\_FiniteStateMachine.RenderBitmapString(-15.0,0.0,-23.0,GLUT\_BITMAP\_TIMES\_ROMAN\_24,"FOUL Hit Wrong Ball");

}

else if(!BallStationary()) { //display relevant game state

m\_FiniteStateMachine.m\_bBallMoving = true;

m\_FiniteStateMachine.RenderBitmapString(15.0,0.0,-23.0,GLUT\_BITMAP\_TIMES\_ROMAN\_24,"Balls Moving");

}

else if(BallStationary()) { //change the gamestate and reset the bool checks when the balls have stopped moving

m\_FiniteStateMachine.m\_bHitFirstBall = false;

m\_FiniteStateMachine.m\_bBallMoving = false;

if(m\_FiniteStateMachine.GetState() != m\_FiniteStateMachine.GAMEOVER)

m\_FiniteStateMachine.m\_bInstaLose = false;

m\_FiniteStateMachine.CheckState();

}

} //UseFSM()

### OGL.cpp

#include "global.h"

extern HDC hDC; // Private GDI Device Context

HGLRC hRC=NULL; // Permanent Rendering Context

HWND hWnd=NULL; // Holds Our Window Handle

HINSTANCE hInstance; // Holds The Instance Of The Application

extern bool fullscreen; // Fullscreen Flag Set To Fullscreen Mode By Default

extern GameObjects GObjects;// Use the GameObjects class here

GLvoid ReSizeGLScene(GLsizei width, GLsizei height) // Resize And Initialize The GL Window

{

if (height==0) // Prevent A Divide By Zero By

{

height=1; // Making Height Equal One

}

glViewport(0,0,width,height); // Reset The Current Viewport

glMatrixMode(GL\_PROJECTION); // Select The Projection Matrix

glLoadIdentity(); // Reset The Projection Matrix

// Calculate The Aspect Ratio Of The Window

glOrtho(-25.0,25.0,-25.0,25.0, 0.0, 200.0); // near and far plane is 0 and 200

glMatrixMode(GL\_MODELVIEW); // Select The Modelview Matrix

glLoadIdentity(); // Reset The Modelview Matrix

}

int InitGL(GLvoid) // All Setup For OpenGL Goes Here

{

glShadeModel(GL\_SMOOTH); // Enable Smooth Shading

glClearColor(0.0f, 0.5f, 0.7f, 0.0f); // Set Background Colour

glClearDepth(1.0f); // Depth Buffer Setup

glEnable(GL\_DEPTH\_TEST); // Enables Depth Testing

glDepthFunc(GL\_LEQUAL); // The Type Of Depth Testing To Do

glEnable(GL\_CULL\_FACE); // Enable Backface culling

glHint(GL\_PERSPECTIVE\_CORRECTION\_HINT, GL\_NICEST); // Really Nice Perspective Calculations

return true; // Initialization Went OK

}

int DrawGLScene(GLvoid) // Here's Where We Do All The Drawing

{

glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT); // Clear Screen And Depth Buffer

glMatrixMode(GL\_MODELVIEW);

glLoadIdentity(); // Reset The Current Modelview Matrix

gluLookAt(0.0,-50.0,0.0,0.0,0.0,0.0,0.0,0.0,1.0); // Setup the camera

glPolygonMode(GL\_FRONT,GL\_FILL); //Draw the polygon in line mode

glPolygonMode(GL\_BACK,GL\_FILL);

//===================== Drawing goes here ===========================

GObjects.UseFSM();

//==================== Create Game Objects ==========================

GObjects.SpawnBalls();

GObjects.CreatePockets();

GObjects.Table();

GObjects.NextBall();

GObjects.AimBall();

//reset the aiming dot on the aiming ball

if(GObjects.m\_CBalls[WHITE].m\_fCoRest == 0.90f)

GObjects.m\_vecPoint.m\_fZ = 0.0f;

GObjects.AimDot();

GObjects.PowerGauge();

//==================== Create Game Objects ==========================

//draw aim line on cue ball

if(GObjects.BallStationary()) {

GObjects.m\_vecAimLine = GObjects.GetCircumPoints(GObjects.m\_fRads, GObjects.m\_CBalls[WHITE].m\_vecPosition, GObjects.m\_fAngle);

GObjects.AimLine();

}

//update ball velocites while balls are moving

GObjects.BallMoving();

GObjects.UpdateMovingBalls();

//collision detection/response

GObjects.RailCollision();

GObjects.PocketCollision();

if(GObjects.BallBallCollision())

GObjects.BallBallResponse();

return true; // Everything Went OK

}

GLvoid KillGLWindow(GLvoid) // Properly Kill The Window

{

if (fullscreen) // Are We In Fullscreen Mode?

{

ChangeDisplaySettings(NULL,0); // If So Switch Back To The Desktop

ShowCursor(true); // Show Mouse Pointer

}

if (hRC) // Do We Have A Rendering Context?

{

if (!wglMakeCurrent(NULL,NULL)) // Are We Able To Release The DC And RC Contexts?

{

MessageBox(NULL,"Release Of DC And RC Failed.","SHUTDOWN ERROR",MB\_OK | MB\_ICONINFORMATION);

}

if (!wglDeleteContext(hRC)) // Are We Able To Delete The RC?

{

MessageBox(NULL,"Release Rendering Context Failed.","SHUTDOWN ERROR",MB\_OK | MB\_ICONINFORMATION);

}

hRC=NULL; // Set RC To NULL

}

if (hDC && !ReleaseDC(hWnd,hDC)) // Are We Able To Release The DC

{

MessageBox(NULL,"Release Device Context Failed.","SHUTDOWN ERROR",MB\_OK | MB\_ICONINFORMATION);

hDC=NULL; // Set DC To NULL

}

if (hWnd && !DestroyWindow(hWnd)) // Are We Able To Destroy The Window?

{

MessageBox(NULL,"Could Not Release hWnd.","SHUTDOWN ERROR",MB\_OK | MB\_ICONINFORMATION);

hWnd=NULL; // Set hWnd To NULL

}

if (!UnregisterClass("OpenGL",hInstance)) // Are We Able To Unregister Class

{

MessageBox(NULL,"Could Not Unregister Class.","SHUTDOWN ERROR",MB\_OK | MB\_ICONINFORMATION);

hInstance=NULL; // Set hInstance To NULL

}

}

/\* This Code Creates Our OpenGL Window. Parameters Are: \*

\* title - Title To Appear At The Top Of The Window \*

\* width - Width Of The GL Window Or Fullscreen Mode \*

\* height - Height Of The GL Window Or Fullscreen Mode \*

\* bits - Number Of Bits To Use For Color (8/16/24/32) \*

\* fullscreenflag - Use Fullscreen Mode (true) Or Windowed Mode (false) \*/

BOOL CreateGLWindow(char\* title, int width, int height, int bits, bool fullscreenflag)

{

GLuint PixelFormat; // Holds The Results After Searching For A Match

WNDCLASS wc; // Windows Class Structure

DWORD dwExStyle; // Window Extended Style

DWORD dwStyle; // Window Style

RECT WindowRect; // Grabs Rectangle Upper Left / Lower Right Values

WindowRect.left=(long)0; // Set Left Value To 0

WindowRect.right=(long)width; // Set Right Value To Requested Width

WindowRect.top=(long)0; // Set Top Value To 0

WindowRect.bottom=(long)height; // Set Bottom Value To Requested Height

fullscreen=fullscreenflag; // Set The Global Fullscreen Flag

hInstance = GetModuleHandle(NULL); // Grab An Instance For Our Window

wc.style = CS\_HREDRAW | CS\_VREDRAW | CS\_OWNDC; // Redraw On Size, And Own DC For Window.

wc.lpfnWndProc = (WNDPROC) WndProc; // WndProc Handles Messages

wc.cbClsExtra = 0; // No Extra Window Data

wc.cbWndExtra = 0; // No Extra Window Data

wc.hInstance = hInstance; // Set The Instance

wc.hIcon = LoadIcon(NULL, IDI\_WINLOGO); // Load The Default Icon

wc.hCursor = LoadCursor(NULL, IDC\_ARROW); // Load The Arrow Pointer

wc.hbrBackground = NULL; // No Background Required For GL

wc.lpszMenuName = NULL; // We Don't Want A Menu

wc.lpszClassName = "OpenGL"; // Set The Class Name

if (!RegisterClass(&wc)) // Attempt To Register The Window Class

{

MessageBox(NULL,"Failed To Register The Window Class.","ERROR",MB\_OK|MB\_ICONEXCLAMATION);

return false; // Return false

}

if (fullscreen) // Attempt Fullscreen Mode?

{

DEVMODE dmScreenSettings; // Device Mode

memset(&dmScreenSettings,0,sizeof(dmScreenSettings)); // Makes Sure Memory's Cleared

dmScreenSettings.dmSize=sizeof(dmScreenSettings); // Size Of The Devmode Structure

dmScreenSettings.dmPelsWidth = width; // Selected Screen Width

dmScreenSettings.dmPelsHeight = height; // Selected Screen Height

dmScreenSettings.dmBitsPerPel = bits; // Selected Bits Per Pixel

dmScreenSettings.dmFields=DM\_BITSPERPEL|DM\_PELSWIDTH|DM\_PELSHEIGHT;

// Try To Set Selected Mode And Get Results. NOTE: CDS\_FULLSCREEN Gets Rid Of Start Bar.

if (ChangeDisplaySettings(&dmScreenSettings,CDS\_FULLSCREEN)!=DISP\_CHANGE\_SUCCESSFUL)

{

// If The Mode Fails, Offer Two Options. Quit Or Use Windowed Mode.

if (MessageBox(NULL,"The Requested Fullscreen Mode Is Not Supported By\nYour Video Card. Use Windowed Mode Instead?","OpenGL",MB\_YESNO|MB\_ICONEXCLAMATION)==IDYES)

{

fullscreen=false; // Windowed Mode Selected. Fullscreen = false

}

else

{

// Pop Up A Message Box Letting User Know The Program Is Closing.

MessageBox(NULL,"Program Will Now Close.","ERROR",MB\_OK|MB\_ICONSTOP);

return false; // Return false

}

}

}

if (fullscreen) // Are We Still In Fullscreen Mode?

{

dwExStyle=WS\_EX\_APPWINDOW; // Window Extended Style

dwStyle=WS\_POPUP; // Windows Style

ShowCursor(false); // Hide Mouse Pointer

}

else

{

dwExStyle=WS\_EX\_APPWINDOW | WS\_EX\_WINDOWEDGE; // Window Extended Style

dwStyle=WS\_OVERLAPPEDWINDOW; // Windows Style

}

AdjustWindowRectEx(&WindowRect, dwStyle, false, dwExStyle); // Adjust Window To true Requested Size

// Create The Window

if (!(hWnd=CreateWindowEx( dwExStyle, // Extended Style For The Window

"OpenGL", // Class Name

title, // Window Title

dwStyle | // Defined Window Style

WS\_CLIPSIBLINGS | // Required Window Style

WS\_CLIPCHILDREN, // Required Window Style

0, 0, // Window Position

WindowRect.right-WindowRect.left, // Calculate Window Width

WindowRect.bottom-WindowRect.top, // Calculate Window Height

NULL, // No Parent Window

NULL, // No Menu

hInstance, // Instance

NULL))) // Dont Pass Anything To WM\_CREATE

{

KillGLWindow(); // Reset The Display

MessageBox(NULL,"Window Creation Error.","ERROR",MB\_OK|MB\_ICONEXCLAMATION);

return false; // Return false

}

static PIXELFORMATDESCRIPTOR pfd= // pfd Tells Windows How We Want Things To Be

{

sizeof(PIXELFORMATDESCRIPTOR), // Size Of This Pixel Format Descriptor

1, // Version Number

PFD\_DRAW\_TO\_WINDOW | // Format Must Support Window

PFD\_SUPPORT\_OPENGL | // Format Must Support OpenGL

PFD\_DOUBLEBUFFER, // Must Support Double Buffering

PFD\_TYPE\_RGBA, // Request An RGBA Format

bits, // Select Our Color Depth

0, 0, 0, 0, 0, 0, // Color Bits Ignored

0, // No Alpha Buffer

0, // Shift Bit Ignored

0, // No Accumulation Buffer

0, 0, 0, 0, // Accumulation Bits Ignored

16, // 16Bit Z-Buffer (Depth Buffer)

0, // No Stencil Buffer

0, // No Auxiliary Buffer

PFD\_MAIN\_PLANE, // Main Drawing Layer

0, // Reserved

0, 0, 0 // Layer Masks Ignored

};

if (!(hDC=GetDC(hWnd))) // Did We Get A Device Context?

{

KillGLWindow(); // Reset The Display

MessageBox(NULL,"Can't Create A GL Device Context.","ERROR",MB\_OK|MB\_ICONEXCLAMATION);

return false; // Return false

}

if (!(PixelFormat=ChoosePixelFormat(hDC,&pfd))) // Did Windows Find A Matching Pixel Format?

{

KillGLWindow(); // Reset The Display

MessageBox(NULL,"Can't Find A Suitable PixelFormat.","ERROR",MB\_OK|MB\_ICONEXCLAMATION);

return false; // Return false

}

if(!SetPixelFormat(hDC,PixelFormat,&pfd)) // Are We Able To Set The Pixel Format?

{

KillGLWindow(); // Reset The Display

MessageBox(NULL,"Can't Set The PixelFormat.","ERROR",MB\_OK|MB\_ICONEXCLAMATION);

return false; // Return false

}

if (!(hRC=wglCreateContext(hDC))) // Are We Able To Get A Rendering Context?

{

KillGLWindow(); // Reset The Display

MessageBox(NULL,"Can't Create A GL Rendering Context.","ERROR",MB\_OK|MB\_ICONEXCLAMATION);

return false; // Return false

}

if(!wglMakeCurrent(hDC,hRC)) // Try To Activate The Rendering Context

{

KillGLWindow(); // Reset The Display

MessageBox(NULL,"Can't Activate The GL Rendering Context.","ERROR",MB\_OK|MB\_ICONEXCLAMATION);

return false; // Return false

}

ShowWindow(hWnd,SW\_SHOW); // Show The Window

SetForegroundWindow(hWnd); // Slightly Higher Priority

SetFocus(hWnd); // Sets Keyboard Focus To The Window

ReSizeGLScene(width, height); // Set Up Our Perspective GL Screen

if (!InitGL()) // Initialize Our Newly Created GL Window

{

KillGLWindow(); // Reset The Display

MessageBox(NULL,"Initialization Failed.","ERROR",MB\_OK|MB\_ICONEXCLAMATION);

return false; // Return false

}

if (!GObjects.InitObjects()) // Initialise the game objects

{

KillGLWindow(); // Reset The Display

MessageBox(NULL,"Game Object Initialization Failed.","ERROR",MB\_OK|MB\_ICONEXCLAMATION);

return false; // Return false

}

return true; // Success

}

### WinMain.cpp

#include "global.h"

bool keys[256]; // Array Used For The Keyboard Routine

bool active=true; // Window Active Flag Set To true By Default

bool fullscreen=true; // Fullscreen Flag Set To Fullscreen Mode By Default

HDC hDC = NULL; // Private GDI Device Context

GameObjects GObjects;

LRESULT CALLBACK WndProc( HWND hWnd, // Handle For This Window

UINT uMsg, // Message For This Window

WPARAM wParam, // Additional Message Information

LPARAM lParam) // Additional Message Information

{

switch (uMsg) // Check For Windows Messages

{

case WM\_ACTIVATE: // Watch For Window Activate Message

{

if (!HIWORD(wParam)) // Check Minimization State

{

active=true; // Program Is Active

}

else

{

active=false; // Program Is No Longer Active

}

return 0; // Return To The Message Loop

}

case WM\_SYSCOMMAND: // Intercept System Commands

{

switch (wParam) // Check System Calls

{

case SC\_SCREENSAVE: // Screensaver Trying To Start?

case SC\_MONITORPOWER: // Monitor Trying To Enter Powersave?

return 0; // Prevent From Happening

}

break; // Exit

}

case WM\_CLOSE: // Did We Receive A Close Message?

{

PostQuitMessage(0); // Send A Quit Message

return 0; // Jump Back

}

case WM\_KEYDOWN: // Is A Key Being Held Down?

{

keys[wParam] = true; // If So, Mark It As true

return 0; // Jump Back

}

case WM\_KEYUP: // Has A Key Been Released?

{

keys[wParam] = false; // If So, Mark It As false

return 0; // Jump Back

}

case WM\_SIZE: // Resize The OpenGL Window

{

ReSizeGLScene(LOWORD(lParam),HIWORD(lParam)); // LoWord=Width, HiWord=Height

return 0; // Jump Back

}

}

// Pass All Unhandled Messages To DefWindowProc

return DefWindowProc(hWnd,uMsg,wParam,lParam);

}

int WINAPI WinMain( HINSTANCE hInstance, // Instance

HINSTANCE hPrevInstance, // Previous Instance

LPSTR lpCmdLine, // Command Line Parameters

int nCmdShow) // Window Show State

{

MSG msg; // Windows Message Structure

BOOL done=false; // Bool Variable To Exit Loop

// Ask The User Which Screen Mode They Prefer

if (MessageBox(NULL,"Would You Like To Run In Fullscreen Mode?", "Start FullScreen?",MB\_YESNO|MB\_ICONQUESTION)==IDNO)

{

fullscreen=false; // Windowed Mode

}

// Create Our OpenGL Window

if (!CreateGLWindow("Jason Luong:11277565 FINAL YEAR PROJECT",1024,768,16,fullscreen))

{

return 0; // Quit If Window Was Not Created

}

while(!done) // Loop That Runs While done=false

{

if (PeekMessage(&msg,NULL,0,0,PM\_REMOVE)) // Is There A Message Waiting?

{

if (msg.message==WM\_QUIT) // Have We Received A Quit Message?

{

done=true; // If So done=true

}

else // If Not, Deal With Window Messages

{

TranslateMessage(&msg); // Translate The Message

DispatchMessage(&msg); // Dispatch The Message

}

}

else // If There Are No Messages

{

// Draw The Scene. Watch For ESC Key And Quit Messages From DrawGLScene()

if (active) // Program Active?

{

if (keys[VK\_ESCAPE]) // Was ESC Pressed?

{

keys[VK\_ESCAPE]=false; // If So Make Key false

// Ask The User If They Want To Quit

if(MessageBox(NULL,"Are you sure you want to quit?", "Quit?",MB\_YESNO|MB\_ICONWARNING)== IDYES)

done=true; // ESC Signalled A Quit

else

done=false;

}

else // Not Time To Quit, Update Screen

{

DrawGLScene(); // Draw The Scene

SwapBuffers(hDC); // Swap Buffers (Double Buffering)

}

}

//=================================== GAME CONTROLS ===================================

if (keys[VK\_F1]) // Is F1 Being Pressed?

{

keys[VK\_F1]=false; // If So Make Key false

KillGLWindow(); // Kill Our Current Window

fullscreen=!fullscreen; // Toggle Fullscreen / Windowed Mode

// Recreate Our OpenGL Window

if (!CreateGLWindow("Jason Luong:11277565 FINAL YEAR PROJECT",1024,768,16,fullscreen))

{

return 0; // Quit If Window Was Not Created

}

}

static bool bPowMax = false;

static bool bPowMin = false;

static int iPrevState = 0;

if (keys[VK\_SPACE]) // Is Spacebar Being Pressed?

{

keys[VK\_SPACE]=false; // If So Make Key false

if(GObjects.BallStationary())

GObjects.WorkoutVel(); // Hit the cue ball if all balls are stationary

}

if (keys[VK\_LEFT]) // Is Left Arrow Being Pressed?

{

if(GObjects.m\_fAngle == 0.0f) // Rotate aim line to aim the ball

GObjects.m\_fAngle = 360.0f;

else

//GObjects.m\_fAngle-=0.2f; //laptop

GObjects.m\_fAngle-=0.5f; //pc

}

if (keys[VK\_RIGHT]) // Is Right Arrow Being Pressed?

{

if(GObjects.m\_fAngle == 360.0f) // Rotate aim line to aim the ball

GObjects.m\_fAngle = 0.0f;

else

//GObjects.m\_fAngle+=0.2f; //laptop

GObjects.m\_fAngle+=0.5f; //pc

}

if (keys[VK\_UP]) // Is Up Arrow Being Pressed?

{

if(!bPowMax) { // Increase the force to hit the cue ball

if(GObjects.m\_fForceCue >= 0.75f) {

GObjects.m\_fForceCue = 0.75f;

bPowMax = true;

}

else {

//GObjects.m\_fForceCue += 0.009f; //laptop

GObjects.m\_fForceCue += 0.02f; //pc

bPowMin = false;

}

} //end if

}

if (keys[VK\_DOWN]) // Is Down Arrow Being Pressed?

{

if(!bPowMin) { // Decrease the force to hit the cue ball

if(GObjects.m\_fForceCue <= 0.02f) {

GObjects.m\_fForceCue = 0.02f;

bPowMin = true;

}

else {

//GObjects.m\_fForceCue -= 0.009f; //laptop

GObjects.m\_fForceCue -= 0.02f; //pc

bPowMax = false;

}

} //end if

}

static bool bSpinMax = false;

static bool bSpinMin = false;

if (keys['T']) // Is T Being Pressed?

{

keys['T']=false; // If So Make Key false

if(!bSpinMax) { // Add Topspin

if(GObjects.m\_CBalls[WHITE].m\_fCoRest > 0.30f) {

GObjects.m\_CBalls[WHITE].m\_fCoRest -= 0.10f;

GObjects.m\_vecPoint.m\_fZ += 0.2f;

bSpinMin = false;

}

else {

GObjects.m\_CBalls[WHITE].m\_fCoRest = 0.30f;

GObjects.m\_vecPoint.m\_fZ = 1.0f;

bSpinMax = true;

}

} //end if

}

if (keys['B']) // Is B Being Pressed?

{

keys['B']=false; // If So Make Key false

if(!bSpinMin) { // Add Backspin

if(GObjects.m\_CBalls[WHITE].m\_fCoRest < 1.30f) {

GObjects.m\_CBalls[WHITE].m\_fCoRest += 0.10f;

GObjects.m\_vecPoint.m\_fZ -= 0.2f;

bSpinMax = false;

}

else {

GObjects.m\_CBalls[WHITE].m\_fCoRest = 1.30f;

GObjects.m\_vecPoint.m\_fZ = -1.0f;

bSpinMin = true;

}

} //end if

}

if (keys['H']) // Is H Being Pressed?

{

keys['H']=false; // If So Make Key false

if(GObjects.m\_FiniteStateMachine.GetState() != GObjects.m\_FiniteStateMachine.HELP) //if the current state is not HELP

iPrevState = GObjects.m\_FiniteStateMachine.GetState(); //store the previous state

if(GObjects.BallStationary() && GObjects.m\_FiniteStateMachine.GetState() != GObjects.m\_FiniteStateMachine.HELP) {

GObjects.m\_FiniteStateMachine.SetState(GObjects.m\_FiniteStateMachine.HELP); //show the help screen if balls are stationary and the current state is not already HELP

}

else if(iPrevState != GObjects.m\_FiniteStateMachine.HELP) {

GObjects.m\_FiniteStateMachine.SetState(iPrevState); //return to the previous state if the current state is already HELP

}

}

//=================================== END GAME CONTROLS ===================================

} //end else

} //end while

// Shutdown

KillGLWindow(); // Kill The Window

return ((int)msg.wParam); // Exit The Program

}