ACW 1: Distributed Table Tennis Report

Design and Implementation

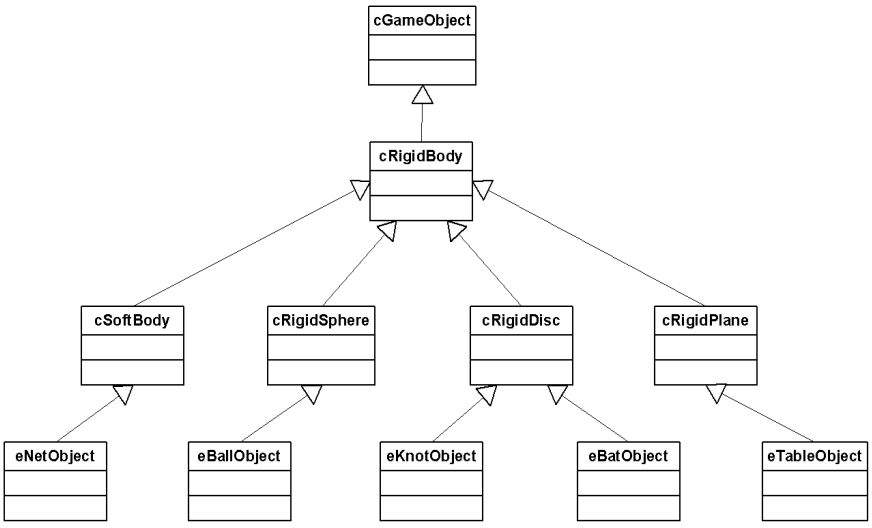
The backbone of the design is to separate the data representations from the graphical representations and by doing so, separate the game logic and simulation from the rendering. The data will have a pointer to the graphic but is only used for updating the position and orientation when the data is modified by the game or simulation.

There is a controller object that contains both a pointer to the data and the graphic and allows access to the graphic directly without having to go through the data class. The design was meant to be a derivative of the Model View Controller design pattern but the controller class was misinterpreted in design and by the time it was realised, it was too late in the development cycle to change the design.

Three managers were used for this project; one for the data representation physics objects, graphic representation and controllers. In hindsight, these three managers should have been made singletons but at present, they just have static variables and functions. All the managers are responsible for maintaining and destroying all the objects that are created at runtime. For example, the Controller manager is maintains a list of pointers to all the controller objects that are created in the application and frees the memory when the application closes.

While the controller manager is fairly generic, the data representations manager and graphics manager serve specific functions in addition. The physics objects manager applies real world physics on the objects in its list such as friction, momentum, forces and collisions. The graphics manager renders all objects in its list to the window.

All the base physic objects in the simulation inherit from a base class which contains all the base level information for the physics objects such as position, orientation, mass, forces etc. The inherited classes include a Rigid Plane, Rigid Disc, Rigid Sphere and Soft Body and the physics manager contains functions to solve collisions between these objects.



The physics objects can only be moved by applying forces to them so that the physics manager is the only class that can change the actual position of an object. Therefore all the input from user does is move apply a force to the object rather then changing the position.

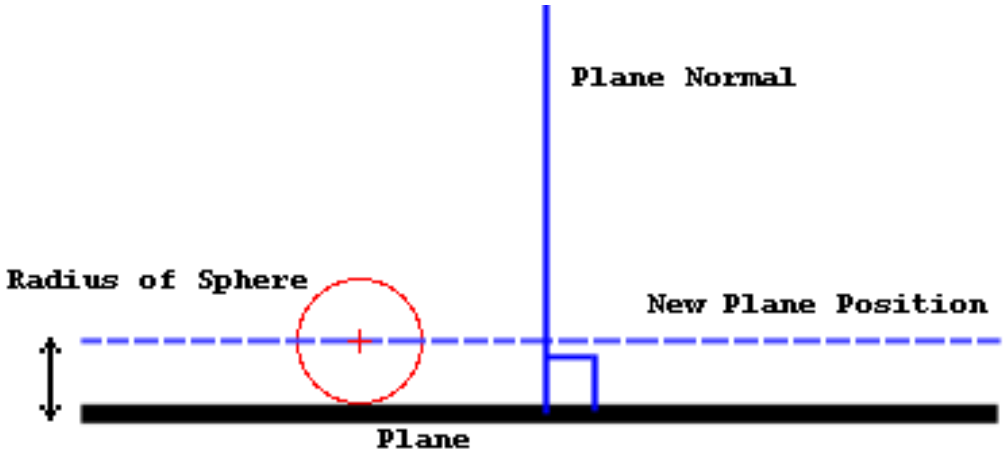
PBM Implementation

The backbone of the physics engine uses the 4th order Runge-Katta to calculate the velocity and new position of the object each time step. The formula was taken from the O'Reilly book of Physics for Game Developers. Euler integration could have been used but tends to become more and more inaccurate as the timestep becomes and larger and larger so was immediately ignored considering Runge-Katta integration was not much more difficult to implement.

The physics engines iterates through the list of objects, calling the update function of the object and then applying the Runge-Katta step simulation. After this loop is completed then the engine checks each object with each other object for collisions within this time step.

Only Plane-Sphere and Disc-Sphere collisions were implemented in the physics engine as the requirements specifically mentioned Ball-Net, Ball-Table and Ball-Bat collisions. Since the Knots of the Net were actually Rigid Discs and discs are basically circular planes, all the collision detection is derived from Ray-Plane intersection calculations.

To simplify the calculation from a Sphere-Plane to a Ray-Plane, the plane is moved along its normal towards the sphere by the radius of sphere for the calculation.

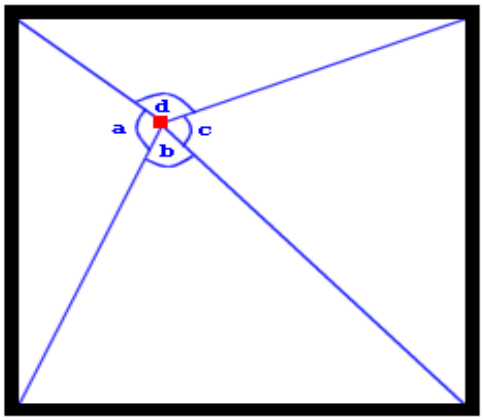


The distance between the sphere and an infinite plane is given by a formula from Nehe's Tutorial 30. Taking the plane position, plane normal, sphere position and sphere direction, the function returns the distance from the plane. If the distance is negative, the sphere is moving away from the plane.

Taking the position prior the physics simulation in the time step and the current position of the sphere, if the distance from the plane is positive from the old position and the distance from the plane is negative in its new position, then a collision has occurred in this time step.

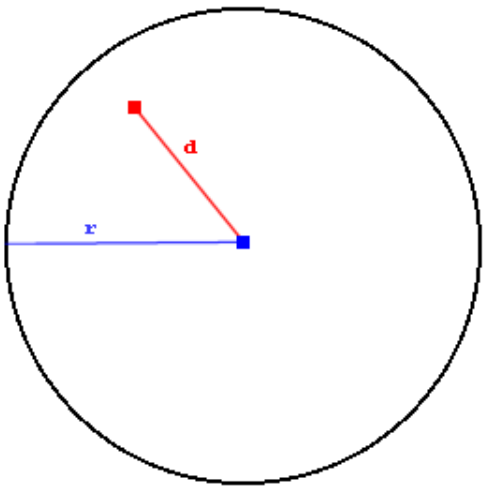
Once it is determined whether a collision has occurred, the time to collision is calculated (using an equation from an unreferenced source) and as long as the time to collision is between 0 and the time elapsed in this time step then a collision has occurred. Using the

sphere's velocity and time to collision, the contact point on the plane is calculated and used to check whether the sphere has collided on the finite plane. If the sum of angles from the contact point to all adjacent pairs of corners on the plane equals 360 degrees, then the collision has occurred inside the plane. If it does not, then no collision has occurred with the finite plane.



if (a + b + c + d) = 360
then a collision has occured
inside the plane

In the case of the disc, the distance between the contact point on the infinite plane and the position of the centre of the disc and the radius of the disc is compared, If the distance is smaller then the radius of the disc, then a collision has occurred, if not then no collision has occurred with the disc.



if (d <= r) then a collision
has occured inside the disc

So far, this collision detection is assuming that the plane/disc is static and the sphere is moving, to make the collision detection suitable for a moving plane/disc to a moving sphere the relative velocity of both objects is used. By taking relative velocity, one object is then considered static (in this situation, the plane/disc) and the other moving (the sphere). The relative velocity is then used to recalculate the current position of the sphere in this time step if it had this velocity and the check for Ray-Plane intersection and time to collision is calculated as normal. Once the time to collision has been calculated, using the

sphere's and plane's original velocity, the objects are moved to their position at the time of collision.

The conservation of momentum is then used to calculate the velocities of both objects after the collision. *mX* represents the mass of the object, *uX* the initial velocity and *vX* the velocity after the collision.

$$(m1 * u1) + (m2 * u2) = (m1 * v1) + (m2 * v2)$$

$$v1 = ((m1 - m2) * u1 + (m2 + m2) * u2) / (m1 + m2)$$

$$v2 = ((m1 + m1) * u1 + (m2 - m1) * u2) / (m1 + m2)$$

This is the standard equation for the conservation of momentum in a 1D collision. For 2D or 3D collisions, the velocity along the line of impact is used instead which forms the following equation with *L* as the line of impact.

$$(m1 * (u1.L) * L) + (m2 * (u2.L) * L) = (m1 * (v1.L) * L) + (m2 * (v2.L) * L)$$

$$(v1.L) * L = ((m1 - m2) * (u1.L) * L + (m2 + m2) * (u2.L) * L) / (m1 + m2)$$

$$(v2.L) * L = ((m1 + m1) * (u1.L) * L + (m2 - m1) * (u2.L) * L) / (m1 + m2)$$

$$v1 = u1 - (u1.L) * L + (v1.L) * L$$

$$v2 = u2 - (u2.L) * L + (v2.L) * L$$

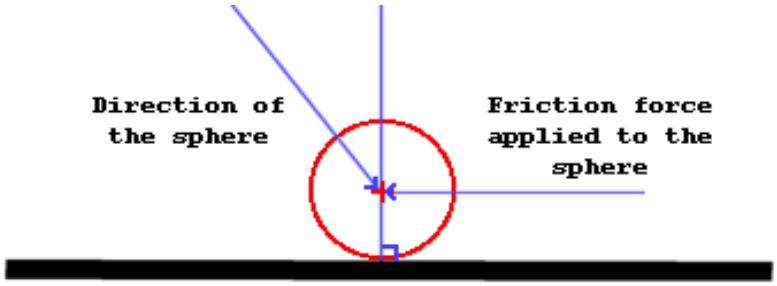
To take elasticity between two materials into consideration, the following lines are changed with *e* representing the elasticity:

$$(v1.L) * L = ((m1 - em2) * (u1.L) * L + (m2 + em2) * (u2.L) * L) / (m1 + m2)$$

$$(v2.L) * L = ((m1 + em1) * (u1.L) * L + (m2 - em1) * (u2.L) * L) / (m1 + m2)$$

Equations taken from Derek Wills Lecture notes

With the velocities after the collision calculated, frictional force is applied to the sphere at the point of contact where the force is along the plane in the opposite direction to the direction of travel along the plane. Ideally rolling friction should be used but for simplicity static friction was applied.



The following equation calculates the opposite vector that the sphere is travelling along the plane with *N* representing the plane normal, *SphereD* as the sphere direction and *FrictionD* as the frictional force direction:

$$FrictionD = (SphereD \text{ [cross] } N) \text{ [cross] } N$$

The friction force is calculated with the static friction equation with *μS* representing the

coefficient of static friction, *N* as the plane normal, *SphereF* as the forces on the sphere, *FrictionD* as the frictional force direction and *FrictionF* as the friction force:

$$FrictionF = (SphereF.N) * N * FrictionD * \mu S$$

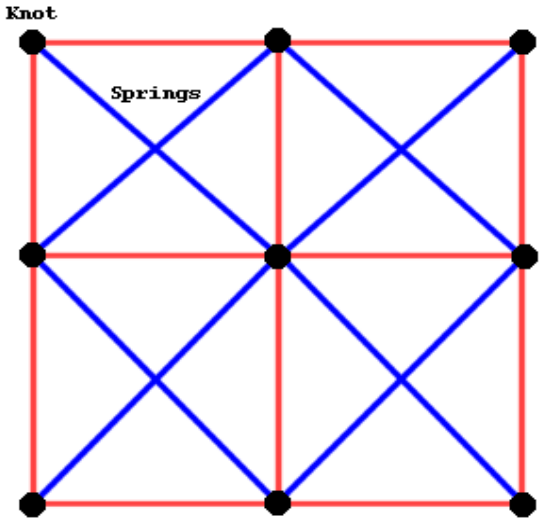
The frictional force is then applied to the sphere in addition to its current forces and Runge-Katta step simulation is used with a time step of 0.001 seconds to simulate the contact time between the two objects. The frictional force then removed from the sphere and the step simulation is used for the reminder of the time step to calculate the sphere's and plane's velocity and position at the end of the time step. At this stage, the engine should check for any new collisions between the point of contact and the objects new position at the end of the time step. This was not implemented due to lack of time and added complexity.

This is admittedly a cheat of sorts of applying friction to an object but works well in implementation although it is not completely correct.

A spring and dashpot model was used for the Net. The net maintains a list of springs that has two knots (point masses) attached to it. The constrains and constants such as the mass of each know, rest length etc. are defined in a configuration file. As each Knot object is a Rigid Disc, the collision detection and collision response are already in place in the physics without any extra code, all that is needed is to pass the Knot objects to the physics manager when they are created by the net and then already included in the collision model.

The Springs are a special case in the simulation as they do not belong to the physics manager but to the Net object and the forces of the Spring are calculated when the Net is updated in the physics object and are then applied to the Knots that are attached to the Spring. This enforces the simulation to apply all the forces in the Springs on the all the Knots in the Net prior to the physics engine running the step simulation on each Knot otherwise the Net would gain energy if the Springs where updated at the same time as the step simulation was applied to the Knot.

The Springs are attached to the Knots in the following pattern:



Only structural and shear springs are used in the Soft Body Net as flexion springs caused the net to become too rigid. The force in the spring that is applied to the masses is calculated in the following equation. Let *F* be force in the spring, *L* the distance between

the two attached knots, *LR* the rest length of the spring, *v1* the velocity of knot1, *v2* the velocity of knot2, *kS* the spring constant and *kD* the damping coefficient.

$$F = (kS * (L - LR) - kD * ((v1 - v2).L / |L|)) * L / |L|$$

Equations taken from Derek Wills Lecture notes

The main problems with the spring and dashpot model is that it is very time dependent. If the time step is too large, the model can become unstable and break. The possible workarounds is to have the soft body extremely stiff and rigid, never let the timestep become higher then a certain value or have a fixed timestep.

Distributed Architectures and Threads

Not taking into account of the Network component, there were four threads used excluding the main thread in the application. One each for the AI controllers on the Bats, the Physics world simulation and the Net simulation. The main thread runs at a constant frame rate of 30fps and handles rendering the representations in the Graphics manager and handles input from the user since both of these tasks are directly reliant on the GXbase OpenGL framework.

The net simulation is separate from the physics simulation purely for performance issues as calculating the forces in all the springs was extremely CPU intensive depending on the number of springs in the net. When it was part of the physics thread, it slowed the frequency of the thread to less then half.

The main shared memory between the threads are the data representations for the Bats, Ball, Net and Table where read and write access are locked by mutexes. The use of mutexs insures that a thread does attempt to write to an area of memory that is currently being written to/read from by another thread.

The threads are implemented through use of a class taken from Warren's Viant tutorial 3a and extended to support suspending and resuming a thread processes, changing the priority of the thread with completeness, a method to immediately terminate a thread if need be although it is not used in this project.

Any class could be given the attributes and functionalities of a thread through inheritance and be given a customisable loop to process until the the thread is terminated. It is recommended that a Sleep time for each iteration of the loop to be at least one millisecond otherwise the thread will maximise the amount of CPU usage that it has access to.

The use of mutexes were also implemented in a similar way where inheriting the mutex class will also give the class the functionality to lock the data that it uses. Whenever a thread attempts to read or write to a GameObject, it will attempt to lock the object and wait till another thread releases the lock.

To minimise the risk of deadlocks, a thread will create copies of the data from the shared memory that it needs, process the data and recommit the results back to shared memory. The locking of the shared memory only occurs during the copying and committing which allows other threads access during the processing of data.

The Networking component made heavy use of Warren Viant's classes for basic networking so all that needed to be developed and implemented was the client class to manage incoming and outgoing packets and a server to manage all the connections.

To simplify the networking, only one packet was used for sending all the data that was always going to be the same size and same format. Due to time issues, the packet was a C structure and was always sent as a struct. The sending of data should have been marshalled to insure that the client that is receiving the data would be able to read the data correctly even if it was on a different platform or architecture.

When a client connects to the server, the server expects a packet to be sent from the client indicating whether it would want to connect as a player or as a spectator. The server checks its connection lists for any available slots and sends a packet back to the client indicating whether the slots are filled, that the client is Player A or B (if the client connects as a player) or that it is now a spectator. In the case that there are no slots, the socket stream is closed and the server disconnects the client. If there are slots, then the socket stream is added to the list.

Both the server and client are given pointers to the three objects that they affect which are the two Bats and the Ball. Data that is sent and received contain the position, orientation and velocity data of these three objects. Although each client has predictive physics, the game action is solely dependent on the server and retains the absolute data for each of the objects.

The Network Client component operates in its own thread once the connection has been established. The Receive and Send functions run in the same thread so once the client has sent data, it expects some data to be sent back. This 'to and fro' system insures that both incoming and outgoing data stays in sync with the server without flooding either of the buffers which occurred when the Send and Receive functions where placed in separate threads.

The Network Server component operates in its own thread but needs to manage many other threads that runs in each client connection. To minimise latency and removing any server locks, each client that connects a new thread is created to receive and send data down the socket stream so that data is being processed and sent almost simultaneous. If this process was all in one thread (iterating through the list of clients send and receiving data) there would be noticeable latency between each client. Also if a client disconnects and the server becomes 'stuck' in a the Receive function call (usually from pulling the cable out) the server cannot iterate through the reminder of clients till the function call times out.

Ideally, the best method for this process is to poll the sockets using Select so that the server only calls Receive if there is data waiting to be read from the socket stream. Using Select also allows the entire process to run in one thread so the overhead of managing multiple connections each with their own thread is negated. However, the project was an exercise in using threads so the use of Select was not permitted.

Testing Harness

The PBM was tested in constant situations such as a ball falling vertically onto a perfectly flat surface and with information such as collision points, the velocity before the collision and after. Any visible anomalies have the equations involved checked by hand. An example would have been a situation that the normal rotation was calculated via matrix mathematics but the normals were being drawn incorrectly on the render window. The reason for the error was due to the input values being modified inside one matrix

mathematical operation (e.g. The angle rotated along the Y axis) and being reused within the same matrix for another operation. Also degree were being used instead of radians.

The locking of shared data via mutexes was tested by removing any releases in one thread, the result should cause all the other threads to deadlock when they try to access the same shared data. If this does not happen, then the thread is reading or writing to the shared memory without attempting to lock the data. A simple test but works effectively to ensure data is locked properly.

To stress the network, multiple clients on separate computers were used to continuously disconnect and reconnect to the server to see if the server could maintain the strain. Also various methods of disconnection were used such as physically disconnecting the network cable from the back of computers, ending the process in task manager, etc. The server was also shutdown when all the slots are filled to see how the clients would handle an abrupt server disconnection.

Checking to see if the packets were being sent correctly was considered the most difficult as none of the packets were being marshalled so any packet sniffer programs were useless. So to compensate, 10 predefined packets were sent from the client to the server and with every packet that was received by the server, the details of the packet were output to the console and cross checked with the details with the predefined packets.