

OBJECT-ORIENTED SIMULATION OF POOL BALL MOTION

Arthur Paul Goldberg

**March 1987
CSD-870016**



UNIVERSITY OF CALIFORNIA

Los Angeles

Object-Oriented Simulation of Pool Ball Motion

**A thesis submitted in partial satisfaction of the
requirement for the degree of Master of Science
in Computer Science**

by


Arthur Paul Goldberg

1984

© Copyright by
Arthur Paul Goldberg

1984

The thesis of Arthur Paul Goldberg is approved.

 18 Oct '84

David Jefferson

 1 Nov. '84

Mario Gerla



D. Stott Parker, Committee Chair

University of California, Los Angeles

1984

**I dedicate this work to my family
and my friends.**

TABLE OF CONTENTS

1. Introduction	1
2. Simulation of Pool Ball Motion	3
3. Physics of Pool Ball Simulation	18
4. Computational Details of Pool Simulation	28
5. Sectored Simulation	45
5. Experimental Results	59
6. A Model of Message Counts in a Sectored Pool Simulation	66
7. Conclusions	72
8. References	77

LIST OF FIGURES

Figure 1. Time Step Simulation Program	4
Figure 2. Event Based Simulation Program	6
Figure 3. Object-Oriented Simulation Program	9
Figure 4a. Ball Broadcasts NewVelocity Message	13
Figure 4b. Objects Schedule Collisions with Ball that Changed Velocity	13
Figure 5. Impact Parameters of Ball-Cushion Collision	20
Figure 6. Ball Which Needs a Corner to Hit	22
Figure 7. Ball Colliding with a Corner	22
Figure 8. Two Balls Colliding	26
Figure 9. Simultaneous Events At Time t for Ball-1	30
Figure 10. Three Balls Colliding Simultaneously	33
Figure 11a. Initial Velocities of Collision Involving Three Balls	36
Figure 11b. Final Velocities of Collision Involving Three Balls	36
Figure 12. Ball About to Strike Another Ball Resting on a Cushion	38
Figure 13. Code for Reversing a Ball's Direction	43
Figure 14. Simulation Program with Sectorred Pool Table	50
Figure 15. Message Flow Graph for Sectorred Pool Simulation	51
Figure 16. Relations between two real roots and the simulation time	55
Figure 17. Messages as a Function of Sector Size - No Collisions	69
Figure 18. Messages as a Function of Sector Size - With Collisions	71

LIST OF TABLES

Table 1. Ball Velocities in Collision Processed Serially	34
Table 2. Simultaneous Collision with 'Lost Update'	34
Table 3. Two Collisions Seen as Two Transactions	35
Table 4. Messages Sent in an Instantaneous Collision	39
Table 5. Message Count of Pool Ball Simulations	60
Table 6. Elapsed Time of Pool Ball Simulations	60
Table 7. Mean and Standard Deviation of Message Counts and Elapsed Time	61
Table 8. Number of Messages Tabulated by Message Type	63
Table 9. Spreadsheet for Estimating Number of NewVelocity Mes- sages	65
Table 10. Message Count of Pool Ball Simulations	68

ACKNOWLEDGEMENTS

I thank my advisor, Prof. D. Stott Parker, for his time, encouragement, enthusiasm and intelligence. I thank Dr. Henry Sowizral, the co-inventor of Time Warp, for proposing the research contained in this thesis and for participating in its development. I thank Dr. Sowizral, the Rand Corporation, of Santa Monica Calif., and the System Development Foundation, grant #58, for providing financial support and facilities to conduct this research. I thank the thesis committee members, Prof. Mario Gerla and Prof. David Jefferson. Lastly, I thank my friends Chet Lanctot and Joe Green, who have supported me through this work and will undoubtedly acknowledge me in their theses.

ABSTRACT OF THE THESIS

Object-Oriented Simulation of Pool Ball Motion

by

Arthur Paul Goldberg

Master of Science in Computer Science

University of California, Los Angeles, 1984

Professor D. Stott Parker, Chair

This thesis presents a simulation model of pool ball motion. The model represents the positions of pool balls as they move around a pool table, colliding with each other, colliding with the sides of the table, and entering pockets.

The primary goal of our study has been to explore methods to *speed up* the pool ball simulation. The simulation techniques we use work towards this goal in two directions.

The main computational expense of a pool ball simulation is scheduling future ball collisions. Our first technique for increasing the speed of a simulation decreases the number of scheduled collisions by dividing the pool table area into *sectors*. We have implemented two pool ball simulation programs - in one the pool table area is continuous, while in the other the pool table area is divided into sectors. Experiments show that as the number of balls on the table increase the simulation runs considerably faster with a sectorized pool table than with a pool table not divided into sectors.

Second, we have implemented the simulation to run on the *Time Warp* [Jefferson and Sowizral 82] distributed simulation system. This is the first large program to use Time Warp. It promises to be an important tool for studying the acceleration of simulation by distributed computing.

1. Introduction

This thesis studies the simulation of pool ball motion. Simulation programs have been implemented in the object-oriented framework, which is well suited for modeling discrete event systems. Our primary goal has been to *speed up* the pool ball simulation. When we speak of speeding up a simulation, we are concerned with reducing the elapsed real time of the simulation, which becomes more important as the number of balls increases. We have worked towards this goal on two fronts. First, we have demonstrated that the computational complexity of the simulation may be decreased by dividing the pool table into *sectors*. Second, we have implemented the simulation on the *Time Warp* distributed simulation system developed by Jefferson and Sowizral [Jefferson 82] so that it may exploit the acceleration of concurrent distributed simulation.

The accomplishments of this work are

- (1) We have implemented an object-oriented simulation of pool ball motion. All essential objects on a pool table are represented: balls, cushions, pockets and corners.
- (2) Experimental results show that a significant decrease in the real elapsed time of a simulation can be achieved by dividing the pool table into *sectors*. A sector is modeled without difficulty as a logical simulation object. In addition, we present an analytic model that predicts an upper bound for the optimal sector size.
- (3) We have discovered that *instantaneous events* (such as a ball colliding into a ball resting against a cushion) occur in a pool ball simulation and are difficult to simulate correctly. We present a simple method for correctly simulating instantaneous events.

- (4) Our pool ball simulation is the first large model implemented on Time Warp at Rand. As such, the simulation will be an excellent research tool for exploring the performance potential of concurrent simulation.

This thesis is divided into several major sections. The next section presents several different approaches to the simulation of pool ball motion, culminating with a description of our object-oriented simulation. The third section rigorously presents the physics of collisions between pool balls. The fourth section discusses the computational details of pool simulation, concentrating on modeling methods for *simultaneous events*, which occur when an object participates in more than one event at the same simulation time. The fifth section presents the design and implementation of *sectored simulation*, which is followed by a section that presents experimental results which show that a sectored simulation executes *faster* than a non-sectored simulation. The next two sections present an analytic model of sectored simulation. We conclude the thesis with a discussion of our ideas for future work.

We assume the reader is familiar with the broad outlines of discrete event simulation. General discussion of simulation methodology and languages can be found in [Fishman 78] and [Franta 77].

2. Simulation of Pool Ball Motion

We all know how pool balls move around a pool table; they bounce off each other and the cushions. Occasionally a ball enters a pocket, although it is usually not the ball and pocket that we had intended.

This thesis describes algorithms that simulate the motion of pool balls on a pool table. The algorithms capture the essentials of pool ball collisions. When a ball runs into a cushion it bounces back, and the direction it moves depends on the angle with which it hit the cushion. When two balls collide the laws of physics are obeyed. The collision conserves momentum and, since we assume it is elastic, kinetic energy. Some of the more subtle physics of pool, such as English and friction between the ball and the table felt, are not represented in these algorithms, but there is no fundamental reason why they could not be included.

There are two major temporal simulation methods, *time step* simulation and *discrete event* simulation. In time step simulation, time advances in a fixed increment, and the state of the entire simulation is updated at each step. In discrete event simulation events in the simulation are scheduled in advance and processed according to the rule 'earliest event first'. In this section we briefly outline pool ball simulation algorithms which use the two major methods. We discuss the merits of each method for simulating pool ball motion.

In these algorithms we limit our concern to representing the physical position of the balls as a function of time. We assume that a moving ball will continue in a straight line at the same speed until it hits another ball, cushion or a pocket. We are interested in events that change a ball's velocity.

2.1. Time Step Simulation of Pool Balls

Let us begin by looking at the time step simulation in figure 1. The essential data in this algorithm are the position and velocity of the balls. The 'while loop' runs until simulation time reaches the end of the simulation. Each pass through the loop increments a pool ball's position one time step. Then the new positions are checked to see if a ball participates in a collision. If two balls collide then they are each assigned a new velocity.

A time step simulation is poorly suited for simulating the discrete motion of pool balls. We support this claim by examining the computational complexity of this simulation. How many processing steps does it take to run this simulation? Standard theory of computation expresses the complexity of an algorithm as a function of the size of the problem. In this case the size of the problem is n , where n is the number of balls. The complexity of a single step of the time step simulation is $O(n^2)$, since all possible pairs of balls are examined to determine if they collide. Expressing the computational complexity of a simulation introduces another parameter, the length of the simulation.

The complexity of an entire time step simulation depends on the size of the time step. If we let T be the length of the simulation and let Δt be the length of the time step we can state that the complexity of the entire simulation is $O(n^2 T / \Delta t)$.

There are two problems with using time step simulation to simulate the discrete motion of pool balls. First, the accuracy of the simulation is compromised because a collision cannot be simulated exactly at the time it occurs. At each time step the algorithm determines if a pair of balls collides by deciding whether they overlap. There is a delay between the instant of the collision and when the balls receive their new velocity, which can be

Time step method

Data:

size of the pool table
set of pool balls
state for each ball: position, velocity

Program:

```
simulate_pool_ball_collisions
initialize ball states
time ← 0
while( time < end AND computing funds remain)
do
time ← time + time step
update the positions of all balls
for each ball on the table
if (the ball has entered a pocket)
then
put it in the pocket
else
if (the ball has collided with a cushion)
then
/* bounce it off the cushion */
send it off in a new direction with a new state
endif
endif
end
for each pair of balls on the table
if (the pair has collided)
then
send them off in their new directions,
with all their new data
endif
end
end
endprogram
```

Figure 1. Time Step Simulation Program

almost as long as the time step Δt .

A second, related problem is that there is a tradeoff between computational complexity and the simulation's accuracy. The smaller the time step the greater the complexity. However, a larger time step makes the simulation less accurate. A significant drawback of time step simulation of pool ball collisions is that there is no simple way to choose the best time step.

2.2. Event-Based Simulation of Pool Ball Motion

Event based pool simulation does not have the drawback of time-step simulation just mentioned, the difficulty of choosing a time step.

The algorithm for an event based simulation is listed in figure 2. The basic idea behind this algorithm is that all possible future collisions are scheduled. When a ball undergoes a collision and changes its velocity, it cancels all other collisions scheduled for it, and schedules a new set of collisions, based on its new trajectory. By trajectory we mean the ball's speed and direction (velocity as a vector) and its last position fix. A trajectory determines a ball's position as a new trajectory are scheduled *before* any other collisions are processed. See section 2.4 for an informal proof that this simulation algorithm is correct.

The data for each ball's state is the same as in the time step algorithm, except that a ball's state also includes the time of the last position fix. The event list is a list of future events. Each entry in the list contains the time and a description of the event. The 'while-loop' runs until the simulation time exceeds the ending time of the simulation. Each pass through the loop processes the next event in the event list. If a collision event is found then the new velocities of the balls involved are calculated.

What is the computational complexity of this event based simulation? Processing a single collision may involve canceling and scheduling $O(n)$ colli-

Data:

Size of the pool table, placement of pockets, etc.

For each pool ball:

$s(t)$ position at time t

v velocity

Event list - a list of pairs (time, event) sorted by time.

Program:

simulate_pool_ball_collisions

initialize ball data and event list

while (balls are still moving AND time < ending time of simulation) **do**

remove the next event from the event list

time ← the time of the next event

/ now process the event */*

case(event type)

ball entering pocket:

take the ball out of play

collision:

/ If a ball is hitting a cushion then there is one ball that changes velocity, if there are two balls colliding then there are two balls that change velocity. */*

for each ball B that changes velocity **do**

change ball B's direction

cancel all future events scheduled for ball B

by removing them from the event list

for all other balls X on the table **do**

if (ball B and ball X will collide)

then

schedule a collision between ball B and ball X

endif

end

for all cushions C **do**

if (ball B will hit cushion C)

then

schedule a collision between ball B and cushion C

endif

end

for all pockets P **do**

if (ball B will enter pocket P)

then

```

        schedule a collision between ball B and pocket P
    endif
end
end
endcase
end
endprogram

```

Figure 2. Event Based Simulation Program

sions, because the ball may schedule a collision with every other ball on the table. Now consider the expense of maintaining the sorted event list. There are at most $O(n^2)$ entries in the event list. If the list is kept as a heap then n deletions and n insertions will cost $n \log n^2 + n \log n^2 = O(n \log n)$. (We ignore the cushions in the complexity calculation since we assume that their number is fixed and small). Letting c be the number of collisions in the entire simulation we see that its complexity is simply $O(cn \log n)$.

We cannot directly compare the computational complexity of the two simulation methods because their complexities are not expressed with the same parameters. However we can say that the event based simulation has the advantage that a ball's position is recalculated only when the ball changes velocity rather than at every time step. The average number of collisions per time step is $c\Delta t/T$. The complexities of the two simulations are equal if the number of collisions per timestep is $n/\log n$. If there are fewer than $n/\log n$ collisions per time step then the event based simulation is less complex. On the other hand if there are more than $n/\log n$ collisions per time step then the time step is probably too coarse. Therefore an event based simulation will most likely be more computationally efficient than a time step simulation.

2.3. Object-Oriented Pool Ball Simulation

An object-oriented simulation is a specialized kind of event simulation in which the simulation program is organized into modular software elements called *objects*. Objects communicate data with each other and schedule events only via simulation *messages*. Object-oriented simulation languages are exemplified by ROSS [McArthur 82] and Time Warp [Jefferson 82]. These simulation languages are the intellectual descendants of the object-oriented programming languages Smalltalk [Goldberg and Kay 76] and Director [Kahn 79].

In this thesis we concentrate on object-oriented pool simulations because this is the simulation methodology of Time Warp. Time Warp chooses an object-oriented simulation method because it is well suited for distributed simulation. The crucial advantage of the object-oriented approach for distributed simulation is that the simulation code which defines how an object processes messages, called the object's *behaviors*, does not access global data structures. A behavior accesses only an object's local data and messages sent to the object. An object-oriented simulation program can therefore be transparently run on a distributed simulation system (which must, of course, provide concurrency control underneath). We will not discuss distributed simulation any further - the interested reader should read Jefferson and Sowizral's Time Warp Rand note [Jefferson 82].

The procedural logic of the object-oriented pool simulation we present next is no different from the logic of the event based pool simulation. The essential difference is in implementation - here the communication is performed via messages, whereas the event simulation program modifies the event list directly. An object-oriented pool ball simulation program is presented in figure 3.

Object-oriented pool ball simulation

Object States:

State name	Properties
BallState	Trajectory - position fix and velocity Pocket the ball occupies Set of planned interactions
CushionState	Position - the cushion's endpoints Set of planned interactions
PocketState	Position - the pocket's endpoints Set of balls in the pocket Set of planned interactions
CornerState	Position Set of planned interactions

Object behavior algorithms:

BallBehavior(messages, BallState)

Eliminate pairs of messages that cancel each other

CollisionHasOccured ← False

for all messages do

case(message_type)

NewVelocity:

identify sender of NewVelocity message as ball X

if (this ball is not in a pocket)

then

cancel all planned collisions with ball X

if (the time of a collision with ball X < the current time)

then

schedule a collision with ball X

add the collision with ball X to the set of planned interactions

endif

endif

Collision:

identify sender of Collision message as object Z

```

CollisionHasOccured ← True
case (object type of object Z)
  Ball:
    BallState:velocity ← BallCollision(BallState:trajectory,
      ball Z's trajectory)
  Cushion:
    BallState:velocity ← CushionCollision(BallState:trajectory,
      cushion Z's position)
  Corner:
    BallState:velocity ← CornerCollision(BallState:trajectory,
      corner Z's position)
  Pocket:
    BallState:pocket ← pocket Z
endcase
BallState:trajectory:position ← current position
BallState:trajectory:fix_time ← current time
remove the collision from the set of planned interactions
endcase
end
if (CollisionHasOccured)
then
  cancel all planned interactions
  empty the set of planned interactions
  for all other balls do
    send a NewVelocity message to the ball
  for all cushions do
    send a NewVelocity message to the cushion
  for all pockets do
    send a NewVelocity message to the pocket
  for all corners do
    send a NewVelocity message to the corner
  endif
endprogram

```

```

CushionBehavior( messages, CushionState)
Eliminate pairs of messages that cancel each other
for all messages do
  case (message_type)
  NewVelocity:
    identify sender of NewVelocity message as ball X
    cancel any planned collision with ball X

```

```

if (the time of a collision with ball X > the current time)
then
    schedule a collision with ball X
    add the collision to the set of planned interactions
endif
Collision:
    identify sender of Collision message as ball X
    remove ball X from the set of planned interactions
endcase
end
endprogram

```

CornerBehavior - same as CushionBehavior except uses a different algorithm to calculate collision time.

PocketBehavior - same as CushionBehavior except add to collision case:
add ball X to the list of balls in the pocket

Figure 3. Object-Oriented Simulation Program

There are four types of objects: balls, cushions, corners and pockets. The top of the figure tabulates the state information associated with each object. A ball state stores the ball's velocity and a position fix; the time and location when the ball last changed velocity. Cushion and pocket states store their object's position as the location of its two endpoints. A pocket state also stores the identities of all the balls the pocket has collected. Finally, each object stores a list of the collisions that it planned, so that if necessary it can cancel a collision later.

There is a behavior procedure for each of the four object types: BallBehavior, CushionBehavior, CornerBehavior and PocketBehavior. A set of messages sent to an object at a particular simulation time is processed by running the behavior procedure of the object's type with the set of messages and the object's state as arguments.

There are strong parallels between the different behavior procedures. In fact, CornerBehavior and PocketBehavior are not written out fully because they are so similar to CushionBehavior. The differences between these two procedures and CushionBehavior are noted at the bottom of figure 3.

In effect, the logic of this object-oriented simulation is the same as that of the previous event based simulation, except that communication between objects is performed via messages. There are three types of messages: NewVelocity, Collision and Cancel. When a pool ball changes its velocity it sends a NewVelocity message to all other objects. The NewVelocity message contains the pool ball's new velocity. An object that receives a NewVelocity message examines the trajectory of the ball that sent the message to determine whether the ball will collide with the object. If they will collide then the object sends two Collision messages to schedule the collision. A Cancel message is sent to cancel a scheduled collision. Cancellation is needed when a ball involved in the collision changes its velocity prior to the collision.

An aid to understanding the simulation program is the message flow graph, shown in figure 4. The nodes of these directed graphs are objects and the arcs represent the transmission of a message from one object to a set of other objects. The arcs are labeled with message types.

Figure 4a indicates that a ball with a new trajectory (after a collision) sends a NewVelocity message to each other simulation object. An object is another ball, a cushion, a pocket or a corner. An object which receives a NewVelocity message from ball X determines whether it will collide with ball X. If the object decides that a collision will occur then it schedules a collision by sending a Collision message to ball X and a Collision message to itself. These two messages are illustrated in figure 4b by the arc and self-arc, each labeled by the single word Collision. Both messages are necessary

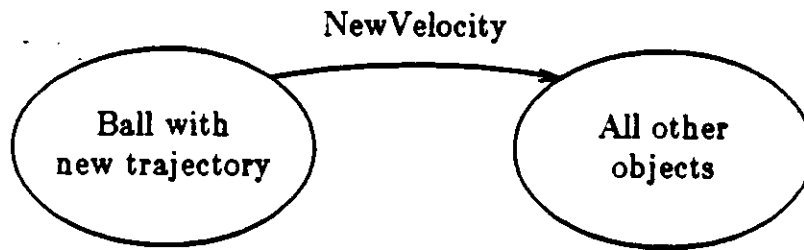


Figure 4a. Ball Broadcasts NewVelocity Message

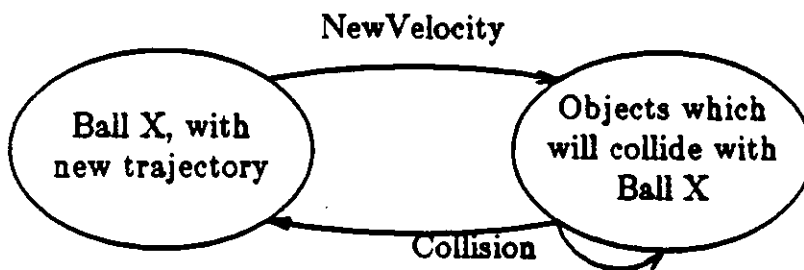


Figure 4b. Objects Schedule Collisions with Ball that Changed Velocity

because each ball independently processes a Collision message to determine its velocity after the collision. The information needed to determine a ball's velocity after a collision are the trajectories of both balls in the collision. The two Collision messages are not identical - each ball receives a Collision message containing the trajectory of the other ball. Therefore, two Collision messages are sent per collision.

We introduce message flow graphs here because they help design object-oriented simulations. This is clearly not the case with these simple graphs, but they do help design simulations which have more types of objects and messages, as we shall see later.

The procedures in the object-oriented simulation which schedule collisions and the procedures which compute a ball's velocity after a collision (BallCollision, CushionCollision and CornerCollision) are presented in the

chapter on physics of pool ball simulation.

The code has been implemented and works.

2.4. Correctness Proof

In this section we present an informal proof that our simulation algorithm correctly models pool ball motion. Assume that the physical collisions are correctly modeled. Then we can be sure that the simulation is correct if it executes the same collisions that physical pool balls would undergo. The proof takes two steps. First, we show that a superset of all possible collisions is scheduled. Second, we show that all collisions that do not physically happen are canceled *before* they can occur.

The proof requires several assumptions. First, assume that an object receives no other messages at the same simulation time that it receives a Collision message. This assumption insures that the simulation time order of events dictates their execution order. Thus, a collision is canceled 'before' it can occur if it is canceled at an earlier simulation time. This assumption implies that the pool balls collide only two at a time. Second, assume that a NewVelocity message is received and processed at the same simulation time that it is sent. (The simulation implementation relaxes these assumptions, and has to make adjustments to compensate.)

Whenever a ball changes velocity it notifies all other objects about its new trajectory by broadcasting NewVelocity messages. An object that receives a NewVelocity message determines whether its trajectory will intersect the trajectory of the ball that sent the NewVelocity message. An intersection is a potential collision. Any object that detects an intersection schedules a collision with the ball that sent the NewVelocity message.

Let the object that schedules the collision be called the 'scheduling object'. The scheduling object sends a Collision message to itself and a Collision message to the ball. The Collision messages are scheduled for receipt at the time of the collision. When (and if) they are processed, each object involved in the collision will change its state appropriately, and the collision will have been executed. This completes the first step of the proof.

When the scheduling object sends the Collision messages it also makes an entry in its list of planned interactions, so that it can cancel the collision. The collision will happen if it is not canceled before the collision time. It must be canceled if some object involved in the collision changes velocity before the collision.

The second step of the proof has two cases, one for each object in the collision. If the scheduling object changes velocity - in which case it must be a ball - then it cancels the collisions which it had scheduled.¹ If a ball involved in a collision that is not the scheduling object changes velocity then the scheduling object receives a NewVelocity message from the ball. The scheduling object will find the ball's name in its list of planned interactions and cancel the interaction by sending collision cancellation (Cancel) messages to itself and the ball. Because the NewVelocity message is transmitted in zero elapsed simulation time (assumed above) the cancellation will happen before the collision is executed. This completes the second step of the proof. We have shown that if either of the two objects involved in a collision changes velocity before the scheduled collision then the collision will be canceled.

¹Note that the ball must be careful to remove the collision that just happened before it cancels all planned interactions.

All that remains is to show that canceled collisions do not happen. At the start of all object behavior algorithms any pair of Collision and Cancel messages that cancel each other are removed from the list of input messages. The pairing is based upon the unique message identifier contained in each Collision and Cancel message. Thus, a canceled collision does not happen. The proof is complete. We have shown that the collisions that occur in the simulation are exactly those that happen on a real pool table.

3. Physics of Pool Ball Simulation

3.1. Scheduling Collisions

When a ball changes its velocity it may head on a collision course. The simulation schedules collisions with all objects the ball will hit. There are four kinds of objects that the ball can collide with - another ball, a cushion, a pocket, or the corner of a pocket. This section presents the geometric calculations that predict when, if ever, a ball will collide with another object. The arguments involve simple geometry and algebra.

The trajectory of the ball is used extensively in these calculations. The position of the ball, $\vec{s}(t)$, is given by the vector equation

$$\vec{s}(t) = \vec{s}(t_0) + (t-t_0)\vec{v}$$

The time of the last position fix is t_0 , and the position at that time is $\vec{s}(t_0)$. The ball's velocity is \vec{v} . If we make the substitution $\vec{D} = \vec{s}(t_0) - t_0\vec{v}$ then we have a simpler expression for the trajectory, which we use extensively:

$$\vec{s}(t) = \vec{D} + t\vec{v}$$

Each kind of object is discussed separately. We begin with the scheduling of a collision between two balls.

3.1.1. Ball Collision Time

Given the trajectories of two pool balls, we are concerned with resolving the question: will they collide, and, if they collide, when will the collision happen? Let the two balls be labeled 1 and 2, and let the parameters be identified by subscripts. Thus,

$$\vec{s}_1(t) = \vec{D}_1 + t_1\vec{v}_1$$

and

$$\vec{s}_2(t) = \vec{D}_2 + t_2\vec{v}_2$$

The distance separating the balls satisfies

$$(d(t))^2 = |\vec{s}_1(t) - \vec{s}_2(t)|^2$$

where $|\vec{x}|$ denotes the magnitude of the vector \vec{x} . Let the balls' radii be r_1 and r_2 . The condition for a collision is that the balls touch, or

$$d(t) = r_1 + r_2$$

We want to solve for the time of the collision. (If no collision occurs then the solution will not be real.) We need to solve

$$(r_1 + r_2)^2 = |\vec{s}_1(t) - \vec{s}_2(t)|^2$$

If we expand the right hand side into the expressions for vector components we get

$(r_1 + r_2)^2 = ((D_{1x} + tv_{1x}) - (D_{2x} + tv_{2x}))^2 + ((D_{1y} + tv_{1y}) - (D_{2y} + tv_{2y}))^2$
 where s_{1x} is the x-coordinate of ball 1 at time t_1 , and so on for the other subscripts. Combining the terms which multiply t , we get

$$(r_1 + r_2)^2 = (t(v_{1x} - v_{2x}) + D_{1x} - D_{2x})^2 + (t(v_{1y} - v_{2y}) + D_{1y} - D_{2y})^2$$

Expanding the right hand side and collecting terms yields a quadratic in t ,

$$\begin{aligned} & ((v_{1x} - v_{2x})^2 + (v_{1y} - v_{2y})^2)t^2 + \\ & 2((v_{1x} - v_{2x})(D_{1x} - D_{2x}) + (v_{1y} - v_{2y})(D_{1y} - D_{2y}))t + \\ & (D_{1x} - D_{2x})^2 + (D_{1y} - D_{2y})^2 - (r_1 + r_2)^2 = 0 \end{aligned}$$

This expression can be simplified considerably by returning to vector notation. Let $\Delta\vec{v} = \vec{v}_1 - \vec{v}_2$ and $\Delta\vec{d} = \vec{D}_1 - \vec{D}_2$. Then we have,

$$|\Delta\vec{v}|^2 t^2 + (2\Delta\vec{v}\Delta\vec{d})t + |\Delta\vec{d}|^2 - (r_1 + r_2)^2 = 0$$

The roots of this quadratic are found by the usual formula $t = (-b \pm \sqrt{b^2 - 4ac})/2a$. The term inside the square root is

$$4(\Delta\vec{v}\Delta\vec{d})^2 - 4|\Delta\vec{v}|^2(|\Delta\vec{d}|^2 - (r_1 + r_2)^2)$$

Let this be 4α . There are three possibilities:

$\alpha < 0$ balls do not collide
 $\alpha = 0$ balls just touch
 $\alpha > 0$ balls collide

If the balls collide then there are two solutions for the collision time. The earlier time corresponds to the actual collision time, since the later time will be the time when the balls just touch after having passed through each other. Thus, if $\alpha > 0$ the balls will collide at time

$$t = \frac{-\Delta \vec{v} \Delta \vec{d} - \sqrt{\alpha}}{|\Delta \vec{v}|^2}$$

Given the trajectories of the two balls, this is an easy function to calculate. We have implemented it in our pool ball simulation.

3.1.2. Cushion Collision Time

Given the trajectory of a ball and the position of a cushion, will the ball hit the cushion, and if the ball will hit the cushion, when will the collision happen? The cushion is a straight side of the pool table. It is described by its two endpoints, \vec{e}_1 and \vec{e}_2 . The endpoints of each cushion are labeled so that in going from \vec{e}_2 to \vec{e}_1 one moves in a clockwise direction around the table.

The pool ball is a moving round object. It has a radius r , and a known trajectory. The ball will hit the cushion when it is a distance r away. See figure 5. The cushion is given by the set of points

$$\alpha \vec{e}_1 + (1 - \alpha) \vec{e}_2$$

where $0 \leq \alpha \leq 1$. Let $\vec{l} = \vec{e}_1 - \vec{e}_2$ be the vector for the length of the cushion. Then, since \vec{l} points clockwise around the table the unit vector pointing into the table perpendicular to \vec{l} , which we label \vec{u} , is given by

$$u_x = l_y / |\vec{l}| \quad u_y = -l_x / |\vec{l}|$$

Let \vec{R} be a vector in the direction \vec{u} whose length equals the radius of the

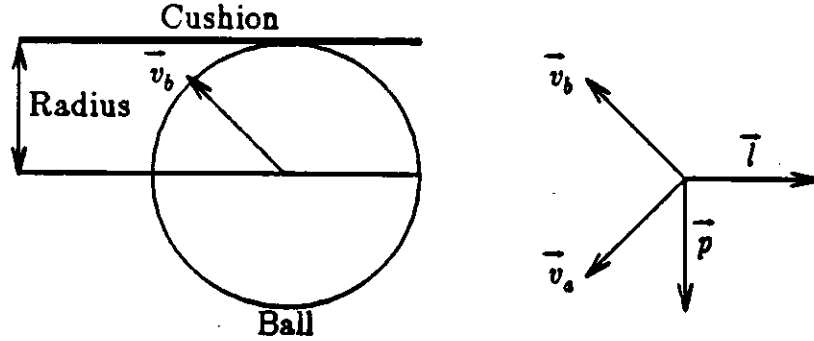


Figure 5. Impact Parameters of Ball-Cushion Collision

ball. Then $\bar{R} = r\bar{v}$. Then the locus of points on the table at a distance r from the cushion is

$$\alpha\bar{e}_1 + (1 - \alpha)\bar{e}_2 + \bar{R}$$

We want to know when the ball will cross this line. Thus, we want to solve

$$\bar{s}(t) = \alpha\bar{e}_1 + (1 - \alpha)\bar{e}_2 + \bar{R}$$

or

$$\bar{s}(t_0) + (t - t_0)\bar{v} = \alpha\bar{e}_1 + (1 - \alpha)\bar{e}_2 + \bar{R}$$

for t and α . As before, we define $\bar{D} = \bar{s}(t_0) - t_0\bar{v}$. Its components are D_x and D_y . The solution will be valid if $0 \leq \alpha \leq 1$. We get one equation for each vector component. For the x and y components they are

$$D_x + tv_x = \alpha e_{1x} + (1 - \alpha)e_{2x} + R_x$$

$$D_y + tv_y = \alpha e_{1y} + (1 - \alpha)e_{2y} + R_y$$

We solve this pair of equations for t and then for α . For t we obtain the solution

$$t = \frac{(e_{2x} + R_x)l_y - (e_{2y} + R_y)l_x + D_y l_x - D_x l_y}{v_x l_y - v_y l_x}$$

provided, of course, that $v_x l_y - v_y l_x \neq 0$. For α we obtain the solution

$$\alpha = \frac{D_x v_y - D_y v_x + (e_{2y} + R_y) v_x - (e_{2x} + R_x) v_y}{l_y v_x - l_x v_y}$$

provided that $l_y v_x - l_x v_y \neq 0$. If $0 \leq \alpha \leq 1$ then the expression for t is the collision time for the ball with the cushion.

3.1.3. Pocket Collision Time

Like a cushion, a pocket is a straight line defined by its endpoints. A ball enters a pocket when it crosses over the line. To determine when (and if) a ball enters a pocket, simply set the ball's radius to zero and use the expressions that determine when (and if) a ball hits a cushion.

3.1.4. Corner Collision Time

Corners are placed at the ends of a pocket. It is not immediately obvious that corners are necessary - to see that they are look at figure 6. Clearly the ball in figure 6 should not enter the pocket. It should bounce off the end of the cushion and move down and to the right.

The cushion will not schedule a collision with the ball, since the ball will never touch the face of the cushion. The pocket will schedule a collision, but we know that the ball should hit the end of the cushion. Therefore, the simulation needs a corner object at the ends of a pocket.

If a ball is headed towards a corner, when will it hit the corner? See figure 7. Let \vec{c} be the position of the corner. The ball will collide with the corner when its position is at a distance of the ball's radius (r) from the corner, or when the relation

$$|\vec{s}(t) - \vec{c}| = r$$

is satisfied. If we insert the expression for the ball's trajectory and square both sides of the expression we get

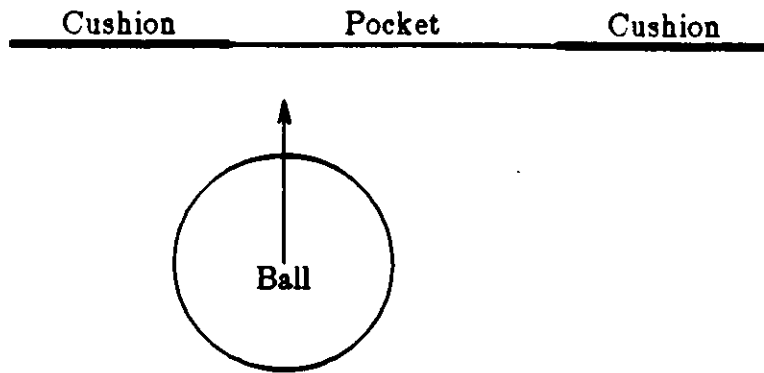


Figure 6. Ball Which Needs a Corner to Hit

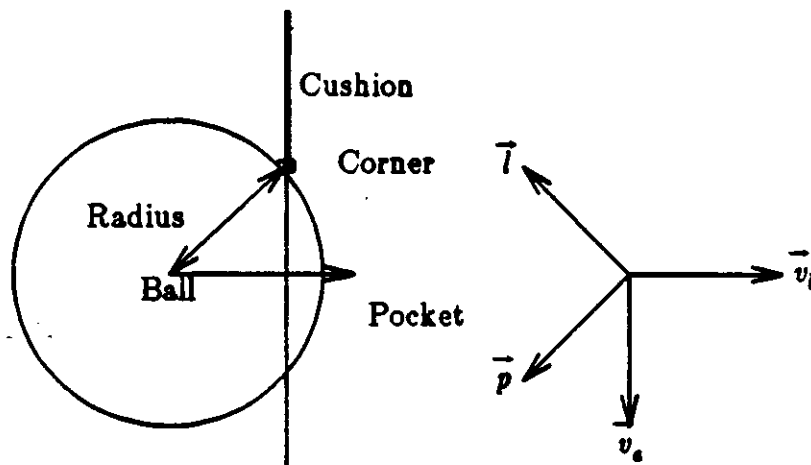


Figure 7. Ball Colliding with a Corner

$$|\vec{D} + t\vec{v} - \vec{c}|^2 = r^2$$

Evaluating the magnitude of the left hand side we get a quadratic in t

$$|\bar{v}|^2 t^2 + 2\bar{v}(\bar{d}-\bar{c})t + |\bar{d}-\bar{c}|^2 - r^2 = 0$$

This is solved by the standard quadratic formula. The term inside the square root is

$$4((\bar{v}(\bar{d}-\bar{c}))^2 - |\bar{v}|^2(|\bar{d}-\bar{c}|^2 - r^2))$$

Let this be 4α . The ball will collide with the corner if $\alpha > 0$. There are two solutions to the quadratic and, as with the time of a ball-ball collision, the actual collision happens at the earlier time, which is given by

$$t = -\frac{\bar{v}(\bar{d}-\bar{c}) + \sqrt{\alpha}}{|\bar{v}|^2}$$

3.1.5. Collisions

As we have seen, a pool ball can collide into another pool ball, a cushion, a pocket, or a corner. In each case the ball departs from the collision with a new velocity, which depends only on the states of the two colliding objects before the collision. In this section we present the calculations of the post-collision velocity for each case.

The ball's parameters before the collision are its velocity \bar{v}_b , and its position at the last time a position fix was taken, $\bar{s}(t_0)$. The velocity after the collision is denoted \bar{v}_a . A position fix is taken at the time of the collision. In the following sections we are concerned with determining \bar{v}_a .

3.2. Cushion Collision

A ball that hits a cushion bounces back. We ignore English and strange caroms and assume that the angle of incidence of the collision equals the angle of reflection. We define the *cushion elasticity* so that the magnitude of the velocity after the collision equals the *cushion elasticity* times the magnitude of the velocity before the collision. The *cushion elasticity* is between 0 and 1. This is

$$|\bar{v}_a| = \text{cushion elasticity} |\bar{v}_b|$$

$$0 \leq \text{cushion elasticity} \leq 1$$

As before, the position of the cushion is defined by its two endpoints \bar{e}_1 and \bar{e}_2 . If we let $\bar{l} = \bar{e}_2 - \bar{e}_1$ be a vector parallel to the length of the cushion then we can easily define the perpendicular to the cushion, \bar{p} . The components of \bar{p} are

$$p_x = l_y \quad p_y = -l_x$$

These vectors are illustrated in figure 5. That the angle of incidence equals angle of reflection implies

$$\bar{v}_a \bar{l} = \text{cushion elasticity} \bar{v}_b \bar{l} \quad \bar{v}_a \bar{p} = -\text{cushion elasticity} \bar{v}_b \bar{p}$$

as figure 5 illustrates. It is straightforward to expand these vector equations into their components and solve for the velocity after the collision. We get

$$v_{ax} = \frac{l_x(\bar{v}_b \bar{l}) - l_y(\bar{v}_b \bar{p})}{|\bar{l}|^2} \text{cushion elasticity}$$

$$v_{ay} = \frac{l_y(\bar{v}_b \bar{l}) + l_x(\bar{v}_b \bar{p})}{|\bar{l}|^2} \text{cushion elasticity}$$

Should the need arise, cushion collisions incorporating more complex physical behaviors could easily be added to the simulation.

3.3. Corner Collision

A ball that hits a corner does bounce back at a predictable angle. We assume that the ball is reflected off a line perpendicular to the line between the corner and the ball's center. See figure 7. Let the position of the corner be denoted \bar{c} , and the position of the ball at the time of the collision be denoted $\bar{s}(t)$. Then the vector, \bar{p} , from the corner to the ball's center is simply

$$\bar{p} = \bar{s}(t) - \bar{c}$$

The vector perpendicular to \bar{p} , denoted \bar{l} , has components

$$l_x = -p_y \quad l_y = p_x$$

Given these definitions we see that a corner collision is exactly the same as a cushion collision. Therefore the velocity of a ball after it hits a corner is given by the equations for the velocity of a ball after it hits a cushion.

3.4. Collision Between Two Balls

When two pool balls collide they bounce off each other. See figure 8.

We study the collision for arbitrary ball masses. Let m_1 be the mass of ball 1 and m_2 be the mass of ball 2.

The impact happens at the point where the balls touch. Thus, the force is along the line of centers at the time of the collision. Let $\vec{s}_1(t)$ be the position of ball 1 at the time of the collision and $\vec{s}_2(t)$ be the same for ball 2. Then the vector between the centers, \vec{c} , is simply

$$\vec{c} = \vec{s}_2(t) - \vec{s}_1(t)$$

The velocity component that is parallel to the direction of \vec{c} is a ball's collision velocity. The velocity component that is perpendicular to \vec{c} does not effect the collision. The vector perpendicular to \vec{c} , \vec{p} , has components

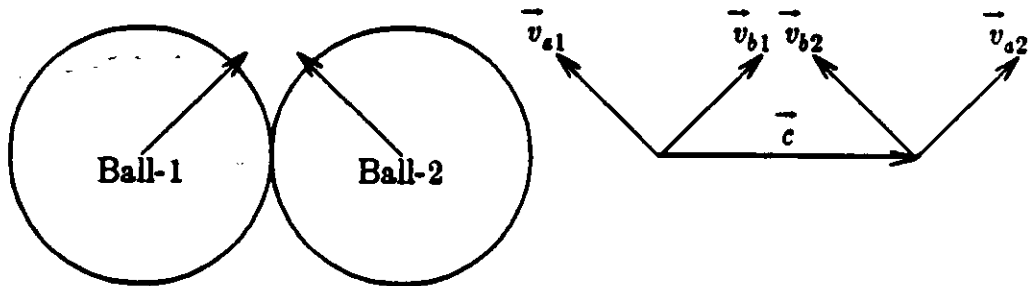


Figure 8. Two Balls Colliding

$$p_x = c_y \quad p_y = -c_x$$

The impact velocity of a ball is given by \bar{v}_b times the unit vector $\bar{c}/|\bar{c}|$.

Thus we have reduced the pool ball collision to a simple linear collision of two masses. If we assume that energy is conserved then we have an *elastic* collision. The final speed of two masses in an elastic collision is derived by Sommerfeld (p. 26) [Sommerfeld 52]. For the pool balls, the final velocity is a vector sum of the velocity along \bar{c} and the velocity perpendicular to \bar{c} . It is straightforward to show that the final velocities of balls 1 and 2 are

$$\begin{aligned} \bar{v}_{1s} &= \left(\frac{m_1 - m_2}{m_1 + m_2} \bar{v}_{1b} \bar{c} + \frac{2m_2}{m_1 + m_2} \bar{v}_{2b} \bar{c} \right) \bar{c}/|\bar{c}|^2 + (\bar{v}_{1b} \bar{p}) \bar{p}/|\bar{p}|^2 \\ \bar{v}_{2s} &= \left(\frac{m_2 - m_1}{m_1 + m_2} \bar{v}_{2b} \bar{c} + \frac{2m_1}{m_1 + m_2} \bar{v}_{1b} \bar{c} \right) \bar{c}/|\bar{c}|^2 + (\bar{v}_{2b} \bar{p}) \bar{p}/|\bar{p}|^2 \end{aligned}$$

This completes the section on the details of scheduling and modeling collisions. We now examine some of the interesting event simulation aspects of pool ball collisions.

4. Computational Details of Pool Simulation

In this chapter we present a detailed discussion of a variety of computational details of pool simulation. The first section of this chapter discusses initialization of the pool simulation. Then we discuss graphics produced by our pool simulation. When an object participates in two events at the same time we call them *simultaneous* events. A problem with simultaneous events is determining their correct processing order. The third section of this chapter discusses our use of message priorities to order simultaneous events.

Next we discuss the issue of instantaneous collisions. This is an important topic, since we have found that it is difficult to model events that appear instantaneous. Lastly, we discuss a clever technique to validate the numerical accuracy of the simulation by running it backwards.

4.1. Simulation Initialization

The simulation starts with no scheduled events. At the beginning, the pool balls should inform each other of their starting velocities. If each ball informs every other ball of its velocity, as it would during the simulation, then there is a problem. Every collision would be scheduled twice - once by each ball involved in the collision. To avoid this problem, a ball sends its initial `NewVelocity` messages only to balls numbered greater than itself. Suppose there are N balls in the simulation. All balls receive an initial `NewVelocity` message from ball 1, but no ball receives one from ball N .

4.2. Graphics

Plate 1² shows a single frame from an animation of a pool simulation. The frame shows two pool balls on a table with five cushions, one pocket

² Plates, which are images of the pool ball simulation, are collected at the end of the thesis.

and corners at the ends of the pocket. The simulation time is 0.0. Ball-2 is about to strike Ball-1 and sink it into Pocket-1. The vector from the center of Ball-2 is its velocity. Ball-1 has no vector because it is stationary. The head of the vector will be the position of the ball in one time unit, if its path is not changed in the interim. Plates 2-4 show the progress of the simulation for the next few time steps. We see that by time 3.0 Ball-1 has entered Pocket-1, as indicated by the list of balls in Pocket-1 under its label.

Animated graphics are easy to generate in an object-oriented simulation. The graphics output is controlled by an object called the Projector. Let the frame time be the time between graphic displays. Every frame time the Projector sends a message called `DisplayYourself` to all objects that can be displayed. When an object processes a `DisplayYourself` message, it draws and labels itself on the display.

4.3. Simultaneous Events

In an event based simulation an object may participate in two or more separate events at the same time. We call these *simultaneous* events. A simultaneous event causes an object behavior algorithm to run with several messages. For example, a ball can receive a `Collision` and a `DisplayYourself` message together. A question naturally arises. In what order should the ball process these messages?

The simulation system does not assign an order to the messages - it simply runs the object with the set of all messages with the same receipt time. (In fact, the simulation system should *not* assign an order to simultaneous messages since it does not know the semantics of the simulation and therefore cannot assign a correct order.)

The next two subsections discuss the issues of ordering the processing of simultaneous events of different types and simultaneous events of the same type, respectively.

4.3.1. Ordering of Simultaneous Event Processing by Message Priorities

It is necessary that simultaneous messages be processed in the same message type order by all objects. We give two examples which demonstrate that message types have to be processed in a prioritized order.

First, consider the graphics animation. We have decided that an animation frame should show the simulation state *after* all events up to and including the time of the frame. Suppose a collision happens at time 1.0. If the frame is at time 1.0 then the frame will show both ball's velocities after the collision. To enforce this constraint, each ball should process the Collision message before the DisplayYourself message.

Second, consider a ball that receives a NewVelocity message and a Collision message simultaneously. Suppose the simulation progresses as shown in figure 9.

There are four balls, moving as shown in the 'Before t ' picture. Focus your attention on ball 1. At time t ball 1 simultaneously gets several messages, including a Collision message indicating a collision with ball 2 and a NewVelocity message indicating that ball 3 is moving towards ball 1.

If ball 1 processes the NewVelocity message before the Collision message then no collision is scheduled between ball 1 and ball 3. This is incorrect - and will cause ball 3 to pass directly through ball 1. The BallBehavior algorithm should process the Collision message before the NewVelocity message, discover that ball 1 is resting in ball 3's new path and schedule a collision

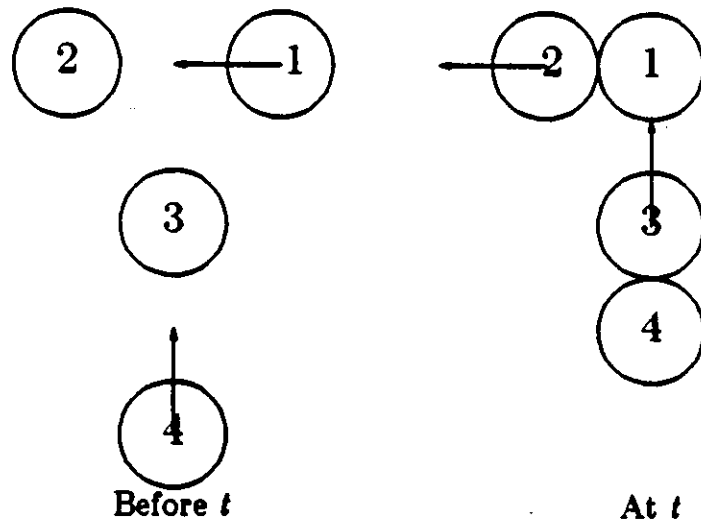


Figure 9. Simultaneous Events At Time t for Ball-1

between them.

These two examples demonstrate that correct operation of the simulation requires that simultaneous events of different types be processed in a particular order. How should the order be enforced?

Priorities based on message types are an effective means for ordering simultaneous messages. Each object type has a list of message types (of messages that it receives) ordered by the message type's priority. The object's behavior algorithm reorders the input messages by their priority before any messages are processed. This is seen in the sector pool simulation algorithm (in chapter 8), where the second line of the `BallBehavior` function is "Order messages by priority".

The list of message types ordered by priority for `BallBehavior` is

Collision
NewVelocity
DisplayYourself

(There are other message types, but this list is sufficient to illustrate our point.) We see that the last two examples will operate correctly with this priority list - Collision messages are processed before both NewVelocity and DisplayYourself messages.

We have shown a priority order for three of the messages received by BallBehavior. How would we choose a priority order for a simulation involving many more message types? If there are n types then $n!$ orders are possible. We cannot give an algorithm but the best strategy, from our experience, is to determine the relationship between all pairs of messages (which will often be 'does not matter which message is processed first') and then find an order that satisfies all relationships. The difficult step, of course, is determining the relationships since they depend on the semantics of the simulation.

4.3.2. Processing of Simultaneous Events of the Same Type

As we discussed in the last section, correct processing of simultaneous messages requires that they be ordered by message type. The algorithm we described will process multiple messages of the same type in an arbitrary order.

For example, suppose a ball hits several other balls at the same time. The ball simultaneously receives several Collision messages, and will process them in an arbitrary order. Will this produce a correct simulation? Unfortunately, the answer is no. The reasons for this answer are, however, quite interesting. There are two levels at which the answer is no. First, there is a problem that is analogous to the 'lost update' problem of concurrent

database access. Second, even if the collisions are processed in a serial order, we find that different serial orders unfortunately produce different final collision velocities.

We now elaborate on these problems. We call the entire multiple ball collision an 'interaction'. We assume that the reader is familiar with the database concept of a transaction. In the pool simulation a single collision between two balls is a transaction. The transaction reads the velocity of each ball before the collision and writes the velocity of each ball to complete the collision.

In the object-oriented simulation a collision is scheduled by sending a Collision message to each ball. The collision is executed by the independent processing of the Collision messages by the colliding ball objects. (See the discussion in section 2.3.) Thus, a transaction involves the processing of two messages. Collisions that occur at distinct simulation times are processed correctly. The corresponding transactions achieve a serial order equal to their simulation time order.

However, multiple ball interactions cause a problem. Suppose that Ball-2 simultaneously collides with Ball-1 and Ball-3, as shown in figure 10. The figure shows three balls colliding in one dimension so we need only

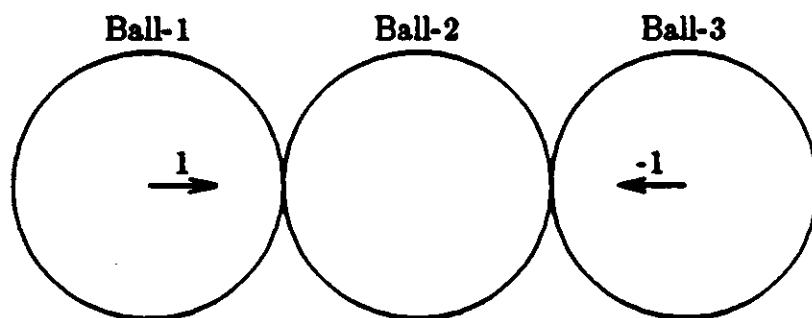


Figure 10. Three Balls Colliding Simultaneously

think about their velocities along the collision axis. When two balls of equal mass collide in one dimension they simply exchange their velocities, as can be derived from the equations at the end of section 3.4. If the interaction were processed as serial collisions, as would be the case in a centralized event simulation, then the three ball interaction progresses as shown in table 1. The column labeled 'Collision' identifies the pair of balls that collide. Notice that the interaction essentially *bubble sorts* the velocities.

However, in an object-oriented simulation the results are different. First, the collision between Ball-1 and Ball-2, and the collision between Ball-2 and Ball-3 are scheduled. Two messages schedule each collision. Ball-2 receives two Collision messages. One message schedules its collision with Ball-1 and the other schedules its collision with Ball-3. If Ball-2 processes these messages in the order Ball-1, Ball-2, then the entire interaction progresses as shown in table 2. The final ball velocities in this table are *not* the same as the velocities in the previous table.

The problem is that the object-oriented simulation interleaves the Ball-1 and Ball-2 collision with the Ball-2 and Ball-3 collision. Suppose we arbitrarily decided to pick a serial order in which the collision between Ball-1 and Ball-2 happens before the collision between Ball-2 and Ball-3. Then Ball-2 should send a new Collision message to Ball-3 *after* Ball-2 processes the Collision message from Ball-1, since otherwise Ball-3 receives a Collision

Collision	Ball-1	Ball-2	Ball-3
Start	1	0	-1
12	0	1	-1
23	0	-1	1
12	-1	0	1

Table 1. Ball Velocities in Collision Processed Serially

	Ball-1	Ball-2	Ball-3
Start	1	0	-1
Running Object			
Ball-1	0		
Ball-2		1	
Ball-2		-1	
Ball-3			0
Ball-1	-1		
Ball-2		0	
Final	-1	0	0

Table 2. Simultaneous Collision with 'Lost Update'

message with Ball-2's velocity from before the collision.

Regardless of the order in which Ball-2 processes the two Collision messages the final velocities will be incorrect. It is easy to see why this is the case when we think of a collision as a transaction with two reads and writes, as shown in table 3. Transaction 1 (in normal type) is the collision between Ball-1 and Ball-2. Transaction 2 (in italics) is the collision between Ball-2 and Ball-3. We see that both transactions read v_2 before either transaction writes v_2 , so there is no serial order of the two transactions.

Ball-1	Ball-2	Ball-3
Read v_2		
		<i>Read v_2</i>
	Read v_1	
	Write v_2	
	<i>Read v_3</i>	
	<i>Write v_2</i>	
Write v_1		
		<i>Write v_3</i>

Table 3. Two Collisions Seen as Two Transactions

The second interesting problem with simultaneous collisions is that different (serial) collision sequences can produce different final velocities. We now display a three ball collision in which the final velocities of the balls *do* depend on the sequence of collisions. The initial velocities are shown in figure 11a. The final velocities are shown in figure 11b. The solid arrows show the final velocities when the collisions are processed in the order Ball-1 and Ball-2, Ball-2 and Ball-3, and the dashed arrows show the final velocities when the collisions are processed in the opposite order. We can verify that both of these collision sequences preserve momentum and kinetic energy. The final velocities are different because of the physics of pool ball collisions. We have shown that the pool collisions are 'unstable' in the sense that an infinitesimal change in timing makes a large change in the final collision velocities.

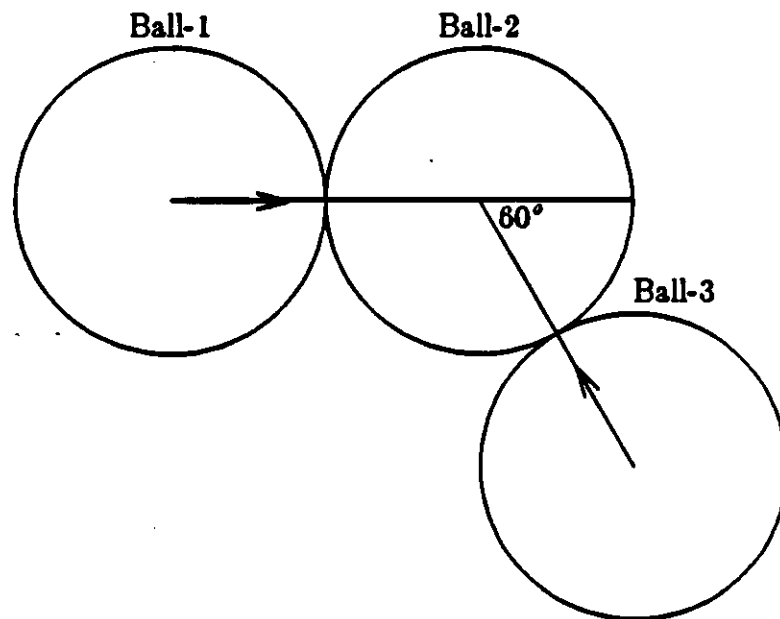


Figure 11a. Initial Velocities of Collision Involving Three Balls

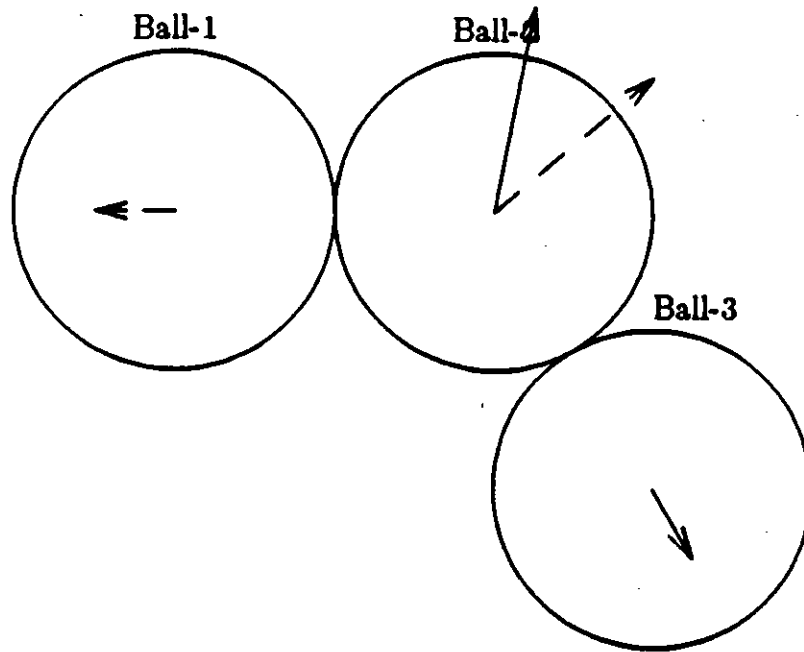


Figure 11b. Final Velocities of Collision Involving Three Balls

In conclusion, we have discovered that it is difficult to simulate simultaneous collisions of many balls.

4.4. Instantaneous Collisions

Let us look at the behavior of a collision in more detail. So far we have assumed that when two balls collide they instantly move in new directions. This simple view is not physically accurate, of course, since it takes a small amount of time for the momentum of one pool ball to be transferred to another, but it seems like a good simplifying assumption. However, we will show a collision, involving two balls and a cushion, which cannot be simulated as an instantaneous event.

In an object-oriented simulation there is a scheduler which runs the objects in chronological order. That is, whenever the processor becomes idle the scheduler runs the object which has the earliest scheduled event. When

an object runs it must process all the messages that it receives at the current simulation time. Therefore, a central rule of object-oriented simulation programming is that a message should be sent only into another object's future. Otherwise, the message might arrive too late to be processed. For example, if two pool balls collide then they should not send instantaneous NewVelocity messages to each other. Consider a simple scenario in which Ball-1 and Ball-2 collide. Suppose Ball-1 processes the collision first. Ball-2 processes the collision after Ball-1. If Ball-2 sends a NewVelocity message to Ball-1 then Ball-1 is cannot run since it should have processed this new message from Ball-1 when it ran earlier.

This problem gets worse when three objects are involved in a collision. Suppose Ball-1 is sitting next to Cushion and Ball-2 is about to strike Ball-1 into the cushion. See figure 12. Using our instantaneous collision model we expect three events to happen simultaneously: Ball-2 bangs Ball-1 into the cushion, Ball-1 bounces off the cushion, and Ball-1 bangs Ball-2 back out

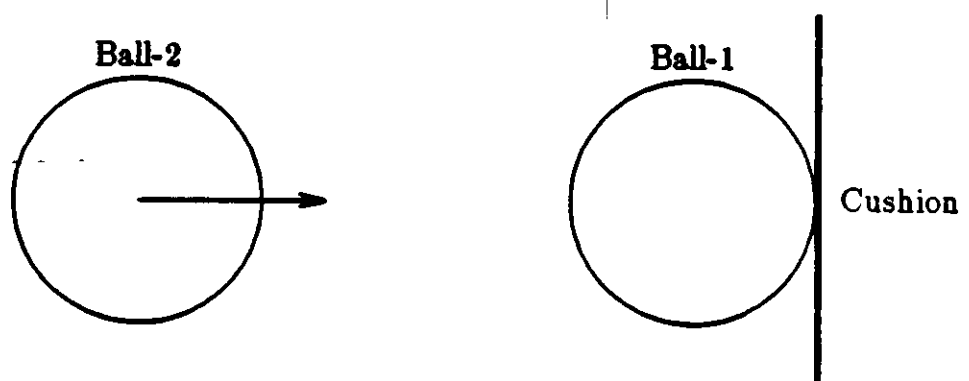


Figure 12. Ball About to Strike Another Ball Resting on a Cushion

into the table. The problem with simulating this collision is that some object will be sent a message that is not in its future.

Let us examine in detail the messages sent in this collision. Suppose the simulation schedules the objects in the order: Ball-1, Ball-2, Cushion. In table 4 we show the processing of messages. The objects run in the sequence in the first column. The rightmost two columns show messages received and sent by an object when it runs. All messages in the table are scheduled for receipt at the time of the collision. The last message in the table causes trouble. It is a collision message from Cushion to Ball-1 which Ball-1 cannot process because it has already run at the time of the collision.

To conclude, we have shown that a NewVelocity message must be scheduled for a time *later than* the collision that causes the change in velocity. There is no alternative. Another question naturally arises. How long after the collision should the NewVelocity message be scheduled? Ideally, the delay should reflect the physical system that is being modeled. In a pool ball collision between a moving ball and a stationary ball there is a delay between the instant when the two balls touch and the instant when the stationary ball has achieved its post-collision velocity. If we think of the balls as *compressible* rather than perfectly rigid then we obtain a consistent picture. The moving ball comes to a rest slightly inside the stationary ball, and

Object	Receives(From)	Sends(To)
Ball-2	Collision(Ball-2)	
		NewVelocity(Cushion)
Ball-1	Collision(Ball-2)	
		NewVelocity(Cushion)
Cushion	NewVelocity(Ball-1)	
		Collision(Ball-1)

Table 4. Messages Sent in an Instantaneous Collision

the stationary ball begins moving at the same time that the moving ball comes to rest.

In our simulation implementation, we have chosen to delay the NewVelocity message so that a fast moving ball compresses a stationary ball about one percent of its diameter.

4.5. Related Work

Our study of pool ball motion was partially inspired by simulation models of air warfare. Interactions between moving aircraft and stationary radar presents scheduling problems similar to those encountered in the pool ball simulation. An air warfare simulation model called SWIRL has been developed at Rand [Klahr 82]. It models the behavior of military objects such as radar and aircraft. A radar is either ground based and stationary or airborne and mobile. An aircraft, such as a bomber, will fly at constant velocity along a straight trajectory until it arrives at a target. In SWIRL, the entrance or departure of an aircraft from radar coverage triggers a chain of events.

It is important for a SWIRL simulation to schedule the beginning and end of all radar contact, since these events initiate other actions. For example, when an attacking bomber enters a defender's radar the bomber is detected (it isn't a stealth bomber) and a fighter is dispatched to intercept the bomber. Faught and Klahr's [Faught and Klahr 81] algorithm determines if an aircraft is within a radar's range by repeatedly testing whether the aircraft is in range. If it is not in range then the algorithm schedules another test. The next test is scheduled at the earliest possible time the objects could interact. This time exists because Faught and Klahr assume that each object has a maximum velocity. The next test is scheduled for the instant when the two objects would interact if they were headed towards

each other at maximum velocity.

This algorithm is computationally expensive. Suppose that an aircraft with a maximum speed of 100 mph is flying at 50 mph, directly towards a radar 100 miles away. Then there are checks performed at the times 0, .5, .75, ... hours after the start. This is an infinite sequence, so the aircraft never arrives. SWIRL avoids the infinite sequence by scheduling events only at multiples of a time granularity. Even so, many checks may be needed to schedule one interaction, and the number required depends exponentially on the size of the time granularity.

The authors concede that their algorithm could be improved and propose that a better scheduling algorithm would 1) schedule all possible future interactions, and 2) recompute future interactions when an object changes its velocity. This is identical to the algorithm that we have designed for pool ball simulation.

Faught and Klahr claim that this latter algorithm would prevent modeling "spontaneous" events, such as a bomber changing course without an interaction with another object. I believe they are incorrect in principle, although the detailed answer depends on the software structure of their simulation. If an aircraft changes its course, for whatever reason, then there must have been an event scheduled for the aircraft at the time of the course change. Thus, at that time the new interaction times between all pairs of objects could be recomputed. Our pool ball algorithm, applied to the air warfare simulation, possesses several advantages over Faught's algorithm.

- (1) It does not assume maximum velocities for any objects.
- (2) It calculates the exact time of interaction rather than rounding the interaction time to a simulation time granularity.

- (3) It is more efficient computationally since an interaction is computed once rather than repeatedly checked.
- (4) It uses no more memory than Faught's algorithm to store future events.

The problem of scheduling an interaction between a circular area, which may be mobile, and a trajectory arise in several different guises in air warfare simulation: when is an aircraft in radar range? when is a target in missile range? when does a jamming transmitter disrupt communications? It is clear that the methodologies we have developed in the pool simulation could be profitably applied to this problem.

4.6. Simulation Reversal

To test the numerical accuracy of the pool ball simulation we use a technique discussed by Aarseth [Aarseth 72] in his article about the simulation of gravitational motion, "Numerical Experiments on the N-Body Problem". He proposes that a good test of the accuracy of numerical solutions "is provided by the time reversibility of the equations of motion". The pool ball simulation can be studied using this method because an elastic collision between two pool balls is reversible. If the collision is run backwards then the balls leave the reverse collision with the negative of the velocity that they entered the forward collision.

It is easy to make a simulation that runs forward and then reverses itself. Let the quitting time of the simulation be q . Start the simulation at time $-q$, reverse it at time 0, and run it until time q . An event that happens at time $-t$ in the forward part of the simulation should happen, in reverse, at time t in the backward part of the simulation. This correspondence makes it easy to compare events between the two parts of the simulation.

A reverse simulation is easily added to the object-oriented pool ball simulation that we have discussed. At the start of the simulation, a message is broadcast to all balls so they reverse their velocity at time 0. This is called a ReverseYourself message. When a ball processes the message it reverses its trajectory, so the entire simulation turns around and restarts at time 0.

To implement a reverse simulation we add another branch to the case statement in the procedure BallBehavior. Figure 13 shows the code which defines the actions of a ball that receives a ReverseYourself message.

Since the simulation restarts at time 0, all future events must be canceled. This is performed in the first two lines of code of the ReverseYourself branch.

When ball runs at time 0 to process the ReverseYourself message there may be several other collisions scheduled. We have decided that a ball executes all events at time 0 before the reversal. Therefore, ReverseYourself is assigned a lower priority than all other message types, except DisplayYourself.

The only difficult aspect of the code is accounting for a ball that entered a pocket. It must return to the table at the correct time.

4.7. Results of Simulation Reversal Experiments

We have implemented the simulation reversal described above. The code has been tested and shown to work, both for balls on the table at time 0 and balls in a pocket at time 0.

We have used the simulation reversal technique to test the numerical accuracy of our implementation of the pool ball simulation.³

³While discussing numerical accuracy, it is appropriate to report the inaccuracy of the SQRT function in Interlisp. We found that (SQRT 10E8) is 10012 and (SQRT 10E10) is 1.1 x 10E5! We have used EXPT with an exponent of .5 as a substitute for SQRT.

```

case(message_type)
...
  ReverseYourself.
  cancel all planned interactions
  empty the set of planned interactions
  if(the ball is in a pocket)
  then
    /* The ball is in a pocket. It obtained a
      position fix when it entered the pocket. */
    reverse the ball's velocity
    time ball returns to the table ← -time ball entered pocket
    broadcast NewVelocity messages for receipt when
      the ball returns to the table
    send a ReturnToTable message to this ball for receipt when
      it returns to the table
  else
    /* The ball is not in a pocket. */
    take a position fix on the ball and reverse its velocity
    broadcast NewVelocity messages for time 0
  endif

```

Figure 13. Code for Reversing a Ball's Direction

The simulation began with the configuration shown in plate 5. The dimension of the pool table are 400x400. The balls were placed in a rectangular array, with random initial directions and initial speeds uniformly distributed from 0 to 100. Their radii was 30. The simulation started at time -3, and ran to time 3. There were over 100 collisions from -3 to 3. At the end of the simulation the balls were close to their starting positions, as can be seen in plate 6. From this experiment we conclude that the simulation is numerically accurate for runs with these conditions.

5. Sectorized Simulation

The primary activities of an object-oriented simulation are sending and processing messages. Let us consider the computational complexity of the pool ball simulation in terms of the number of messages sent. As for the centralized event list simulation, the complexity of the object-oriented simulation is $O(n \log n)$ per collision, where n is the number of balls. Each time a ball changes its velocity it sends a *NewVelocity* message to each other ball on the table. Thus, the complexity of simulating n events among n moving objects is $n^2 \log n$.

Our intuition says that this algorithm is overly complex - only balls which might collide soon with the ball which changed velocity should be examined for possible collisions. There should be no need to test for a possible collision with a ball at the far end of the pool table. How do we exploit our intuition? Suppose the pool table is divided into *sectors*. Then we can see that a ball is only concerned about hitting other balls that share a sector it occupies. We propose that the computational complexity of the simulation can be decreased by subdividing the table into sectors and considering only collisions between balls that occupy the same sector.

A sectorized pool table is attractive since pool ball motion is localized - within time δt a ball moves only $|v\delta t|$ distance. When a ball changes velocity it determines if it will collide with only the other balls in its sector, not all the balls on the table. This is a fixed number of calculations, limited by the number of balls that can fit in a sector. Thus, in the best case the complexity of a sectorized simulation is $O(\log n)$ per collision. However, there are additional costs. When a ball changes velocity, in addition to determining collisions, it schedules two events - leaving its current sector, and entering the next sector on its trajectory. These events must take place so that sec-

tors can keep track of the balls inside it and balls can keep track of the sectors they occupy. The complexity of these operations will depend on the size of the sectors and the velocity distribution of the balls.

In the remainder of this chapter we discuss the design of a sectorized pool simulation algorithm. First, we discuss the sectors, and the messages passed between them and the balls. Then we present the object behavior algorithms for sectorized simulation. Lastly, we discuss the scheduling problems of sectorized simulation, and their solution.

Let us consider the design of a pool ball simulation on a sectorized table in more detail. Suppose that the table is covered by fixed, nonoverlapping sectors. (You may want to think of a rectangular grid of sectors, since it is simple, but the algorithm will work for any shape. A sectorized pool table is shown in plate 7. There are four square sectors: Sector-1, Sector-2, Sector-3 and Sector-4, and seven balls. In the middle of each sector is the name of the sector. Under the name of each sector is a list of the balls in the sector. For example, under Sector-3 is listed the balls in Sector-3, Ball-2 and Ball-3. Likewise, the name of each ball is displayed in the middle of the ball, and the sectors the ball occupies are listed under its name.)

Since both the balls and the sectors have size, a ball may occupy several sectors and several balls may be in the same sector. Note that if the diameter of the largest ball is less than the smallest sector dimension then a ball may occupy at most four sectors. The number of balls in a sector is similarly limited.

In this simulation the significant events are collisions between balls and the passage of balls in and out of sectors. As in the nonsectorized algorithm, all predictable future events are scheduled when a ball changes its velocity. Therefore, to design the algorithm we ask, what future events can be

predicted when a ball changes its velocity?

First consider ball collisions. The ball which changes velocity must consider collisions with all balls in the set of sectors that it occupies. Second, consider sector departures. These must also be considered for all the sectors that the ball occupies. Lastly, consider sector entrances. Sector entrances must be considered for all the sectors that are adjacent to the set of sectors that the ball occupies. Accurate scheduling of sector departures and sector entrances insures that each ball always knows the sectors it occupies and each sector knows the balls that are on its territory. This knowledge in turn insures that all ball collisions are properly scheduled and executed.

An algorithm for an object-oriented simulation of pool balls on a sectored table is given in figure 14. There are two objects in the simulation - balls and sectors. (Cushions, pockets and corners have been left out because their behaviors are the same for sectored and non-sectored simulations.) The algorithm consists of a description of the state and behavior of each object.

In figure 15 we show the message flow graph for the sectored simulation algorithm.

There are five types of messages, which can be categorized into two groups. First, the *VelocityChange* and *NewVelocity* messages (which are labeled in italics in figure 15) convey information about a ball's change in velocity. Second, the *Collision*, *SectorEntry* and *SectorDeparture* messages schedule future events, involving two balls or a ball and a sector, based on a ball's new velocity. Note that each of these messages is sent by the object scheduling the interaction to both itself and the other object in the interaction.

Sector simulation

Object States:

State name	Properties
Ball	Position at time t Velocity Set of planned interactions Set of sectors occupied List of message types, ordered by priority
Sector	Perimeter Set of balls in the sector Set of adjacent sectors List of message types, ordered by priority

Object behavior algorithms:

BallBehavior(Messages, BallState)

Eliminate pairs of messages that cancel each other

Order messages by priority

/* Even if there are several collisions, only one
VelocityChange message needs to be sent */

CollisionHasOccured ← False

for all messages **do**

case (message_type)

 NewVelocity for ball X:

 cancel all collisions with ball X

if the time of a collision with ball X > the current time)

then

 schedule a collision with ball X

 add ball X to the set of planned interactions

endif

 Collision with ball X:

 CollisionHasOccured ← True

 update this ball's trajectory

 remove the planned interaction for ball X

 from the set of planned interactions

 SectorEntry:

 add the sector to the set of occupied sectors

```

SectorDeparture:
    remove the sector from the set of occupied sectors
endcase
end
if(CollisionHasOccured)
then
    cancel all planned interactions
    empty the set of planned interactions
    for all sectors the ball occupies
        send a VelocityChange message to the sector
    end
endif
endball

SectorBehavior(Messages, SectorState)
Eliminate pairs of messages that cancel each other
Order messages by priority
for all messages do
    case (message_type)
        VelocityChange from ball X
            for all adjacent sectors
                send the sector a NewVelocity message for ball X
            end
            for all balls in this sector
                if (ball  $\neq$  ball X)
                    then
                        send the ball a NewVelocity message for ball X
                    endif
                end
                cancel any planned interaction with ball X
                NewVelocity message for ball X
                if (ball X in not in the sector)
                    then
                        cancel any planned interaction with ball X
                        if (the ball will enter the sector on its new trajectory)
                            then
                                schedule a SectorEntry
                                add the ball to the set of planned interactions
                            endif
                        endif
                    end
                end
                SectorEntry for ball X
            end
        end
    end
end

```

```

for all adjacent sectors
    send the sector a NewVelocity message for ball X
end
for all balls in this sector
    send the ball a NewVelocity message for ball X
end
add ball X to the set of balls in the sector
remove the interaction with ball X from the set
    of planned interactions
SectorDeparture for ball X
remove ball X from the set of balls in the sector
remove the interaction with ball X from the set
    of planned interactions
endcase
if (message_type = VelocityChange OR message_type = SectorEntry)
then
    if (the velocity of ball X  $\neq$  0)
    then
        schedule a SectorDeparture for ball X
        add ball X to the sector's set of planned interactions
    endif
    endif
end
endprogram

```

Figure 14. Simulation Program with Sectored Pool Table

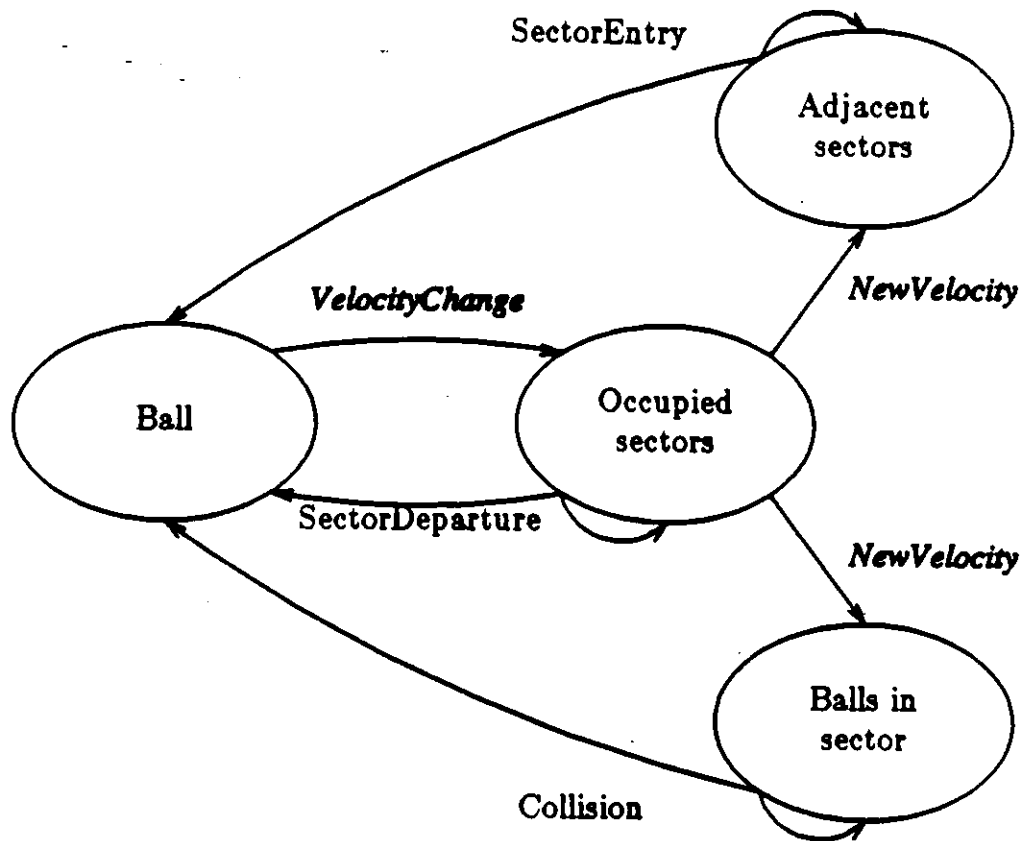


Figure 15. Message Flow Graph for Sector Pool Simulation

In detail, the algorithm works as follows. When a ball changes its velocity it sends a *VelocityChange* message to each sector that it occupies. Now consider one such sector. Unless the ball is stationary, the sector schedules a *SectorDeparture* for the ball. The sector sends *NewVelocity* messages to all adjacent sectors and to other balls within the sector itself. The *NewVelocity* messages identify the ball whose velocity changed, and contains its new trajectory. The adjacent sectors receive the *NewVelocity* messages. If the ball's trajectory intercepts an adjacent sector's perimeter the sector schedules a sector entry by sending a *SectorEntry* message to the ball whose velocity changed and one to itself.

A NewVelocity message is received by each ball that shares a sector with the ball whose velocity changed. These balls determine whether they will collide with the ball whose velocity changed. If a ball predicts a collision then it schedules a collision by sending a Collision message to the ball whose velocity changed and one to itself.

In summary, information about a ball's velocity change is conveyed from the ball to the sectors it occupies and then to the balls in these sectors and to adjacent sectors. These balls and sectors are responsible for scheduling interactions with the ball whose velocity changed.

The algorithm includes data structures and procedures for remembering scheduled interactions and canceling these interactions, in the same way that interactions were canceled in the nonsectored simulation. The messages that schedule future interactions, Collision, SectorDeparture and SectorEntry, may be canceled. The messages that relay the ball's changed velocity, VelocityChange and NewVelocity, are never canceled.

5.1. Sector Entry and Sector Departure Scheduling

We have already shown how to calculate the time of a collision between a ball and another ball, or a ball and a stationary object. The expressions given earlier for the collision time of these interactions clearly hold for sectored simulations as well as non-sectored simulations. In addition, the sectored simulation program must predict the time of sector entries and sector departures.

The perimeter of a sector is represented by a list of its corners. The edges of the sector connect adjacent corners in the list. A rectangular sector has a list of five corners, since the first corner repeats at the end of the list. The sector departure time of a ball from a sector is the instant a ball leaving

the sector crosses its perimeter. Thus, the departure time is simply the maximum of the times at which the ball crosses the sector's edges and corners. This maximum is easy to calculate since it is straightforward to calculate the time when the ball crosses over each edge and corner.

Conversely, the sector entry time is the time when a ball entering a sector first crosses its perimeter. This is simply the minimum of the times when the ball crosses the edges and corners of the sector.

Representing the sector perimeter by a list of its corners has a nice bonus - the simulation algorithms work for any polygon shaped sectors.

5.2. Instantaneous Messages in Sectorized Simulation

Consider again the message flow graph (figure 15) of the sectorized pool ball simulation. We would like all VelocityChange and NewVelocity messages to be instantaneous. An instantaneous message is received at the same simulation time that it is sent. It is sensible to want VelocityChange and NewVelocity messages to be instantaneous because they do not schedule future events. Rather, they convey information about the current state of the simulation. When a ball changes velocity all objects which may need to react to the change in velocity should be notified immediately. We now show that it is not possible to satisfy our desire - NewVelocity messages cannot be instantaneous.

In an object-oriented simulation an object may receive several messages at one instant. For example, in the pool ball simulation an object receives a cancellation message and the collision message being canceled at the same time. All messages received at an instant must be processed together or else the simulation may not be correct. This is an important point. It implies that there must not be a cycle of instantaneous messages between particular

objects. If there is a cycle at some simulation time t , then the first object in the cycle to run at time t must later receive another message at time t . This object has not run correctly at time t . Therefore, there must not be a cycle of instantaneous messages. We now give an example of a cycle of NewVelocity messages involving two objects.

Suppose NewVelocity messages were instantaneous and consider the following example. Ball-1 occupies two sectors, Sector-1 and Sector-2. Ball-1 collides with another ball and changes velocity. It sends VelocityChange messages to Sector-1 and Sector-2. Because Sector-1 and Sector-2 are neighboring sectors they send each other NewVelocity messages, thereby creating a cycle of instantaneous messages between Sector-1 and Sector-2.

We resolve this problem by adding a small delay to the NewVelocity message. Thus, it takes a little simulation time for the NewVelocity message to convey its information. If the time between events is greater than the delay then the simulation is still exactly correct. Therefore, simulation algorithm is correct in the limit as the delay approaches zero. On the other hand, the delay introduces another problem - which we discuss in the next section.

5.3. Ball Collision Time for Sectorized Simulations

In our initial tests of the pool ball simulation balls passed right through each other, despite our careful scheduling of collisions. For example, suppose Ball-1 is resting against the side of a sector and Ball-2 hits Ball-1 just as it enters the sector. Because of the delay in the NewVelocity message discussed in the last section, Ball-1 will not know about Ball-2 until they already overlap. Since the collision would be in the past, not the future, Ball-1 would not schedule it and Ball-2 would pass right through Ball-1. This cannot be allowed. There needs to be strong protection in the

simulation against balls passing through each other. We now discuss how we implement this protection.

Recall that balls collide when their surfaces touch, and that the time of collision, t , is given by the solution to a quadratic in t . A quadratic has three possible solutions: one real double root, two imaginary roots, or two real roots. Only if there are two real roots might there be a collision to schedule. There are five possible relations between the two real roots and the current simulation time - the time at which the NewVelocity message is received. These times are shown schematically in figure 16.

Situations 1 and 2 are collisions that are over - or that may never have happened. Situation 5 is a collision to be scheduled. Situations 3 and 4 are problems. They may indicate a collision that just happened, so the balls overlap and are moving apart, or they may indicate two balls that

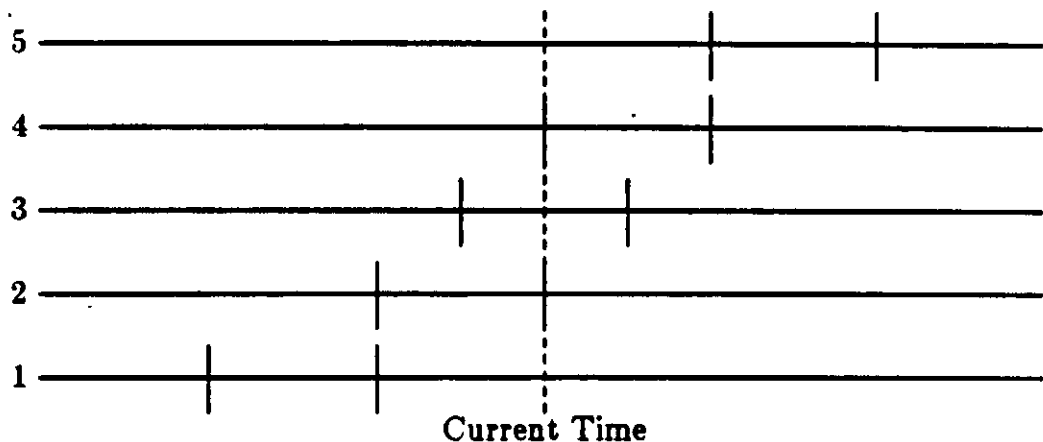


Figure 16. Possible relations between two real roots and the simulation time

overlapped before receiving a NewVelocity message and will pass through each other. -If the latter is true, then we want to schedule an immediate collision. That is, the naive algorithm:

```
if (two real roots)
  then
    collision time ← earlier of two roots
    if (collision time > current simulation time)
      then
        schedule a collision
      endif
    endif
  endif
```

should be changed to an algorithm that prevents two balls from passing through each other:

```
if (two real roots)
  then
    collision time ← earlier of two roots
    if (collision time > current simulation time)
      then
        schedule a collision
      else
        if (later time of two real roots > current simulation time
           AND the balls are closing)
          then
            schedule an immediate collision.
          endif
        endif
      endif
    endif
```

Implementing this algorithm is straightforward. The only detail that has not been elaborated is determining whether two balls are closing on each other. This can be found analytically by taking the time derivative of the distance between the two balls.

It is straightforward to show that the balls are closing at time t , or $d'(t) < 0$, if

$$\Delta \bar{s}(t) \Delta \bar{v}(t) < 0$$

where

$$\Delta \bar{s}(t) = \bar{s}_1(t) - \bar{s}_2(t)$$

$$\Delta \bar{v}(t) = \bar{v}_1(t) - \bar{v}_2(t)$$

In most cases, scheduling an immediate collision prevents a ball entering a sector from passing through another ball sitting on the edge of the sector. However, if the ball entering the sector is moving very fast then it might still tunnel through the stationary ball. How can this occur? Observe that a collision is scheduled if the distance between the two balls is decreasing d time units after their edges touch. If the moving ball is traveling so fast that it passes through the stationary ball by time d , then the stationary ball will not schedule a collision. Thus, for a collision to be scheduled the moving ball must move slower than v_m where

$$v_m = (r_1 + r_2)/d$$

Note, again, that in the limit as d approaches zero the simulation becomes exact.

5.4. Sectorized Simulation Initialization

The simulation starts with no scheduled events. In the beginning a pool ball must learn about the velocity of other balls in its sector. If each pool ball sends a VelocityChange message to its sector, and the sector sends NewVelocity messages to each ball then there is a problem - every ball collision would be scheduled twice. To avoid this problem a ball sends a VelocityInitialization message to its sector. The sector has a list of the balls within it. When the sector receives a VelocityInitialization message from a particular ball it sends NewVelocity messages to only the balls that *follow* the particular ball in the list of balls in the sector. This method transmits exactly one NewVelocity message between each pair of balls, so that each

ball collision is scheduled exactly once.

6. Experimental Results

In this section we present the results of experiments that compare the computational complexity of sectored and non-sectored pool simulation. The results demonstrate that as the number of balls increase, a sectored simulation runs faster than an identical non-sectored simulation. In fact, in experiments with a large number of balls the non-sectored simulation cannot run because it requires too much memory, whereas the sectored simulation runs successfully. In this section we first discuss the parameters of the experiments, and then we present their results.

The parameters of the experiments were chosen so that the ball motion would closely resemble the motion of atoms in a gas. Random ball motion provides a good initial test of the performance of our pool simulation programs.

The starting situation of one simulation is shown in plate 8. All simulations discussed in this section use a square table with four cushions and no pockets (which is how the game often appears to me!).

Balls are placed in a grid pattern, as shown in the plate. The units of length are arbitrary and could be scaled without influencing our conclusions. The unit we use is convenient for plotting the simulation on the Xerox SIP 1100 (Dolphin) display. A ball has radius 30. The length of the table is 50 times the number of balls along the side. For example, a simulation with 6 by 6 balls uses a 300 by 300 table. Each ball is assigned a random initial direction, and an independent random initial speed sampled from a uniform distribution between 0 and 100. All simulations run from time 0 to time 4. Plates 9 to 13 show the progress of a 36 ball sectored simulation from time 0 until time 4.

The results from our simulation experiments are shown in tables 5 and 6. Two parameters vary, the number of balls and the sector size. Table 5 shows the total number of simulation messages in the simulation, whereas table 6 shows the elapsed time of the simulation. Each entry in both tables reports the result of one simulation.

To improve the statistical validity of the experiments we considered running several simulations with different random number seeds for each case, but the long duration of the simulations with many balls made this impossible. However, for some experiments with a small number of balls we ran several simulations with different random number seeds and found little variation in the number of messages and elapsed time. For example, for the non-sectored simulations each experiment was run with six different seeds.

Number of Messages						
Balls	Sector Size					
	No Sectors	200	150	100	50	25
4	632	-	-	1001	1627	3445
16	4936	7677	-	5366	6549	11831
36	17313	-	13942	11492	13837	-
64	mo	-	-	17801	21330	mo

Table 5. Message Count of Pool Ball Simulations

Elapsed time (sec)						
Balls	Sector Size					
	No Sectors	200	150	100	50	25
4	580	-	-	1114	1807	4086
16	4540	5956	-	5562	8426	18564
36	38575	-	15926	13097	20234	-
64	mo	-	-	36256	68843	mo

Table 6. Elapsed Time of Pool Ball Simulations

The mean and standard deviation of the number of messages and elapsed time are shown in table 7. The small size of the standard deviation indicates that it is safe to rely on the results of single experiments for the more lengthy simulations.

The results in table 5 show that sectored simulations can use fewer messages than non-sectored simulations. Each entry in the table is a total of all messages involved with the logic of the simulation (but not collecting statistics or running the display). The sum includes messages of type VelocityChange, NewVelocity, SectorDeparture, SectorEntry, Cancel and Collision. There are two important observations to make about the data. First, for simulations with 36 or more balls the sectored simulation employs fewer messages than the non-sectored simulation with the same number of balls. In fact, the non-sectored simulation was not able to run with 64 balls. (The entry 'mo' indicates a simulation that encountered memory overflow on the Dolphin and could not be completed.) The 64 ball non-sector simulation did not even get started before it ran out of memory. The initialization requires that each pair of balls exchange a message. The initialization could not finish because the virtual address space allocated on the Dolphin was not large enough to hold all 2048 messages.

Balls	Messages		Elapsed Time	
	Mean	SD	Mean	SD
4	452	109	545	55
16	3850	708	3536	612
36	14293	1412	26336	5903

Table 7. Mean and Standard Deviation of Message Counts and Elapsed Time

The second important observation is that the speed of the simulation depends on sector size. For both 16 and 36 ball simulations the fewest messages are used when the sector size equals 200. The tradeoffs are that as the sector size becomes small, excessive effort is spent processing sector entries and departures, whereas as the sector size becomes large there are many balls in a sector and each velocity change is broadcast to a large local population.

Finally, examine table 6, which reports simulation elapsed time. For simulations with fewer than 15,000 messages the average time per message is about a second. However, as the number of messages climbs the time per message increases. We attribute the increased time per message to increased virtual memory paging. Since the non-sectored simulation schedules more messages it uses more memory and incurs paging delays with fewer balls in the simulation. Thus, the 36 ball sectored simulation runs nearly 3 times as fast as the non-sectored simulation, although it sends only 30% fewer messages.

While discussing memory use it is appropriate to recall our discussion about canceling messages at the end of section 1.3. We pointed out that an advantage of implementing cancellation in the simulation system is that it allows immediate reclamation of the storage for both messages.⁴ Currently, there is no cancellation function in the Time Warp system so cancellation was performed by the simulation objects themselves and message space was not reclaimed until after the cancellation message and the message being canceled were received. A lesson that our experiments teach is that memory management is important in object-oriented simulations.

6.1. Influence of Sector Size

Let us study our experimental results from another angle. Consider the total count of messages send, broken down by message type. Table 8 reports the count of each message type in the simulations reported in table 5. Examining message counts confirms our intuition regarding the tradeoffs that effect the selection of optimal sector size.

As the sector size decreases we expect the number of SectorEntry and SectorDeparture messages to increase since a ball path will cross more sectors, which we see in the data. In addition, the count of VelocityChange messages rises, because the average number of sectors occupied by a ball when it changes velocity increases as the sector size decreases. We expect fewer Collision messages since decreasing the sector size decreases the number of collisions that are scheduled and then canceled. Our expectation is confirmed by the data in table 8.

A decrease in sector size has a more complex influence on the count of the other two message types, NewVelocity and Cancel. Since the most numerous message type is NewVelocity, let us try to predict its count. A NewVelocity message is broadcast by a sector to balls in the sector and

Message counts by type for 36 balls				
Message type	Sector size			
	No sectors	150	100	50
SectorDeparture	-	1080	1206	1952
SectorEntry	-	480	680	1282
VelocityChange	-	341	373	609
Collision	2880	1746	1280	1070
Cancel	2304	2616	2404	3092
NewVelocity	12129	7679	5549	5832

Table 8. Number of Messages Tabulated by Message Type

adjacent objects when a ball in the sector changes velocity and when a ball enters the sector. Thus the number of NewVelocity messages, $\#(\text{NewVelocity})$, is approximately given by

$$\#(\text{NewVelocity}) \approx (\text{number of balls in a sector} + \text{number of adjacent objects}) * (\#(\text{VelocityChange}) + \text{number of actual sector entries})$$

(We denote the number of messages of type MessageType by $\#(\text{MessageType})$.) As sector size decreases, the first term in the product decreases, but the second term increases. Let us find expressions for the four counts in the last approximate expression. $\#(\text{VelocityChange})$ is given in table 8. The number of actual sector entries can be estimated by

$$\text{number of actual sector entries} \approx c * \text{total ball path length} / \text{sector size}$$

where the total ball path length is the total distance traveled by all balls in the simulation, and the sector size is the width of a sector. We have multiplied by a constant c because a ball has dimension and may cross several sectors when it travels a sector length. For sectors that are not unreasonably small c is about 3, but the exact value, which is not important to this discussion, depends on the sector and ball sizes. The number of adjacent objects is exactly 4 for all sectors.

The average number of balls in a sector can be estimated by

$$\text{number of balls in a sector} \approx \text{average number of sectors occupied by a ball} * (\text{number of balls in the simulation} / \text{number of sectors})$$

This estimate is actually low, since collisions are more likely to occur in

crowded sectors. What we really would like here is the expected number of balls in a sector when a ball changes velocity.

Finally we can estimate the average number of sectors occupied by a ball by

$$\text{average number of sectors occupied by a ball} \approx \frac{\#(\text{VelocityChange})}{\text{number of actual collisions}}$$

We have carried out these calculations using the data in table 8. The results are reported in the spreadsheet table 9. The data in table 9 were calculated for a simulation with the parameters:

Number of balls = 36
 Total free path = 7749
 Number of collisions = 278
 Table dimensions = 600x600

The results nicely validate our model. The estimate for $\#(\text{NewVelocity})$ is within 20% of the actual number of NewVelocity messages in all cases.

We find that studying these tables has improved our intuition for selecting a good sector size. In the next section we extend these ideas to an analytical model of message traffic.

Number of Sectors	4	9	36
VelocityChange	341	373	609
Estimated sector entries	155	232	465
Sectors occupied by a ball	1.23	1.34	2.19
Estimated balls per sector	11.0	5.4	2.2
Estimated NewVelocity messages	7459	5671	6648
Actual NewVelocity messages	7679	5549	5832

Table 9. Spreadsheet for Estimating Number of NewVelocity Messages

7. A Model of Message Counts in a Sectored Pool Simulation

We have shown that dividing the pool table into sectors can decrease the number of messages sent in a pool simulation. This result naturally provokes the optimization question, what sector configuration generates a simulation that sends the fewest messages? Assuming, for simplicity, a square table and square sectors of equal size the question becomes, what size sector is optimal? In this section we formulate a simple model which shows that an optimal sector contains 8 balls or fewer on average, depending on the number of collisions in the simulation.

Let F be a function giving the number of messages in a simulation. F is a sum of the number of messages of each type. Let $\#(MessageType)$ denote the number of messages of type $MessageType$. Then

$$F = \#(Collision) + \#(Cancel) + \#(NewVelocity) +$$

$$\#(SectorEntry) + \#(SectorDeparture) + \#(VelocityChange)$$

where we have organized the sum such that message types in the first row will be *more numerous* as the sector size increases, while the message types in the second row will be *less numerous* as the sector size increases. Since F is a sum of increasing and decreasing functions of the sector size we expect that F will have a minimum as a function of sector size.

Now let us define parameters for the simulation. Let the dimensions of the pool table be the unit square. Time units are arbitrary. We define the parameters

N	number of balls
V	ball speed
s	sector width
m	number of ball velocity changes
T	duration of simulation

We assume that N is constant and all balls travel at speed V . The definition of m means that when two balls collide m increases by two.

Let us begin the analysis by making the simplifying assumption that no collisions between balls occur. Let us also assume that the table is so large ($V \ll 1$) that boundary effects at the cushions are negligible. Therefore $m=0$, no collisions are scheduled and there are no Collision messages. Since a ball never changes velocity there are also no Cancel messages and no VelocityChange messages.

$$\#(Collision) = \#(Cancel) = \#(VelocityChange) = 0$$

The total distance all balls travel during the simulation is NVT . A ball crosses into a new sector every sector distance traveled, approximately. (The average sector crossing distance could be refined by averaging over all ways to cross a sector, but this stochastic analysis is not necessary for our argument.) Two SectorEntry messages are sent for each sector entered, one to the sector itself and one to the ball entering the sector. Thus

$$\#(SectorEntry) = 2NVT/s$$

The number of SectorDeparture messages equals the number of SectorEntry messages, so

$$\#(SectorDeparture) = 2NVT/s$$

Since the pool table is the unit square there are $1/s^2$ sectors on the table. The average number of balls in a sector is therefore $N/(1/s^2) = Ns^2$. NewVelocity messages are sent to all balls in a sector when a ball enters the sector and to the sector's 4 adjacent sectors when the ball enters the sector. Thus, if we assume that the balls are uniformly distributed then

$$\#(NewVelocity) = (NVT/s)(Ns^2 + 4)$$

We have given expressions for all types of messages, so now we can sum them to obtain F

$$F = (NVT/s)(Ns^2 + 8) \tag{1}$$

The minimum of F occurs at $s = 2\sqrt{2}/\sqrt{N}$, or, alternatively, where the aver-

age number of balls per sector, Ns^2 , equals 8. Thus, we have the result that if there are no collisions a simulation sends the fewest messages if there are eight balls per sector.

This is an attractive result. On reflection we realize that we have obtained an *upper bound* to the optimal sector size. To see this consider the effect of introducing collisions into the model. This will increase the count of Collision, Cancel and NewVelocity messages which, as we saw above, will tend to *decrease* the optimal sector size. It will not change the count of SectorDeparture or SectorEntry messages. (Although introducing collisions will increase the $\#(\text{VelocityChange})$ its increase will be overshadowed by the increase in $\#(\text{NewVelocity})$). Therefore, a sectored pool simulation should not have more than eight balls per sector on average.

We now show that this statement is validated by our experimental results. Table 10 reports the count of messages in our experimental simulations, modified to show average number of balls per sector rather than sector size. Table 10 shows that for all simulations the fewest messages are sent when there are an average of four balls per sector. In the 16 and 36 ball simulations we see that a simulation with more than 8 balls per sector sends more than the minimum number of messages. Thus, these experimental results do not contradict our statement that 8 balls per sector is an upper

Balls in Simulation	Balls per Sector				
	16	9	4	1	1/4
4	-	-	1001	1627	3445
16	7677	-	5368	6549	11831
36	-	13942	11492	13837	-
64	-	-	17801	21330	mo

Table 10. Message Count of Pool Ball Simulations

bound on the optimal sector size.

Figure 17 plots F as given in equation 1 versus sector size for four values of N : 64, 32, 16 and 8.

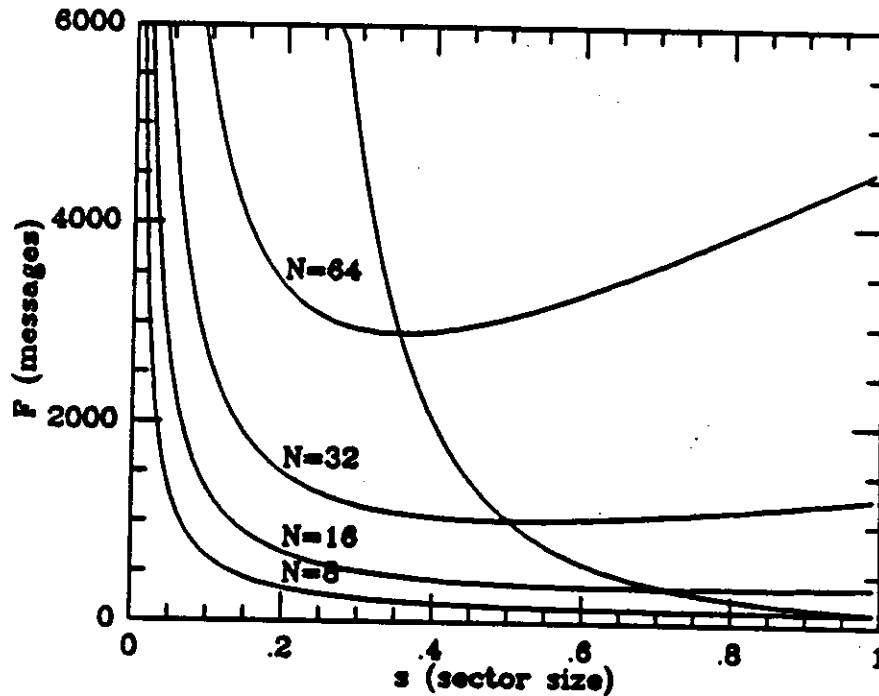


Figure 17. Messages as a Function of Sector Size - No Collisions

The curves shown assume that $VT=1$, which means that on average a ball crosses the table once during the simulation. Figure 17 also shows a curve connecting the minima of the set of F curves. The curves confirm that F has a minimum.

Now suppose that collisions happen, $m > 0$. Let $\#(\text{Cancel, Message-Type})$ denote the number of messages that cancel messages of type Message-Type. Whenever a ball changes velocity its previously scheduled sector interactions are canceled and new ones are scheduled. Thus we can write

$$\#(SectorEntry) = 2NVT/s + 2m$$

$$\#(Cancel, SectorEntry) = 2m$$

$$\#(SectorDeparture) = 2NVT/s + 2m$$

$$\#(Cancel, SectorDeparture) = 2m$$

When a ball changes velocity a NewVelocity message is sent to each ball in the sector and to the 4 adjacent sectors. Assuming as before that balls are uniformly distributed so the number of balls per sector is Ns^2 we can write

$$\#(NewVelocity) = (NVT/s)(Ns^2 + 4) + m(Ns^2 + 4)$$

The number of messages of the other message types is given by

$$\#(VelocityChange) = m$$

$$\#(Collision) = 2m$$

$$\#(Cancel, Collision) = 0$$

Therefore, the total number of messages is given by

$$F = (NVT/s)(Ns^2 + 8) + m(Ns^2 + 15) \quad (2)$$

We take the derivative of F and set it equal to 0.

$$NVT(N - 8/s^2) + 2mNs = 0$$

The analytic solution to this cubic in s can be obtained, but it is so complex that it provides no insight into the value of s that minimizes F .

Figure 18 plots F as given in equation 2 versus s , assuming that $VT=1$ and $m=N$, which would occur if each ball participated in one collision during the simulation.

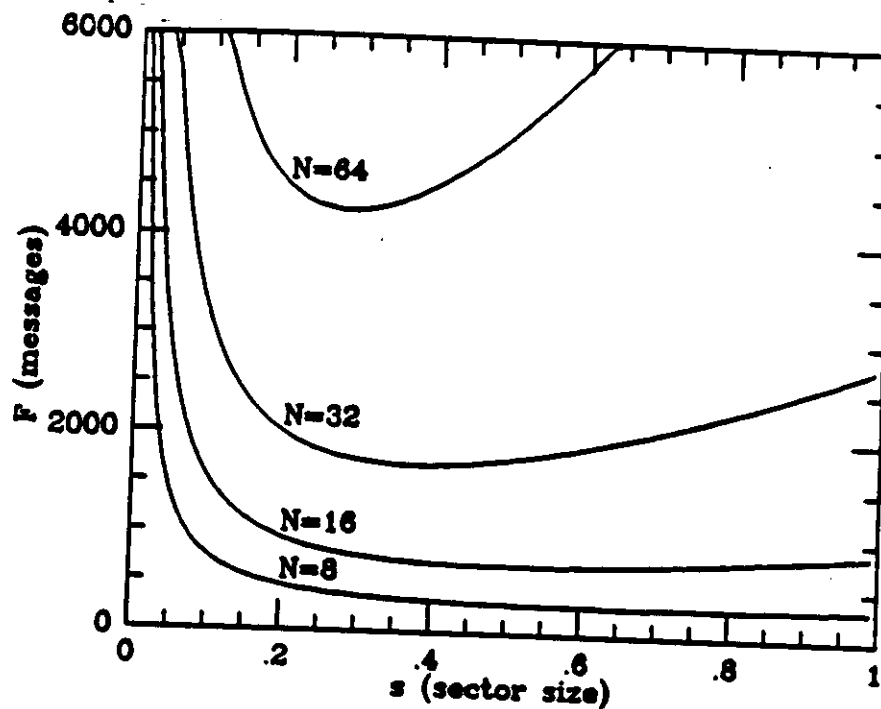


Figure 18. Messages as a Function of Sector Size - With Collisions

Comparing this figure with the previous one shows that increasing the number of collisions decreases the value of s which minimizes F , just as we predicted. It is also worth noting that for this example 8 balls per sector, as indicated by the curve of minima from figure 17, remains a good sector size choice.

This section has presented a simple model of message traffic in the pool simulation. The main result has been that the pool table should be divided into sectors so that there are fewer than 8 balls per sector on average.

8. Conclusions

8.1. Coping with Instantaneous Interactions

One lesson we have learned from this work is that it is difficult to use an object-oriented simulation system to model instantaneous interactions. We have already presented an example of an instantaneous interaction - the situation in which one ball sits by a cushion and another ball hits it, discussed in section 4.3.

In this section we present three main points. First we explain why object-oriented simulation exacerbates the modeling of instantaneous interactions. Second, we present several examples that explain it is necessary to model instantaneous interactions. And third, we present our temporary solution to the problem.

8.1.1. Instantaneous Interactions and Object-Oriented Simulation

In a message passing simulation system events are ordered in time by message timestamps. Programming an instantaneous interaction is difficult because the simulation system cannot allow an object to run twice at any given simulation time. The problem is that running an object twice at a given simulation time can produce an incorrect simulation. The results can be incorrect because an object that runs several times at a given simulation time to process all the messages scheduled for that time may produce different simulation results than an object that processes all of the messages at once.

For example, suppose a Ball object in the pool simulation receives a Collision message and its corresponding Cancel message. If it processes both messages together the ball cancels the collision. However, if the ball first receives and separately processes the Collision message, then it simulates the

collision - which is incorrect. When the ball later receives the Cancel message there is no matching Collision message to cancel.

In a centralized event queue simulation, instantaneous interactions are not a problem because an object can run one more than once at a given simulation time. The constituent events in an instantaneous interaction are ordered by the procedural sequence of the simulation. For example, the situation in which one ball sits by a cushion and another ball hits it, discussed in section 4.3, is easy to simulate instantaneously in a centralized simulation - the events are processed in their physical order, although they all occur at the same simulation time.

The same approach cannot be used in a message passing simulation, because the second collision between the two balls occurs at the same simulation time as the first collision. The simulation system must not allow the second collision to be scheduled after the first collision has been processed.

8.1.2. Instantaneous Interactions Are Common

Instantaneous events are not unique to pool ball simulations. Using our experience with the pool simulation, we now argue that simulation designs will frequently include instantaneous events. We have wanted instantaneous interactions for two reasons.

First, although a physical pool ball collision actually takes finite time, the duration of the collision is extremely small compared with other time periods in the model. Therefore we chose to assume instantaneous collisions. By analogy we infer that most modelers of discrete event systems will try to simplify their modeling task by making similar assumptions.

Second, we have invented objects, the sectors, which do not physically exist on the pool table. As we have seen, the sectors were introduced to

accelerate the simulation. Because sectors are independent of the physics of the ball interactions, interactions involving sectors, such as ball entries and ball departures, should be instantaneous. We feel that simulation design tactics will often encourage invention of objects to accelerate a simulation or simplify its design.

For example, consider a multi-user computer system, which can be modeled by a closed queueing network. Suppose the computer system enables a user to create a background task which requests service independently of commands entered at the user's terminal. This computer system can be modeled by a queueing network in which one customer 'spawns' another customer. In this model it is pedagogically convenient to imagine a 'spawning' service center with service time zero [Goldberg 83]. When a customer enters the spawning service center the customer is immediately released and another customer is immediately produced. Suppose that this model is simulated. We again find an instantaneous event - in this case, the service of a customer at the 'spawning' service center.

The last two examples have shown that it is reasonable to expect that simulation designs will frequently include instantaneous events.

8.1.3. Tentative Solution

We have developed a technique for programming instantaneous events in a message passing simulation system. We analyze a graph of message transmissions between the objects involved in an instantaneous interaction. If an object can receive a message at the event time *after* already running at the event time then there is a cycle in the message transmission graph which must be broken. We break the cycle, at an ad hoc place, by introducing a small delay into one message. In the limit as the delay goes to zero the simulation is exact. However, the delay introduces another problem - there is a

small gap in time where some of the events in the instantaneous interaction have occurred but others have not. This can introduce additional problems, as we saw in the case of a ball entering a sector at the point where a ball rests on the perimeter. We were able to solve this problem in the pool simulation, but in general the problems introduced by the short delay technique are difficult and appear to demand clumsy solutions.

Our experience indicates that the above technique is less than satisfactory. Adding a slight delay to a message that should be instantaneous solves one problem by creating another. Our conclusion is that a more satisfactory methodology for programming instantaneous interactions in object-oriented simulations is needed.

8.2. Contributions of This Work

- (1) We have demonstrated that pool ball motion lends itself to object-oriented simulation. We have implemented two types of pool simulations, simple (non-sectored) and sectored.
- (2) We have shown that a correct simulation can be implemented in which the pool table is divided into sectors that relay messages between balls. Provided a sector always knows which balls intersect its area and a ball always knows which sectors it occupies, then sectors can relay messages so that the simulation correctly schedules and executes all ball collisions.
- (3) Our experimental results show that dividing the pool table into sectors can accelerate simulation. In our experiments sectored simulation with 36 or more balls were faster than corresponding non-sectored simulations.

- (4) Our analytic results indicate that a pool simulation will run most quickly if there are fewer than eight balls per sector on average.

9. References

- [Aarseth 72] Aarseth, S.J., Numerical Experiments on the N-Body Problem, pp. 29-43, in M. Lecar (ed.), *Gravitational N-Body Problem*, D. Reidel Publishing Company, Dordrecht-Holland, 1972.
- [Alexandridis 78] Alexandridis, N. and Klinger, A., Picture Decomposition, Tree Data Structures, and Identifying Directional Symmetries as Node Combinations, *Computer Graphics and Image Processing*, Vol. 8, No. 1, 1978, pp. 43-77.
- [Fishman 78] Fishman, G.S., *Principles of Discrete Event Simulation*, John Wiley & Sons, New York, 1978.
- [Franta 77] Franta, W.R., *The Process View of Simulation*, in The Computer Science Library, Operating and Programming Systems Series, Elsevier North-Holland, Inc., New York, 1977.
- [Goldberg 76] Goldberg, A. and A. Kay "Smalltalk-72 Instruction Manual", SSL 76-6, Xerox PARC, Palo Alto, 1976.
- [Goldberg 83] Goldberg, Arthur, Lavenberg, S.S. and Popek, G., Performance Modeling of a Distributed Computer System, in Performance '83, the 9th International Symposium on Computer Performance Modeling, Measurement and Evaluation, College Park, Maryland, May 25-27, 1983.
- [Jefferson 82] Jefferson, D. and Sowizral, H., Fast Concurrent Simulation Using the Time Warp Mechanism, Part I: Local Control, Rand Note N-1906-AF, December 1982.
- [Kahn 79] Kahn K. M. "Director Guide", AI Memo 482B, Massachusetts Institute of Technology, Cambridge Mass. 1979.
- [Klahr 82] Klahr, P., McArthur, D., Narain, S. and Best, E., SWIRL: Simulating Warfare in the ROSS Language, Rand Note N-1885-AF, September 1982.
- [Klinger 76] Klinger, A. and Dyer, C.R., Experiments on Picture Representation Using Regular Decomposition, *Computer Graphics and Image Processing*, Vol. 14, No. 1, 1976, pp. 68-105.
- [McArthur 82] McArthur, D. and Klahr, P., The ROSS Language Manual, Rand Note N-1854-AF, September 1982.
- [Nance 81] Nance, R.E., The Time and State Relationships in Simulation Modeling, pp 173-179, Comm ACM, 24, 4, April 1981.
- [Novak 83] Novak, Gordon S., GLISP User's Manual, Technical Report HPP-21-1, Heuristic Programming Project, Computer Science Dept., Stanford University, February, 1983.
- [Sommerfeld 52] Sommerfeld, A., *Mechanics*, Academic Press Inc., New York, NY, 1952.

POOL SIMULATION AT 8.8

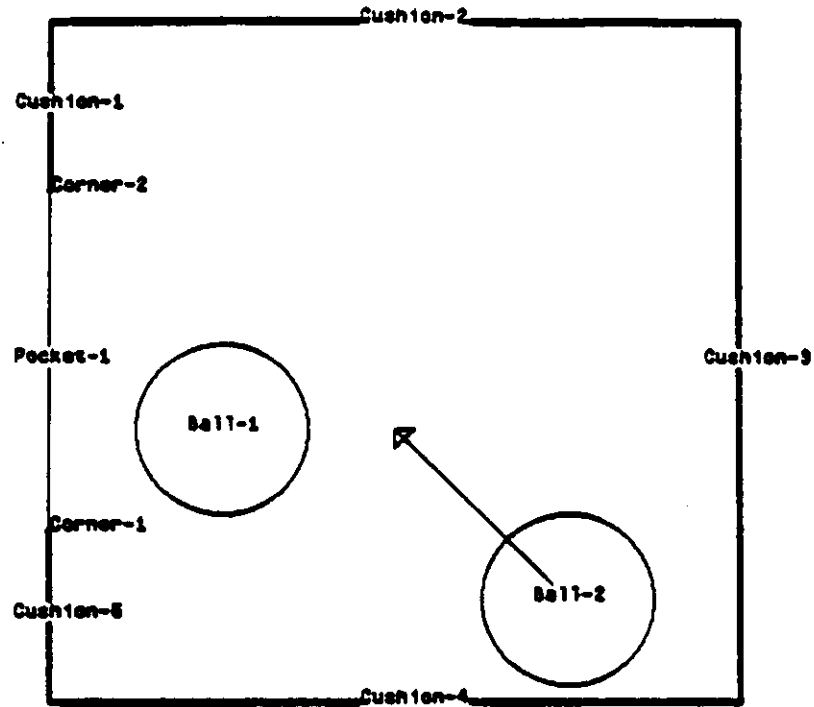


Plate 1. A Computer Graphics Picture of the Pool Simulation

POOL SIMULATION AT 1.0

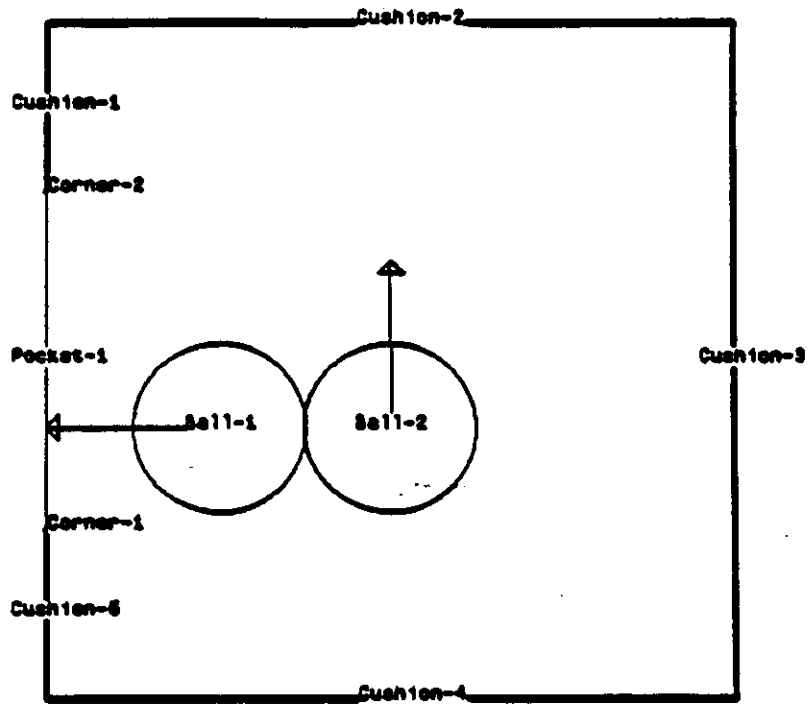


Plate 2. Pool Simulation at Time = 1.0

POOL SIMULATION AT 2.0

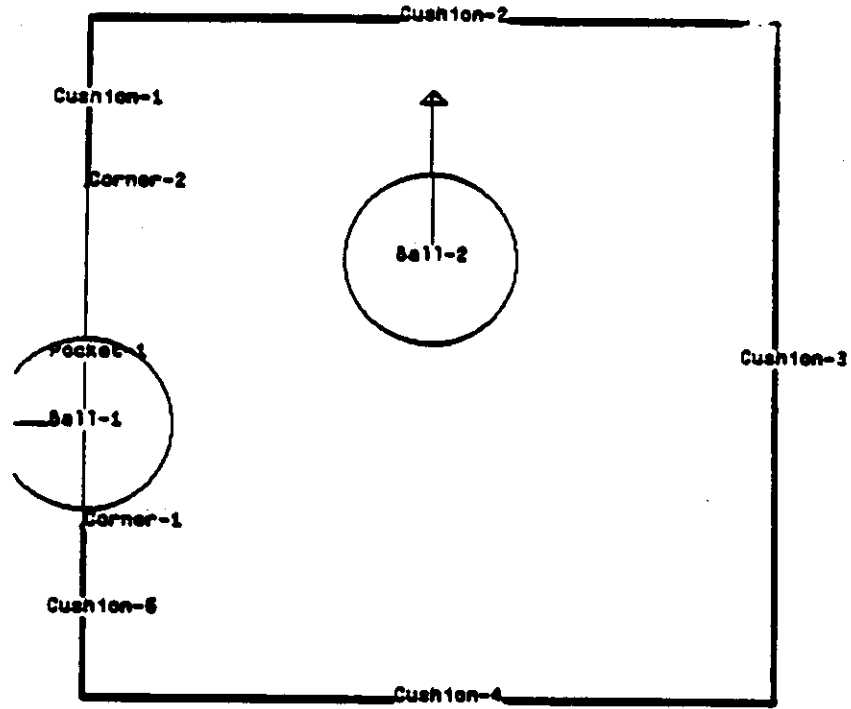


Plate 3. Pool Simulation at Time ≈ 2.0

POOL SIMULATION AT 3.0

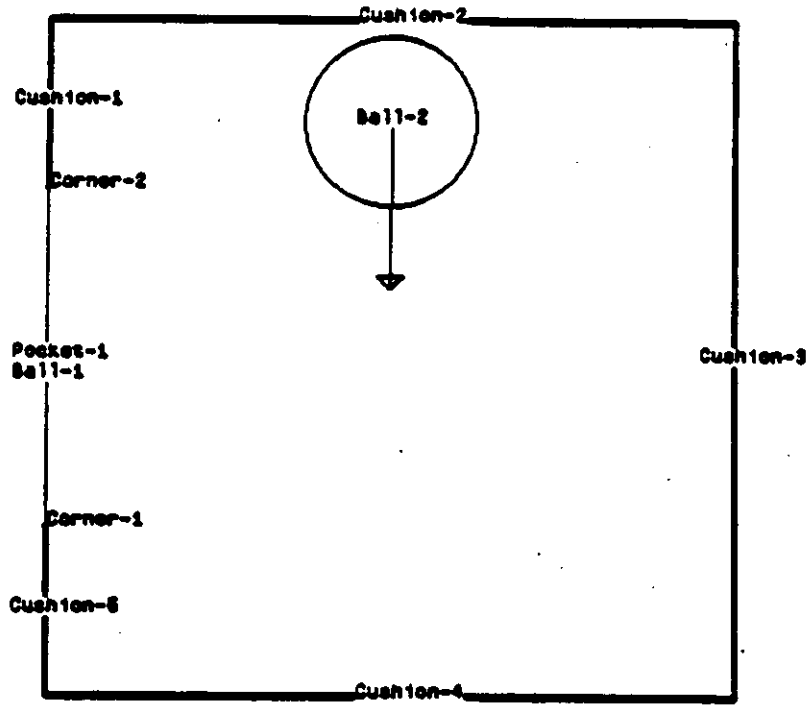


Plate 4. Pool Simulation at Time = 3.0

POOL SIMULATION AT -3.0

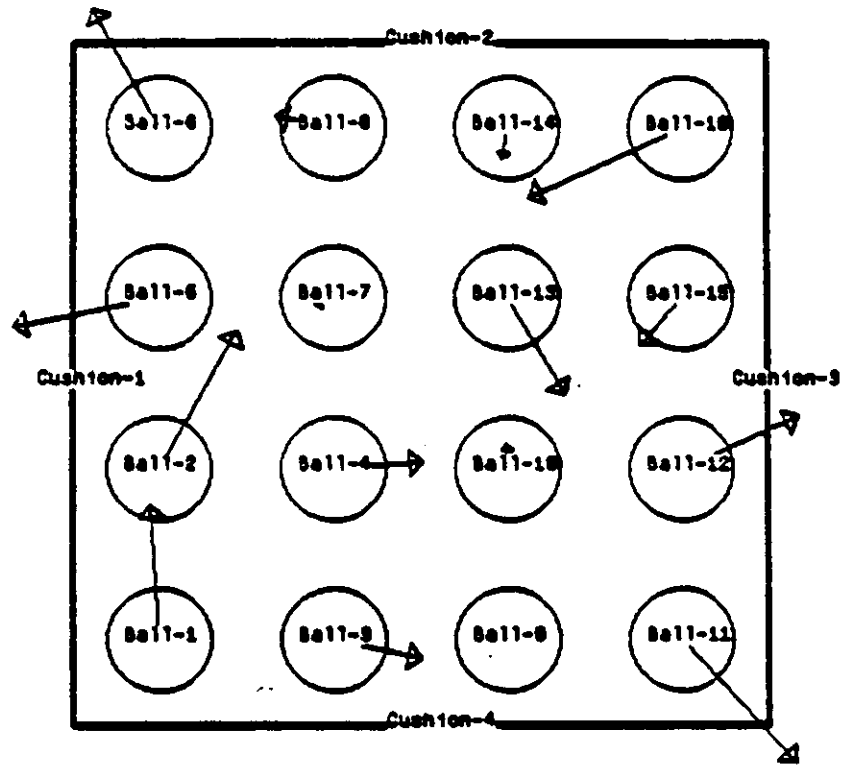


Plate 5. Pool Simulation at Start of Reversal Experiment

POOL SIMULATION AT 9.8

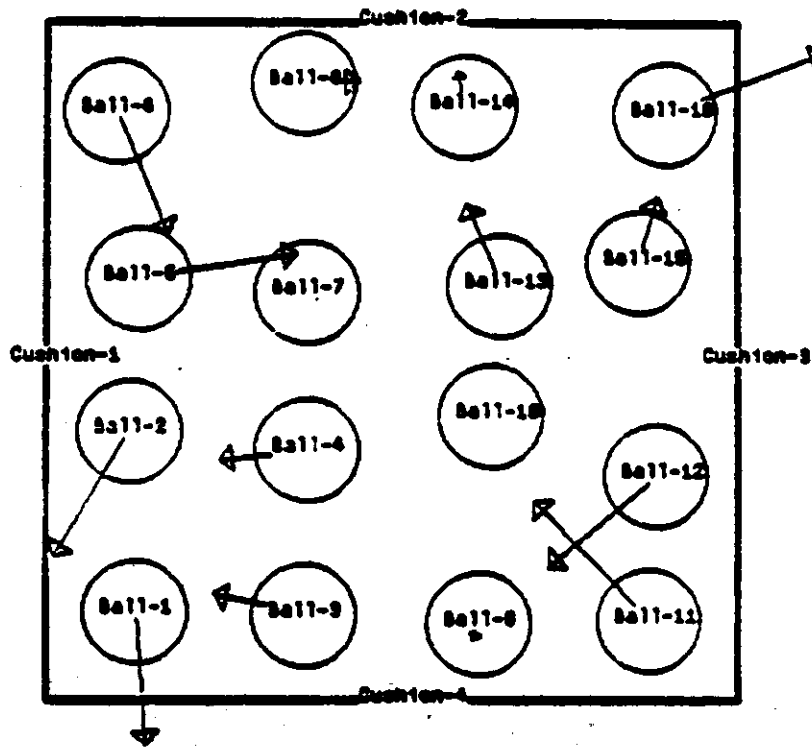


Plate 6. Pool Simulation at End of Reversal Experiment

POOL SIMULATION AT 9.8

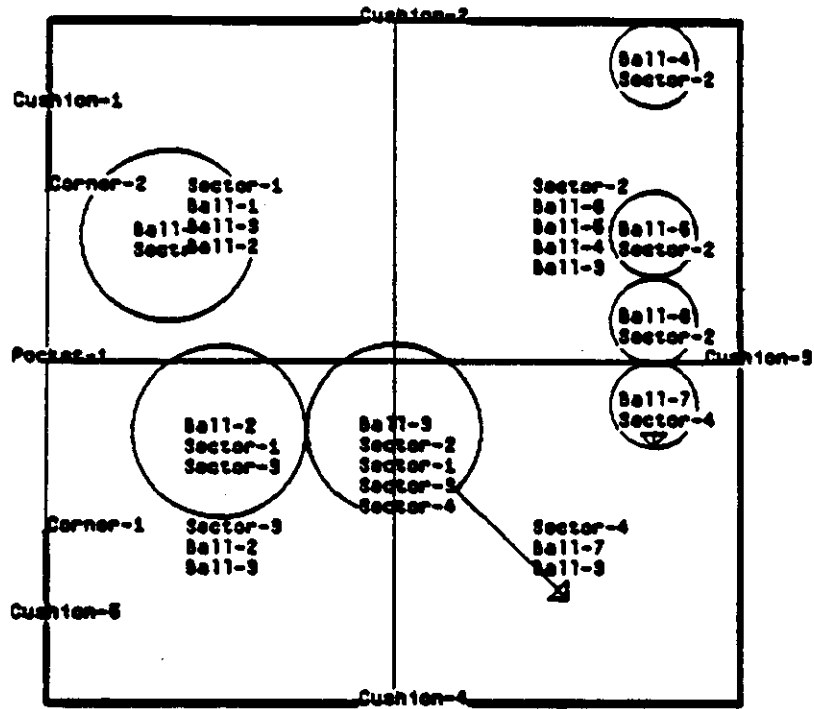


Plate 7. Picture of a Sectored Pool Table

POOL SIMULATION AT 0.0

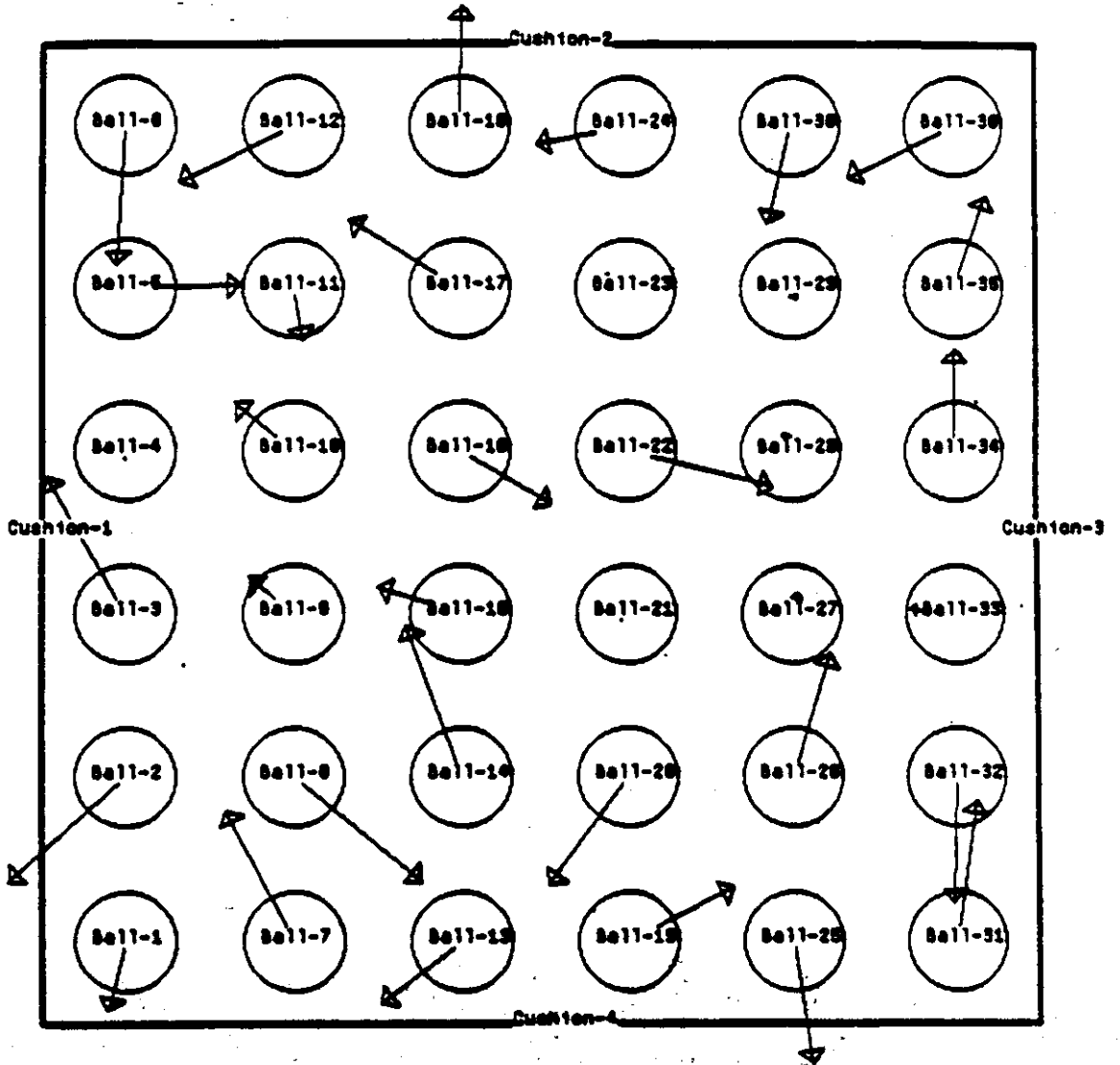


Plate 8. 36 Ball Non-Sectored Pool Simulation

POOL SIMULATION AT 0.0

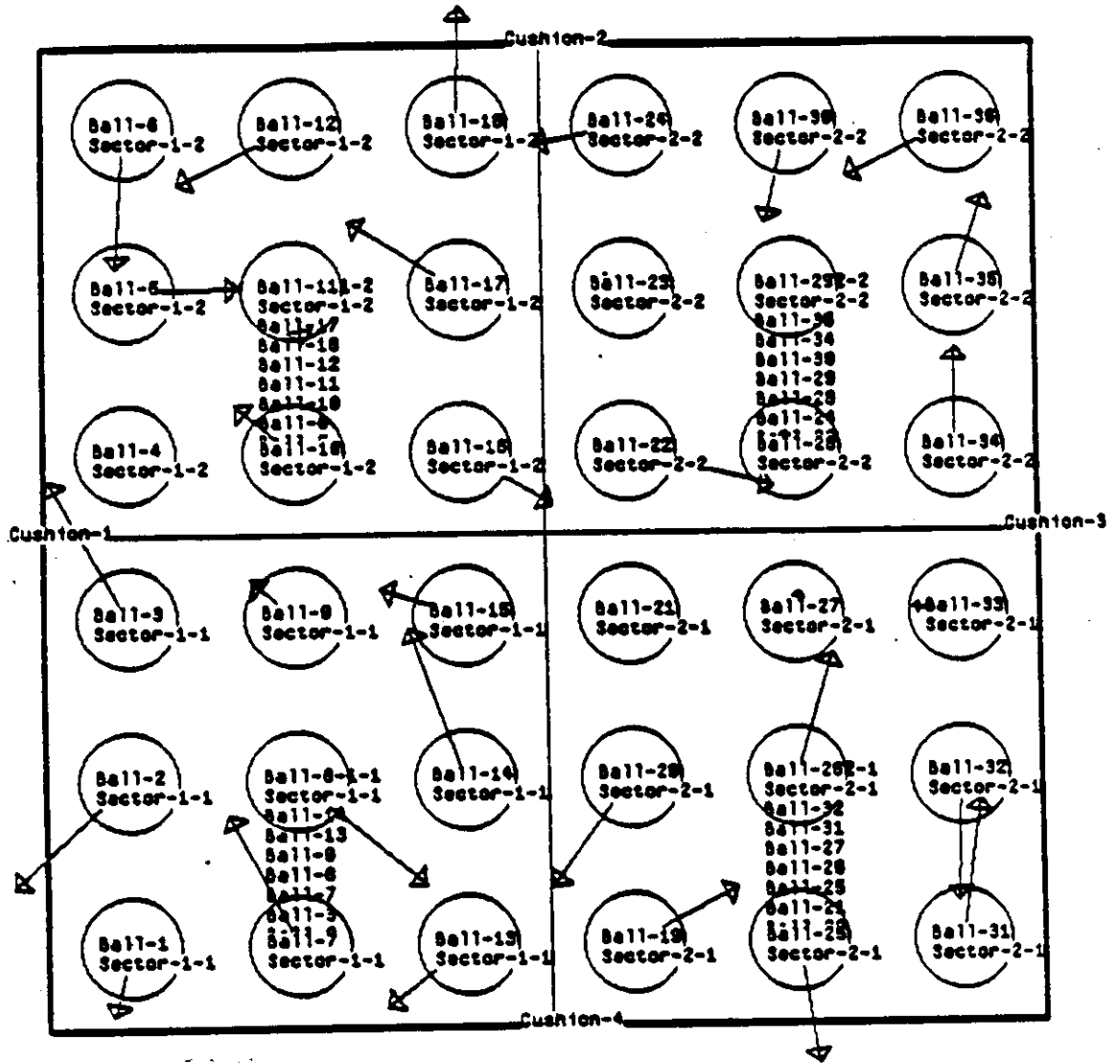


Plate 9. 36 Ball Sectored Pool Simulation

POOL SIMULATION AT 1.0

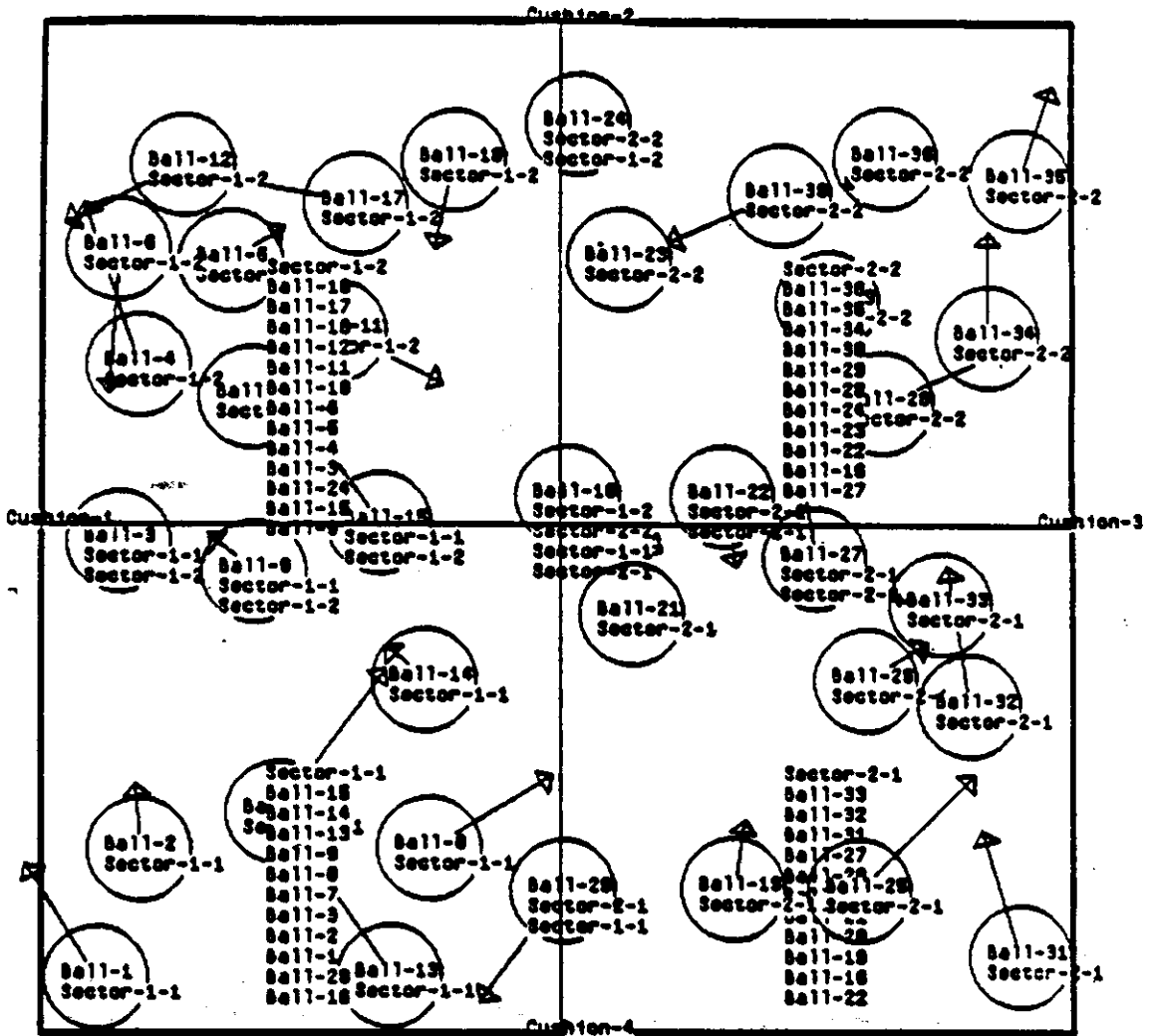


Plate 10. 36 Ball Sectored Pool Simulation

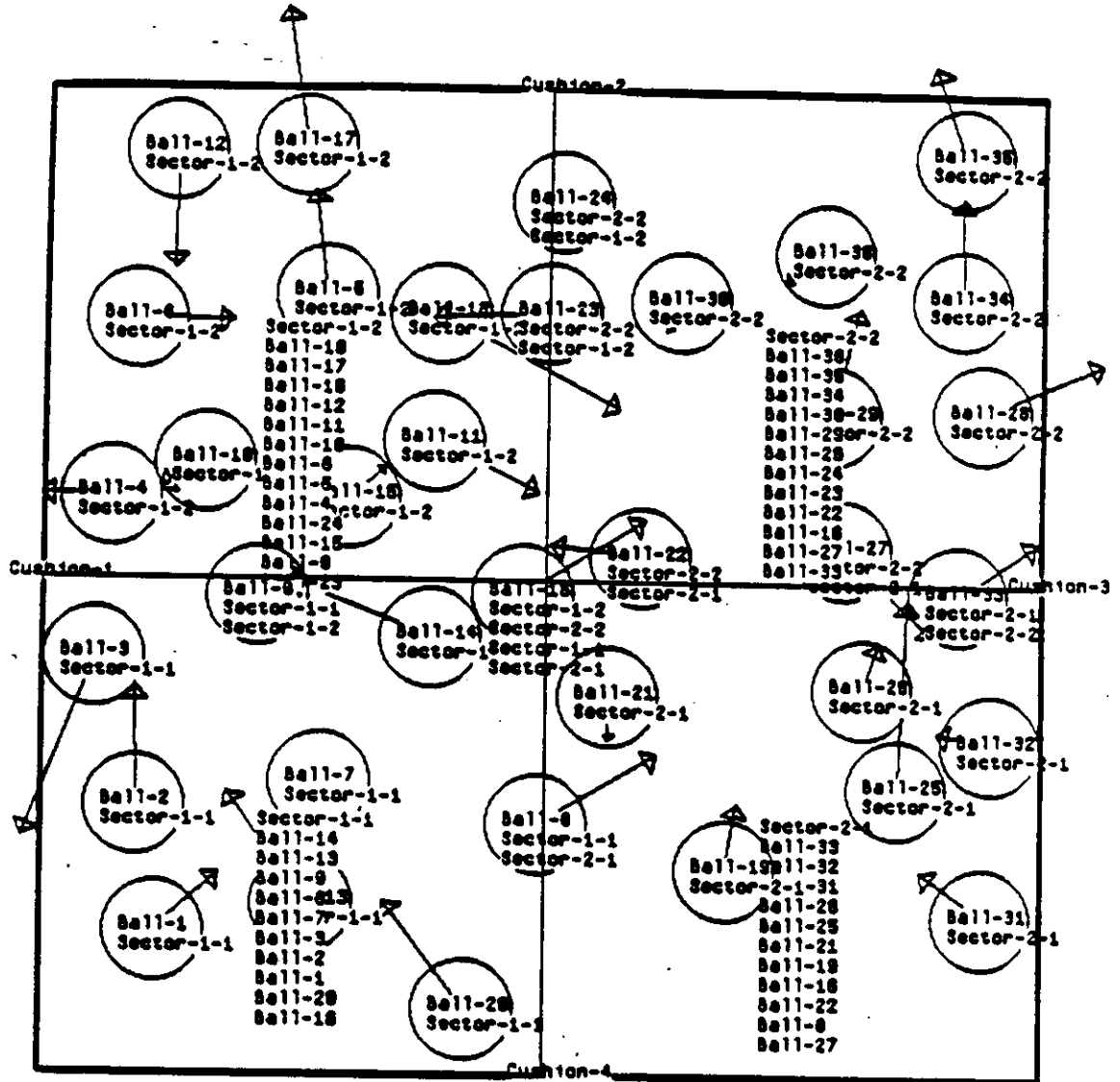


Plate 11. 36 Ball Sectored Pool Simulation

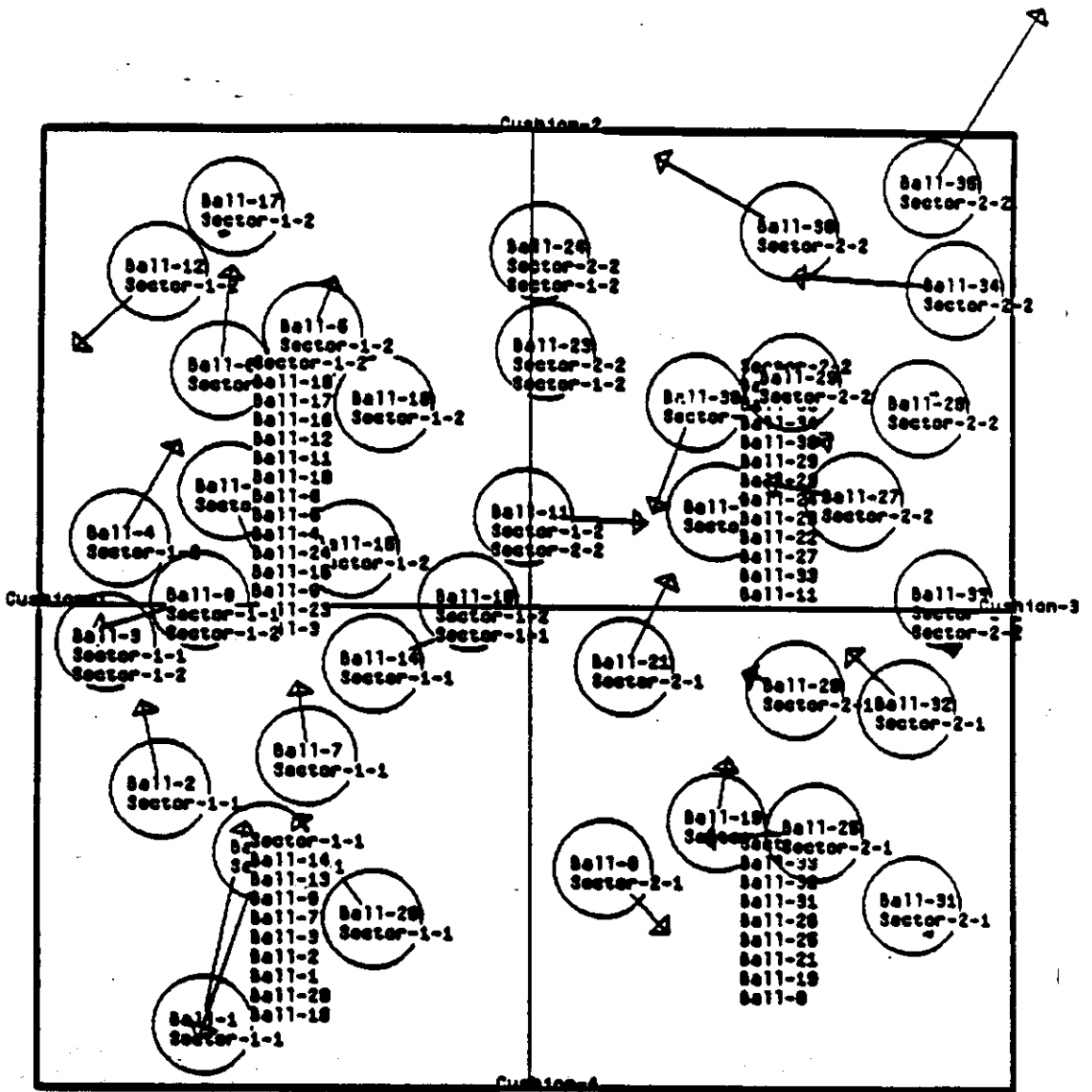


Plate 12. 36 Ball Sectored Pool Simulation

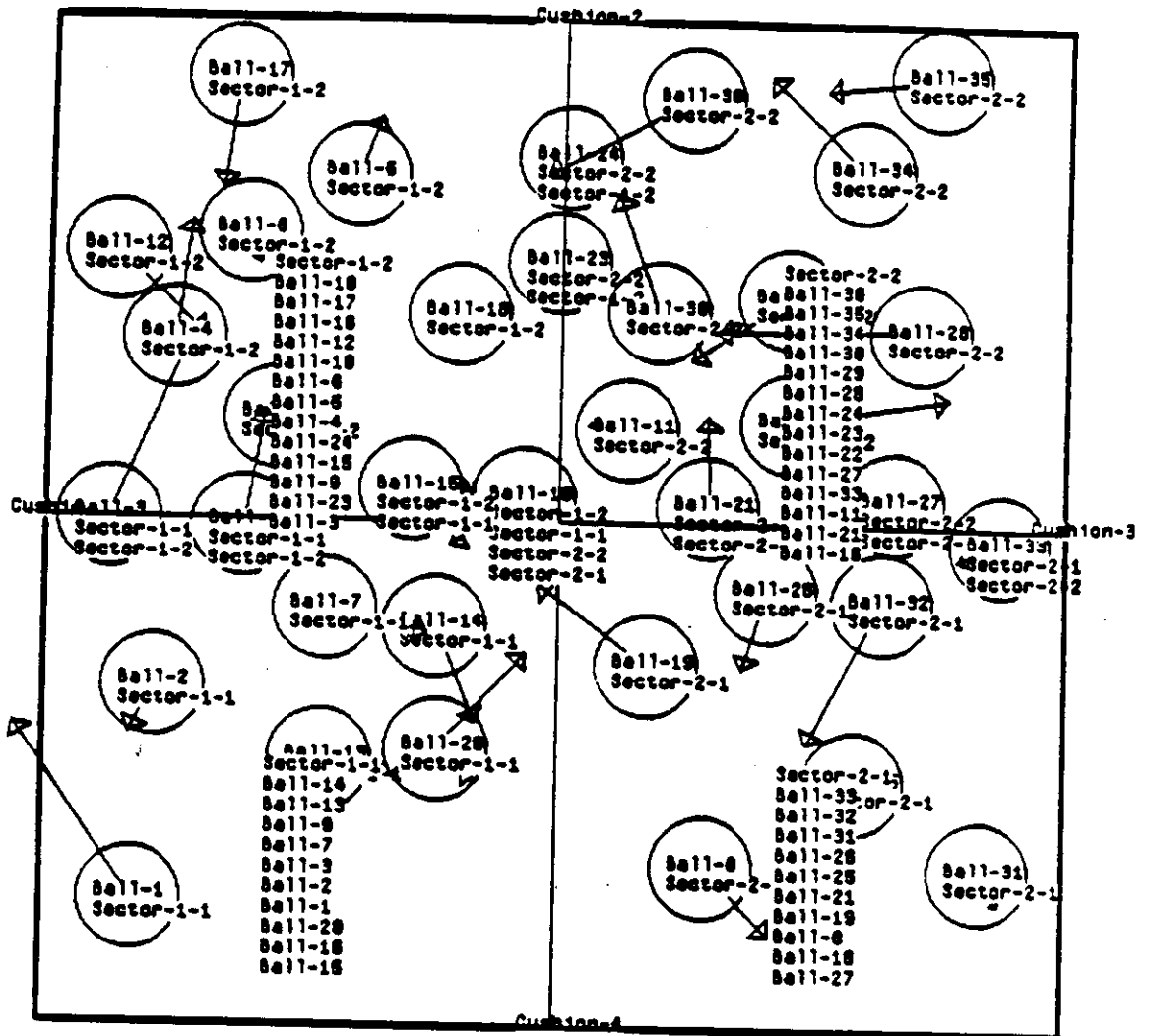


Plate 13. 36 Ball Sectored Pool Simulation