THE WAY THE BALL BOUNCES

Brian Hayes

A reprint from

American Scientist

the magazine of Sigma Xi, the Scientific Research Society

Volume 84, Number 4
May–June, 1996
pages 331–335

THE WAY THE BALL BOUNCES

Brian Hayes

The billiard ball rolls smartly along the baize, aimed for another ball at the far end of the table. The player watching its trajectory anticipates the solid *tlock!* of contact, to be followed by the familiar textbook demonstration of the conservation of energy and momentum. But something has gone awry on this billiards table. The moving ball passes right through the stationary one, without altering its own motion or disturbing the repose of its target. And then the rolling ball glides wraithlike through the rail at the edge of the table and continues imperturbably sailing across the room, with no inclination to fall to the floor.

What's going on here? It's not quantum tunneling; it's a failed computer simulation. On a real billiards table, collisions just happen. Billiard balls require no special instrumentation to detect each other's presence; they simply obey the "law of nature" that says two solid bodies cannot occupy the same space at the same time. You might think that a simulated billiard ball could be made to behave in the same way—that it could be given the property of solidity, so that it would automatically rebound from obstacles. But such autonomy of action is an illusion in the land of computer simulation. When you create a simulated world, nothing falls to earth unless you remember to turn the gravity on. Nothing bounces unless you tell it exactly where and when to bounce.

This need for continual supervision and intervention may come as a surprise if you know the art of simulation only by observing inputs and outputs. From the outside, a physics simulation looks like a clockwork universe, which can be set ticking and left to work out its own destiny. Events inside the computer seem to unfold just as they do in nature, following the same rules of cause and effect. Accordingly, you come to expect a one-to-one mapping between processes in the physical world and algorithms in the simulated one. But much of what goes on inside a simulation has no counterpart in the real world.

In pointing out this disparity between real and virtual physics, my aim is not to question the validity of computer simulation as a tool in science and engineering. Simulation is a valuable and productive technique; I dabble in it myself. But I am intrigued as I continually rediscover that what looks effortless in nature can be so laborious to compute.

Brian Hayes is a former editor of American Scientist. *Address: 211 Dacian Avenue, Durham, NC 27701. Internet: bhayes@amsci.org.*

The Naive Algorithm

The problem of detecting when two objects touch or overlap comes up in many computational contexts. It is the major challenge in planning the motion of robots and computer-controlled machine tools. It is also the crucial step in studies of molecular "docking," such as simulations of how an enzyme binds to its substrate. Video games and computer animation also rely on algorithms for collision detection. Another obvious example is the simulated crash testing of automobiles. Still, the most convenient place to continue our study of the basic issues is at the billiards table.

The naive algorithm for simulating billiards—"naive" in the sense that it's the first thing that pops into my head—goes like this. At some initial instant t_0 we know the positions and velocities of all the balls on the table. We plug these numbers into the equation of motion for each ball to find its position and velocity at a future time, $t_0+\Delta t$. The results serve as the starting data for another iteration of the same process. Continuing in this way, we calculate a sequence of snapshot images, showing the state of the system at successive intervals of Δt.

In the simplest model of billiard dynamics, the equation of motion for each ball is just $\mathbf{x}=\mathbf{x}_0+\mathbf{v}\Delta t$. (Here the boldface variables $\mathbf{x}$ and $\mathbf{v}$ are vector quantities, representing the position and the velocity of the ball on the surface of the playing table.) A more sophisticated model would include the effects of friction, angular momentum and aerodynamics, which would complicate the equation considerably but would not change its nature in any fundamental way; $\mathbf{x}$ and $\mathbf{v}$ would remain well-defined functions of t. But even with the most elaborate equation describing each individual ball's motion, we are still playing ghost-billiards, with balls that slide through each other and drift off the table.

To make the balls bounce, we have to encumber the algorithm with a deliberate check for

collisions after every time step. That is, whenever we move a ball to a new position, we have to calculate its distance from every other ball on the table. If the distance, measured center-to-center, is less than or equal to the sum of the two balls' radii, a collision has occurred, and we need to do something about it. (We also need to watch for collisions with the edges of the table and with other fixed obstacles, but these complications will be neglected here.)

Several aspects of this algorithm call for comment. First there is the matter of efficiency—how the amount of computation needed varies as a function of n, the number of balls on the table. The equation-of-motion part of the algorithm is said to have $O(n)$ efficiency, meaning that the effort is simply proportional to n. Doubling the number of balls doubles the number of velocity and position calculations. The collision-detection part, in contrast, is classified as an $O(n^2)$ algorithm: Because each ball might conceivably collide with any other ball, the effort rises as the square of n. Doubling the number of balls brings a fourfold increase in computational labor. (To be precise, the number of possible collisions is $(n^2 - n)/2$, but the "big-O" notation of computer science neglects everything but the leading term of such a polynomial.)

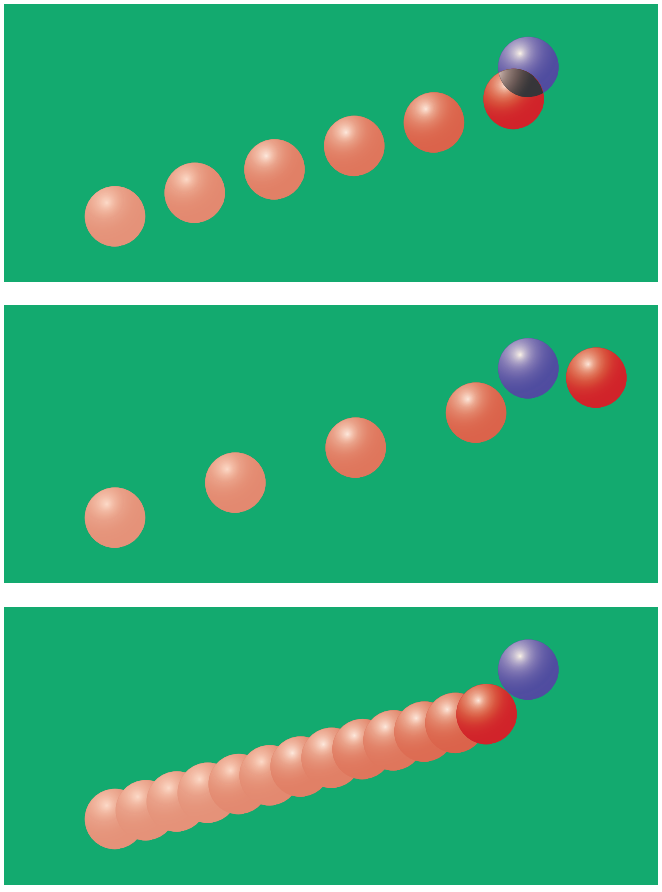

Figure 1. Failings of the naive algorithm include unphysical interpenetration *(top)* and missed collisions *(middle)*. The cure is to reduce Δt *(bottom)*, but this is computationally expensive.

The difference between an $O(n)$ algorithm and an $O(n^2)$ one might not matter much on a pool table (where n is never greater than 16), but it becomes critical when there are thousands or millions of interacting objects, as in a molecular-dynamics simulation. A number of programming tricks can lighten the computational load. For example, a program might partition the billiards table into several square blocks and look for collisions only between balls that lie within the same block or in adjacent blocks. But this gain in efficiency has a steep cost in program complexity; moreover, even with such a strategy, collision detection remains the most expensive phase of the algorithm.

And inefficiency isn't the only drawback of the naive algorithm; what bothers me more is that the method of detecting collisions seems artificial—perhaps one might even say *unnatural*. If the purpose of a simulation is to help us understand events in the real world, then the simulation should not only yield correct answers but should also mimic the natural process as closely as possible. The equation-of-motion part of the simulation passes this test: It's not too outrageous a metaphor to say that nature is evaluating the equation of motion of each ball at each instant to determine the ball's next position. But surely there is no process in nature that resembles the naive collision-detection algorithm. A ball on a billiards table does not have to stop and look around before it moves, checking the distance to every other ball on the table. Real billiard balls are blind; they don't sense each other's presence until they touch.

A final complaint about the naive algorithm is that it simply doesn't work! Generally, by the time a collision is detected, the balls have already penetrated each other's volume. (One thing real billiard balls never do is overlap and then back up in order to properly collide and rebound.) Worse, in some cases the algorithm may miss a collision altogether. If the balls are not touching at time t and they are again not touching at $t + \Delta t$, there is no way of knowing that they passed through each other at some moment between these times. The cure for all these ills is to reduce Δt, but that is computationally costly. You wind up watching the whole movie in extreme slow motion. Sophisticated algorithms adjust Δt dynamically, taking large time steps when the balls are far apart and smaller ones when they come closer, but as with the partitioning of space, there is a cost to be paid in program complexity. And the results are never exact except in the limit where Δt goes to zero.

Mathematics to the Rescue

A mathematician looking at the naive algorithm might well recoil in disgust. Why make such a laborious and plodding effort to find the approximate moment of collision, when there is a perfectly well-defined procedure for identify-

ing the exact place and time that two balls come together? It's just a matter of solving some simultaneous equations.

Suppose two pointlike particles are moving in straight lines on an infinite plane. Their paths are described by equations of the form $y = Ax + B$, where A and B are constants. The point at which the two paths cross (if indeed they do cross) can be found by any of the methods taught in elementary algebra for solving two simultaneous linear equations. This crossing point is not necessarily a point of collision, however, because the two particles may not arrive there at the same time. To detect collisions, we have to examine the particles' trajectories in *space-time*, where the parameter t is treated as if it were just another spatial dimension. Thus if the two trajectories share a point with identical x, y and t coordinates, the particles collide at that space-time point, or *event*.

This algebraic approach to the collision problem has some very pleasant properties. Because there is no need to step through time in units of Δt, detecting a distant collision is just as quick and easy as finding a nearby one; the algorithm wastes no time on the monotonous intervals between collision events. Furthermore, if all the calculations are done with sufficient precision, and if the coefficients A and B are rational numbers, then the coordinates of the collision can be determined exactly rather than approximately. Getting exact results is often important in geometric problems, since small errors tend to accumulate.

The algebraic method also has some troubling complications and potential weaknesses. In the first place, billiard balls are not point particles; they have a nonzero radius. Hence their trajectories do not cross at a single point, and devising a reliable procedure to find the first point of contact requires considerable care. Also, the roundness of the balls introduces points with irrational coordinates, which means that computations can no longer be numerically exact. Curved paths and accelerated motion (which make the equations of motion nonlinear) bring similar complications.

Then too, the algorithm must deal with more than two balls on the table. As with the naive algorithm, $O(n^2)$ possible collisions have to be considered. Suppose a simulation program examines the balls in numerical sequence when predicting their future collisions. It finds that ball ❶ collides with ❷ at time t, whereas balls ❸ and ❹ collide at a later moment, $t+\alpha$. Can the program now place these two predicted collisions on the schedule of events to be simulated? No, because ❶ and ❷, on the rebound after time t, might interfere with the motion of ❸ and ❹ before $t+\alpha$. Indeed, some other ball, whose path has not yet been computed, might come along and disturb ❶ or ❷ before time t. To avoid errors of this kind, the program must first catalogue all the potential collisions, then sort them in chronological order, and simulate only the earliest predicted event. Once the new trajectories are established after this event, the complete $O(n^2)$ collision analysis has to be repeated.

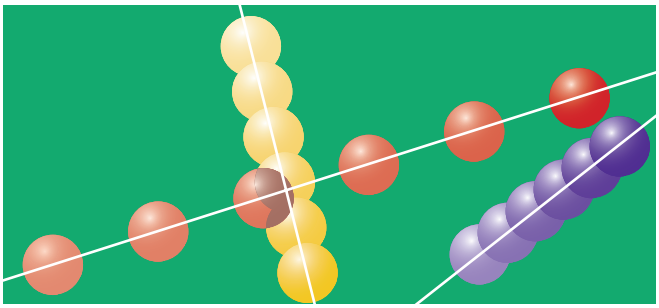

Figure 2. An algebraic method solves pairs of simultaneous equations to identify collision events. Only the earliest collision is retained (here between a yellow ball and a red ball).

Even with this handicap, the algebraic method can be quite efficient, and it yields results of high accuracy. But I have a hard time seeing in it any clue to how nature plays billiards. Are we to imagine some physical process that looks into the future, tabulates all possible collisions, sorts them into their proper sequence, and finally enacts the earliest predicted event? There is some question whether such a method of analysis ought to be called a simulation at all. It does not evolve over time to reveal an outcome, as the real world seems to do; instead it jumps ahead to discover a solution and then returns to the present to construct the sequence of states leading to that solution. Some people might call that cheating.

Physics to the Rescue

A physicist looking at the billiards problem might suggest a very different approach. From a physicist's point of view, a suspect element of the entire simulation is the supposed "law of nature" declaring that two solid bodies cannot occupy the same place at the same time. This notion is not in fact a fundamental law but instead emerges as a consequence of deeper principles. Solid bodies do not overlap or interpenetrate for a reason—namely because short-range repulsive forces keep atoms and molecules apart. A physics-based billiards simulation would focus on these forces.

To the casual observer, two billiard balls rolling across a table are utterly oblivious of each other's presence until they make direct contact, at which point they instantly change direction. But the collision and rebound cannot be truly instantaneous, for that would imply infinite acceleration. For an alternative model of how billiard balls bounce, consider two electrons on a collision course. Each electron is surrounded by an electric field, which the other electron detects and responds to even at a distance. As the two particles approach, the repulsive forces generated by these fields grow stronger, and so the electrons are deflected before they ever touch.

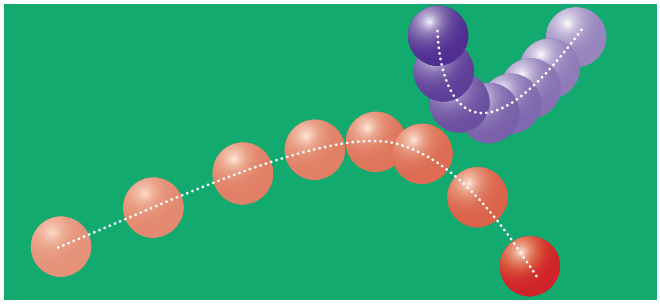

Figure 3. Physicist's billiard balls are surrounded by interacting fields, so that they smoothly repel one another without touching.

A simulated game of electron billiards is just an approximate solution of the classical n-body problem. The simulation works something like the naive algorithm for billiards, in that snapshots of the system's state are calculated at intervals of Δt, but there is no need for a special collision-detection procedure. Instead, the forces acting on each particle are summed up at each time step; from the forces the acceleration can be calculated, and the acceleration in turn yields a new velocity and position. Strictly speaking, no particles ever collide; as they get closer, the repulsive forces between them increase sharply, and so they smoothly and veer away.

The physicist's algorithm shares certain drawbacks of the naive algorithm. In particular, choosing Δt wisely is just as crucial if you want to get sensible results in a reasonable amount of time. The interval between snapshots has to be short enough that the fields and forces do not change drastically during a single time step; otherwise two electrons might stray closer than they would in the real world, generating implausibly large forces and velocities. Also, the n-body method is another $O(n^2)$. Nevertheless, the algorithm does seem somewhat more realistic as a model of how particles interact in nature. Just as evaluating an equation of motion seems like a fairly "natural" computation, summing up the forces acting on a body corresponds closely to processes that appear to go on in the real world.

The physicist's algorithm works well for simulating the motions of electrons or atoms. It has also proved effective in astronomical contexts, where the "particles" are stars or galaxies, and the dominant force (gravitation) is attractive rather than repulsive. But what about billiard balls? In principle their interactions can be described in the same way. Near the surface of a billiard ball, an electrostatic field repels other matter; this field is the underlying reason that colliding balls bounce. The trouble is, the field has an extremely short range. At macroscopic distances the field is effectively zero, and it remains negligibly small until you approach to within a few atomic diameters of the ball's surface; then it rises steeply to become a powerful repulsive force at the surface itself. With a force that has such an abrupt onset, all the advantages of smoothness and continuity are lost.

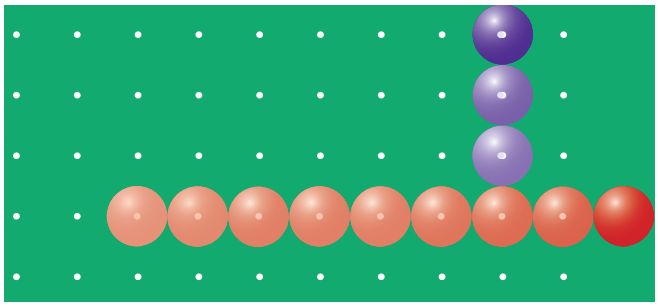

Figure 4. Lattice model confines the balls to a set of discrete points, making collision-detection easy. But the real world is not a lattice.

The Lattice World

Many of the troubles that beset computer simulations stem from the need to chop up the continuum of space-time into byte-size pieces. Some of the algorithms mentioned above address this issue directly. The mathematician's algorithm effectively restores the continuity of time, by identifying the exact moment of a collision. The physicist's algorithm imposes a different kind of continuity by turning discrete collision events into gradual interactions. Curiously, the opposite strategy can also help: Insisting on the discreteness of both space and time creates a world where billiards is a good deal simpler to simulate. The idea is to play billiards on a lattice.

In the simplest case the lattice is a grid of evenly spaced lines parallel to the x and y axes. Billiard balls can occupy only the lattice points where grid lines intersect; intermediate positions are nonexistent. Time is quantized in a similar way, so that the state of the system evolves in discrete jumps. A ball can move only by hopping from one lattice site to another.

In lattice billiards, two balls collide if they occupy adjacent lattice sites at the same time. This rule makes collision detection remarkably simple, reliable and efficient. To find out if a ball is colliding with any others, you merely have to look at the four neighboring sites to see if any of them are occupied. Note that this is an $O(n)$ process, since the number of operations needed is directly proportional to n. Moreover, if Δt is kept small enough that no ball can move more than one lattice spacing in a time step, there is no hazard of ever missing a collision or having two objects improperly overlap.

A particularly elegant way of writing a program for lattice billiards begins with the abstract computational model called a cellular automaton. Here the universe is not only divided into discrete lattice sites, or cells, but in addition each cell is equipped with its own internal computer to calculate the cell's future states. Thus where other schemes turn billiard balls into intelligent agents, computing their own trajectories, the cellular-automaton model makes the billiard *table* the seat of intelligence. The individ-

ual cells need little computational power; they merely look at their own present state and at the states of a few near neighbors to decide on a next state. All the cells apply the same rule in making this decision; computers throughout the lattice run the same program in lockstep.

In the case of a billiards simulation, the most conspicuous aspect of a cell's state is whether or not it is occupied by a billiard ball. A cell's internal instructions for computing its own next state might go something like this: If you are currently empty but the cell on your left is occupied, then in the next time step be occupied; if you are currently occupied and the cell to your left is empty, then in the next state be empty. Applying this rule consistently in all the cells allows balls to roll across the array from left to right, at a speed of one lattice site per time step. A complete rule that provides for other directions of motion and for collisions is more complicated, but the basic principles are the same.

Not only can this primitive computer simulate billiards; what's more remarkable is that the simulated game of billiards can in turn simulate any computer! It turns out that billiard balls bouncing around a tabletop can be interpreted as implementing the AND, OR and NOT functions of Boolean logic. A place on the table where two trajectories cross—where a collision will take place if two balls are traveling those paths at the same time—represents a logic gate. The gate has a logic value of TRUE if the two balls collide and is FALSE otherwise. By properly arranging many such gates one can build a universal computer—a machine that can emulate any other digital computer. Thus a PC might emulate a cellular automaton emulating a billiard-ball computer emulating a PC.

If only the real world were built on a lattice! Computing would be so much easier, simulation so much more direct. The integers would supply enough numbers for all computational purposes; there would be no need for the infinitesimal intricacies of the real number line. A simulation could set up a one-to-one mapping between addresses in the computer's memory and places in space, between ticks of the computer clock and the progression of time. But alas the real world does not have a lattice structure, or at least none is visible at any scale yet probed by experimental physics.

Lattice models have proved useful anyway. For instance, there have been interesting and fruitful explorations of knot theory on a lattice, and studies of how proteins and other polymer chains would fold if they were confined to a lattice. Perhaps the most celebrated examples are lattice studies of the force that binds together quarks inside subnuclear particles. The quark models attempt to recover the continuum by looking at what happens as the lattice spacing shrinks toward zero. Unfortunately, that strategy may not always work, because some properties of the lattice survive even at the smallest spacings. For example, a rectilinear lattice has a discrete fourfold symmetry that persists even at the finest spacing. Also, in continuous space and time it's easy to prove that a simultaneous collision of three billiard balls has a probability of zero, but on a lattice such three-way collisions are commonplace. As a result the statistics of collisions will be different in the lattice world.

Making Do

The conceptual thickets surrounding simulation algorithms have not stopped anyone from using those algorithms—nor should they. All four of the basic approaches described here are in daily service, and they work. Limitations and caveats have to be respected—but then again you are well advised to read the instructions and always wear eye protection when operating *any* power tool.

If simulation techniques give the right answers, why worry about whether or not they are made from all-natural ingredients? Why should we confine ourselves to the same limited stock of algorithms that nature draws on? Sometimes the simulator is surely licensed to innovate. If you are building a video game, any shortcut or algorithmic trickery that yields the desired illusion ought to be acceptable practice. If the motions of the objects on the screen look right, then they are right. There is no higher standard of judgment.

But in other contexts a little caution would be prudent. Simulation is now celebrated as a "third way" of doing science, an adjunct to theory and experiment. Accordingly, some simulations mimic processes where no one knows the right answer, and the simulation is supposed to reveal it. The imitation of nature is the only warrant of truth such a simulation can offer. Without it, we may well be watching ghostly billiard balls wander off the table, and never know there's anything wrong.

Bibliography

Greenspan, Donald. 1982. Deterministic computer physics. *International Journal of Theoretical Physics* 21:505–523.

Hoffmann, Christoph M. 1989. The problems of accuracy and robustness in geometric computation. *IEEE Computer*, 23 (March):31–41.

Hubbard, Philip M. 1995. Collision detection for interactive graphics applications. *IEEE Transactions on Visualization and Computer Graphics* 1:218–230.

Lubachevsky, Boris D. 1991. How to simulate billiards and similar systems. *Journal of Computational Physics* 94:255–283.

Margolus, Norman. 1984. Physics-like Models of Computation. *Physica D* 10:81–95.

Minsky, Marvin. 1982. Cellular vacuum. *International Journal of Theoretical Physics* 21:537–551.

Reif, John H., and Stephen R. Tate. 1993. The complexity of *N*-body simulation. *Proceedings of the 20th International Conference on Automata, Languages and Programming*, July 1993.

Toffoli, Tommaso. 1982. Physics and computation. *International Journal of Theoretical Physics* 21:165–175.

Uchiki, Tetsuya, Toshiaka Ohashi and Mario Tokoro. 1983. Collision detection in motion simulation. *Computers and Graphics* 7:285–293.