Realistic Billiards Simulation with Variable Time-Step

Luke Anderson
Rensselaer Polytechnic Institute, Troy, NY

anderr4@rpi.edu

Abstract

This paper presents a method for creating a robust system for modeling physical interactions between objects. The system avoids problems such as objects escaping or passing through each other due to undetected collisions caused by excessive velocities or insufficient time resolution.

1 Introduction

In the example presented here, the method is applied to pocket billiards (“pool”). We will describe all of the parts required to create the pool simulation. We will also demonstrate a method for increasing the visual believability of the simulation.

2 The System

The pool balls in the simulation are represented by a collection of spheres, and the pool table is represented by a rectangle. In order to make the pool simulation come to life, we will use a model that is well-suited to detecting physical interactions between the table and the balls.

2.1 Pool Balls

Each pool ball has a vector for its position and a vector for its velocity. As specified by the World Pool-Billiard Association, the balls have a radius of 2.25 inches and weigh 5.5 oz. [5]

2.2 Pool Table

The pool table is represented by four distinct points in 3D space, one for each corner, effectively making up a rectangle. The lines formed between two points are used as the edges of the table. The table is centered at the origin, and the surface lies in the plane perpendicular to the Y-axis. It is an 8-foot table, measuring 92 inches by 46 inches as specified by the WPA. [5]

2.3 Pool Cue

The pool cue is represented by an angle and a magnitude, representing the direction of the cue stick with respect to the cue ball, and the intensity of the strike. In the simulation, this vector is represented as a red line.

3 Interactions

There are four distinct interactions we are concerned with in this simulation: collisions between ball and table, collisions between ball and ball, frictional forces between ball and table, and the initial impact between the cue and cue ball.

3.1 Ball/Wall Collisions

Since each sphere has a position and velocity vector, we can think of it as a parametric ray with an origin and direction. The same can be done for the boundaries of the table: given two corners, we can create a ray representing the edge. Thus, the ball/wall collision simplifies down to a simple collision between two rays. The parametric equations for the rays, where *O* is the origin, *D* is the direction, and *T* is the parameter, are:

$$R_1(T_1) = O_1 + D_1 \cdot T_1$$
$$R_2(T_2) = O_2 + D_2 \cdot T_2$$

At the intersection of the two rays, the X and Z dimensional components must be equal:

$$O_{1,x} + T_1 \cdot D_{1,x} = O_{2,x} + T_2 \cdot D_{2,x}$$
$$O_{1,z} + T_1 \cdot D_{1,x} = O_{2,z} + T_2 \cdot D_{2,z}$$

Solving for the *T* values gives:

$$T_1 = \frac{D_{2,x} \cdot (O_{2,z} - O_{1,z}) + D_{2,z} \cdot (O_{1,x} - O_{2,x})}{D_{1,z} \cdot D_{2,x} - D_{1,x} \cdot D_{2,z}}$$
$$T_2 = \frac{D_{1,x} \cdot (O_{2,z} - O_{1,z}) + D_{1,z} \cdot (O_{1,x} - O_{2,x})}{D_{1,z} \cdot D_{2,x} - D_{1,x} \cdot D_{2,z}}$$

We only want to consider positive *T* values, since time is only moving forward. So, if T_1 and T_2 are both positive, the ball and the wall are going to collide, and we have the exact point in time of their intersection.

Once a ball/wall collision has occurred, the ball's velocity must be reflected about the normal of the table edge. The normal can be found by taking the cross product of the edge vector with the up vector $\langle 0,1,0 \rangle$. The reflection can be found using the following equation, where *v* is the velocity and *n* is the normal.

$$R = v - 2n \cdot (v \cdot n)$$

3.2 Ball/Ball Collisions

For determining when two spheres will collide, the simulation uses an implementation of the algorithm described in "Simply Bounding-Sphere Collision Detection" [3]. This algorithm solves the quadratic equation representing the intersection of spheres:

$$dv^2 \cdot T^2 + 2 \cdot (dp + dv) \cdot T + dp^2 - (2r)^2 = 0$$

First, we calculate the difference in velocities and positions between the two spheres:

$$dv = v_2 - v_1$$
$$dp = p_2 - p_1$$

The dot product of *dp* with itself represents the square of its length. If this value is less than the square of the sphere's diameter, then the spheres have already intersected:

$$pp = dp \cdot dp - (2r)^2 \overset{?}{\leq} 0$$

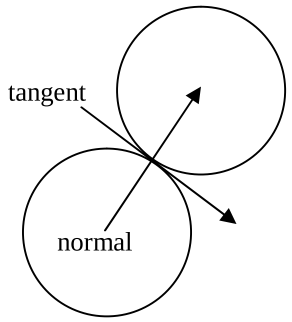
Next, we calculate the dot product of *dv* and *dp*. If this result is greater than zero, then the spheres are moving away from each other:

$$pv = dv \cdot dp \overset{?}{\geq} 0$$

Finally, we compute the dot product of *dv* with itself and the determinant of the quadratic equation. If the determinant is positive, then the spheres intersect, and we can solve for *T*:

$$vv = dv \cdot dv$$
$$d = (pv \cdot pv) - (pp \cdot vv) \overset{?}{\geq} 0$$
$$T = \frac{-pv - \sqrt{d}}{vv}$$

This gives us the exact amount of time that will pass until the spheres collide. Once we know that a ball/ball collision has occurred, we must update each ball's velocity vector. As described in [2], each sphere has a normal and tangent vector.



We calculate the normal component by subtracting the positions of the two spheres:

$$n = p_2 - p_1$$

The tangential component is simply:

$$t = \langle -n_z, n_y, n_x \rangle$$

Since all of the spheres have equal mass, we only need to consider their initial velocities. The equations for the resulting velocities after the collisions are:

$$v_1' = n \cdot (n \cdot v_2) + t \cdot (t \cdot v_1)$$
$$v_2' = n \cdot (n \cdot v_1) + t \cdot (t \cdot v_2)$$

3.3 Physics

In modeling the physics in our system, we will use two of the equations of motion.

The first equation is responsible for moving the position of the balls based on their velocity, taking into account the oppositional force of friction:

$$p = p_0 + vt + \frac{1}{2}at^2$$

The second equation is responsible for reducing the balls' velocities due to the effects of friction:

$$v = v_0 + at$$

In our simulation, *a* will always be a negative value, because nothing should be adding energy to the system except for the initial strike of the cue ball. The energy from this strike is added by simply setting the velocity component of the cue ball to the appropriate value.

3.4 Friction

There are two different types of frictional forces that occur in our simulation: static friction and kinetic friction. Static friction occurs when a ball is rolling across the table and is a very small value, while kinetic friction occurs when a ball is sliding across the table and is a slightly larger value. When a ball's kinetic energy exceeds that required to overcome the static frictional force, it is acted on by kinetic friction as it slides across the table. However, once the ball's speed drops below the threshold of the kinetic frictional force, it begins to roll and is acted on only by static friction. [4]

4 Time

Our simulation has to progress through time. The balls have specific velocities and positions, which change over time according to the physical laws of motion. This raises the question: how do we represent the passage of time in our system?

4.1 The Time-Step Problem

Physics simulations typically use a discrete time-step value, which represents the amount of time passed since the last physics update. For example, an object moving at 100in/s is traveling one inch every 0.01 seconds, so using a time-step of 0.01 would cause the object to be moved one inch on every iteration.

While this additive time-step method may be useful for simple motions, it presents problems when trying to create the interactions required for our realistic simulation. Most critically, the quantification of time requires collision detection to be preemptive (checking where the object will be after the next timestep, and adjusting for a collision) or corrective (checking for collisions that have just occurred, and fixing the objects involved). Both detection methods are difficult to implement successfully and are likely to have problems detecting collisions with very fast-moving or very thin objects.

A discrete time-step also raises problems of granularity: how small should the time-step be, in order to efficiently balance rendering speed and physical accuracy? A small time-step will be more accurate, but it will also dramatically increase the processing required. Likewise, a big time-step will be less accurate but much faster to calculate.

Finally, when we consider that the animation may not be rendered at a constant frame rate, a third problem arises. How do we maintain time at a consistent speed? In real life, the speed of time does not change. However, the frame rate in our simulation may change, and if the time-step is incremented a fixed amount after drawing each frame, the speed of the animation could rapidly fluctuate.

4.2 The Time-Step Solution

For our real-time simulation, pool balls will have a very wide range of velocities (the fast-moving cue ball could hit another non-moving ball). We need to guarantee that they can never pass through each other. Therefore, the problems associated with an additive time-step system are unacceptable and we have to find another way to model time in our system.

First, we want the simulation to run in real-time. In order to do this, we determine how much actual time has passed between each rendered frame of the animation using the high-resolution timer of the CPU. This gives us an accurate representation of how much time has actually passed, and we use this as the ΔT value. This value represents how far we need to progress the simulation.

At this step, we iterate through all of the balls in the system and determine the time until the next ball/wall or ball/ball collision using the algorithms described in sections 3.1 and 3.2. Then, instead of incrementing by a fixed timestep, we progress the simulation up until the time of the next collision. At this point, we run the collision algorithm which sets the new directions of the objects involved. Finally, we recursively call the progress function with any remaining time, until all of the original ΔT value has been added. Here is pseudocode of the algorithm:

```
Function Progress(float T)

 For Each Ball i
  If Hit(Ball[i], Table, $T_n$) <= T
    Hits.Add(Table)
  End If
  For Each Ball j
   If Hit(Ball[i], Ball[j], $T_n$) <= T
    Hits.Add(Ball[j])
   End If
  Loop
 Loop

 For Each Ball i
  Ball[i].Move($T_n$)
 Loop

 For Each Table Hit
  If HitPocket(Ball)
   Remove(Ball)
  Else
   Reflect(Ball)
  End If
 Loop

 For Each Ball Hit
  Collide(Ball1, Ball2)
 Loop

 Progress(T - $T_n$)

End Function
```

This method prevents any of the typical collision detection problems from occurring. Instead of incrementing time and trying to figure out when collisions will occur or have occurred, we increment time exactly until the next collision and adjust for it. Balls cannot escape the table, nor can they pass through each other.

This also makes the frame rate completely independent of the system, since the time-step can be any value. A large value, like 0.25 seconds, will cause the same result as many small values, like 0.01 seconds. In this way, the simulation is deterministic.

5 Randomization

Even though the results of the simulation are accurate, they may appear unrealistic. This is because, given the same starting conditions, the same input to the system will result in exactly the same result each time—it is *too* accurate. To make the actions of the simulation more believable, we add random perturbations to the normals in the collision correction algorithms as described in [1]. This random variation can be accounted for in reality (for example, imperfections in the table or balls), so the simulation still appears plausible but delivers different results each time. This puts the finishing touch on our realistic billiards simulation.

6 Screenshots

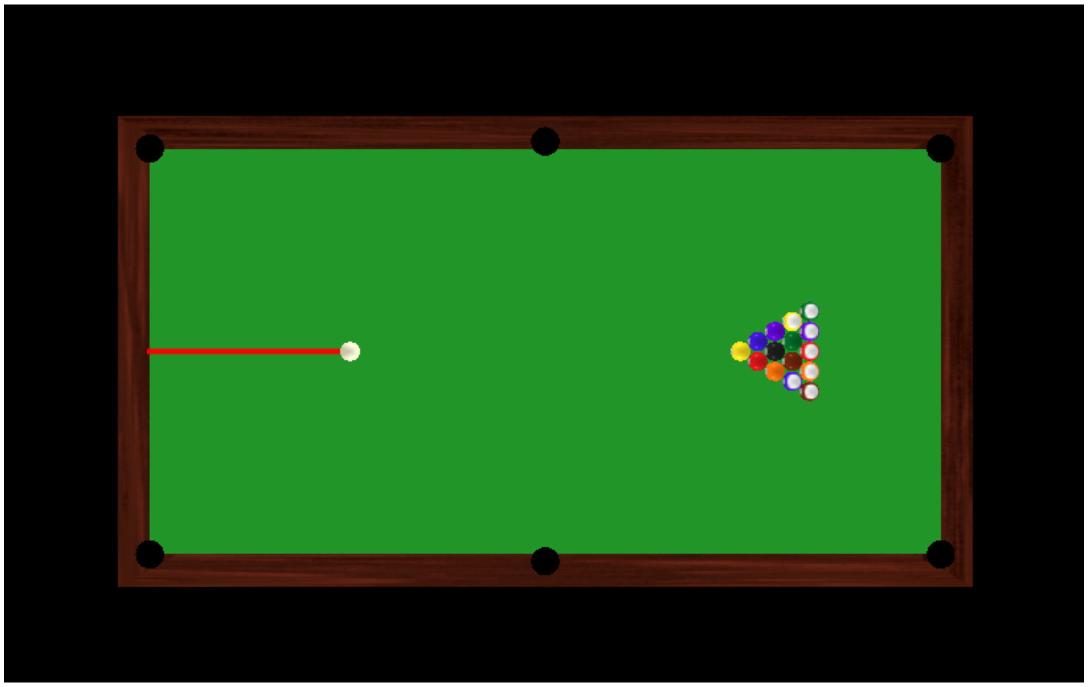


Figure 1: Default layout

The red vector represents the direction and intensity of the cue stick. The cue ball is placed at the one-quarter mark of the pool table in the horizontal direction, and centered in the vertical direction.

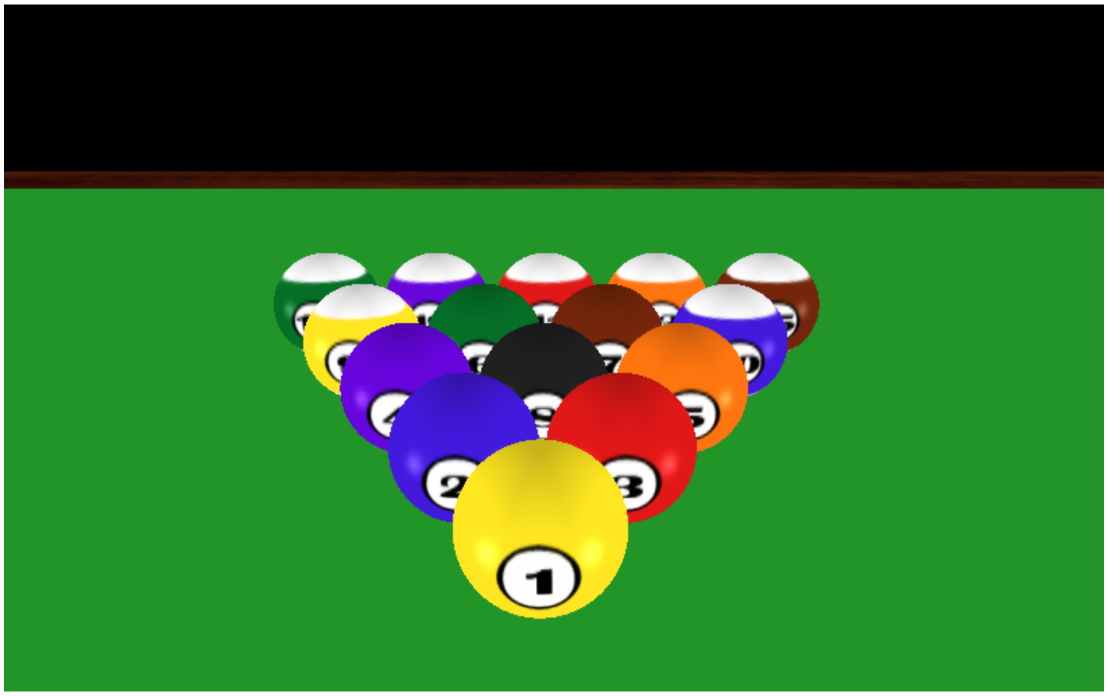


Figure 2: Rack

The balls are racked 1/10” apart. Without randomization, hitting it the same way will cause it to break the same way every time.

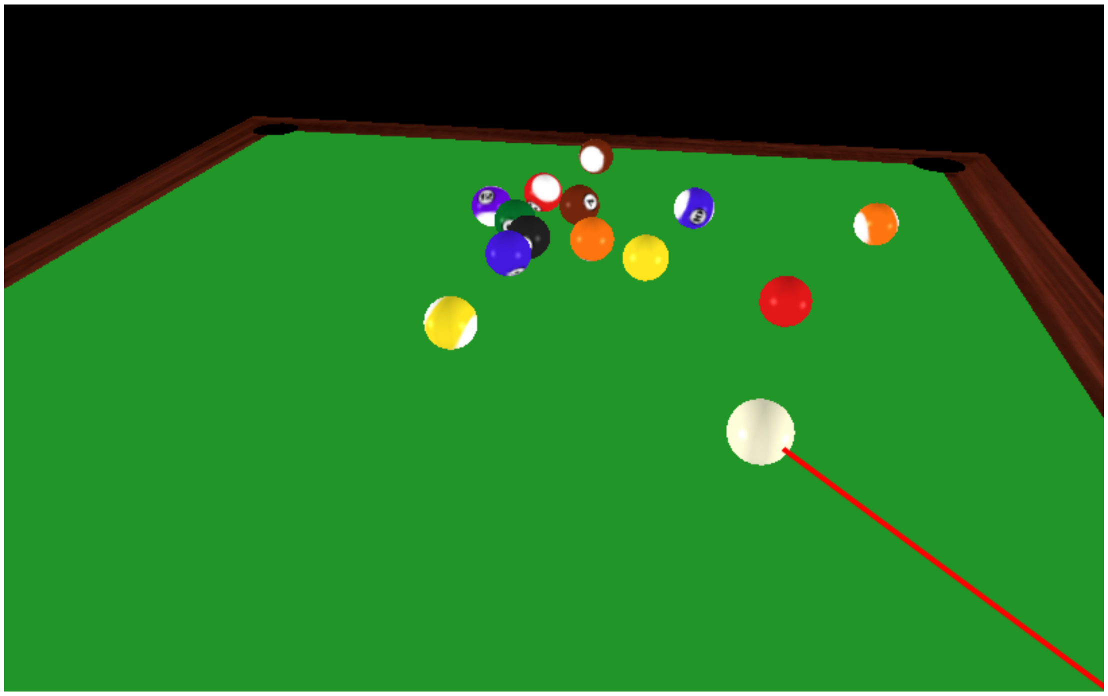


Figure 3: Break
The balls scatter realistically after being hit by the cue ball.

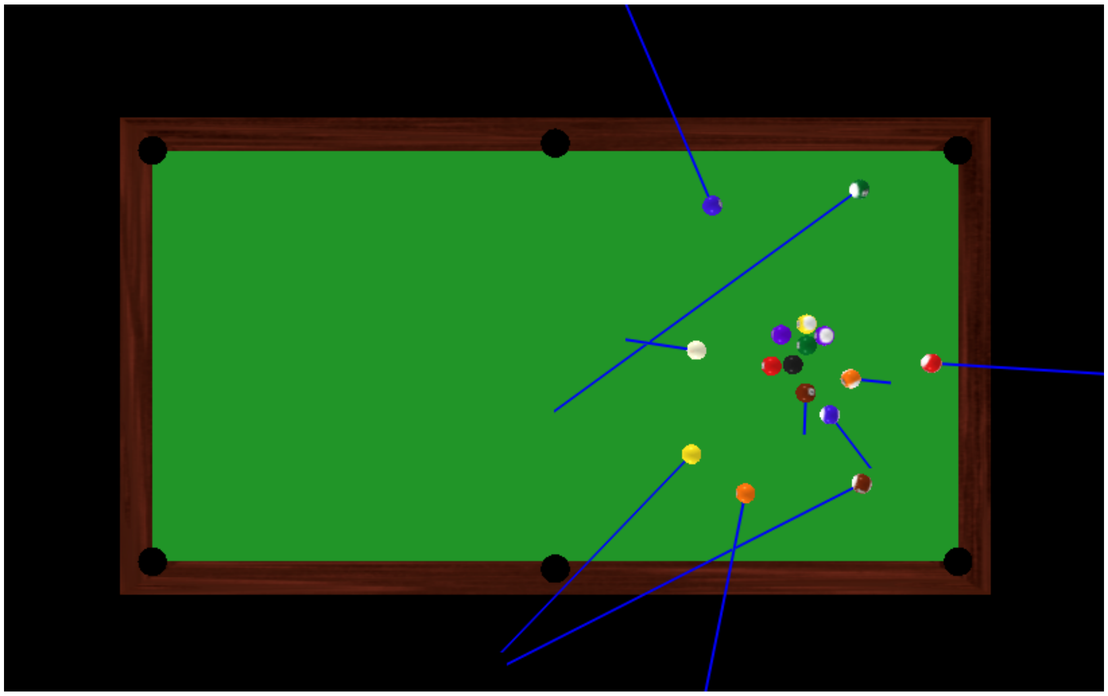


Figure 4: Velocity Vectors
The blue vectors show the direction and speed of each ball in the simulation.

7 References

[1]. Barzel, Ronen, John F. Hughes, and Daniel N. Wood. Plausible Motion Simulation for Computer Graphics Animation. 1996. University of Washington. <http://www.ronenbarzel.org/papers/plausible/plausible.pdf>.

[2]. Berchek, Chad. “2-Dimensional Elastic Collisions Without Trigonometry.” 2006. <http://www.geocities.com/vobarian/2dcollisions/2dcollisions.pdf>.

[3]. Dopertchouk, Oleg. “Simple Bounding-Sphere Collision Detection.” 12 Nov. 2000. <http://www.gamedev.net/reference/articles/article1234.asp>.

[4]. Lee, Soo. “Physics of Pool: Drag Force.” 2000. Oracle ThinkQuest Library. <http://library.thinkquest.org/C006300/data/seven3_1.htm>.

[5]. “WPA Tournament Table & Equipment Specifications.” World Pool-Billiard Association. Nov. 2001. <http://www.wpa-pool.com/index.asp?content=rules_spec