Billiards Sim

Team #11

Ira Cooper (ira@mit.edu)

Miro Jurisic (meeroh@mit.edu)

Krishnan Sriram (sriram@mit.edu)

Abstract

Billiards Sim is a system that allows you to go from an initial scene description of a billiards game to an animation with sound.

Billiards Sim is a based on a time-step based simulation. At every time step a basic set of computations is performed to see what actions have to take place. Billiards Sim then takes all of the actions and runs a physics model across them, which produces the state for the next step. While all this is running Billiards Sim deals with the cinematography in addition.

The cinematography adds an additional dimension to the problems in this project. The major problem added was the question of how the camera should act as balls collide with each other and impact the rails. This all ends up being solved in an automated and reasonably simple manner. In addition, from all the data the system generates it can create a sound track that is synchronized with the rest of the action in the simulation.

Finally, based on the simulation and the cinematography, the rendering layer emits frames and a sound track. The frames are fed to a third party renderer, and assembled along with the sound track into the final animation.

Introduction

Billiards Sim was conceived because the team was interested in creating a full simulation, in addition to being interested in the sport of billiards. First, the team members find an actual joy in playing the game of pool, and in the beauty of its physics. In addition, there is the challenge of actually producing a system that goes from beginning to end - from an initial simulation to a final video.

Billiards Sim is a time-step based simulation, that creates an animation of a game of billiards. It is assembled using many different techniques of simulation, graphics, cinematography, and animation.

The simulation portion of Billiards Sim is based on a standard time stepped architecture. This allows for the easy forward propagation of state, and makes that propagation easy to segment, allowing multiple people to work on the project.

To create the simulation, Billiards Sim needed objects, a way to render the graphics, models of all the objects in the scene and finally a way to animate them. The team chose the freely available renderer

POV-Ray to render the graphics. The objects in the scene were all hand modeled by members of the team.

Lastly, and most importantly, Billiards Sim has cinematography and animation. These components are what allow the system to actually transform the graphics components into the final animations. This part of the system deals with problems such as moving balls to the right location, getting the cue stick in and out of the scene, and determining the correct location for the camera. From all of this the system assembles frame descriptions. These are rendered by the rendering components of the system and put together to produce the final animation.

Goals

Primary Goal:

To produce a believable computer generated movie of billiards being played.

Primary Supporting Goals

- Object models:
  - Table model
  - Ball models
- Physical simulation:
  - Infrastructure for time-stepped based simulation
  - Ball motion without spin
  - Friction
  - Restitution
- Cinematography:
  - Scriptable camera movement relative to the balls and table

Supporting Goals

- Object Models:
  - Cue stick
- Physical simulation:
  - Realistic trajectories of balls into pockets
  - Balls spinning
  - Transfer and activity of angular momentum in collisions
- Cinematography:
  - Music sound track
  - Scriptable cue stick
  - Scriptable camera

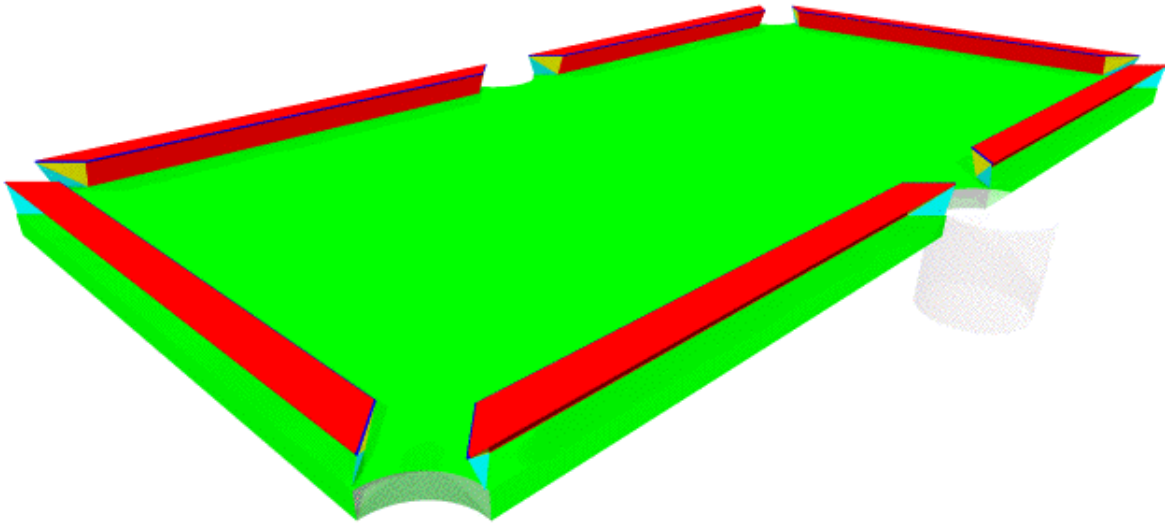
Achievements

Complete Set of Physical Models

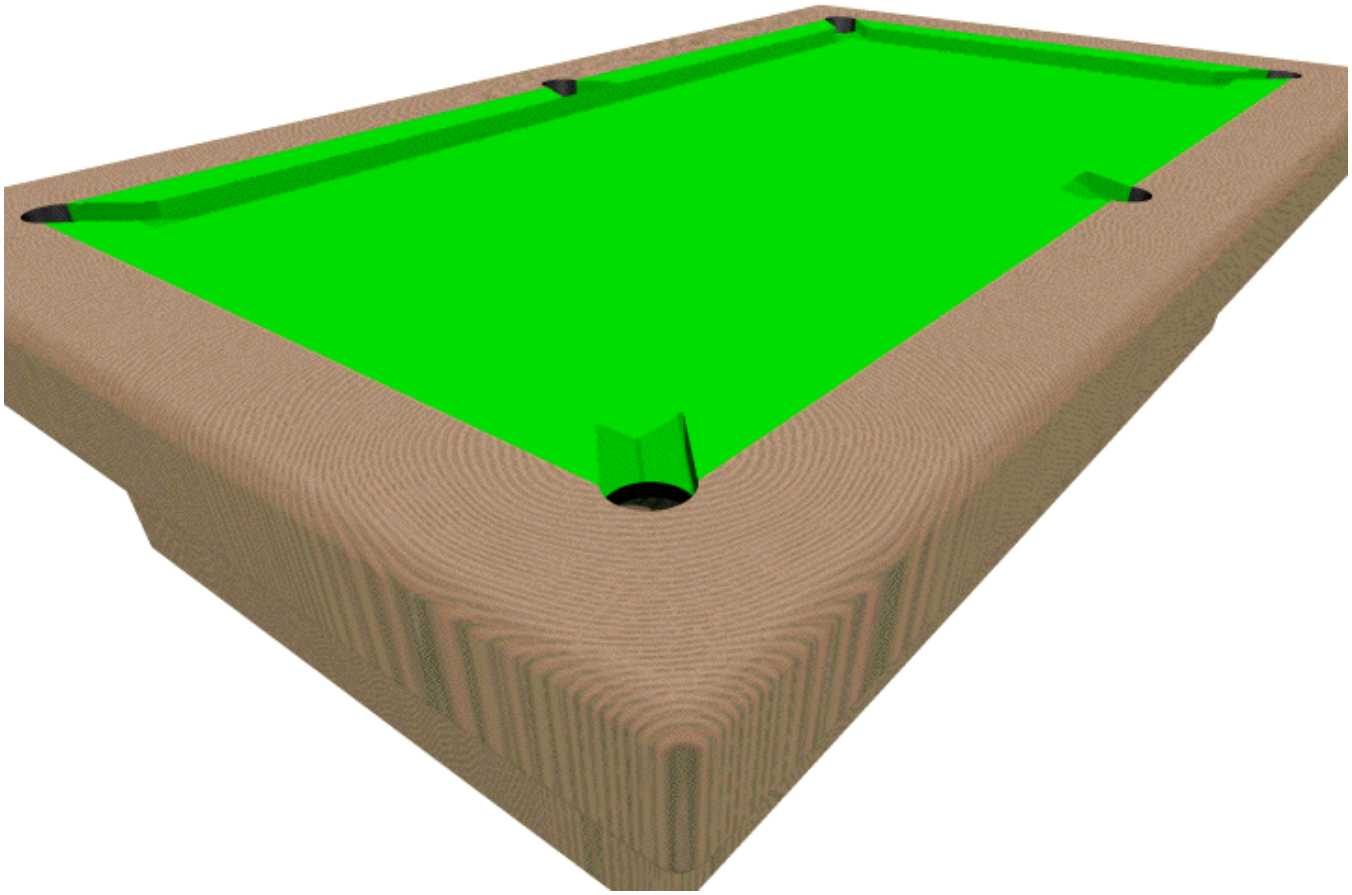
Pool Table

The pool table was the most difficult model due to its geometric complexity, and in fact took two tries to do correctly.

The first attempt at the table failed due to the fact that seams could be seen in the bumpers' interaction with the table surface. This is a common problem in using our renderer of choice, POV-Ray, if the model is created without using Constructive Solid Geometry (CSG).



The second attempt at the table used CSG, and was done without using a modeling package. Due to the creator's inexperience with CSG, producing the table model was a lengthy process.

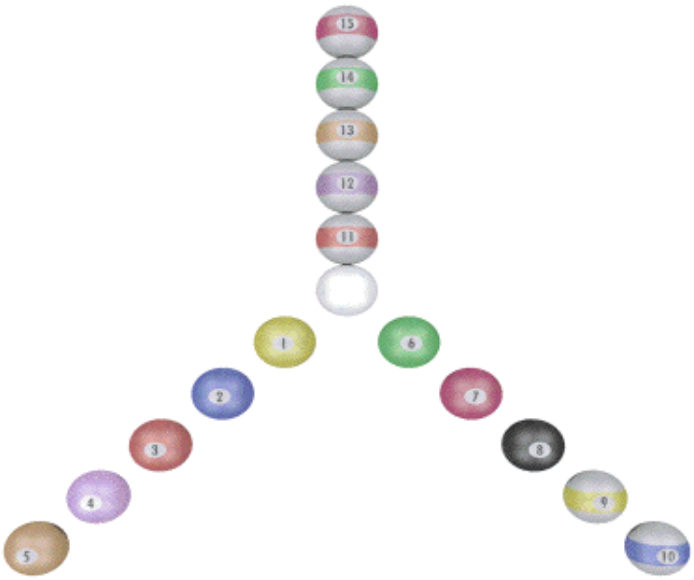


Pool Cue

The pool cue is modeled using cones and cylinders. The tip of the cue is actually a cylinder; in most of the scenes the difference from a real lens shape is minimal.

Pool Balls

Texture was applied to the pool ball to attain the white spot and numbers.



Physical Simulation

Overview

Problems faced

The team spent too much time trying to figure out multi-ball impacts before realizing that this is in fact a difficult problem, especially considering angular momentum and non-ideal surfaces. In the end, despite some progress, this was dropped due to implementation complexity and lack of time.

It is possible that some of the time taken above could have been lessened through a better allocation of staff; this was learned after the production of the third set of collision equations.

Solutions

The physics for Billiards Sim was simplified by resolving all the collisions in a pairwise manner instead of considering all the balls as a system. This also allowed for more complexity in the physics of the simulation in terms of angular momentum and restitution.

Actual implementation

System Architecture

The system architecture is broken down into several parts. The first of these is the simulation’s architecture itself.

The simulation's architecture is a simple object oriented architecture. The primary object of the simulation is the world object. The world object manages the balls, the rails, and the system clock for the simulation. The world time-steps all of the objects forward in time and detects and lists all collisions between objects.

The other components of the simulation architecture are classes subclassed from the virtual base class Impactor. These include the rails, balls and pockets. These classes have a method which allows them to detect if a ball has impacted them. It is only necessary to detect impacts by balls, because balls ore the only moving objects. There is a method in the world object which will then take all the collisions and set the balls to their new states. This allowed an incremental approach to the physics of the simulation, and independent work on collision detection, friction, and the physics of impact.

The world calls the rendering layer on the appropriate steps of the simulation. Here the rendering layer transforms the current state of the world into a description that our renderer, POV-Ray, would accept.

After the simulation is run and all the frames are completed, the operator just runs POV-Ray on each frame one at a time, or in parallel on multiple machines. The operator then takes all the resultant frames and collapses them into a QuickTime movie, which is the final product of the simulation.

Ball Trajectories in Pockets

The trajectory of a ball falling into a hole is modeled in a simple manner by Billiards Sim. As a ball in the pocket will not interact with other balls and will only spend a small amount of time on camera, there is no need for high accuracy in the physical model.

The actual physics model for the trajectories deals with two cases. The first case is where the ball's velocity is less than the square root of the radius times gravity, where the ball rolls over the edges; in the second case the velocity is greater, this is when it flies into the hole. Once the ball enters the pocket, the first case simplifies to the second case.

Additional collisions with the inside of the pocket are handled with a simple collision mechanism, so that the ball appears to bounce off the end of the pocket.

Once the ball hits a fixed depth below the table plane, it stops moving. The ball stays in the pocket forever, and its collision handling is disabled, so that the collision handling code will not be run if two balls fall into the same pocket. This implies that if someone were to look at the bottom of a pocket that more than one ball fell into, they would see several inter-penetrating balls. This will never be seen in the movie due to the choices that the team will make in cinematography.

Ball Motion without Spin

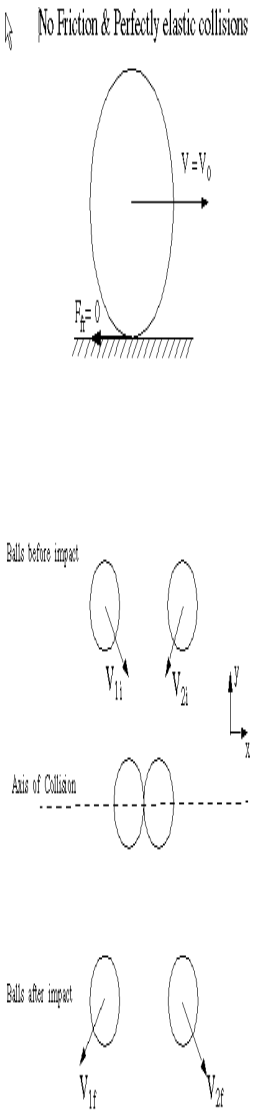
Ball motion without spin is implemented using the velocity of the ball to determine location and quaternions to deal with the rotation of the ball.

Ideal Ball Collisions

This model is the case of ideal Newtonian Mechanics. It was very simple to get up and running.

Algorithm to update the velocity and spin of a ball

```
Update ( object ball ) {
        Vector r = ( 0, 0, -ball.radius);
            V_rel = V_0 + ω × r ;
            Vector Vrel = ball.velocity + crossProduct( ball.omega , r );
            if( length(Vrel != 0) ) {
                findForceDirection ( ) {
                        Vector fDirection = - Vrel;
                        normalize ( fDirection );
                        }
                findForce( ) {
                        Vector force = mu_surface*9.81*fDirection;
                        }
                findTorque( ) {
                        Vector torque = crossProduct ( r, force );
                        }
                ball.velocity += force*delta_t;
                ball.omega += 5*torque*delta_t/(2 * ball.radius * ball.radius );
            else {
                Reduce ball.velocity and ball.omega using empirical models
                }
}
```



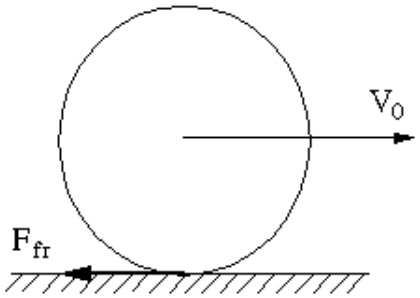
Friction and Restitution

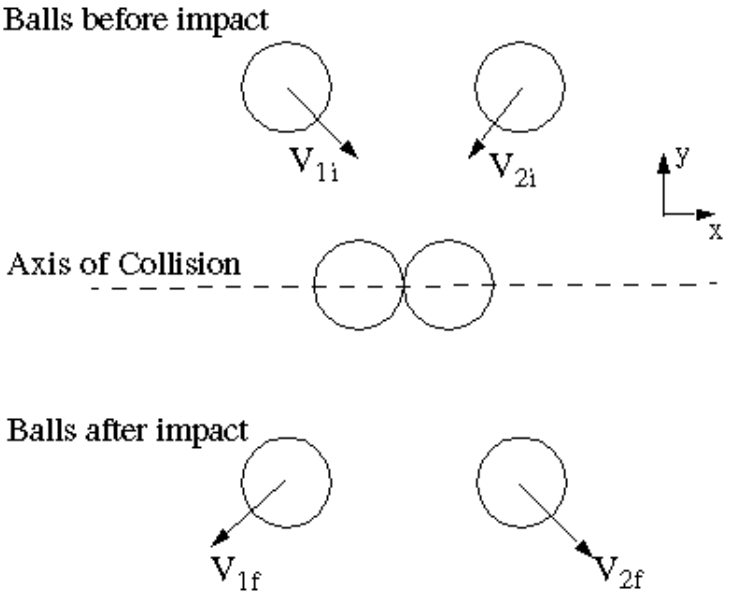
Friction was taken care of by the standard physics equations as shown below. This effectively slows balls down by a constant amount regardless of their weight. The constant amount is determined by a constant called table speed which is the reciprocal of the typical friction coefficient mu.

Restitution is handled by incorporating the restitution factor "e". This models all effects properly except for issues dealing with spin.

The equations used to implement this are the same as the ones listed for Ball Collisions with Spin Transfer, but there is no spin transfer and therefore there are no equations involving omega.

Motion with friction & inelastic collisions





Ball Collisions with Spin Transfer

When two balls with initial spin collide, there is linear momentum transfer from collision and angular momentum transfer from relative slipping of the objects at the point of contact. By implementing both of the above phenomena, a simulation as close to reality as possible is achieved. In addition, it was found that the simulation could detect when balls jump due to spin transfer, but this effect was discarded due to the complexity inherent in implementing it.

Collision algorithm for 2 ball collision

Collision (ball1, ball2) {

find Axis Of Collision ( ) ;

find RelativeVelocity with respect to IRF fixed to ball 2 ( ) ;

// the relative velocity values are $\mathbf{V}_0$ and ω_0

Transform Velocity from World Frame IntoCollisionFrame ( ) ;

find Gamma ( ) {

// γ is the angle made with Y axis by the contact force between the two balls.

$$\gamma = \arctan((-R\omega_{0y}) / (V_{0y} + R\omega_{0z}))\ ;$$

}

find V_ball2 ( ) { // e is restitution

$$V_{by} = (1+e)V_{0x}/(2)\ ;$$

$$V_{by} = V_{bx}\mu_{bb}\cos(\gamma)\ ;$$

$$V_{bz} = 0\ ;$$

}

find V_ball1 ( ) {

V_ball1 = $\mathbf{V}_0$ - V_ball2;

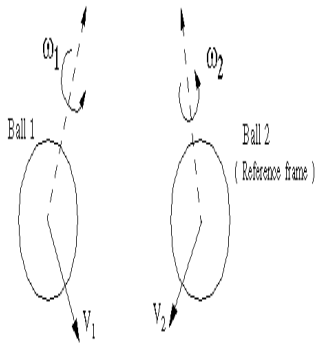
}

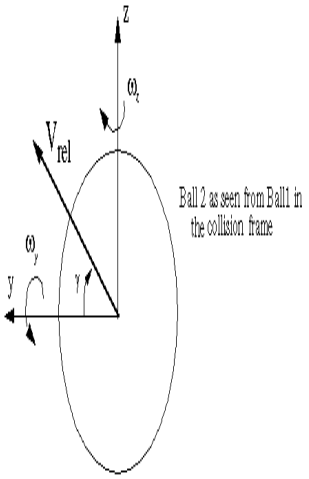
findOmega_ball2 ( ) {

$$\omega_{by} = -\frac{5\mu_{bb}V_{bx}\sin(\gamma)}{2R}\ ;$$

$$\omega_{bz} = \frac{5\mu_{bb}V_{bx}\cos(\gamma)}{2R}\ ;$$

Ball Collision with spin transfer





```
                    ω_bx = 0 ;
                }
            find Omega_ball 1 ( ) {
                omega_ball1 = omega_0 - omega_ball2;
                  }
            Transform from Collision Frame IntoWorld Frame ( ) ;
            Find Absolute velocity values by adding the V and ω values of
            the Refenence Frame fixed to ball 2 ( ) ;
}
```

Cinematography

Cinematography consists of three parts: scriptable camera movement, scriptable cue stick movement, and a sound track. Scriptable camera movement was added so that camera would be automatically generated for each frame, without hand editing. Scriptable cue stick movement adds realism to the scenes. The sound track also adds realism to the simulation, as it provides the sounds the viewer would expect to hear.

Scriptable Camera Movement

Scriptable camera movement relies on the idea of events. Any collision between a ball and a ball, a bumper, or the cue stick is considered an event. There are no events that relate to pockets, and therefore camera movement can not automatically synchronize with balls falling into pockets. In addition, there is a synthetic event at the beginning and the end of every scene.

The desired camera movement can be specified between any two events, including the start and end of the scene. The movement can be specified in one of three ways:

- Hovering: The camera is at a fixed location and looking at a fixed point between two events.
- Looking: The camera is at a fixed location and looking at a specified ball between two events.
- Following: The camera is at a location relative to a specified ball between two events, looking at the ball.

In the case when the camera is following a ball, the position of the camera is given by a vector which is interpreted relative to the velocity of the ball. This allows for more intuitive specification of camera movement, as <-1, 0, 0> always means "one unit behind the ball" regardless of the direction the ball is moving in.

Using only the specific camera movement would clearly result in jerkiness, as the location and direction of the camera would not have C1 continuity. To solve this problem, camera is interpolated between

every two consecutive events.

For every event, camera setup time is specified as two fractions between 0 and 1: pre-event setup time and post-event setup time. Pre-event setup time specifies how much before the event the camera starts interpolating, and post-event setup time specifies how long after the event the camera stops interpolating. For pre-event setup time, the times are relative to the preceding event; for post-event setup time, the times are relative to the following event.

For example, if events 1, 2, and 3 occur at t = 0, t = 6 and t = 10, respectively, and event 2 has pre-event setup time of 2/3 and post-event setup time of 3/4, then:

- between t = 0 and t = 2, camera uses the location and direction specified between events 1 and 2,
- at t = 2, camera starts interpolating around event 2 (because 2/3 of the time between events 1 and 2 are spent interpolating camera),
- at t = 6, camera is still interpolating, and event 2 occurs,
- at t = 9, camera interpolation ends (because at t = 9 3/4 of the time from event 2 to event 3 have elapsed),
- between t = 9 and t = 10 camera uses the location and direction specified between events 2 and 3.

There are some desirable properties for the camera movement when interpolating camera around an event:

- camera trajectory should have at least C1 continuity,
- camera direction should be continuous,
- camera object (what the camera is looking at) should not deviate much from the object(s) being followed.

There are cases when the object followed will make sharp turns or even move discontinuously, like when following a ball through a bumper collision or following one ball before a collision and another ball after the collision. Therefore, camera object is interpolated during the time that the camera is interpolated. At the beginning and the end of the interpolation period, the camera object coincides with the point being observed. Between those two points, the object follows the Beziér curve with the following control points:

- camera object at the beginning of the interpolation,
- specified camera object at the beginning of the segment from the event to the next event,
- camera object at the end of the interpolation.

The velocity of the object that the camera is observing is not a boundary condition for interpolation. If it were, the interpolation could lead the camera past the object being followed and then back, along a self-intersecting curve, which would be unnatural.

The second part of interpolation is the camera location. Initially, camera location was going to be interpolated using another Beziér curve dependent on the location and velocity of the object the camera is observing. This yields C2 continuity for camera movement. After this was implemented, however, it was noticed that the camera location and object were varying in a way that made the distance from the camera to the object change significantly between every two events. This lead to the strange feeling that the table was swaying closer and farther from the observer, so this implementation was scrapped.

It was replaced by the second version of camera location interpolation. This calculates distance from the object at the beginning and the end, interpolates it with a C1-continuous curve, separately calculates the direction of the camera at the beginning and the end, and slerps the camera direction between the two vectors. Camera location is then finally determined by placing the camera at the calculated distance from the object.

Therefore, if t = 0 at the beginning of the interpolation, t = 1 at the end, and d0, v0, p0, and d1, v1, p1 are distance to the object, direction to the object, and the location of the object at t=0 and t=1, respectively, then at time 0 <= t <= 1, camera is calculated with:

CameraObject (t) = Beziér (p0, p2, p1, t)

CameraDirection(t) = slerp (v1, v2, t)

CameraDistance (t) = Smooth (d1, d2, t)

CameraLocation (t) = CameraObject (t) - CameraDistance (t) * CameraDirection (t)

where p2 is the object being observed at the time of the event being interpolated around, and

Smooth (x) = 2*x^2, x<.5

Smooth (x) = 1 - 2 (x-1)^2, x >= .5

This interpolation gives a reasonable display of the camera following balls through collisions.

Scriptable Cue Stick

The scriptable cue stick is specified by the ball it strikes and the angle of impact.

The movement of the stick consists of 4 phases:

- Approach: The stick is brought from elsewhere to the field of view, until its tip is near the target ball.
- Preflight: The stick moves back and forth twice in preparation for impact.
- Impact: The stick accelerates linearly, hits the ball, and then slows down.
- Retraction: The stick is removed from the field of view, and then disappears after a short period of time.

Care was taken to ensure that the stick enters and leaves the frames in a reasonable way - it will avoid penetrating the table, and it will not disappear while still in sight. The latter is achieved by moving the stick far enough and then removing it from the scene.

The stick is smoothly slerped into the view. Then the stick follows the path of two sinusoidal waves to prepare for impact. After accelerating and decelerating at the appropriate time to give the illusion of setting the ball in motion, it is slerped away from the view and removed.

During the impact, the maximal velocity of the cue tip and the initial velocity of the target ball are

related, although not as realistically as might be desirable. The amplitude of the preflighting displacement is not related to the initial velocity of the target, although it would be in reality.

Sound track

Since event listing was already generated for the scriptable camera, it was extended to include event type with each event. This extension allows the creation of code that generates a composite audio file from audio cues for every type of event. Therefore, given the sounds of a ball-ball collision, ball-bumper collision, and ball-cue collision, and the event listing, the system can generate the sound track which has the sounds appropriately synchronized with the actions.

Playing the sound files in perfect synchronization with events made the sound track appear lagged. This was probably because the sound files used have a very short buildup sound leading up to the peak of the impact. It was empirically determined how much to shift the sound track. Therefore, the sound track is imperfectly synchronized when examined in a sound editor, but well-synchronized when viewed on screen.

Individual Contributions

Ira

- Project proposal editor
- Second version of the collision physics (added restitution)
- Friction
- Initial software architecture and software for physical simulation
- Pool table model
- Cue stick model
- Background research:
  - Researched pool physics and papers
  - Investigated other billiards and physics software
  - Found the texture maps used on the balls
  - Matrix library to use
- Editor of the final paper

Miro

- First version of the collision model
- Selection of rendering software
- Setup and configuration of source control system
- Implementation of the third collision model
- Interface between the simulation and the renderer
- Cinematography of the camera
- Cinematography of the pool stick
- Quaternion implementation
- Ball models
- Debugging and integration of the project
- Physics and implementation of balls falling in holes

- Final production/cinematography

Sriram

- Author project proposal
- Physics behind the third physics model (angular momentum and angular momentum's effect on collisions between both balls and balls, and balls and rails)
- Physics behind ball motion with spin
- First implementation of the object models

Lessons Learned

Please note that much of this is talked about in the achievements section.

1. Animation is more difficult than it looks.

   Animation has many aspects to it, just like the rest of computer graphics. Often the largest issues are not the technical ones but the artistic ones. Issues like how a ball should fall into a pocket, which ball should the simulation try to hit next, where should the camera be, what options should it use so it can follow the action going on, on the table best. Other artistic things are also needed such as, a model of the pool table and a model of each of the balls. When all of these factors are finally added up, the amount of minutia can be overwhelming.

2. Get simple models to work and work on complicated models if one has time.

   This actually turned out to be the most painful lesson. The team did quite a bit of research into multi-ball collisions, and ended up discarding all of it. The team believes that it should have gotten more of the physics working for the basic two ball case and then finally if it had time really look at the multi-ball case. As the project went the physics did get done, but it was more rushed and less detailed than it should have been, due to the early focus on complicated models.

Acknowledgments

Thanks to Dr. Roger W. Webster, Nathaniel G. Auvil, Paul Flory, and Patrick Gillisse of Millersville University for the ball texture maps, and for source code of a similar system.

Thanks to Cathy Coury for being the other set of eyes that every editor needs.

Bibliography

Amateur Physics for the Amateur Pool Player: by Ron Shepard, Argonne Pool League, Argonne National Laboratory.

Engineering Mechanics : Dynamics by J. L. Meriam, L. G. Kraige, 4th edition Vol 2 (March 1997)

Appendix 1

Scene description file format

Scene description file consists of tokens separated by whitespace. There is no syntax for comments in a scene description file. Tokens can be integers written in decimal, floating point numbers written out in decimal, or keywords.

Keywords used for the scene description file are:

```
<keyword> := ball | rail | cue | camera | endcamera | event | follow | hover | look
```

Vectors are represented as 3 floating point numbers, and quaternions as 4 floating point numbers:

```
<vector> := <float> <float> <float>

<quaternion> := <float> <float> <float> <float>
```

A scene description file contains a cue description, a number of ball descriptions, a number of rail descriptions, and a camera description:

```
<scene> := <cue-description> <ball-description>* <rail-description>*
<camera-description>
```

Cue description consists of a number which refers to a ball (the target ball of the cue) and a number which is the angle of the impact:

```
<cue-description> := cue <integer> <float>
```

The target number of the cue has to be greater than or equal to zero, and less than the total number of balls in the scene description. It is the zero-based index of the ball in the file (i.e. 0 is the first ball in the file, 1 the second, etc.)

Ball description consists of an integer ball number, the initial position of the ball as a vector, the initial velocity of the ball as a vector, and the initial orientation of the ball as a quaternion:

```
<ball-description> := ball <interger> <vector> <vector> <quaternion>
```

Rail description consists of two vectors that are the two endpoints of the rail. They must be listed in clockwise order (as seen from above the table):

```
<rail-description> := rail <vector> <vector>
```

Camera description consists of a sequence of event camera descriptions:

```
<camera-description> := camera <event-camera-description>* endcamera
```

An event camera description specifies motion of camera between two consecutive events and consists of an event descriptor and a camera specifier. An event descriptor specifies the first of the two consecutive events and the camera pre-event and post-event interpolation time. A camera specifier consists of camera motion type (look, hover, or follow) and a description of camera location and camera object which depend on the motion type:

```
<event-camera-description> := <event-descriptor> <camera-specifier>

<event-descriptor> := event <integer> <float> <float>

<camera-specifier> := <hover-camera-specifier> | <look-camera-specifier> |
<follow-camera-specifier>

<hover-camera-specifier> := hover <vector> <vector>

<look-camera-specifier> := look <vector> <integer>

<follow-camera-specifier> := follow <integer> <vector>
```

The numbers in an event descriptor, in order, are the event number, the pre-event interpolation time, and the post-event interpolation time. The event number has to be greater than or equal to zero, and either smaller than the total number of events (including the two synthetic events), or equal to 65535 (which is considered equal to the highest event number). The special meaning of 65535 is a magic value that would ideally be implemented as a keyword. The pre- and post-event interpolation times must be greater than or equal to zero, and less than or equal to one.

The vectors in the hover camera specifier are the location vector and the object vector, in that order.

The vector in the look camera specifier is the location of the camera. The integer is the ball the camera is looking at.

The integer in the follow camera specifier is the ball the camera is following. The vector is the location of the camera relative to the coordinate system in which the ball is at <0, 0, 0> and moving along the +x axis, and z is up from the table plane. For example, the vector <-1, 0, 1> means "one unit behind the ball in the direction opposite from its velocity, and one unit up away from the table".

The actual parsing code is somewhat more liberal than this specification: it allows any order of cue, ball, rail, and camera specifications.

This specification should be treated as the authoritative description of valid scene description files.