gl-billiards

Dan Lapp
Taras Mychaskiw
COSC 3P98 Project

January 8th, 2014

Contents

1 Controls

Refer to Figure 1 for a view of the gl-billiards window. In the figure, the user interface components are labelled. Below is the description of each of the labelled controls.

1. Informational text area. This will contain information about balls pocketed, fouls, scratches, whose turn it is and which ball type (solids or stripes) they are shooting for etc.

2. Cue rotation. Click the double arrow and drag the mouse left or right to rotate the cue during your shot.

3. Shot power and control. Adjust the power of your shot by clicking the up or down arrows. You can also manually enter a shot power in the input (shot powers are between 0 and 1). Click the Shoot button to take the shot at the current angle and power selected.

4. Camera rotation. Click the cross and drag the mouse in any direction to rotate the entire scene.

5. Pick the center of the camera's rotation. If "Follow Ball," the camera will rotate around the ball selected in item 7. The camera will also follow the ball as it moves around the table during a shot. If "Center of Table," the camera will rotate around the center of the table, and will not move as the balls are moving around the table.

6. Resets the camera's rotation to the default.

7. Choose which ball to follow. If "Follow Ball" is selected in item 5, then the camera will follow the selected ball around the table as it moves. The center of the camera's rotation will also be the selected ball here.

8. Camera zoom controls. Self explanatory.

Additionally, during a time when the player is allowed to move the cue ball freely, the arrow keys will move the cue ball around. You may need to click the graphics window (the pool table area) first.

Figure 1: The gl-billiards window. Described above is what each of the controls labelled by red numbers do.

2 Physics Engine

2.1 Taking a Shot

Once a ball is hit by the cue, there are two ways energy is transferred to the ball. First, and more noticeable of the two is the transfer of linear momentum. This sends the ball moving around the table in straight lines. Second is the transfer of angular momentum, which applies spin to the ball. Unfortunately in gl-billiards, spin is not implemented (though some time was spent formulating equations for spin).

To simplify the shot making process, the pool cue is assumed to act like a spring. That is, the further you pull the cue away from the ball you are going to hit, the more powerful

your shot will be. Spring motion can be described using Hooke's Law,

$$\vec{F} = -k\vec{x} \tag{1}$$

where $\vec{F}$ is the force, $\vec{x}$ is how much the spring is stretched and k is the spring constant, which describes basically how powerful (springy) the spring is.
Hooke's Law is equivalent to Newton's Second Law, which states

$$\vec{F} = m\vec{a} \tag{2}$$

where $\vec{F}$ is the force, m is the mass of the object and $\vec{a}$ is the acceleration of the object. Setting Equation 1 equal to Equation 2 gives

$$\begin{aligned} -k\vec{x} &= m\vec{a} \\ -k\vec{x} &= m\frac{\Delta\vec{v}}{\Delta t} \\ \Delta\vec{v} &= \frac{-k\Delta t}{m}\vec{x} \end{aligned}$$

where $\vec{v}$ is the velocity of the object and t is the time. For pool, the initial velocity of the ball being struck is always zero, so we can simplify the above equation to

$$\vec{v} = \frac{-kt}{m}\vec{x} \tag{3}$$

where $\vec{x}$ is the distance between the ball and the tip of the cue, m is the mass of the ball, t is how much time the cue is in contact with the ball and k is the spring constant of the cue. The values for t and k needed to be experimentally found though trial and error, essentially trying values until what happened in the simulation closely represented real life. The values for t and k were found to be 0.01s and 200N/m respectfully.

2.2 Event System

For the purposes of this program, an event is a ball-ball collision, or a ball banking off of the cushion (a ball-wall collision). The way gl-billiards works is each time the screen is redisplayed, some physics updater function is called with a parameter dt, which is how much time has passed since the last frame. The engine simulates the pool game up until a time dt has passed. For example, if 1s has passed, then it would simulate 1s of play before returning, then the screen would be updated as if 1s had passed.
The engine determines which events occur within time dt. Event detection then becomes *when* do events happen instead of *if* they happened. The entire update function works as described in Algorithm 2.1 below.

Algorithm 2.1: UPDATE(dt)

$event \leftarrow$ earliest event that occurs with time dt
if no events
  then roll balls for dt

  else { rolls balls until $event$ occurs
  handle $event$, update velocities
  UPDATE($dt - event.getTime()$)

In this manner, all events that happen within dt and handled automatically before the frame updates and the balls slow down due to friction. Note that "roll balls" simply means each ball moves at it's (constant) speed for some amount of time. Pocketing detection needs to be done there, as it is not a standard event. That is, if a ball is pocketed, it cannot possible participate in any events in time dt, since it would bank off the cushion or collide with another ball first.
After $update(dt)$ is finished, each ball gets affected by friction and slows down slightly. Friction takes the form of a slight decrease in the velocity of the ball.

2.3 Event Detection

2.3.1 Ball-Ball Collision Detection

A ball-ball collision will occur when the centers of the balls are two radii apart. In gl-billiards, the trajectory of each ball is simplified to be a straight line per frame since friction is only applied at the end of each frame. Thus, the trajectory of each ball becomes,

$$\vec{r} = \vec{r_0} + \vec{v}t \tag{4}$$

where $\vec{r}$ is the position of the ball, $\vec{r_0}$ is the starting position, $\vec{v}$ is the velocity of the ball and t is the amount of time elapsed. Then, for each possible collision to occur, the program needs to find at what times the distance from one ball to the other is equal to twice the radius of the ball. Mathematically, this becomes

$$\begin{aligned} |\vec{r_1} - \vec{r_2}| &= 2R \\ \sqrt{(\vec{r_1} - \vec{r_2}) \cdot (\vec{r_1} - \vec{r_2})} &= 2R \end{aligned}$$

where R is the radius of a ball. Squaring both sides, and subbing in Equation 4 for both $\vec{r_1}$ and $\vec{r_2}$ (using $\vec{dr_0} = \vec{r_{0_1}} - \vec{r_{0_2}}$ and $\vec{dv} = \vec{v_1} - \vec{v_2}$),

$$(\vec{dr_0} \cdot \vec{dr_0}) + (\vec{dr_0} \cdot \vec{dv})t + (\vec{dv} \cdot \vec{dv})t^2 = 4R^2 \tag{5}$$

This is a quadratic equation for t which can be solved easily. If there are real roots to the equation, then a collision occurs, and it occurs at the minimal possible t value which is a solution to Equation 5.

2.3.2 Ball-Wall Collision Detection

Ball-wall collision detection is similar to ball-ball, except the ball is a distance one radius away from the wall when it collides. Since the wall doesn't move, a different strategy had to be employed to detect when a ball-wall collision occurs. In this case, both the and wall positions are described using parametric equations.

$$\vec{r} = \vec{r_0} + (dt\ \vec{v})T_b \tag{6}$$
$$\vec{w} = \vec{s} + (\vec{f} - \vec{s})T_w \tag{7}$$

Where $\vec{r}$ is the position of the ball, $\vec{r_0}$ is the initial position of the ball, $\vec{v}$ is the velocity of the ball, dt is the amount of time that will pass this frame; $\vec{w}$ is a point on the wall, $\vec{s}$ is one endpoint on the wall, and $\vec{f}$ is the other endpoint of the wall. T_b and T_w are the parametric variables, and range between 0 and 1.
To determine when the ball hits the wall, where the ball hits must be determined first. The center of the ball hits the wall when $\vec{r} = \vec{w}$, thus the ball actually collides with the wall when $\vec{r} - R\vec{n} = \vec{w}$, where $\vec{n}$ is the normal vector to the wall. For simplicity below, we will ignore this offset value, as it is a fairly simple transformation. So, the collision occurs when

$$r_{0_x} + (dt\ v_x)T_b = s_x + (f_x - s_x)T_w$$
$$r_{0_y} + (dt\ v_y)T_b = s_y + (f_y - s_y)T_w$$

Solving for T_b and T_w gives

$$T_b = \frac{(f_x - s_x)\cdot(s_y - r_{0_y}) - (f_y - s_y)\cdot(s_y - r_{0_y})}{(dt\ v_y)\cdot(f_x - s_x) - (dt\ v_x)\cdot(f_y - s_y)}$$
$$T_w = \frac{(dt\ v_x)\cdot(s_y - r_{0_y}) - (dt\ v_y)\cdot(s_y - r_{0_y})}{(dt\ v_y)\cdot(f_x - s_x) - (dt\ v_x)\cdot(f_y - s_y)}$$

If both T_b and T_w are between 0 and 1, then a ball-wall collision occurs. The point at which can be found by subbing in T_b or T_w into Equations 6 or 7 respectfully. Then the amount of time passed can be solved for using Equation 4.

2.4 Event Handling and Velocity Updates

2.4.1 Ball-Ball Collisions

The first step in handling a ball-ball collision is to break up the velocity into two components: normal to the collision plane and tangential to the plane. The normal components of each ball's velocity are given by

$$\vec{v}_{n_1} = (\vec{v}_1 \cdot \vec{n})\vec{n}$$
$$\vec{v}_{n_2} = (\vec{v}_1 \cdot (-\vec{n}))(-\vec{n})$$

where $\vec{n}$ is the normal to the plane. Then, the tangential components can be obtains from

$$\begin{aligned} \vec{v}_{t_1} &= \vec{v}_{n_1} - \vec{v}_1 \\ \vec{v}_{t_2} &= \vec{v}_{n_2} - \vec{v}_2 \end{aligned}$$

During the collision, the tangential components of the velocity of each ball do not change. Thus, the collision can be reduced to a $1D$ collision along the normal components. For a $1D$ collision, the final velocities are given by

$$\begin{aligned} \vec{v_1}' &= \frac{(m_1 - \epsilon m_2)\vec{v}_1 + (m_2 + \epsilon m_2)\vec{v}_2}{m_1 + m_2} \\ \vec{v_2}' &= \frac{(m_1 + \epsilon m_1)\vec{v}_1 + (m_2 - \epsilon m_1)\vec{v}_2}{m_1 + m_2} \end{aligned}$$

where ϵ is the coefficient of restitution. In our case $m_1 = m_2$, and $\epsilon = 1$ since all balls have an equal mass and the collision is totally elastic. The above equations simplify to

$$\begin{aligned} \vec{v_1}' &= \vec{v}_2 \\ \vec{v_2}' &= \vec{v}_1 \end{aligned}$$

Thus, the normal components of the velocities of the balls simply switch. The final velocities of each ball after the collision are

$$\begin{aligned} \vec{v_1}' &= \vec{v}_{t_1} + \vec{v}_{n_2} \\ \vec{v_2}' &= \vec{v}_{t_2} + \vec{v}_{n_1} \end{aligned} \tag{8}$$

2.4.2 Ball-Wall Collisions

Ball-wall collisions are much simpler than ball-ball, as the velocity of the ball is reflected through the vector normal to the surface of the wall. The velocity after the event becomes

$$\vec{v}' = \vec{v} - 2(\vec{v} \cdot \vec{n})\vec{n} \tag{9}$$

However, there is some energy loss due to the rubberiness of the wall. To account for this, the resultant velocity is scaled down by a small percentage to represent the loss. In gl-billiards, this speed loss factor is 0.3, which equates to about a 20% energy loss from the collision with the wall.

2.4.3 Friction

At the end of each frame, friction slows down every moving ball slightly. There are two types of friction, kinetic and static. Kinetic friction slows down objects which are moving, and static friction must be overcome to either start or continue moving. Simpler of the two in gl-billiards is static friction, which is just a simple speed check. If the speed of a ball is

less than some threshold (1cm/s in gl-billiards), then the ball's velocity is set to 0. Kinetic friction acts in the opposite direction of the ball's velocity, and is equal to

$$\vec{F}_f = -\epsilon mg \tag{10}$$

where ϵ of the coefficient of kinetic friction between the table surface and a pool ball. Using Equations 2, this final velocity of the ball can be obtained after the frictional force is applied.

$$\vec{v}' = \vec{v} + \vec{F}_f \cdot dt \tag{11}$$

where dt is the amount of time that passed in the frame. And that's it! That's all of the physics in gl-billiards in a nutshell.

3 3D Engine

3.1 Loading Models

To use 3d models that are more complex than simple primitive shapes, it is important to import custom models. gl-billiards employs 3D Studio Max to create and texture all of the models in the game. The main advantage of this approach is that all vertex coordinates and texture mapping coordinates can be handled by 3D Studio Max and loaded into OpenGL. Programming a pool table directly in OpenGL would be virtually impossible.
gl-billiards uses the *.3ds* file format to read models into OpenGL. This is a binary file format that is fairly simple to parse, well documented and natively supported by 3D Studio Max. The data is organized into a hierarchy of chunks. An imported model is abstracted into it's own class *Model*. Each object in the scene, such as *Table* or *Cue* have a *Model*. An example of the model initialization is shown below,

$$Model\ \ model = Model(std :: string\ \ modelPath,\ \ std :: string\ \ texturePath)$$

The texture specified must be a *.bmp* file for the texture parser to read. When the model is initialized, the 3ds parser skips through the file chunks and stores three things. First are the actual *(x, y, z)* coordinates of the vertices, then the polygon list, which is the order to draw the vertices. Last is the texture mapping coordinates.
The bitmap parser reads in a *.bmp* file used as a texture for the model. The texture must have a width and a height that is a power of 2, and must be maximum 2048 pixels in any dimension. The textures are then handed to OpenGL and a texture id is saved in the model.

3.2 Calculating Normals

After the model and texture is loaded, normals have to be calculated to be used with game lighting. gl-billiards calculates normals itself. This way we could get the vertex normals without having to parse the smoothing groups in the *.3ds* file. Smoothing groups

are a different representation of smoothed vertex normals used by 3D Studio Max, and are not directly used with OpenGL's $glNormal()$. Since every polygon in this file format is a triangle, calculating normals was fairly simple. The pseudocode for this is shown in Algorithm 3.1.

Algorithm 3.1: CALCULATENORMALS($vertexList, polygonList$)

local $numPolygons[vertexList.size]$
for each $polygon \in polygonList$
do
$\vec{v}_1 \leftarrow polygon.\vec{P}_1 - polygon.\vec{P}_2$
$\vec{v}_2 \leftarrow polygon.\vec{P}_1 - polygon.\vec{P}_3$
$\vec{n} \leftarrow \vec{v}_1 \times \vec{v}_2$
for each $coord \in polygon$
do
$normalList[coord] \leftarrow normalList[coord] + \vec{n}$
$numPolygons[coord] \leftarrow numPolygons[coord] + 1$

for each $i \in normalList$
do $normalList[i] \leftarrow normalList[i] \div numPolygons[i]$

First, for each triangular polygon (P_1, P_2, P_3) polygonList, calculate two vectors $\vec{v}_1, \vec{v}_2$ to use for the cross product. The cross product returns a vector $\vec{n}$ perpendicular to the polygon face. *numPolygons* keeps track of which surfaces are next to each vertex. Then in the final *for* loop the polygon normals are averaged to give each vertex normal a smoothed effect.

3.3 Optimizations

To optimize 3d rendering, OpenGL display lists are implemented for *Model*. Large floating point arrays used to store vertex coordinates require a lot of memory to manage these arrays. Using display lists significantly reduces the overhead associated with storing vertex arrays. From the results of running benchmarks on gl-billiards, the user interface used $\sim 43\%$ of the program's computation time, and physics used $\sim 27\%$. Therefore speeding up the model rendering provided a noticable speed increase of at least $5\times$.

References

[1] Ross, B. "COSC 3P98 Lecture Slides" Brock University, Fall 2013

[2] Fowles & Cassiday "Analytical Mechanics, 7th Ed." Thomas Brooks/Cole, 2005

[3] Paul Rademacher, "GLUI - A GLUT-Based User Interface Library" Version 2.0, 1999

[4] Damiano Vitulli, "3D Engine Programming Tutorials", Retreived December 2013 from http://www.spacesimulator.net/wiki/

[5] World Pool-Billiard Association, "Table & Equipment Specifications", Retreived December 2013 from http://www.wpa-pool.com/