SFML Billiards

This assignment must be completed individually. Working in groups is not permitted.
Due Wednesday, March 18th at the start of class.

For this project we will use the SFML "game loop":

```
initialize(/*references*/);
while (window.isOpen()) {
    sf::Event event;
    while (window.pollEvent(event)) {
        if (event.type == sf::Event::Closed)
            window.close();
        if (event.type == sf::Event::MouseButtonPressed) {
            //handle event
        }
        if (event.type == sf::Event::MouseButtonReleased) {
            //handle event
        }
        window.clear();
        //draw background image
        //draw balls
        //updateBalls(balls, vel);
        window.display();
    }
}
```

The objects of interest in a billiards game must include, at minimum, a table (rectangular png object) including features of the shape of the perimeter with the six pockets, the cueball and the 15 other balls, 1-8 solids and and 9-15 striped. See https://opengameart.org/content/8-ball-pool-assets

1. The first `initialize()` function prompted looks like this:

```
void initialize(/*reference to container(s) for billiard balls*/) {
    //set up billiard ball objects with radii, positions, textures, and velocities
}
```

   However, this suggests that there are three separate, parallel containers for the `sf::CircleShape`, `sf::Texture` and the `vel(sf::Vector2f)` objects. A better design concept here might be to have, as in the 2-body project, a `Ball` or `Body` class (call it something evocative of what it is) the has these as member variables.

2. The `update()` function:

```
void update(std::vector<sf::CircleShape>& Cans, std::vector<sf::Vector2f>& vel) {
    for (/*every ball*/) {
        //move the ball according to its velocity
        //bounce off the edges of the table, if needed
        //delete the ball if it falls in a pocket
        //simulate the loss of velocity due to friction
        //stop the ball if its speed is less than some threshold
    }
    //call the collision function
}
```

To model play you will need to associate with each ball a `sf::Texture` (see the `assets.zip` file), a `sf::CircleShape` and a velocity, which will need to have both `x` and `y` components.

3. Provide a routine for handling the physics of elastic collisions between balls.

```
void collision(/*balls and velocities*/) {
    for (int i = 0; i < numbCans - 1; ++i) {
        for (int j = i + 1; j < numbCans; ++j) { //for every pair of billiard balls
            if (/*the balls collide*/) {
                //compensate for penetration by repositioning
        //compute the collision normal
        //compute the unit normal
        //compute the relative velocity
        //compute component of the relative velocity along the collision normal
        //update the velocity of the balls that collided to simulate actual physical
    collision
      }
    }
  }
```

Impulse Based System

Suppose `sf::CircleShapes C[i]` and `C[j], i!=j` have collided. Then the direction of the collision normal is $\vec{d}$ = `C[i].getPosition() - C[j].getPosition()`. The penetration distance is `P = 2*ballRadius - distance(C[i],C[j])`, where the distance is measured between their centers. If the penetration is large then you'll want to move the balls in opposite directions along the collision normal, each by `P/2`.

The point of collision is then $\vec{p}$ = `C[i].getPosition()` + $\vec{d}/2$. The unit normal is found by dividing the collision normal by its length: $\hat{n} = \dfrac{\vec{d}}{\|\vec{d}\|}$

Assume that near the time of collision, forces and positions are nearly constant, but there will be a discontinuity in velocities. The impulse of collision will push the bodies apart. If the masses are equal, then the impulse of each ball will be equal in magnitude but opposite in direction. So we need to generate a scale factor j for $\hat{n}$ and then add $j \cdot \hat{n}$ and$-j \cdot \hat{n}$ to the velocities `vel[i]` and `vel[j]` to get the outgoing velocities.

The relative velocity is $\vec{V}_{ij}$ = `vel[i]` − `vel[j]`. From that we compute the amount of relative velocity that is applied along the collision normal. The dot product of any vector with a normalized vector gives the projection along the normalized vector, which is what we want: $\vec{V} = \|\vec{V}_{ij}\| \cos(\theta)\hat{n}$ where θ =the angle between the collision normal and the relative velocity.

4. Once you have the collision system working, add a feature to "hit" the cue ball with an imaginary stick (or you can use the actual png stick provided) by clicking on the cue ball and dragging out a direction and velocity. Set it up so that player can simulate the play of "eightball".