Billiards Simulation

Project Documentation

Version *<1.0>*

<2019-04-14>

420-204-RE Projected Management Plan

Presented to

Samad Rostam Pour

Vanier College

Team Members : Daniel Cicciarelli

Karim Botros

Dylan Bobb

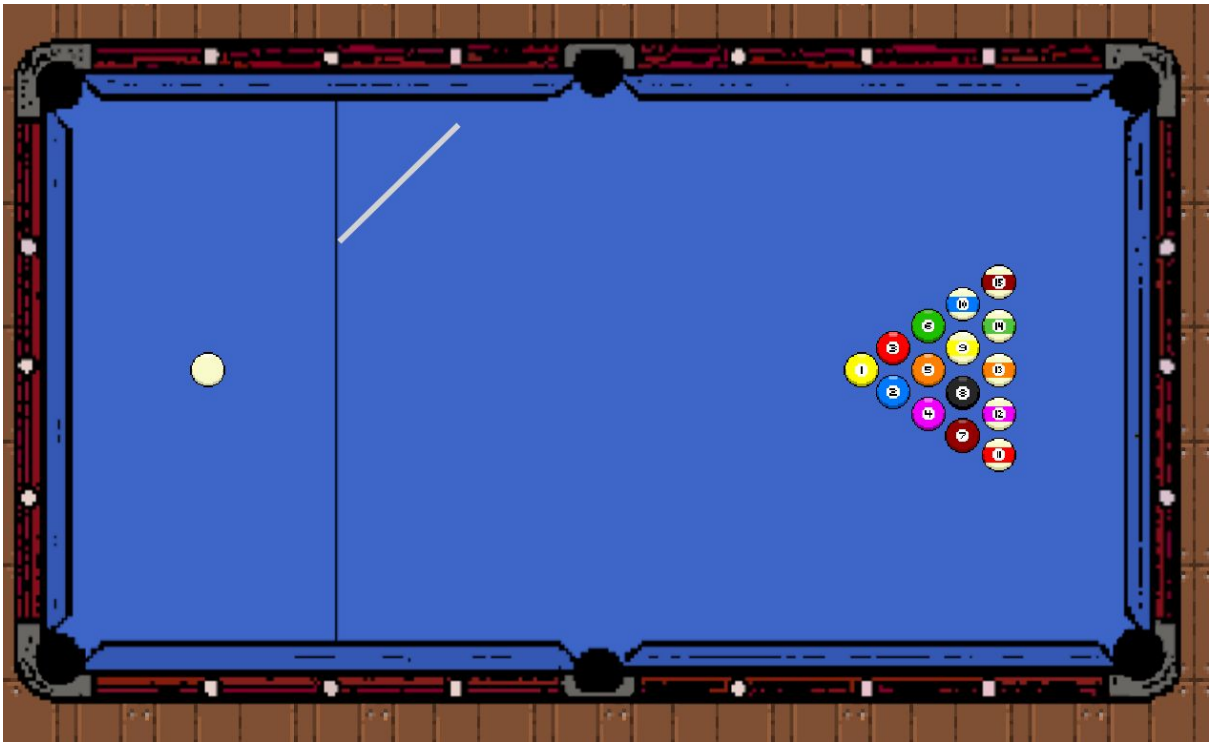
Vincent Bruzzese

User Documentation

Upon starting up the program the user will be faced with the startup menu. This consists of a gif playing in the background, along with three buttons and three menu items. Should the user wish to play the sandbox portion of the game, he will press on the first button. This is the button that has two balls in sand, continued by the word box. This leads him to the sandbox gamemode. If the user wishes to play the 8-ball game of billiards implemented in the game, he must press the button with the 8 ball design. Upon doing so, he will be redirected to the 8 ball portion of the game. At least, if turning on the game was a giant mistake, the exit button will allow him to close the program.

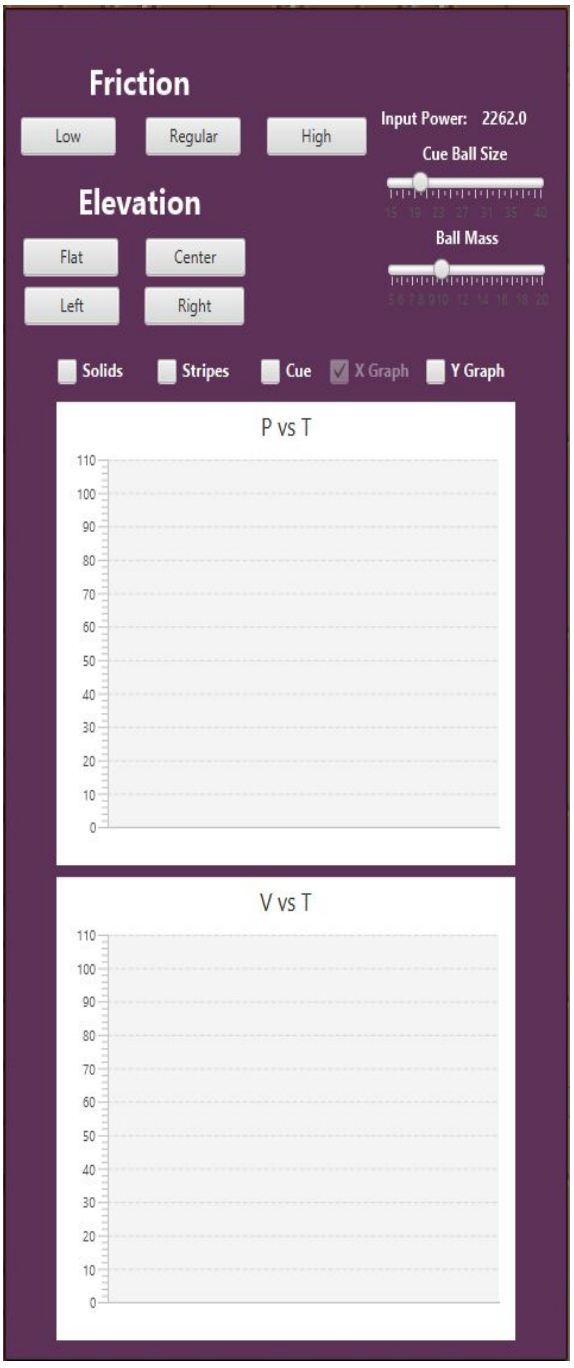
Accessing the game modes is also available through the menu at the top of the screen . The user would have to press on the options portion, and to access the sub menu items. These items are SandBox (ctrl+S) and 8-Ball (Ctrl+B). Upon using the shortcuts indicated, the user will be able to bypass the menu screen.Of course, upon pressing these menu options, you will be redirected to the appropriate section of the game. If the user desires to read the rules of the game, he must click on the rules portion of the menu. The two submenus SandBox Rules (Ctrl+F1) and 8-ball Rules (Ctrl+D) will give the user access to the rules of the two game modes. The last menu item, will Exit the game, Quit (Ctrl Q).

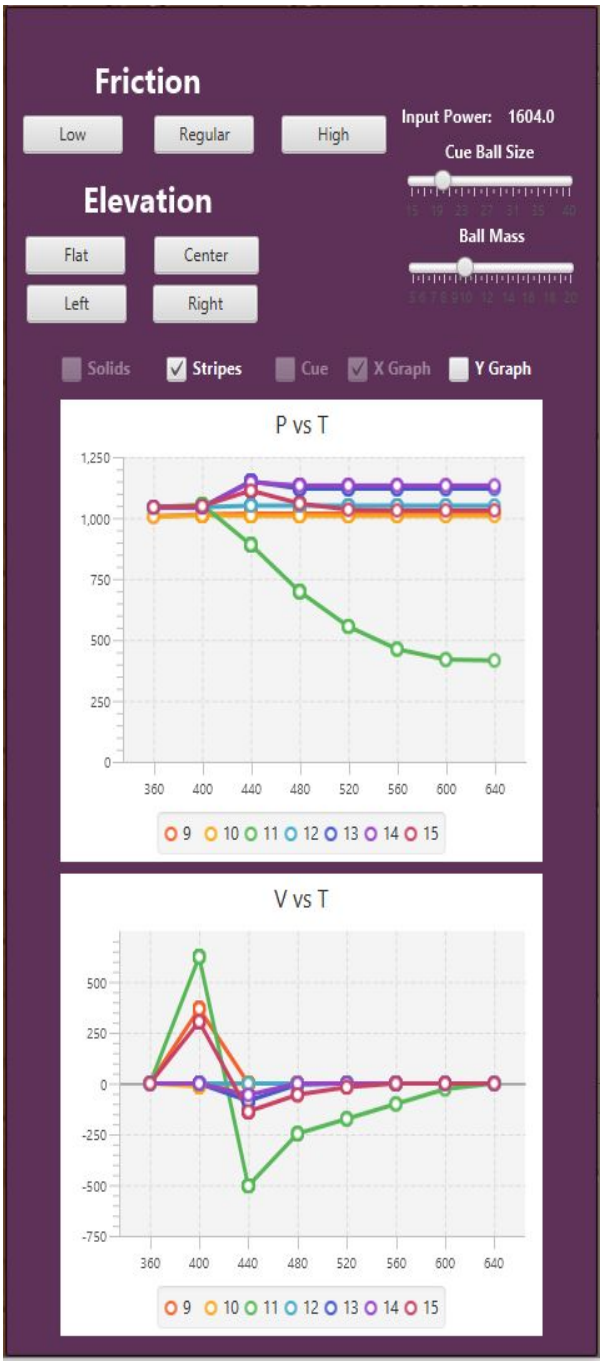
Upon entering the sandbox game mode, the user will be looking at a table, with a white cue ball, and 16 balls in a triangle formation on the right side of the table. His cue, the white stick, rotates around the ball, mimicking an actual cue. Depending on the location in which the user places the cue, it will determine the angle and the distance will determine the power input. The power input can be located on the side menu. When the user manages to get a ball into a pocket, one of the black corners, the ball will disappear, as it has been removed from the game. The cue ball cannot be sunk in sandbox mode, to allow the player complete freedom to do what he pleases to all the other balls. Once the user clicks with the mouse, the shot is released, following the trajectory and power inputted.

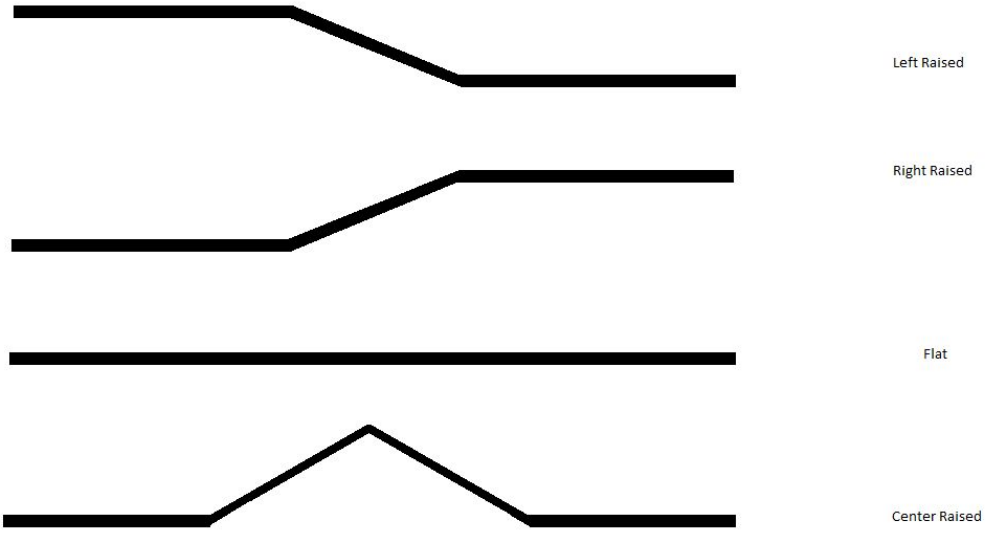


Now, the user can always modify the friction of the table or the elevation that he prefers with the buttons located on the side menu. There are three friction options, which lead to different friction coefficients, working on the ball. These work to slow down the cue ball,

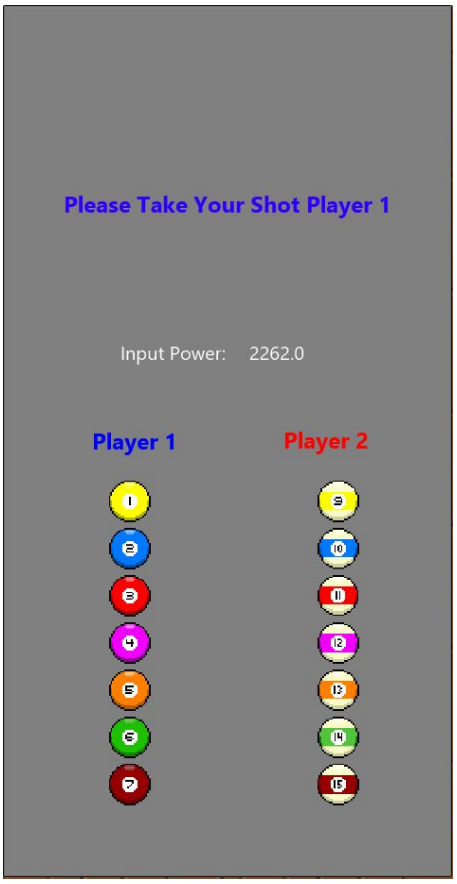
and other balls. The elevation buttons will modify the elevation of the table. Center, raises the center of the table, making the balls fall to either side of the table, Left and Right will raise either the left or right side of the table, splitting it into thirds, and allowing for the ball to fall, to the other side, on the middle portion of the table. If the user desires, the Cue ball size and Ball masses can be modified with the sliders. Of course, the user can also decide whether or not he wishes to graph certain aspects of the game. This can be done through the toggles located above the two graphs. All that needs to be done is to decide what you would like to have graphed.







If the user decides to enter the 2-player 8-ball game mode rather than the sandbox mode, he will once again be faced with a table a white cue ball, and 16 balls in triangle formation.



The user will once again have to shoot the ball in order to pot his balls and win the game. Player 1 has to pot all the solid balls and then the 8-ball to win, while player 2 has to pot all

the striped balls and the 8-ball to win. Unlike the sandbox mode, there are scratches in this game mode, and scratching could give the other player the upper hand. If the user pots the cue ball, does not hit any balls, hits a ball of the opposite suit first, or if none of the balls collide with the table, the opposite player will get to place the ball wherever he/she chooses in the scratch area. Another difference in the 8-ball mode is that there are no physics-related toggles on the side. Apart from those differences, the game runs the same as the sandbox mode, just with a more competitive nature with two players.

Code Documentation:

Our program was rather complex, and it was difficult to recreate billiards and physics manipulations for our project. We had 12 classes and an FXML document in this project: WinterProject, AssetManager, Vector2D, GameObject, FXMLDocumentController, Ball, Cue, Collision, Player, Table, Pocket and Wall.

WinterProject:

The main class of this project is our WinterProject class. This class contained the information for stage and scene, and called them up when the method start was called.

Asset Manager:

We have a class named AssetManager, which, as its name implies, takes care of all the images and audio used in this project. This class contains all the URLs of each image used in our project, as well as all the different cases for the balls and tables. There were 16 cases total for all the balls, and cases for the tables.

```
static public void preloadAllAssets() {
    //backgrounds for the game
    Image Sandboxbackground = new Image(fileURL("./assets/images/sandboxBackground01.png"));
    Image background = new Image(fileURL("./assets/images/background.gif"));
    backgroundImage = new Background(
            new BackgroundImage(background,
                    BackgroundRepeat.NO_REPEAT,
                    BackgroundRepeat.NO_REPEAT,
                    BackgroundPosition.DEFAULT,
                    new BackgroundSize(100, 100, true, true, true, true)));

    SandboxbackgroundImage = new Background(
            new BackgroundImage(Sandboxbackground,
                    BackgroundRepeat.REPEAT,
                    BackgroundRepeat.REPEAT,
                    BackgroundPosition.DEFAULT,
                    new BackgroundSize(250, 250, false, false, false, false)));
    //backgrounds for the buttons
    Image standardButtonBackground = new Image(fileURL("./assets/images/8button.png"));
    Image sandButtonBackground = new Image(fileURL("./assets/images/sandboxbutton.png"));
    Image exitButtonBackground = new Image(fileURL("./assets/images/exitbutton.png"));
```

Vector2D:

As well as a class for our assets, we have a class named Vector2D. This class is for all the vectors involved in our project. As our project was very physics-oriented, having a class for vector manipulation was essential. This class involves basic adding, subtracting and multiplying vector methods, and includes an adjustMagnitude method that takes into account the friction of the table when calculating the magnitudes of the velocity vectors. This was a crucial method as friction played a big part in all game modes of this project.

GameObject:

Following the Vector2D class, we made a GameObject class for creating objects in our project. Our project has both circular and rectangular objects, so we made two constructor methods for both shapes. This class also included getters and setters for velocity, position, and acceleration, and getters for the circles and rectangles of objects. This was important for adding and removing the objects from the panes, as well as updating their positions and velocities while the game was in action. The most important method this class contained was the update method, which continuously updated all of the vector components while the animation timer was running.

```
public void update(double dt) {
    // Euler Integration
    // Update velocity
    Vector2D frameAcceleration = getAcceleration().mult(dt);
    velocity = getVelocity().add(frameAcceleration);

     //Update position
    if (Math.abs(getVelocity().getX()) < 1) {
        getVelocity().set(new Vector2D(0, this.getVelocity().getY()));
    }

    if (Math.abs(getVelocity().getY()) < 1) {
        getVelocity().set(new Vector2D(this.getVelocity().getX(), 0));
    }
    position = getPosition().add(getVelocity().mult(dt));
    if (this.getCircle() != null) {
        circle.setCenterX(position.getX());
        circle.setCenterY(position.getY());
    } else {
        rectangle.setY(position.getY());
        rectangle.setX(position.getX());
    }
}
```

FXMLDocumentController:

The FXMLDocumentController is our project's most important class. This class handles the phases of the game, the functions of all the buttons, the mouse events, and the game's workings.

Our billiards simulation has four phases: a cue phase, an action phase, a check phase and a place phase. The game starts on the cue phase, and all the physics toggles on the side are available to change. Once the player shoots, a method called beginActionPhase is activated and all the toggles are disabled. Once all the balls' velocities reach 0, the game switches into check phase. This phase checks what happened in the turn before, and determines if the player shoots again, if it switches to the other player, or if there was a scratch. If there was a scratch, the game switches to place phase, and the user can place the ball where he wants before the game switches to cue phase. If there was no scratch, the game switches straight to the cue phase.

This class also includes all the actions for the physics toggles. There are five toggles for the graphs, which include a toggle for the positions and velocities along the x-axis, the y-axis, for the striped balls, the solids balls, and for the cue ball. There's also the actions for the quit buttons and the start up buttons for 8-ball and sandbox mode. Finally, there's the logic for the friction and elevation toggles, which sends the information to the table class.

The FXMLDocumentController class contains the mouse events for our game. There's the onMouseMoved event, which, when in the cue phase, updates the shot input (the position of the mouse will determine the power of the shot and the direction). When in the place phase, the mouse event moves the cue ball wherever the user's mouse is. The onMouseClicked will shoot the ball in the cue phase, and will place the ball when in the place phase.

The last part of code in this class was the entirety of the game rules. The method that determines whether there was a scratch or not (returns a boolean variable equal to true when the cue ball has no collisions, if the cue ball is sunk, if the 8 ball is sunk, if the user does not hit a ball of the correct suit first or if none of the balls hit a wall) is a major part of this code. There is also the logic for switching the turns, winning and losing, and the graphing of all the balls.

Ball:

We created a Ball class that extended our GameObject class. This class is essential for our program to function correctly, as it contained the logic for all the collisions between the balls and other balls, walls, and pockets. The first method we made was the constructor, which needed to include position, velocity, ball case, and the mass. The int that was inserted for the ball case would be passed along to the asset manager, which would then in turn assign the ball its proper image. This class contained all the calculations for the collisions, as well as

the calculations for friction and table elevation. A lot of the game physics is within the Ball class.

Cue:

The cue class only contained the constructor for the cue. Despite only having the one method, this class was still slightly complicated because we needed to come up with a way for a rectangle to rotate around a fixed point (the cue stick needed to rotate around the cue ball). In order to do this, we made our cue a rectangle that had the length and width of a circle's radius, that way the image would be rectangular yet we could still rotate the object like a circle.

Collision:

This class was made to keep track of all the collisions that occur in our simulation. There's a constructor that takes two balls, and checks the numbers of both balls and puts the ball with the lower number as the first ball. Then we have methods to determine which balls collided, if the cue ball collided, if a striped ball collided, and if a solid ball collided.

Player:

The player class starts with a constructor as well, taking a player number, a name, and a boolean variable that is dependent on if it is there turn or not. This class is very important for the classic billiards game mode, as it contains methods relating to the player suits. First, there is a getter and setter for a boolean variable suit, that returns true for solids and false for stripes. Then there is a method that returns a boolean called ballMatchesSuit. This checks the suit of the ball, checks the suit of the player, and returns true if the player hit a ball that matches his or her own suit. There are also methods for switching the player turn and adding potted balls to an arrayList.

Table:

Our table class is mainly used for updating the table's image to that of the user's selections. The different elevations and frictions values all produce different images for the table, and this class runs through every scenario's case number. There are also getters and setters for the elevation cases, angles, and friction.

```
switch (number) {
    case 0:
        friction = low;
        elevationAngle = flat;
        elevationCase = 1;
        break;
    case 1:
        friction = low;
        elevationAngle = left;
        elevationCase = 2;
        break;
    case 2:
        friction = low;
        elevationAngle = right;
        elevationCase = 3;
        break;
    case 3:
        friction = low;
        elevationAngle = center;
        elevationCase = 4;
        break;
    case 4: // 8Ball Default Table //
        friction = regular;
```

Pocket:

Pocket is our projects simplest class, only containing the classes constructor. This class takes in a position and radius, and is transparent.

Wall:

The wall class is also rather simple. It contains a constructor that takes position, width, height, cushion and ID. There is a getter and setter for the wall's cushion, as well a getter for the walls ID. The walls were of course transparent, and were simply placed on the table image's walls.

Obstacles:

Throughout the weeks, we encountered several obstacles that needed a bit more time to flatten out. For the GUI, we had a significant amount of trouble trying to get the graphs to function properly. Sometimes the balls wouldn't appear, sometimes the program would crash, and other times the balls velocities and positions wouldn't be displayed correctly. It took nearly a week to get our graphs to become fully functional. A major issue with the physics section of our project was dealing with balls moving at low velocities. When the balls were moving at low speeds, they would phase in and out with one another, and animation of the ball would be jittery. This took a lot of trial and error to work, but once we got past it there was only one other major obstacle. Recognizing when there was a scratch took a while to get working. When we first started working on that, the game would return the method checkForScratch as true on every shot. Finally, after several days of focus only on that one method, we got it to function properly.