Project documentation

Walter Berggren
thesauri

1 Introduction

The goal of this project was to create a billiard game and implement the physics for it on my own.

To make a realistic simulation of a game of billiard, the angular velocity of the balls has to be taken into account. For instance, hitting the top of the cue ball will give it a forward spin causing it to continue forward after hitting other balls (instead of stopping). To allow for this kind of shots, the physics engine takes both the velocity and angular velocity into consideration as the balls roll over the board and collide with each other.

The game can be played by two players taking turns on the same device. The full set of rules specified by the World Pool-Billiard Association have not been implemented. Only the most essential rules, such as allowing the same player to continue shooting if one of the player's balls is pocketed,are implemented.

This game uses the Java game framework libgdx for rendering. However, none if its physics and mathematics libraries were used in this project. These have been implemented on my own. The main reason why libgdx was chosen was its cross-platform support.

Games can be saved and returned to at a later point in time.

The project is based on an empty project template for using Scala and SBT with libgdx[1].

2 Installation and usage

Compiling the project requires `sbt`. Compiled binaries for desktop and Android devices can be found in the `bin/` folder.

2.1 Compiling

2.1.1 Desktop

To run the application without creating a stand-alone jar file, type `sbt desktop/run` in the root project folder.

To create a standalone JAR file, type `sbt assembly`

[1] `https://github.com/ajhager/libgdx-sbt-project.g8`

2.1.2 Android

Type `sbt android:package-debug` to create an Android package. It will be saved as `android/bin/android-debug.apk`.

Compiling for Android requires the Android SDK. Make sure to set the `ANDROID_HOME` path to point at the Android SDK directory.

2.2 Installation (Android)

Transfer the file to the device and open it using a package manager. Make sure that installing from unknown sources is enabled in the security settings.

OR

Install it using `adb install <package>` with the phone connected by USB in debugging mode.

2.3 Testing

The tests can be run by typing `sbt core/test`.

2.4 Program usage

Once the program is opened, the user is presented with a grid of screenshots of the saved games. If it is the first time that the user opens the game, there will be one screenshot containing a new game.

A game is started by tapping on the desired screenshot.

When the game is started the players take turns playing. The balls are divided into two colors: solids (numbers 1–7) and stripes (numbers 9–15). In the beginning, none of the players have any assigned balls and may shoot whichever ball they like. Once one of the players have pocketed a ball, they are assigned that color and should shoot at those.

If a player pockets manages to pocket a ball, without pocketing one of the opponents balls or the cue ball, the player may shoot again. Pocketing the black ball will cause the player to lose.

The game is won by pocketing all the assigned balls and then the black ball.

The cue stick is moved by dragging on the screen. It will always point towards the cue ball, but the rotation can be locked by pressing with a second finger (or by holding the left shift button on a desktop device). Shooting is done by dragging the finger over the cue ball.

In the upper-right corner a small ball with a target can be found. This ball is used to choose where on the ball to hit. For instance, choosing a low point on the ball will cause the ball to be shot with some back spin.

The game can be quit at any time by clicking the cross in the upper-left corner. This will automatically save the game and take the user back to the main menu.

3 Program structure

The game is divided into three parts: `core`, `desktop`, and `android`. The `core` folder contains the source code and tests that is shared across all platforms. Consequently, all the code related to the actual game is stored there. `desktop` and `android` contain device-specific launchers for running the game on the different devices.

The `core` project is divided into a set of packages. We will inspect these one by one.

3.1 Main package

The main package contains one class: `Eightball`. This class is instantiated by the device-specific launchers when the game is run. After the game has been launched, this class will create an instance of the main menu (which is separated to another class) and render it.

3.2 State

The `state` package contains classes related to storing and manipulating the state of the game.

To store the state of game the class `GameState` is used. It contains variables storing instances of the balls present on the board, who's turn it is, where the cue stick is etc. The variables are mutable.

The class also acts as a controller to some extent. For instance, the `nextRound()` method can be called to advance the game to the next state.

The GameState class has a companion object providing methods to load and save game states to the file system.

Manipulation of the state of the balls is separated into the class `PhysicsHandler` where `update` is the "magic" method. By passing a set of balls and a time step, the method will update the position, velocity, and angular velocity of the balls according to the events (collisions and friction) that will occur within that time step. Under the hood the method divides the task into a set of submethods performing individual, easily testable parts.

3.3 UI

The main menu and the game have been separated into two different classes: `MainMenuScreen` and `GameScreen`.

Both of these classes are views that retrieve their data from a `GameState` object.

To load the screenshots for the saved games in the `MainMenuScreen` class, the standalone `GameState` object is used to retrieve a set of screenshots and paths to the files storing the associated game state. Once a game is picked, the standalone `GameState` object is once again retrieve an instance of the saved game state. This object is then passed to a new instance of the `GameScreen` class.

The `GameScreen` class draws the game board, the balls, and the cue stick according to the state specified in the passed `GameState` object. This is done in the `render` method which is called approximately 60 times per second. Before rendering, the `update` method from the `PhysicsHandler` object is called to update the state of the balls.

When aiming, the `GameScreen` class acts as a controller manipulating the state of the cue stick in the `GameState` object. Once it determines that the player has attempted to shoot, the `shoot` method is called, passing the cue ball and the cue stick's velocity, to update the state of the cue ball.

Two UI elements are also drawn, a cross for exiting the game and a ball to choose where on the ball to shoot.

3.4 Objects

The `objects` package contains the renderable objects `Ball`, `Board`, and `CueStick`. They store their positions and dimensions as well as providing `render` methods to easily render themselves.

3.5 Math

The `math` package contains a `Vector3D` class representing vectors in 3D space. It also defines functions to manipulate vectors by implementing vector operations such as addition, subtraction, scaling, dot product, and cross product.

As the libgdx framework uses its own vector class, implicit definitions have been written to seamlessly pass `Vector3D` to libgdx methods and vice-versa.

3.6 Libgdx

This package contains two classes `SAction` and `SInputListener` to implement `libgdx` `Actions` and `InputListeners` in a Scala-like way. Instead of implementing interfaces to react to user input, as often done in Java, the programmer can pass the code as a function literal.

4 Algorithms

As previously mentioned, the `update` method within the `PhysicsHandler` object does the heavy lifting when it comes to simulating the balls.

When calling the `update` method the caller passes a `GameState` and a time step. The time step is the amount of time between each render call, which often is around 16 ms (as 60 is the target FPS of the game).

First, the algorithm updates both the angular and linear velocity. The linear velocity is affected by a force of friction at the point of contact. The change in velocity is calculated as follows:

$$\Delta\vec{v} = -\mu g \frac{\vec{v}}{|\vec{v}|}\Delta t \tag{1}$$

If the ball is sliding, the force of friction will also cause the ball to eventually start rolling. The change in angular velocity is calculated in the following manner, where r is the radius of the ball and R is a vector pointing from the middle of the ball to the touching point:

$$\Delta\vec{\omega} = \frac{5}{2}(\vec{R} \times (-\mu m g r \frac{\vec{v}}{|\vec{v}|}))\frac{\Delta t}{mr^2} \tag{2}$$

After updating the velocities, the time until the next collision(s) is/are determined. This is done analytically by assuming that the balls move linearly

within the time step (an assumption that can be made because of the small time step).

To determine when ball-ball collision occur, the lineary trajectories of all possible pair of balls are compared to determine if and when the distance between them is two times the radii of the balls. This is done by solving the following equation for t, where $\vec{p}$ designates a position, $\vec{v}$ a velocity, and r a radius[1]:

$$\vec{p}_{ab} = \vec{p}_a - \vec{p}_b \tag{3}$$

$$\vec{v}_{ab} = \vec{v}_a - \vec{v}_b \tag{4}$$

$$0 = t^2(\vec{v}_{ab} \cdot \vec{v}_{ab}) + 2t(\vec{p}_{ab} \cdot \vec{v}_{ab}) + (\vec{p}_{ab} \cdot \vec{p}_{ab}) - (r_a + r_b)^2 \tag{5}$$

The time until a ball-pocket collision is determined in the same manner where the pocket is represented by a circle with a radius slightly larger than the ball.

As the game only has horizontal and vertical walls, the time until wall collisions can be determining the distance along the x- or y-axis to a ball radius length from the wall. This value is then divided by the x- or y-component of the velocity along that axis.

When the next upcoming collisions are known, all balls are moved along their linear trajectories for the time t to align the balls for the collision.

Next, the new angular and linear velocities of colliding balls are determined. Ball-ball collisions are assumed to be perfectly elastic, preserving the total kinetic energy, momentum, and angular momentum.

To calculate the new linear velocity a collision plane is formed along the impact point between the balls. The balls will conserve their velocity component tangential to the collision plane while the velocity components perpendicular to the collision planes are swapped.

The following equation is used to determine the change in angular velocity, where $\vec{r}$ is a vector to the touching point, $\vec{v_{n_a}}$ the normal velocity component, $\vec{v_{t_a}}$ the tangential velocity, and $\vec{v_{p_R}}$ the relative velocity at the touching point[2]:

$$\Delta\vec{\omega_a} = \frac{5}{2}(\vec{r_a} \times (-\mu\frac{m\Delta\vec{v_{n_a}}}{\Delta t} \cdot \frac{\vec{v_{p_R}} + \vec{v_{t_a}}}{|\vec{v_{p_R}} + \vec{v_{t_a}}|}))\frac{\Delta t}{mR^2} \tag{6}$$

When a ball collides with a wall, it is assumed to be in contact with the wall during a certain time t. During this time, the friction between the contact point and the (possibly) rotating ball will dampen its angular velocity. The norm of the angular velocity vector will be shortened by the following amount:

$$\Delta|\omega| = \frac{1}{2}\mu v_t^2 \tag{7}$$

If the ball is spinning around the z-axis (up from the game board), the ball will also deviate slightly as a result of the collision. The loss in angular velocity in the z-axis is transfered into linear momentum along the x- or y-axis.

In some cases, multiple collisions may occur at the same time. In these cases all changes in collisions are calculated individually without applying the changes of the previously calculated collisions (otherwise the results would vary depending on the order their calculated in). When all changes in velocity have been calculated, the average for each ball is added.

Once the angular and linear velocities have been updated, the update method is called recursively for the time remaining in the time step. This is because multiple collisions at different times can happen within the same time step.

5 Data structures

The `GameState` class acts as a model storing the game state. All of the variables and collections in the game state are mutable. Balls are stored in a buffer, from which they are added and removed as the game state evolves. Besides that, the model also stores the current game state (aiming, rolling, or lost), optionally the player that has the solid balls, the cue stick, and boolean value determining whether it is player one's turn or not. Furthermore, it also stores boolean values regarding what balls that have been pocketed during this round.

Scores are not saved explicitly, instead there are methods to determine this from the state.

The class also provides some methods to manipulate the state, such as resetting it to its default state, pocketing a ball, and proceeding to the next round.

6 File formats

The purpose was to load and save game states as JSON-files. However, due to time constraints I did not have time implement saving and loading in this manner.

Instead, the `GameState` and the classes used in the contained variables extend the `Serialized` interface. To save a game state, the method `save` of the `GameState's` companion object is used to serialize the game state which is then written to a file. Similarly, loading is done by unserializing these files.

Unfortunately, the saved game states cannot easily be manipulated by hand as a result.

Screenshots are saved by storing the screens pixel color values to a PNG using a method provided by a libgdx library.

On desktop devices, the files are saved locally in the same directory as the executable in the folder `saves/`. On Android devices, the save files cannot be accessed.

7 Testing

As the game screens primarily act as views, they can rather easily be tested by visual inspection. The `PhysicsHandler` and `Vector3D`, on the other hand, have been thoroughly tested using unit tests.

The `PhysicsHandler` class has been implemented submethod by submethod in three steps. First, the behavior of a submethod was specified, what should it do? What parameters should it take? What should it return? Then, unit tests have been written to verify that the method has been implemented accordingly. Finally, the method has been implemented.

A similar approach was used for the `Vector3D` class, where the methods were implemented using the steps above as they were needed.

Writing unit tests for the controllers of the `GameState` class would in retrospect have been a good idea as well. For instance, there was some trouble with enumerations not unserializing properly that was not noticed until a couple of days after implementation.

8 Problems and shortcomings

8.1 Balls sticking to each other

In some cases, the balls get stuck in walls or in each other. This seems to be caused by the `update` method, in some cases, moving balls slightly into each other before updating the velocities during a collision. When examining one such situation, the distance between the balls was exactly two times the radius when calculating the value using a calculator. However, when determining the distance between the balls using the `Vector3D` class, the distance between the balls was slightly less than that. Thus, the problem is probably due to floating point imprecision.

A semi-working solution is to move the balls by 99% of the time step, which leaves a tiny bit of extra space between them. This fixes the problem in most cases, but they still do get stuck sometimes. A better solution would probably be to in reverse find the position so that the distance is exactly two times the radius for the program.

8.2 Odd physics behavior

The physics also seem to behave a bit oddly in some cases. For instance, balls often change direction the moment before they stop, as if it was a result from some odd angular spin (which you do not normally see in billiard games). As the game is in 2D, I did not figure out any good way to visualize the spin of the balls, making it hard to visually determine what is actually happening. While print lines do suggest that the spin often is rather realistic (or not at least extreme), there is something odd going on in some cases which I do not know the reason behind.

Sometimes the balls also get unrealistic amounts of spin, making them bounce along the walls towards a pocket.

Touching on the cue ball sometimes causes it to hit the ball with enormous speed.

8.3 Removing saved games

Games are automatically saved whenever the user hits the exit button, but there is no way to remove them from the game UI. This can be especially annoying on Android devices as there is no way to access the saved game files and remove them manually.

8.4 Memory management

`Libgdx` requires the programmer to dispose certain resources explicitly when they are not needed any more. This is not taken into consideration in this project and resources are left undisposed until the game is exited. While this is a major concern for larger games, the only images used in this game are the screenshots and the two in-game UI buttons. Besides that, everything is drawn using shapes. Thus, the game should not crash because of this at least..

9 The good and the bad

9.1 Separation of the model

Modeling the game state in a separate class proved to be a good choice. By doing this, the game screen remained relatively uncluttered and made serializing the game state easy.

9.2 Vector3D

The `Vector3D` class is more convenient to use than the `Vector3` class provided. For instance, adding two vectors and multiplying them by a scalar using the provided `Vector3` class is done by writing `vector1.add(vector2).scl(3)`. The same can be expressed in a more readable way using the `Vector3D` class: `3 * (vector1 + vector2)`.

9.3 Unintuitive shooting on mobile

When shooting, the tip of the cue stick is positioned at finger tip. This makes it hard to determine in what direction the ball will shoot as the finger is blocking the line of sight.

9.4 Setting spin has marginal effect

While the user may determine where on the ball the cue stick should hit, it has barely any conceivable impact on the game. That part is also implemented in a physically unrealistic manner where the angular spin is set linearly depending on the distance from the center and the velocity of the cue stick.

Links

[1] The Two-Bit Coder, *Circle Collision Detection - Linear Trajectories*, Accessed 2016-04-14 http://twobitcoder.blogspot.fi/2010/04/circle-collision-detection.html

[2] Brian Townsend, *An Attempt at a Real-Life Simulation of the Mechanics of Billiard Balls*, Accessed 2016-02-11, http://archive.ncsa.illinois.edu/Classes/MATH198/townsend/