Bitte lösen Sie Aufgaben 33-34 bis zum 19.07.2020 **und geben Aufgabe 33 ab!**

Aufgabe 33

Billard zu zweit

Wir optimieren unser Billard-Spiel und implementieren erste Regeln. Erstellen Sie zunächst das Paket ***aufgabe33*** als Kopie von *aufgabe31*.

Optimierter Spielstart und Speicherbedarf

Unser Billard Spiel liest sage und schreibe **249** Klang-Dateien ein!

- 1-mal atmo_bar.mp3
- 16-mal billard_kugel_rand.mp3
- 16-mal billard_kugel_queue.mp3
- 120-mal billard_kugel_kugel.mp3
- 96-mal billard_kugel_loch.mp3

Deshalb dauert bei einem Rechner ohne SSD das Laden des Spiels sehr lange. Sie können das überprüfen, indem Sie im Konstruktor der Klasse ***Klang*** *system.out.println(filname);* einfügen.

Zur Verbesserung machen wir die entsprechenden Variablen *static*, so dass jeder Sound nur einmal vorkommt, also nur **fünf** Sound-Dateien geladen werden. Nur wenn viele Kugeln gleichzeitig aneinanderstoßen, hören wir überhaupt einen Unterschied.

In ***ElastischerStoss*** sparen wir die ersten 119 Dateien ein:

- Fügen Sie die **Klassen**variable *klang* hinzu und initialisieren sie gleich bei der Definition:
  `private static Klang klang = new Klang("/sound/billard_kugel_kugel.mp3");`
- geben Sie beim Aufruf von ***super*** für den Klang die statische Variable *klang* weiter,
- spielen Sie am Ende von reaktion den Sound ab: *klang.play(0.1*f);*

In ***BillardLochReaktion*** sparen wir auf die gleiche Weise nochmal 96 Dateien ein, rufen Sie dort *play(1.2)*, denn der Sound wenn die Kugel im Loch verschwindet ist unabhängig von der Geschwindigkeit.

Weitere 32 Dateien sparen wir in ***BillardKugel***, wenn wir die Variable ***queueKlang*** *static* deklarieren und gleich bei der Definition initialisieren und eine weitere statische Klang variable für den Klang beim Aufprall auf den Rand verwenden, die wir mit super an den Basisklassenkonstruktor weitergeben.

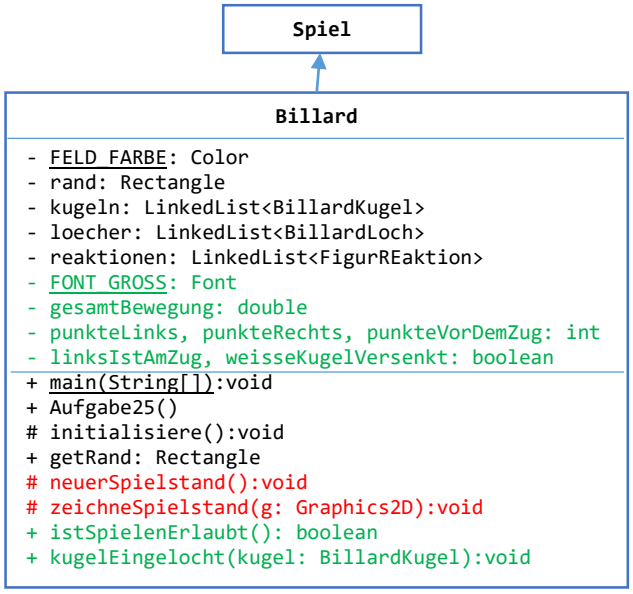
Jetzt sollte das Spiel schneller laden und braucht deutlich weniger Speicher, da nur noch fünf MP3-Dateien gelesen werden müssen.

Einfache Billard Regeln

Wir implementieren folgende Regeln:

- Es darf nur gestoßen werden, wenn alle Kugeln zur Ruhe gekommen sind.
- Wenn die weiße Kugel versenkt wird, wechselt der Spieler.
- Wenn ein Spielzug zu Ende geht, ohne dass eine Kugel versenkt wurde, wechselt der Spieler.

Um die diese Regeln zu programmieren, erweitern wir die Klasse ***Billard*** wie rechts gezeigt. Die grünen Member kommen hinzu, die roten werden erweitert.



Die Variable *FONT_GROSS* wird mit `("SansSerif",Font.BOLD,25)` initialisiert.

Die Methode ***istSpielenErlaubt*** gibt *true* zurück, wenn *gesamtBewegung* keiner als 0.1 ist (alle Kugeln ruhen).

In BillardKugel prüfen wir in ***mousePressed*** und ***mouseReleased***, ob Spielen erlaubt ist, bevor wir den Queue aktivieren:

```
if(((Billard)spiel).istSpielenErlaubt()) {
```

hier verwenden wir einen **Cast** um dien *SpielFigur* geerbte Variable *spiel* in Billard zu wandeln.

Die neue Methode ***kugelEingelocht*** in **Billard** macht folgendes:

- Falls es sich um die weiße Kugel handelt, wird *weisseKugelVersenkt* auf *true* gesetzt.
- Andernfalls wird der Punktestand (*punkteLinks* bzw. *punkteRechts*) für den gerade aktiven Spieler (*linksIstAmZug* abfragen) hochgezählt.

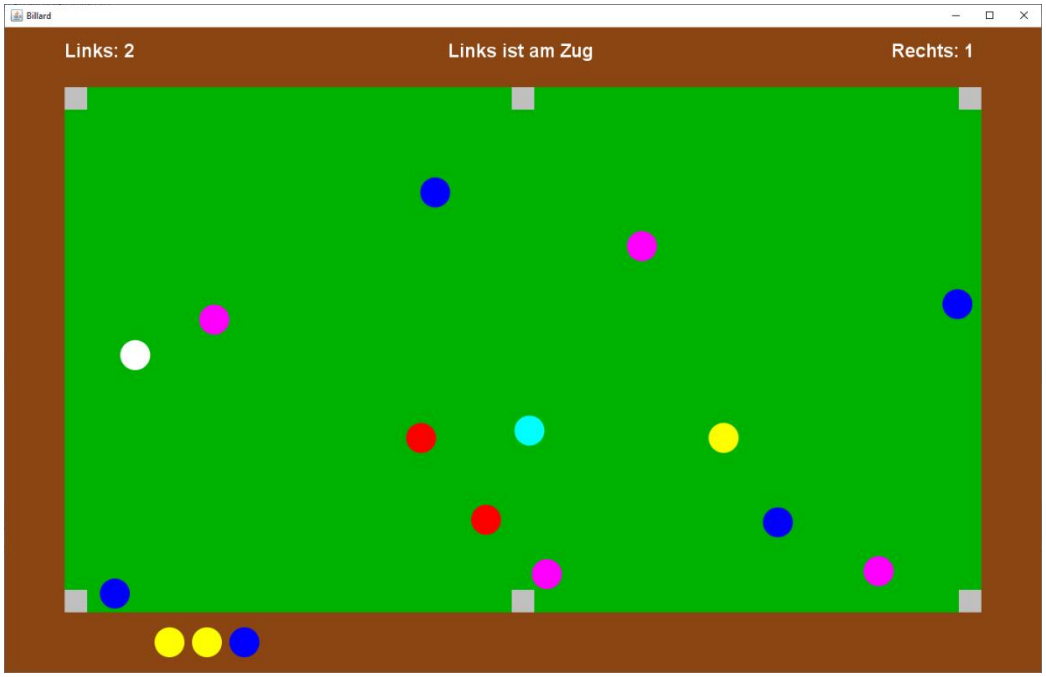
In der Klasse ***BillardLochReaktion*** rufen wir am Anfang von ***reaktion*** *spiel.kugelEingelocht* auf, dazu müssen wir wieder einen **Cast** verwenden: `((Billard)spiel).kugelEingelocht(kugel);` .

Die Methode ***neuerSpielstand*** in Billard erweitern wir wie folgt:

- Nach der Reaktion der Figuren merken wir uns den Wert von *gesamtBewegung* in der lokalen Variable ***bewegungVorher***.
- Dann setzten wir *gesamtBewegung* auf 0.0. und summieren in einer Schleife die Bewegungen aller Kugeln in *gesamtBewegung* auf:
  `for(BillardKugel f:kugeln) gesamtBewegung +=f.getBewegung().distance(0,0);`
- Am Ende der Methode prüfen wir, ob gerade ein Spielzug zu Ende ist, das definieren wir als `(bewegungVorher>=0.1 && gesamtBewegung<0.1)`
  Wenn ein Zug zu Ende ist:
  - Prüfen wir, ob der Spieler wechselt: das ist der Fall, wenn die weiße Kugel versenkt wurde **oder** wenn im letzten Zug kein Punkt gemach wurde, d.h. wenn die Summe der Punkt beider Spieler gleich der Punkte vor dem Zug (*punkteVorDemZug*) ist. Um den Spieler zu wechseln ändern wir den Wert von *linksIstAmZug*.
  - Dann setzen wir *punkteVorDemZug* auf die Summe der Punkte der beiden Spieler und *weisseKugelVersenkt* auf false.

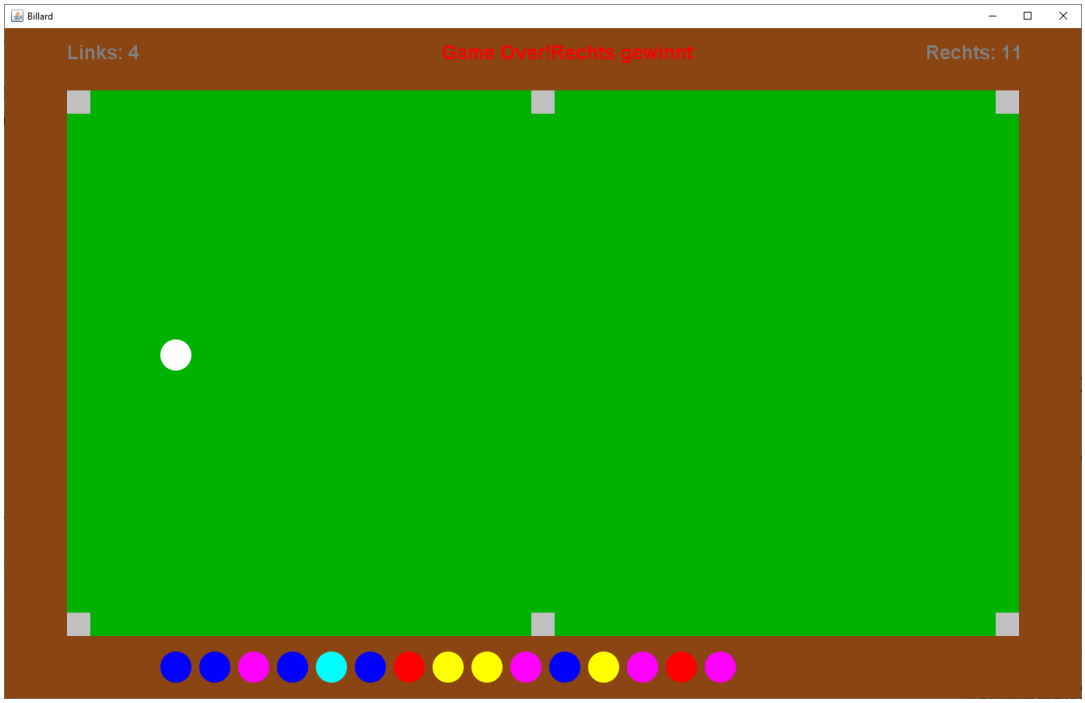
In ***zeichneSpielstand*** zeigen wir am oberen Rand des Spielfeldes links und rechts die Punkte der beiden Spieler an. In die Mitte schreiben wir, welcher Spieler gerade am Zug ist. Dazu setzen wir die Farbe auf Weiß, wenn Spielen erlaubt ist und auf Grau, solange kein Zug möglich ist. Das ist ein schönes Beispiel für den Bedingungsoperator: g.setColor(istSpielenErlaubt()?Color.WHITE:Color.GRAY).

Dann setzen wir den Font auf *FONT_GROSS* und rufen dreimal die Methode *g.drawString*. Auch bei der Darstellung, wer gerade am Zug ist, bietet sich der Bedingungsoperator an.



Eine weitere Verbesserung ist die Anzeige von „Game Over“:

- Geben Sie in ***istSpielenErlaubt*** nur dann *true* zurück, wenn die Kugeln stehe**n und die Summe der Punkte kleiner als 15 ist.**
- Fragen Sie in ***zeichneSpielstand*** ab, ob die Summe der Punkte kleiner ist als 15. Wenn ja, schreiben Sie wie bisher „Links ist am Zug“, ansonsten Game Over und wer gewonnen hat:



Jetzt kann man das Spiel zu zweit spielen. Es fehlt noch eine schönere Grafik und wir haben ein ganz passables Computerspiel.

Aufgabe 34

Bits und Bytes

Sie fahren zu einem Bekannten und bringen ihm eine DVD mit 4,5 Gigabyte Daten. Ihre durchschnittliche Geschwindigkeit beträgt 72 km/h. Bis zu welcher Entfernung haben Sie eine höhere Datenrate als eine Internetverbindung mit 6 Megabit/Sekunde?