CT-Projekt:
Pi-Collision

Tobia Ippolito

8. Januar 2020

Inhaltsverzeichnis

1 Tagesberichte

1.1 Erste Schritte zu dem Projekt (18.09.2019)

Dies ist der erste Tag des Projektes, beziehungsweise der Vorbereitungsphase.
Ich bin die verschiedenen Projektvorschläge durchgegangen. Zudem habe ich auch eigene Ideen überdacht. Da meine Ideen meist etwas zu ambitioniert/aufwendig sind, wollte ich mich bei diesem Projekt für einen fremden Projektvorschlag entscheiden.
Zur Auswahl stand das Erzeugen einer zufälligen Zahl, das Programmieren einer physikalische Engine (welche die Kollisionen von zwei Vierecken berechnet) und das Programmieren von einer Abi-Aufgabe.
Dazu sollte man erwähnen, dass Aufgaben für/von dem Abitur meist eher seltsam und unpraktisch konzeptioniert sind. Daher stelle ich mir dieses Projekt recht witzig vor. Jedoch stelle ich mir dieses Projekt auch sehr langweilig vor. Schließlich muss ich dabei nichts programmieren, sondern nur abschreiben.
Das Erzeugen von zufälligen Zahlen finde ich sehr interessant und werde mich damit aufjedenfall beschäftigen, jedoch nicht in diesem Projekt.
Ich entschied mich für das Programmieren der Physikengine. Ich denke dieses Projekt ist sehr spannend und ich werde (hoffentlich) viel dabei lernen.

1.2 Konzeptionierung (25.09.2019)

Heute ist der erste Tag, an dem ich 'wirklich' angefangen habe an dem Projekt zu arbeiten.
Ich habe mir ein Konzept gemacht, wie das Projekt aussehen soll. Dieses kann man in einem anderen Abschnitt ansehen. Dieses Konzept werde ich nach und nach ausbessern (Details hinzufügen). Dieses Konzept habe ich dann noch initiiert. Heißt: Ich habe die entsprechenden Klassen schon mal erstellt. Momentan sind diese jedoch ziemlich überschaubar (leer).
Es gibt eine Start-Klasse, eine Klasse zum Steuern des Programms, eine Klasse für die Grenzen, eine für die Vierecke (welche sich später bewegen sollen). Die Start-Klasse ist logischerweise die Main-Klasse. Aber ich habe schon geplant mehr Klassen hinzuzufügen. Beispielsweise eine Klasse zum Laden von Bildern und eine Klasse zum Abspielen von Sounds. Wahrscheinlich kommen aber noch mehr Klassen hinzu, welche ich noch nicht auf dem Schirm habe.
Ich habe mich außerdem dazu entschieden keine anderen 'Pi-Collision'-Projekte anzusehen. Ich gehe davon aus, dass es recht viele Referenzen in diesem Bereich gibt, jedoch will ich mich verbessern und das Programm selbst konstruieren. Mein Ergebnis sieht dann höchstwahrscheinlich schlechter aus, aber meine Fähigkeiten profitieren davon.

1.3 Arbeiten am Grundgerüst (02.10.2019)

Heute habe ich mit der Programmierung der zuletzt erstellten Klassen begonnen. Ich habe es geschafft das Grundgerüst zu programmieren und damit steht schon mal der wichtigste Teil des Programms. Ich habe zum ersten Mal eine Schnittstelle verwendet. Dies finde ich wirklich überzeugend, da ich ziemlich einfach andere Objekte einbinden kann.
Ich bin mit dem Gerüst zufrieden und habe Zuhause die Start-Klasse und die Klasse für die Vierecke geschrieben. Auch die Wall Klasse wurde verfasst.
Das Programm läuft, leider habe ich aktuell noch das Problem, dass sich die Objekte aus dem Rand bewegen und dass die Performance etwas schlecht ist.

Außerdem habe ich mit der Konzeptionierung der Methode zum Berechnen der Kollision begonnen und möchte diesen recht 'realistisch'. Bedeutet, ich habe in meinen alten Physik Unterlagen gekramt und schließlich den unelastischen Stoß inklusive dessen Formel gefunden.

1.4 Jetzt kommt die Physik ins Spiel! (16.10.2019)

Heute habe ich mithilfe des Internets meinen Game-Loop etwas modifiziert und nun läuft dieser performanter. Zusätzlich habe ich die Formel eingebunden. Diese hat erst nicht funktioniert. Ich hatte nämlich Schreibfehler, da die Formel im Editor nicht so übersichtlich aussieht. Nun funktioniert sie aber.
Ich habe noch das Problem, dass man das Programm nicht einfach stoppen kann. Um dieses Problem kümmere ich mich aber erst später. Um das Problem wenigstens einzudämmen habe ich einem Durchgang eine zeitliche Begrenzung gegeben.
Ich habe heute noch ein Menü hinzugefügt und Kästchen, in denen man die Werte ändern kann. In Zukunft kann ich hier sehr viel Content hinzufügen.

Zuhause habe ich eine Klasse zum laden von Bildern hinzugefügt und eine entsprechende Option im Menü hinzugefügt. Sehr cool ist, dass ich Bilder neben der Auswahl eingesetzt habe. Dies war ein guter Einfall, denn es sieht echt nice aus. Aber ich habe noch keine Bilder, sondern nur das Viereck in anderen Farben. Ich muss also noch schauen, wie ich die neu-erstellte klasse verwende.
Ich habe Zuhause auch die Sichtbarkeit von Werten verändert. Diese kann man nun im Menü verändern.

1.5 Neue Features sind nicht immer gut (13.11.2019)

Ich habe heute eine Verwendung für die Klasse zum Laden von Bildern gefunden. Ich habe nämlich Pfeile hinzugefügt, welche die Richtung mit verändern. Dies ist über das Menü veränderbar.
Zudem habe ich das Design verändert, da ich ein Koordinatenfeld hinzugefügt habe um die Daten aufzuzeichnen. Dies klappt eigentlich ganz in Ordnung. Ich finde es wirkt etwas instabil. Ich spiele mit dem Gedanken es lieber wegzulassen, auch wenn es mich viel Zeit gekostet hat.
Zudem habe ich heute hinzugefügt, dass es anzeigt, wie oft es in einer Sekunde updatet (Frames-Per-Second).

Zuhause habe ich dann noch die Klasse 'Timebar' hinzugefügt. Diese Klasse soll anzeigen, wann der Durchgang endet. Die Idee hinter der Klasse ist simple und ist mir im Alltag eingefallen. Ich habe einfach zwei Rechtecke. Sie liegen auf der selben Position. Unterschied ist, dass das eine Rechteck nicht ausgefüllt ist und nur einen Rand hat. Dieses stellt den Rand dar.
Das andere Rechteck ist ein ganz normales, ausgefülltes Rechteck, nur dass sich seine X-Werte ständig ändert. Heißt so viel wie: Das Rechteck verändert seine Breite in Abhängigkeit von Variablen, welche zum stoppen des Durchgangs verwendet werden.

1.6 Geht das auch ohne Grafiken? (20.11.2019)

Heute habe ich einen Modus hinzugefügt, bei dem nichts gezeichnet wird und nur updatet. Ich habe den Loop geändert und es klappt recht gut. Jedoch könnte man ihn wohl etwas besser programmieren. Trotzdem werde ich diesem Modus nicht mehr Zeit zur Verfügung stellen, da er nur eine nette Dreingabe ist.
Heute habe ich außerdem eine Musik Klasse für die Musik/Sounds hinzugefügt. Im Menü kann man den Sound bei einer Kollision ändern. Dies hat viel Zeit erfordert.

Zuhause habe ich die Klasse Timebar überflüssig gemacht. Ich habe nämlich einen Algorithmus entworfen, welcher einen Durchgang in Abhängigkeit von der letzten Kollision beendet.
Dies bedeutet, dass ein Durchgang beendet wird, wenn eine bestimmte Zeit lang keine Kollisionen stattfinden. Eine eindeutig bessere, aber noch nicht perfekte Lösung.
Denn nun endet zwar ein Durchgang und zwar meist zur Rechten Zeit (die letzte Kollision wird manchmal nicht abgewartet, dafür muss man in anderen Momenten nicht so lang warten), aber während der Durchgang läuft kann man nichts machen.
Diesem Problem möchte ich mich nächstes Mal widmen.

1.7 Ein separater Arbeitstisch (27.11.2019)

Heute habe ich mich dem Problem von letztem Mal gestellt und habe auch eine Lösung gefunden. Diese wollte jedoch sehr lange nicht funktionieren. Am Ende habe ich sie dann doch zum Funktionieren gebracht. Ich führe nun die Methode 'start' in einem neuen Thread aus. Dadurch kann ich es Parallel ablaufen lassen. Ein Kolleg hat mich auf diese Lösung gebracht und darüber bin ich ziemlich glücklich.
Es gibt nur ein Problem, denn wenn ein Durchgang läuft und ich einen neuen starten will, klappt es nicht und spackt sehr seltsam rum. Ich finde dies seltsam, da ich es nach meiner Meinung gut gemacht habe. Denn ich schließe den aktuell laufenden Thread und starte ihn neu. Ich habe zuhause sehr viele Varianten ausprobiert. Aber aus irgendeinem Grund macht es immer noch so komisch. Aber ich widme diesem Problem erst mal keine Zeit, da ich wichtigeres zu lösen habe.
Ah und es gibt noch einen komischen Fehler, welcher mir nun aufgefallen ist. Wenn ich eine Variable über Swing ändere, legt sich Canvas über die Anzeige des Menüs. Es wirkt als würde Swing den Fokus beanspruchen, aber ich weiß nicht wie ich das ändere. Aber auch das ist nur ein kleines Problem mit wenig Priorität.

1.8 Performanteres Herz (04.12.2019)

Heute habe ich den Game-Loop etwas angepasst. Nun zeichnet das Programm öfters, als das es Updatet. Zwar ist das Updaten eindeutig schneller als das Zeichnen, jedoch ist genau das das Problem. Wenn man genauso viel Zeichnet wie Updatet, läuft das Programm eindeutig zu schnell ab. Man muss diesen die Geschwindigkeit zurückfahren. Wenn das Programm oft zeichnet ist das nicht schlimm, denn dadurch sieht es nur ruckelfreier aus. Durch das Updaten passieren schließlich alle Handlungen und deswegen muss dieser auch seltener ausgeführt werden.
Ich habe das so geregelt, dass der Update-Loop nur dann ausgeführt wird, wenn er langsamer(x) als gedacht ist. Heißt: Ich habe eine Zeit festgelegt und solange soll das Updaten dauern. Falls es mal länger braucht zum Updaten heißt das, dass es gerade etwas zum rechnen gibt und deswegen darf der Loop noch einmal ausgeführt werden.
Es ist kein perfekter Loop, aber ich finde ihn ganz in Ordnung. Ich habe auch die FPS Berechnung angepasst und nun sehen die Zahlen auch besser aus.
Der vorherige Game-Loop war noch einmal Primitiver, aber ging im Kern auch in die gleiche Richtung und hatte den selben Kerngedanken. Hier wurde er sozusagen erweitert oder besser in die Praxis umgesetzt.
Ich habe den Tag nur damit verbracht. Ich habe viel herumexperimentiert und mich auch im Internet anreichern lassen. Ich habe geschaut wie viel Zeit was in Anspruch nimmt und habe das Konzept überarbeitet. Auch wenn das Ergebnis noch verbesserungswürdig ist, so habe ich einige Schritte in die richtige Richtung gemacht und etwas dabei gelernt.
Mein Ziel wäre es (wenn ich mehr Zeit hätte), dass der Update-Loop beim Aufprallen von den Rechtecken im kritischen Bereich besonders viele Durchläufe hätte. Ich habe mir dazu auch Gedanken gemacht, aber noch keine Überzeugende Lösung gefunden. Meist mangelt es dann zu Beginn oder am Schluss, denn ich muss mir auch überlegen, wie oft der Update-Loop dann durchgeführt werden soll. Ich denke mit mehr Zeit oder mit einer Recherche im Internet würde ich dafür eine gute Lösung finden.
Zuhause habe ich dem Programm noch Content, wie neue Sounds und andere Designs (sehen schrecklich aus, aber der Grundgedanke ist gut und mit etwas mehr Zeit würden diese auch ein nettes Feature abgeben -> Ich habe jedoch keine Zeit :P) hinzugefügt.

1.9 'The Purge' (18.12.2019)

Heute gab es eine große Säuberung. Alle alten Klassen und Methoden, sowie alte Variablen wurden ohne Gnade entfernt.
Mittlerweile haben sich sehr viele nicht mehr verwendete Dinge angestaut, wie eben Klassen, Methoden und Variablen. Diese bereiten Chaos in meinem Quellcode und somit hatte ich keine andere Wahl als sie zu beseitigen. Diese Tat hat auch mehr Zeit in Anspruch genommen als mir lieb ist. Was wohl noch einmal zeigt wie viel 'Müll' noch herumgelegen ist. Ich denke ich würde jetzt immer noch ein paar unnötige/nicht verwendete Dinge finden, aber es ist VIEL weniger geworden.

Außerdem habe ich einen kleinen Kniff vollzogen. Ich habe mich nochmals mit dem Problem befasst,

dass sich die Objekte durch die Wand durch boundsen. Dies sollte eigentlich nicht gehen und ich habe dies schon versucht zu verhindern, jedoch hat dies nie geklappt. Am Schluss war entweder alles wie zuvor oder es klappte, aber dafür die 'Pi-Collision' nicht mehr. Also entweder hat sich nichts verändert oder die Kollision wurde seltsam.
Jetzt da das Projekt zu Ende ist musste eine Lösung her. Da die Kollisionen korrekt sind und es sich vielleicht nur um einen grafischen Fehler handelt, habe ich die Grafik etwas manipuliert.
Wenn die Rechtecke in kritische Bereiche eintreten, wird ihre Position anders gezeichnet. Und zwar mit festgelegten Werten. Somit sieht das Ergebnis besser aus und ist nicht manipuliert (nur die Grafik ein bisschen). Ich hätte das auch weglassen können, aber so ist es schöner.
Über dieses Problem hatte ich auch mit Kollegen geredet. Sie meinten ich solle das Prüfen der Kollision nicht auf einen Wert beschränken, sondern einen Bereich angeben (anstatt '==' '>=' verwenden). Dies habe ich jedoch schon von Anfang an so gemacht, denn ich versuche 'defensiv' zu programmieren. Also so viele Fehler wie möglich zu verhindern, obwohl die erwarteten Werte dies überflüssig erscheinen lassen.

1.10 Grobe Zusammenfassung

Am Anfang war nur eine Idee, welche immer weiter konzeptioniert wurde. Zu Beginn des Programmierens waren die Scheinwerfer auf das Grundgerüst gerichtet. Dieses musste gut sein, davon ist ein Haus wie ein Programm abhängig. Mein Grundgerüst wurde zufriedenstellend und Schnittstellen fanden nützliche Verwendung darin.
Danach fanden Objekte Platz darin und man konnte es durch eine andere Klasse starten (und initiieren). Die Objekte Interagierten miteinander und eine Verbesserung des Herzens (des Game-Loops) war erforderlich. Dies sollte nicht die letzte Änderung dessen sein.
Nun kamen immer mehr Features (wie das Menü und dessen Unterpunkte). Aber auch so manche Features mussten das Programm verlassen. Sie waren entweder nicht gut genug oder sie wurden durch bessere ersetzt.
Natürlich wurden all diese Schritte von Problemen begleitet, welche entweder gelöst, nicht gelöst, teilweise gelöst, nicht beachtet oder vertuscht wurden.
Zuletzt wurde der Quellcode bereinigt.

2 Quellcode - Klassen

Ich werde kurz etwas zu der allgemeinen Struktur meiner Klassen sagen und dann im Detail die Klassen begutachten.
Den Kern meines Programms bildet die Steuerklasse 'PhysicEngine'. Alle Handlungen gehen von dieser Klasse aus. Die 'Start'-Klasse initiiert alle Objekte und übergibt dann das Wort einem Objekt von der erwähnten Steuerklasse.
Für den Kern meines Programms ist das Interface 'PhysicalObject' von großer Bedeutung. Ich benutze es um der Steuerklasse Objekte hinzuzufügen. So stelle ich sicher, dass alle Objekte, die in einer ArrayList in der Steuerklasse sind, Methode haben, welche in der Steuerklasse ausgeführt werden. Also wie eine Schablone. Ohne sie wüsste das Programm nicht sicher, ob alle Objekte in der ArrayList so eine Methode haben...Es würde ohne also nicht funktionieren.
Dann habe ich noch eine Klasse für die Rechtecke und für die Wände. Diese haben die passende Schablonen-Form. Also sie besitzen alle nötigen Methoden um gezeichnet zu werden.
Die Sprite und MusicLoader Klasse sind für das Laden von Bildern und Sounds verantwortlich. Da diese in meinem Projekt nicht unbedingt nötig sind, sind diese Klassen auch nur eine nette Dreingabe.
Eine wichtige Rolle spielt die 'CollisionProcess'-Klasse, welche einen Durchgang darstellt. Sie sagt dem Thread was er tun soll. In diesem Fall soll er einen Durchgang durchführen. Dadurch habe ich die Möglichkeit einen Durchgang, sowie das Userinterface (Start-Button,...) laufen zu lassen. Somit ist nicht mein ganzes Programm in der While-Schleife gefangen und man kann zum Beispiel das Programm schließen.
Dafür besitzt ein Objekt dieser Klasse das Objekt von der Steuerklasse. Damit es dessen Methode aufrufen kann.
Das ist die grundlegende Struktur meines Quellcodes.

2.1 Start

Diese Klasse ist zwar nicht der Kern meines Programms, aber hier beginnt alles.
Hier werden alle meine Objekte erzeugt, initiiert und miteinander verknüpft, denn viele Objekte brauchen andere Objekte für spezielle Dinge.

```
import java.awt.Color;

public class Start {

        public static void main(String[] bvsus) {

                //2 Rechtecke – Erzeugung und Konfiguration
                RectangleObject r1 = new RectangleObject();
                RectangleObject r2 = new RectangleObject();

                r2.setX(200);
                r2.setV2(0.0d);
                r2.setColor(Color.CYAN);
                r1.setColor(Color.BLUE);
                r1.setM(100);
                r1.setGrenzwertR(150);

                //zwei Mauern – Erzeugung und Konfiguration
                Wall leftW = new Wall();
                leftW.setY(75);
                leftW.setX(95);
                Wall bottomW = new Wall();
```

```
                bottomW.setHeight(5);
                bottomW.setWidth(300);
                bottomW.setY(325);

                //Thread-Klasse, braucht das Objekt von PhysicEngine
                CollisionProcess cp = new CollisionProcess();

                //physicEngine - Erzeugung und Konfiguration
                PhysicEngine engine = new PhysicEngine(r1, r2, bottomW, leftW, cp);
                engine.addRendering(r1);
                engine.addUpdating(r1);
                engine.addRendering(r2);
                engine.addUpdating(r2);
                engine.addRendering(leftW);
                engine.addRendering(bottomW);

                cp.init(engine);

                engine.init();
        }

}
```

Die Game-Objekte werde dann noch in einer ArrayList hinzugefügt, welche sich in der PhysicEngine (Steuerklasse) befinden.
Eine Liste für die Objekte zum Updaten und eine für die Objekte zum Zeichnen. Die Wände/Begrenzungen müssenbeispielsweise nicht geupdatet werden, da ihre Position konstant ist.

2.2 PhysicEngine

Hier befinden wir uns im Herz des Quellcodes. Diese Klasse stellt die Steuerklasse dar und von ihr geht nach dem Start alles aus.
Deswegen hat sie auch von jeder Klasse ein Objekt, außer von 'Sprite' und von 'MusicLoader', da diese nicht erzeugt oder schon in der Klasse erzeugt werden.

Zu aller erst kommen die Imports und die Variablen. Beides werde ich nicht zeigen, da sie viel Platz einnehmen (94 Zeilen) und trotzdem nicht so viel aussagen.
Der Konstruktor sieht wie folgt aus und weißt die entgegengenommenen Parameter seinen deklarierten (aber noch nicht erzeugten) Variablen/Objekten zu.

```
//Konstruktor
public PhysicEngine(RectangleObject r1, RectangleObject r2, Wall bottomW,
                                        Wall leftW, CollisionProcess cp) {
        this.r1 = r1;
        this.r2 = r2;
        this.bottomW = bottomW;
        this.leftW = leftW;
        this.cp = cp;
}
```

Noch einmal erwähnen wollte ich, dass die Klasse 'CollisionProcess' ein Interface namens 'runnable' besitzt. Dadurch kann man ein Objekt von dieser Klasse einem Thread übergeben, aber weiteres in dem Abschnitt 'CollisionProcess'. Ich finde nur, dass der Name nicht so erklärend/passend ist...aber ich bin zu faul ihn zu ändern.

Als nächstes kommen wir zu den Methoden der Klasse 'PhysicEngine'. Dabei sind die ersten beiden für das hinzufügen von Objekten zu einer ArrayList gedacht. Diese findet wiederum in update() und render()

Verwendung.
Die hinzugefügten Objekte müssen jedoch einer Norm entsprechen, also bestimmte Dinge erfüllen. Dies bedeutet sie folgen einer Schnittstelle. Und damit weiß ich, dass alle Objekte in einer der beiden ArrayLists bestimmte Methoden besitzen.

```
//Methoden
public void addUpdating(PhysicalObject phyObj) {
        updating.add(phyObj);
}

public void addRendering(PhysicalObject phyObj) {
        rendering.add(phyObj);
}
```

Die nächste Methode ist die 'init()'-Methode und ist sehr groß. Deswegen werde ich sie nur teilweise darstellen.
Sie erzeugt all die Swing/AWT-Komponenten und initiert auch andere Dinge. Beispielsweise lädt sie die Sounds.
Besonders ist auch, dass ich diese Methode verwende: 'EventQueue.invokeLater(()[...]);'. Diese sorgt dafür, dass die Befehle, welche darin stehen nach und nach durchgeführt werden. Ich habe diese Methode benötigt, da es Probleme mit der grafischen Oberfläche (Swing) gab. Es scheint nämlich nicht möglich, dass zwei Dinge auf die grfaische Oberfläche zugreifen und somit kommt dieser Befehl ins Spiel.
Durch diesen Befehl, müssen sich konkurrierende Zugriffe in eine Warteschlange reihen und werden nach und nach abgearbeitet. Dies habe ich aus einem Udemy-Kurs über Swing gelernt.
Bei AWT scheint es dieses Problem nicht zu geben.

In dieser Methode ('init()') werden die Swing-Komponenten einem Container von dem Fenster hinzugefügt.
Viel Platz nimmt übrigens den JMenuBar ein.

```
public void init() {
        //Sounds werden geladen
        click0 = new MusicLoader();
        click0.loadPackage(songName0);
        click1 = new MusicLoader();
        click1.loadPackage(songName1);
        click2 = new MusicLoader();
        click2.loadPackage(songName2);
        click3 = new MusicLoader();
        click3.loadPackage(songName3);
        click4 = new MusicLoader();
        click4.loadPackage(songName4);


        EventQueue.invokeLater(() -> {          //regelt die Zugriffe
                //Fenster und Canvas wird erzeugt und Konfiguriert
                window = new JFrame("Pi Collision");
                window.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                window.setResizable(false);
                window.setSize(width, height);
                window.setLocationRelativeTo(null);
                window.setVisible(true);

                Container contentPane = window.getContentPane();

                //start Button
                start = new JButton(new ImageIcon("src/images/start-png-44882.png"));
                start.setSize(100, 100);
                start.setLocation(600, 75);
```

```
                start.addActionListener(new ActionListener() {
                        public void actionPerformed(ActionEvent e) {
                                if(!run) {
                                        lastTime = System.currentTimeMillis();

                                        run = true;
                                        thread1 = new Thread(cp);
                                        thread1.start();

                                        firstStart = false;
                                }
                        }
                });

                [...] -> andere Swing-Komponenten


                //Menu-Bar-Init
                //Menu: JmenuBar -> Jmenu -> JMenuItem
                [...]
                window.setJMenuBar(jMenuBar);  //komplette MenuBar dem Fenster adden
                //Menu Ende

                //Komponeten hinzufuegen (dem Container des Fensters)
                contentPane.add(vLabelR1);
                contentPane.add(vR1);
                contentPane.add(vR1Slider);
                contentPane.add(vLabelR2);
                contentPane.add(vR2);
                contentPane.add(vR2Slider);
                contentPane.add(weightLabelR2);
                contentPane.add(weightR2);
                contentPane.add(weightR2Slider);
                contentPane.add(weightLabelR1);
                contentPane.add(weightR1);
                contentPane.add(weightR1Slider);
                contentPane.add(start);
                contentPane.add(renderer);

        });
}
```

Die nächste Methode wird jedes Mal vor dem starten eines Durchgangs (nicht einen Schleifendurchgang, sondern einen kompletten Durchgang durch die Methode) ausgeführt. Die Methode liest dabei den Input des Benutzers ein. Dieser kann schließlich das Gewicht und die Geschwindigkeit beider Rechtecke verändern. Zudem kann der Benutzer dies entweder durch JSlider oder durch JTextFields tun. Dies muss aber erst abgefragt werden (der Modus muss abgefragt werden).
'InputMode' ist eine Variable vom Datentyp Boolean und gibt den Modus zurück.

```
public void setInstances() {
        if(inputMode) {         //true -> JTextField-Mode
                r2.setM(weightR2.getText());
                r2.setV(vR2.getText());

                r1.setM(weightR1.getText());
                r1.setV(vR1.getText());
        }else {                 //false -> JSlider-Mode
                r2.setM(weightR2Slider.getValue());
```

```
                r2.setV(vR2Slider.getValue());

                r1.setM(weightR1Slider.getValue());
                r1.setV(vR1Slider.getValue());
        }
        //wird von der update-loop-Abfrage gebraucht
        nextGameTick = System.currentTimeMillis();
}
```

In der nächsten Methode geht es um die FPS.
Diese Methode wird nur jede Sekunde durchgeführt. Dies wird durch die Variable 'timeAtLastFPSCheck' und der aktuellen Zeit überprüft.
Dann lasse ich die Variable 'ticks' ausgeben. Diese Variable wird bei jedem Update um eins hochgezählt und jetzt nach einer Sekunde wieder zurückgesetzt. Außerdem muss man die Variable 'timeAtLastFPSCheck' der aktuellen Zeit zu ordnen, da der FPS-Check durchgeführt wurde.
'ticks' gibt also an wie oft das Programm in einer Sekunde geupdatet hat. Die Zahl sollte nicht zu hoch sein, aber auch nicht zu niedrig. Denn entweder das Programm läuft dann zu schnell oder zu langsam.

```
public void fps() {
        if(System.currentTimeMillis() - timeAtLastFPSCheck >= 1000) {
                window.setTitle("Pi Collision - FPS: "+ticks);
                ticks = 0;
                timeAtLastFPSCheck = System.currentTimeMillis();
        }
}
```

Die nächste Methode überprüft die Zeit von der letzten Kollision und in Abhängigkeit dieser Zahl beendet sie den Durchgang (der Methode).
Diese Methode kam erst später hinzu und war eine kleine Bereicherung, denn lange gab es das Problem, dass das Programm nicht wusste wann Schluss ist. Ich kam dann auf die Idee, dass ich die Zeit seit der letzten Kollision prüfe und nach einer gewissen Zeit die While-Schleife schließe.
Natürlich muss lastTime bei jeder Kollision um eins hochgezählt werden und 'limit', also die Grenze des Wartens, könnte man zur Sicherheit etwas erhöhen. Dadurch müsste man an verschiedenen Stellen aber auch länger warten.
Insgesamt endet das Programm meist rechtzeitig.
Der Aufruf der Methode ist in einer If-Verzweigung, aber darauf gehe ich später ein. Generell kann man sagen, dass diese Methode jede Runde am Ende der update()-Methode ausgeführt wird (außer am Anfang des Programms).

```
public void timeCheck() {
        if(System.currentTimeMillis() - lastTime > limit) {
                run = false;
        }
}
```

Für den Sound bei einer Kollision ist die nächste Methode verantwortlich.
Je nachdem was für einen Sound man gewählt hat, wird dieser beim Aufruf mit 'play(File sound)' abgespielt. Um dies genauer zu verstehen sollte man sich die Klasse 'MusicLoader' ansehen.
Zur Auswahl stehen 5 Sounds und für jeden gibt es ein MusicLoader-Objekt. Diese werden von einer Variable des Typs Integer repräsentiert ('sound') und der Benutzer kann diese Variable in der Menu-Leiste ändern.

```
public void soundPlay() {
        switch(sound) {
                case 0: click0.play(click0.getSong());
                break;
                case 1: click1.play(click1.getSong());
                break;
                case 2: click2.play(click2.getSong());
```

```
                break;
                case 3: click3.play(click3.getSong());
                break;
                case 4: click4.play(click4.getSong());
                break;
        }
}
```

Nun kommen wir zu der Herzmethode des ganzen Programms. Wenn man sie ausführt, wird ein Durchgang ausgeführt.
Es befindet sich also der Game-Loop mit der While-Schleife und der update()- und render()-Methode.
Tatsächlich handelt es sich eher um einen Update-Loop. Aber dazu gleich mehr.
In den ersten Zeilen werden die zwei Rechtecke zurückgesetzt. Dabei wird ihre X-Position und der Kollisions-Zähler zurückgesetzt.
Dann wird die Mehtode 'setInstances' von gerade Eben aufgerufen. Diese liest das eingegebene Gewicht und die Geschwindigkeit aus und passt sie bei den Rechtecken an. Zudem wird 'nextGameTick' auf die aktuelle Zeit geändert. Der Grund folgt sehr bald.

```
public void start() {
        r1.reset();
        r2.reset();

        setInstances();
        [...]
```

Nun folgt eine Abfrage, denn das Programm kann entweder in einem Modus sein, wo gezeichnet wird und einer wo es nicht gemacht wird.
Dies wird über einen Boolean entschieden und kann über die Menu-Leiste geändert werden.
Dies befindet sich schon in der While-Schleife, welche von dem boolischen Wert von 'run' abhängig ist. Dieser Boolean wird in der Methode 'timeCheck()' geprüft und entsprechend verändert.
Zuletzt wird in jedem Schleifendurchgang die 'fps()'-Methode durchgeführt. Sie berechnet in jeder Sekunde die FPS und setzt ihre dafür verwendeten Variablen zurück('ticks', 'timeAtLastFPSCheck').

```
[...]
        while(run) {
                if(normalMode) {
                   [...]
                }else {
                   [...]
                }

                fps();
        }
}
```

Schauen wir uns zunächst den normalen Modus an.
In diesem wird zuerst die Variable 'loops' auf 0 gesetzt. Diese Variable zählt jeden Durchgang des Update-Loops. Sie funktioniert also ähnlich wie 'ticks'. Nur das 'ticks' erst nach einer Sekunde zurückgesetzt wird und 'loops' nach jeder Runde (vor jeder Runde).
Nun folgt der Update-Loop und danach die render()-Methode. Die update()-Methode befindet sich in einem Loop. Tatsächlich hatte ich zu Beginn eine Verzweigung benutzt und dies hat ebenfalls funktioniert. Zuerst gehe ich auf die Frage ein, warum es eine Bedingung für die update()-Methode geben muss und danach warum ich mich für die While-Schleife entschieden habe.
Wenn die update()-Methode genauso oft ausgeführt werden würde, wie die render()-Methode, so wäre das nicht schön für Auge. Denn das Programm wäre viel zu schnell. Deswegen darf es nicht so oft updaten. Zeichnen darf es ruhig öfters, denn damit sieht es nur ruckelfreier aus. Tatsächlich bin ich noch nicht ganz zufrieden, denn am Besten wäre es, wenn es am kritischen Punkt öfters updaten würde als an Stellen wo es nicht so oft kollidiert.
In meinem Code darf die update()-Methode nur ausgeführt werden, wenn die aktuelle Zeit größer als die

vorgesehene Zeit ist. Wie ist das jetzt zu verstehen? Ich habe zu Beginn 'nextGameTick' auf die aktuelle Zeit gesetzt und bei jedem Durchgang (durch den update-loop) wird 'nextgameTick' um eine bestimmte Zahl erhöht.
Umso öfters der update-loop durchgeführt, desto schneller wird 'nextGameTick' größer. Jedoch wird dies mit der aktuellen Zeit verglichen und wird in diesem Fall gestoppt. Nun wird der update-loop so oft ausgesetzt, bis er wieder unter dem Wert der aktuellen Zeit ist. Danach wird der update-loop wieder so oft durchgeführt bis der Wert von 'nextGameTick' zu groß wird.
Mit diesem System kann ich die Rate der update-loop-Durchgänge herunterschrauben und den Fähigkeiten des Rechners anpassen.
Bei einem langsamen Rechner würde es zwar den update-loop öfters durchlassen, da der PC aber langsamer ist würde das Programm trotzdem nicht zu schnell sein. Zu mindestens in der Theorie.
Bei einem schnellen Rechner würde der update-loop weniger oft durchgeführt werden, aber dieser schafft auch mehr gesamt Durchgänge und sollte auf ein vergleichbares Ergebnis stoßen.

Warum habe ich nun eine While-Schleife, anstatt einer If-Verzweigung genommen.
Dies ermöglicht dem Programm im Notfall mehrere update-loop-Durchgänge zu unternehmen. Dies wäre bei einem erwähnten langsamen PC hilfreich, denn dieser könnte somit performanter laufen. Rechnen kann der PC nämlich eindeutig schneller als zu zeichnen, was ich selbst auch getestet habe. Auf dem langsamen PC würde dann seltener gezeichnet werden, was zu rucklern führt. Außerdem sollte man eine Grenze einrichten, denn wenn der PC so langsam ist, das es nur updatet, dann würde das Programm nur noch updaten und nur sehr selten zeichnen. Zumindest denke ich mir das, aber ich baue trotzdem keine Grenze ein, denn der alte PC ist nur ein neben Grund. Der eigentliche Grund ist die Hoffnung, dass der PC bei vielen Kollisionen länger zu Rechnen hat und damit mehr Zeit benötigt. Dann würde sich die Anzahl der update-loop-Durchgänge anpassen, aber momentan ist dies noch nicht der Fall. Ich bastle noch daran und kann es hoffentlich rechtzeitig verbessern.

```
[...]
        if(normalMode) {
                loops = 0;

                while(System.currentTimeMillis() > nextGameTick) {
                        update();
                        nextGameTick += TIME_PER_TICK;
                        ticks++;
                        loops++;
                }
                render();
        }
[...]
```

Der andere Modus ist grundlegend ähnlich, aber der update-loop wird eindeutig häufiger durchgeführt und gezeichnet wird eindeutig weniger.
Die FPS-Zahl ist durchschnittlich um 40500 Updates in der Sekunde schneller und damit klar ein Modus, welcher recht interessant sein sollte.

```
[...]
        else {
                loops = 0;

                while(loops < 10000) {
                        update();
                        ticks++;
                        loops++;
                }

                renderMode2();
        }
[...]
```

So wir haben es fast geschafft. Jetzt schauen wir uns noch die update()-Methode an.
Zu Beginn werden alle zu updatende Objekte geupdatet. Diese befinden sich in einer ArrayList und deswegen weiß der Compiler, dass es so eine Methode geben muss. Die Rechtecke bewegen sich also nach ihren Geschwindigkeiten fort.
Danach wird erst geprüft, ob das linke Rechteck mit der Wand kollidiert. Falls ja wird die Richtung des Objektes in die andere Richtung gekehrt. Zudem wird der Kollisions-Zähler um eins erhöht, ein Sound wird abgespielt und die Variable 'lastTime' wird auf die aktuelle Zeit gesetzt. 'lastTime' gibt die Zeit der letzten Kollision an und wird in einer her vorige Methode verwendet um die While-Schleife zu beenden.
Falls diese Kollision nicht stattfindet, wird geprüft, ob sich die zwei Rechtecke berühren. Wenn ja wird ebenfalls ein Sound abgespielt, der Kollisions-Zähler erhöht, 'lastTime' aktualisiert und nun wird die Kollision berechnet.
Diese Berechnung ist wie ein realer unelastischer Stoß und nimmt die Geschwindigkeit und das Gewicht beider Rechtecke zur Berechnung.
Außerdem wird ein Boolean auf 'true' gesetzt. Dieser ist weiter unten wichtig und will eigentlich nur die erste Kollision notieren und es der Methode 'timeCheck' sagen. Nun wird es jede Runde auf true gesetzt und in der Wand Kollision muss ich es nicht einbinden, da die Kollision der Rechtecke die erste Kollision sein muss.
Nach dem Checken der Kollision werden die zu Anzeigenden Daten aktualisiert um in der render()-Methode angezeigt werden zu lassen.
Zu Letzt wird überprüft, ob die letzte Kollision zu lange her war. Dabei muss dies eine Bedingung erfüllen. Es muss die erste Kollision vorbei sein, denn zu Beginn bewegen sich die zwei Rechtecke recht lange aufeinander zu ohne sich zu berühren. Und hierfür habe ich eine Brücke eingebaut. Es wird also erst nach der ersten Kollision angefangen zu prüfen wie lange die letzte Kollision her war.

```
public void update() {
        for(PhysicalObject i:updating) {
                i.update();
        }

        if( r2.getX() <= leftW.getX()+5) {
                soundPlay();
                lastTime = System.currentTimeMillis();
                r2.setV2(-(r2.getV()));
                r2.collisionCounter();
        } else if(r2.getX()+50 >= r1.getX()) {
                soundPlay();
                lastTime = System.currentTimeMillis();
                r2.calculate(r1, leftW);
                r2.collisionCounter();
        }

        x2 = "x2: "+String.valueOf((int) r2.getX());
        x1 = "x1: "+String.valueOf((int) r1.getX());
        v2 = "v2: "+String.valueOf((int)r2.getV());
        v1 = "v1: "+String.valueOf((int)r1.getV());
        collisionC = "collison: "+String.valueOf(r2.getCollision());

        if(r2.getCollision() > 0) {
                timeCheck();
        }
}
```

Nun kommen wir zu einer ebenfalls wichtigen Methode. Die Sprache ist von der render()-Methode.
Zu aller erst nehme ich die 'BufferStrategy' von dem oben erstellten Canvas-Objekt. Danach stelle ich sicher, dass ich genug 'BufferStrategy's habe. Dies mache ich indem ich prüfe, ob ich keine habe und wenn ja erstelle ich zwei 'BufferStrategy's.
Diese 'Bufferstrtegy's kann man sich wie Leinwände vorstellen. Man kann auf ihnen zeichnen und sie dann zeigen. Genau das tun wir. Wir zeichnen etwas auf einer Leinwand und zeigen sie. Dann zeichnen wir auf die zweite Leinwand etwas. Der Zuschauer sieht immer noch die erste und sieht die zweite erst, wenn sie fertig gezeichnet ist. So entsteht ein 'sauberer' Übergang. Der Benutzer sieht nicht wie es gezeichnet wird, sondern er sieht es wenn es schon fertig ist.
Nun überreichen wir einem 'Graphics2D'-Objekt die BufferStrategys.
Dieser löscht erst einmal alle gezeichneten Dinge in einem Radius eines Rechtecks, welcher die Größe von dem Canvas-Objekt hat.
Dann wird die Kollisionsanzahl gezeichnet, sowie die Geschwindigkeit und die X-Position. Dies wird jedoch nur gezeichnet, wenn der entsprechende Boolean 'true' ist. Dies ist dafür, dass der Benutzer einstellen kann, ob er dies sehen will.

Schließlich werden alle Objekte in der Render-ArrayList gezeichnet (es wird deren Methode aufgerufen).
Die Grafiken werden nun noch disposesed, damit beim Neuzeichnen nicht mehr vorhanden sind.
Oder anders gesagt, werden die Grafiken weggeschmissen. Sie wurden schon gezeichnet und würden jetzt nur noch Speicher beanspruchen.
Da wir ja mit Leinwänden arbeiten muss unsere Arbeit noch gezeigt werden, was der letzte Befehl macht.
Wir haben also mit 'Graphics2D' auf unserem BufferedStrategy gezeichnet und zeigen es jetzt wo wir es fertig haben. Als nächstes zeichnen wir wieder auf die nächste und zeigen diese dann wenn sie fertig ist.

```
public void render() {
        BufferStrategy preRender = renderer.getBufferStrategy();
        if(preRender == null) {
                renderer.createBufferStrategy(2);
                return;
        }

        Graphics2D g = (Graphics2D) preRender.getDrawGraphics();
        g.clearRect(0, 0, renderer.getWidth(), renderer.getHeight());

        g.setFont(collisionFont);
        if(collision) {
                g.drawString(collisionC, 400, 150);
        }

        g.setFont(vFont);
        if(velocity) {
                g.drawString(v2, 150, 200);
                g.drawString(v1, 250, 200);
        }
        if(x) {
                g.drawString(x2, 150, 225);
                g.drawString(x1, 250, 225);
        }


        for(PhysicalObject i:rendering) {
                i.render(g);
        }
        g.dispose();
        preRender.show();
}
```

Für den anderen Modus, in welchem nicht gezeichnet wird, wird ebenfalls gerendert. Dies geschieht jedoch nur sehr selten und es wird auch nur die Kollisionsanzahl gezeichnet.
Unnötig ist, dass ich mehrere Buffer benutze, aber so war es nun mal schneller und trägt keine großen Schäden mit sich (sonst hätte ich es geändert).

```
public void renderMode2() {
        BufferStrategy preRender = renderer.getBufferStrategy();
        if(preRender == null) {  werden
                renderer.createBufferStrategy(2);
                return;
        }

        Graphics2D g = (Graphics2D) preRender.getDrawGraphics();
        g.clearRect(0, 0, renderer.getWidth(), renderer.getHeight());

        g.setFont(collisionFont2);
        g.drawString(collisionC, 100, 200);


        g.dispose();
        preRender.show();
}
```

Dies war nun die Komplette Klasse mit all seinen Variablen und Methoden.

2.3 RectangleObject

Aus dieser Klasse gehen zwei Rechtecke hervor, welche sich bewegen und verschiedene Informationen besitzen (wie Gewicht und Geschwindigkeit).
Sie unterstützt das Interface 'PhysicalObject' und besitzt somit eine Methode zum Zeichnen und zum Updaten.
Außerdem wird hier die Kollision berechnet, aber nur von einem Objekt.
Damit die 'PhysicEngine' auf die Daten zugreifen kann, besitzt diese Klasse noch einige Getter und Setter. Diese sind auch in der Start-Klasse von Wichtigkeit.

```
import java.awt.Color;
import java.awt.Graphics2D;
import java.awt.image.BufferedImage;

import Images.Sprite;


public class RectangleObject implements PhysicalObject{

        private double x = 400, y = 275, xBegin = x;
        private int width = 50, height = 50;

        private int grenzwertR = 100;
        private long collisionCount = 0;

        private double m = 1.0, v = -80.0, v2;

        private boolean arrowOn = false;
        private Color color = Color.BLACK;
        private int pic = 0;
        private BufferedImage arrowL, arrowR;
[...]
```

Der Konstruktor ladet die Bilder. In diesem Fall gibt es nur die Pfeile als Bilder. Dafür wird eine andere Klasse verwendet, welche im Abschnitt 'Sprite' detailiert beschrieben wird.

```
public RectangleObject() {
        try {
                arrowL = Sprite.getSprite("left-arrow.png");
                arrowR = Sprite.getSprite("right-arrow.png");
        }catch(Exception e) {
                System.err.println(e.getMessage());
                System.exit(1);
        }
}
```

Nun folgen die ganzen setter und getter. Diese sind von großer Wichtigkeit. Da über sie sehr viele Informationen ausgetauscht und verändert werden. Ich denke jedoch, dass es unnötig ist alle von ihnen zu zeigen. Schließlich sind sie immer gleich aufgebaut. Man erhält oder gibt einen Parameter zurück.
Interessant wird es erst ab der 'reset' und der 'collisionCounter'-Methode. Beide machen genau das was ihr Name uns schon verrät. Die 'collisionCounter'-Methode erhöht eine Variable und wird nur von dem linken Rechteck verwendet.
Die 'reset'-Methode setzt den Kollisionszähler und die X-Position zurück. Die anderen Werte werden von der Methode 'setInstances' (in der Klasse 'PhysicEngine') zurückgesetzt, da die Werte aus den JSlider oder aus den JTextField ausgelesen werden muss.

```
public void reset() {
        x = xBegin;
        collisionCount = 0;
}

public void collisionCounter() {
        collisionCount++;
}
```

Erstrecht bei der Methode 'calculate(RectangleObject r1)' wird es interessant. Diese Methode wird zum berechnen verwendet und zwar nur vom zweiten Rechteck. Die Rechnung ist wie die normale Formel von einem unelastischen Stoß: 2*(((m*v)+(m*v)) / (m+m)). Nur das dies der aktuellen Geschwindigkeit abgezogen wird.
Die Variable v2 dient dazu die Geschwindigkeit zwischen zu speichern, da sie in der zweiten Berechnung noch benötigt wird. Danach wird die Geschwindigkeit auf v2 gesetzt.
Die erste Berechnung dient dem linken Rechteck (welches die Berechnung auch durchführt) und die zweite Berechnung dient dem rechten Rechteck.

```
public void calculate(RectangleObject r1) {
        v2 = 2* (
                ( (m * v) + (r1.getV()*r1.getM()) )
                / (r1.getM() + m)
                ) -v;


        r1.setV3( 2*
                  ( (m * v) + (r1.getV()*r1.getM()) )
                     / (m + r1.getM()) - r1.getV() );

        v = v2;
}
```

Nun kommen wir schon zum updaten und zum rendern der Klasse. In der update()-Methode wird um dessen Geschwindigkeit erhöht. Die Werte der Geschwindigkeit sind realitätsnah und müssen deswegen noch verkleinert werden.

```
public void update() {
        x = x + (v/100);
}
```

Das rendern ist in zwei Bereiche, also zwei Modi eingeteilt. Und sieht also etwa so aus.
Die Methode benötigt zudem ein Graphics2D-Objekt, um zeichnen zu können. Dieses erhält sie in der render()-Methode in der 'PhysicEngine'-Klasse.
Die zwei Modi sind eine kleine Manipulation, die ich mir erlaubt habe. Dies ist keine Lösung für ein Problem, sondern eine Vertuschung. Aber meine Absicht war es nicht, dass Problem geheim zu halten und kann detailliert im Bereich 'Probleme' -> 'Out of the box' untersucht werden.
Ich überprüfe die Rechtecke und lasse sie normal zeichnen. Wenn sie jedoch den 'kritischen Orbit' überschreiten, also Bereiche betreten die sie nicht betreten dürfen und auch nicht sollten, dann zeichne ich eine festgelegte Position.
Wenn sie die Verzweigung auskommentieren, dann werden sie feststellen was ich meine.
Ich bin mir nicht ganz sicher woran das liegt.

```
public void render(Graphics2D g) {
        if(x >= grenzwertR) {
                [...]
        }else {
                [...]
        }
}
```

Der normale Zeichnen-Mode funktioniert so, dass ein Dreieck in Abhängigkeit der x-Variable gezeichnet wird.
Davor wird überprüft, welches Design für das Rechteck gewählt wurde. Dies geschieht durch eine Variable vom Typ Integer.
Der Pfeil wird ebenfalls anhand einer Variable (hierbei vom Typ Boolean) gezeichnet oder eben nicht gezeichnet.
Dabei werden Methode von graphics2D für das Zeichnen verwendet.

```
[...]
        if(x >= grenzwertR) {
                switch(pic) {
                        case 0: g.setColor(color);
                        g.fillRect((int)x, (int)y, width, height);
                        break;
                        case 1: g.setColor(Color.BLACK);
                        g.fillRect((int)x, (int)y, width, height);
                        break;
                        case 2: g.setColor(Color.MAGENTA);
                        g.fillRect((int)x, (int)y, width, height);
                        break;
                }

                if(arrowOn) {
                        if(v < 0) {
                                g.drawImage(arrowL,(int) x,(int) y, null);
                        }else if(v > 0) {
                                g.drawImage(arrowR, (int) x, (int) y, null);
                        }
                }
        }
[...]
```

Der Zeichnen-Mode, in dem ich die Position anhand einer festgelegten Position selbst bestimmte, funktioniert identisch, nur das ich eine konstante Variable als Punkt zum Zeichnen weitergebe.

```
[...]
        else {
                switch(pic) {
                        case 0: g.setColor(color);
                        g.fillRect((int)grenzwertR, (int)y, width, height);
                        break;
                        case 1: g.setColor(Color.BLACK);
                        g.fillRect((int)grenzwertR, (int)y, width, height);
                        break;
                        case 2: g.setColor(Color.MAGENTA);
                        g.fillRect((int)grenzwertR, (int)y, width, height);
                        break;
                }

                if(arrowOn) {
                        if(v < 0) {
                                g.drawImage(arrowL,(int) grenzwertR,(int) y, null);
                        }else if(v > 0) {
                                g.drawImage(arrowR, (int) grenzwertR, (int) y, null);
                        }
                }
        }
}       //Ende der Methode
```

2.4 Wall

Zu dieser Klasse gibt es nicht viel zu sagen, da sie nur ein Objekt zum Zeichnen ist. Außer den Kollisionscheck (durch eine getX-Methode) mit dem Rechteck, passiert nichts.
Die Klasse besitzt trotzdem ein paar setter()-Methoden, da es zwei Wände gibt und bei einer Wand zu Beginn die Werte geändert werden müssen.
In der render-Methode passiert nicht viel. Es wird ein Objekt der Klasse graphics2D benutzt um Rechtecke zu zeichnen.

```
public void render(Graphics2D g) {
        g.setColor(color);
        g.fillRect(x, y, width, height);

}
```

2.5 CollisionProcess

Diese Klasse ist dafür dar, dem Thread in 'PhysicEngine' zu sagen was er machen soll und wird diesem auch übergeben.
Damit dies geht muss diese Klasse der Schnittstelle 'Runnable' folgen. Dies stellt sicher, dass es eine run()-Methode gibt.
Diese Methode braucht das erzeugte Objekt von der Klasse 'PhysicEngine' um dort die start()-Methode aufrufen zu können.

```
public class CollisionProcess implements Runnable{

        private PhysicEngine engine;

        public CollisionProcess() {
        }

        public void init(PhysicEngine pe) {
                engine = pe;
        }

        public void run() {
                engine.start();
        }
}
```

2.6 MusicLoader

Von dieser Klasse gibt es mehrere Objekte. Jedes Objekt steht für einen Sound.
Dabei ist die Methode 'loadPackapge(String songname)' für die Initiierung notwendig. So lädt das Objekt 'seinen' Sound und zwar erst einmal als File. So besitzt das Objekt schon mal den Pfad zu dem Sound.

```
public void loadPackage(String songName) {
        sound = new File("src/Sound/"+songName+".wav");
}
```

Die Methode play() wird verwendet um den Sound schließlich abzuspielen. Da man mit der obigen Methode schon den File geladen hat man diesen schon.
Dafür wird das 'sound'-Package von javax verwendet. Um genau zu sein das Package 'sampled', welches Klassen bereitlegt um Audio abzuspielen. Die Klasse 'Clip' kann genau dies und mit der Klasse 'AudioSystem' verwerte ich die Rohdaten einer wav-Datei, wessen Standort ich durch einen File angebe.
Zuerst wird der clip einem clip vom AudioSystem zugewiesen. Damit ist unser clip ein richtiger clip. Danach kann die Sound-Datei in den clip geladen werden und schließlich geladen werden.
Der clip startet also den Sound und ladet ihn mithilfe des AudioSystem. Das AudioSystem weißt zudem den clip dem Audio hinzu.

```
public void play() {
        try {
                clip = AudioSystem.getClip();
                clip.open(AudioSystem.getAudioInputStream(sound));
                clip.start();
        }catch(Exception e) {
                e.printStackTrace();
        }
}
```

2.7 Sprite

Diese Klasse lädt (die wenigen) Bilder. Vo ihr wird kein Objekt erzeugt, da nur ihre Methode verwendet wird.
Diese Methode benutzt das awt-Package und versucht eine Datei mit den Namen, wie der eingegebene String, einzulesen und zwar relativ zur sich selbst.
Die Klasse gibt ein BufferImage zurück, welches dann von Graphics2D gezeichnet wird.

```
public static BufferedImage getSprite(String fileName){
        try {
                return ImageIO.read(Sprite.class.getResourceAsStream(fileName));

        }catch(Exception e){
                e.printStackTrace();
        }
        return null;
}
```

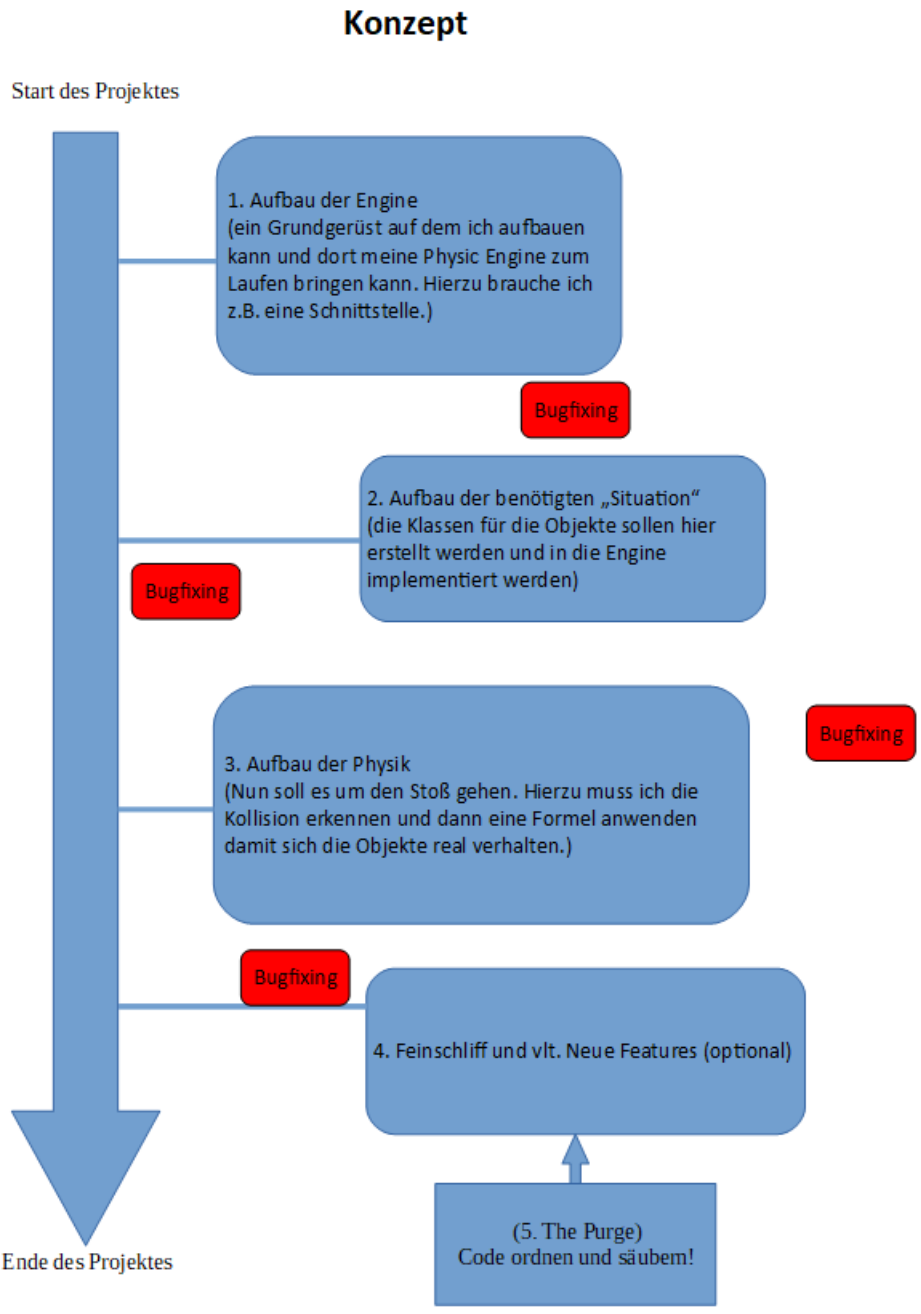
2.8 PhysicalObject

Dies ist die Schnittstelle. Mit dieser weiß 'PhysicEngine', dass die Objekte in den zwei ArrayLists gerendert und geupdated werden können. Sonst könnte man diese Methoden nicht Aufrufen, da die 'PhysicEngine' nicht weiß, ob die Objekte diese Methoden besitzen.
Zudem weiß die 'PhysicEngine', dass alle Elemente in den ArrayLists das Package Graphics2D kennen.

```
import java.awt.Graphics2D;

public interface PhysicalObject {
        public void update();
        public void render(Graphics2D g);
}
```

3 Konzept


Konzept
Start des Projektes
1. Aufbau der Engine
(ein Grundgerüst auf dem ich aufbauen kann und dort meine Physic Engine zum Laufen bringen kann. Hierzu brauche ich z.B. eine Schnittstelle.)
Bugfixing
2. Aufbau der benötigten „Situation“
(die Klassen für die Objekte sollen hier erstellt werden und in die Engine implementiert werden)
Bugfixing
3. Aufbau der Physik
(Nun soll es um den Stoß gehen. Hierzu muss ich die Kollision erkennen und dann eine Formel anwenden damit sich die Objekte real verhalten.)
Bugfixing
Bugfixing
4. Feinschliff und vlt. Neue Features (optional)
(5. The Purge)
Code ordnen und säubern!
Ende des Projektes

4 Diagramme zum Programm

Die Diagramme wurden digital verschickt.
Man kann die Diagramme nämlich nur schwer drucken und was dabei rauskommt können sie an dem Klassendiagramm sehen. Man kann sie wenn dann nur sehr schlecht lesen.

4.1 Klassendiagramm

Die Klassendiagramme wurden mit dem (alten) Programm 'Dia' erstellt. Dieses bietet alle nötigen Funktionen um ein Klassendiagramm zu erstellen. Leider konnte ich nicht das Gitter ausschalten (die anderen Gitter konnte ich ausschalten), aber dies kann man zu akzeptiert.
Störender fand ich die Pfeile bei einer Assoziation. Diese sind komisch in der Mitte und sehen nicht gut aus. Aber vielleicht war dies die alte UML-Syntax.

4.2 Sequenzdiagramm

Digital nachzusehen.
Das Sequenzdiagramm wurde in der Online-Entwicklungsumgebung 'GenMyModel' entwickelt. Diese lässt sich leicht bedienen, hat jedoch Fehler beim Übertragen zu einem Bild oder ähnliches.
Dies ist nervig, da die Entwicklungsumgebung eigentlich echt gut ist und seine eigentlich fertige Arbeit noch herauszögert.

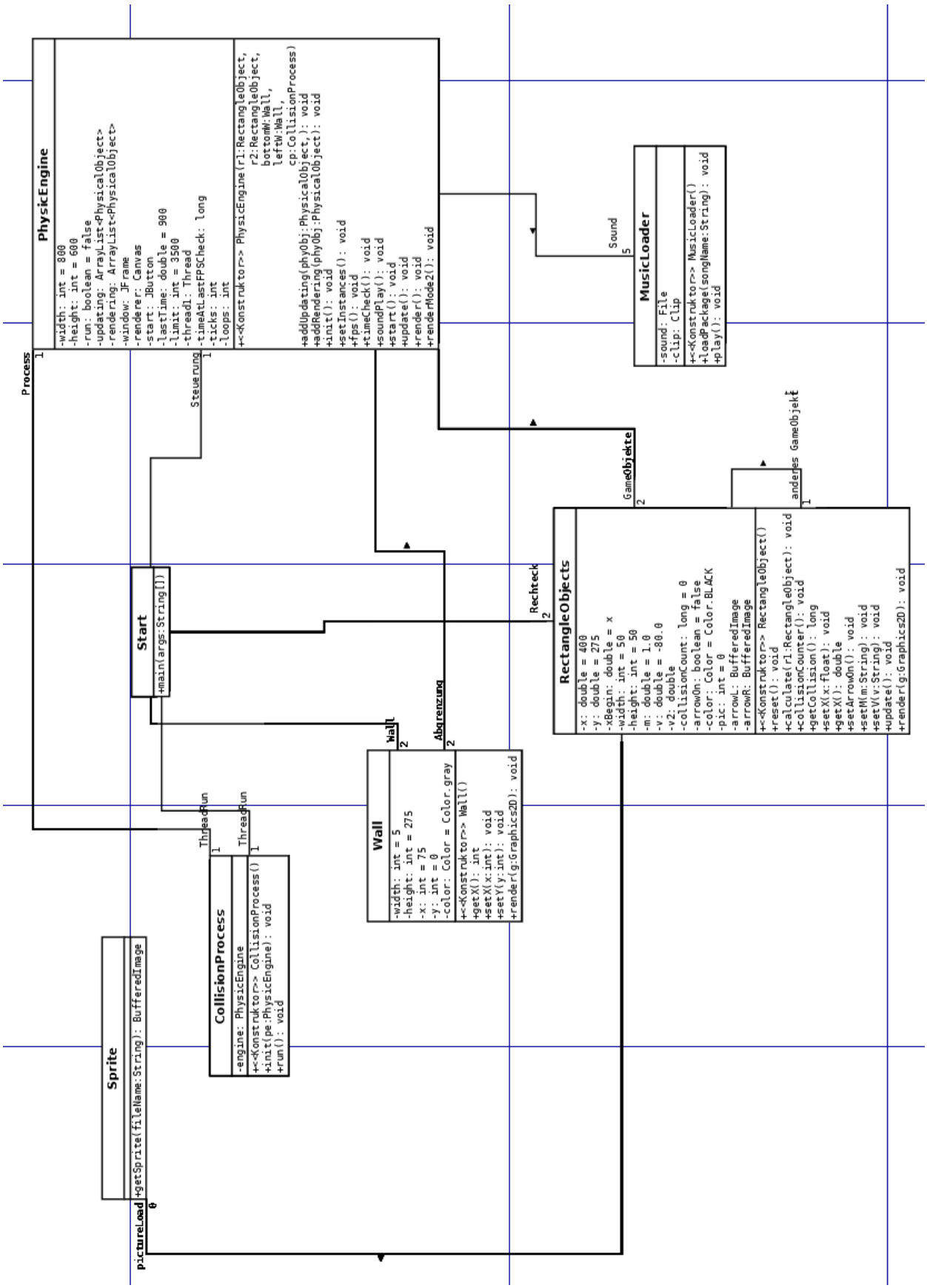


Abbildung 4.1: Klassendiagramm

5 Probleme

5.1 Hidden JMenu

Dieses Problem habe ich erst später entdeckt und es geht dabei darum, dass sich das JMenu, sobald ich auf ein JTextField klicke, hinter Canvas versteckt.
Ich habe unter anderem erfahren, dass ich hierbei einen nicht so schönen Fehler begangen habe. Ich habe Swing mit der Vorgängerversion AWT vermischt. Durch Canvas. Ob es wirklich daran liegt weiß ich nicht, aber ich habe das Gefühl, dass sich dieses Problem eigentlich leicht lösen ließe. Ich dachte man müsse einfach den Fokus von dem Textfield lösen... vielleicht stimmt dies oder so ähnlich sogar, aber ich hatte mit dem Lösen noch keinen Erfolg.

5.2 Out of the box

Bei diesem Problem geht es darum, dass sich die Rechtecke durch die Wand buggen. Komisch ist jedoch, dass die Kollisionen trotzdem richtig erscheinen.
Die Rechtecke kommen auch wieder zurück. Ich denke es leigt an den zu hohen Werten. Wenn ein Rechteck die Wand berührt kehrt sich seine Geschwindigkeit um. Somit sollte es eigentlich nicht möglich sein diese zu überqueren, aber dennoch passiert es. Wie gesagt ist das mit den zu hohen Werten meine einzige logische Erklärung.
Da ich die Kollision so messe, dass die Kollision stattfindet, wenn die Position kleiner als die der Wand ist, finden die Kollisionen immer noch statt und deswegen kehren die Rechtecke wohl wieder zurück.
Somit sieht es seltsam aus, sollte dennoch korrekt sein. Deswegen habe ich die Grafik etwas manipuliert. Nun sieht man nicht mehr, dass die Rechtecke die Wand ('kritischer Orbit'- von mir genannt) durchqueren.
Ich wüsste nicht wie ich dieses Problem beheben soll. Ich habe verschiedene weitere Effekte zu der Kollision hinzugefügt, damit die Mauer nicht durchquert werden kann. Dann stoße ich jedoch auf seltsame Kollisionen und es verhält sich seltsam. Damit habe ich es dann nach verschiedenen Anläufen aufgegeben.

5.3 Wie lange soll ich arbeiten?

Ein weiteres Problem war die Zeit herauszufinden, ab welcher das Programm enden soll. Zunächst lief die Methode 'start()' beim Aufruf für immer.
Dann setzte ich als Lösungsansatz eine zeitliche Begrenzung ein. Um dies zu 'verschönern', habe ich eine Klasse programmiert mit welcher ich ein Ladebalken einfügen konnte. Somit konnte man gut sehen, wie lang der Durchgang noch lief. Dies sah echt gut aus und der Gedanke kam mir im Alltag, denn der Aufbau ist echt simple. Der Ladebalken bestand aus zwei Rechtecken, einer war die Hülle und der andere veränderte sich relativ (prozentual) zu der maximalen Größe und der maximalen Zeit.
Trotzdem war diese Lösung eine sehr schlechte Lösung, aber ich konnte wenigsten weiter programmieren.
Irgendwann musste dann eine bessere Lösung her und mir viel auch eine ein. Hierbei prüfe ich wann die letzte Kollision war. Wenn dieser Wert zu groß wurde, wurde der Durchgang beendet (ein Boolean wurde auf 'false' gesetzt und die while-Bedingung war nicht mehr erfüllt).
Dies war eine richtig gute Lösung, da ich sich das Programm selbst beendet und zwar eigentlich immer rechtzeitig. Ich habe den Wert nicht ganz so groß gemacht. Dadurch gibt es zwar Situationen, wo die letzte Kollision gerade so nicht mitgezählt wird, aber dadurch muss man oft nicht so lang warten.

Zu Beginn dachte ich die Lösung wäre ein Fehltritt, denn das Programm schaffte nicht einmal die erste Kollision. Dies liegt daran, dass die beiden Rechtecke bei der ersten Kollision recht viel Abstand zueinander haben. Es dauert also ein bisschen bis die erste Kollision erfolgt.
Ich konnte nun den Wert für die maximalen Zeit der letzten Kollision hochdrehen, dann musste man am Ende jedoch sehr lang warten. Es gab auch die Möglichkeit die beiden Rechtecke näher aneinander spawnen

zu lassen. Ich finde es jedoch schön, dass sie sich zu Beginn erst aufeinander zu bewegen, deswegen gab es es eine andere Lösung.
Hierbei wird die Zeit seit der letzten Kollision erst nach der ersten Kollision abgefragt (Bedingung: r2.getCollision() > 0). Somit prüft das Programm, ob es die While-Schleife beenden soll erst nach der erste Kollision.

5.4 Paralleles Arbeiten

Dieses Problem habe ich erst recht spät beseitigt. Es handelt davon, dass mein Programm entweder die Kollision durchgeführt hat (die 'start()'-Methode in 'PhysicEngine') oder im Menü war. Wenn ich also einen Durchgang gestartet habe, so konnten keine Inputs vom Benutzer wahrgenommen werden.
Deswegen öffne ich einen neuen Thread zum durchführen der Kollisionen. Damit der Thread weiß was er zu tun hat, gebe ich ihm ein Objekt der Klasse 'CollisionProcess' mit. Dieses besitzt ein Interface und somit kann der Thread die Methode 'run()' aufrufen. Diese startet einen Durchgang. Hierzu braucht die Klasse 'CollisionProcess' das Objekt von der Klasse 'PhysicEngine', schließlich ruft sie dort die Methode 'start()' auf.
Nun kann ich mit dem Benutzerinterface interagieren, während ein Durchgang läuft.

5.5 Restart verboten!

Ein sehr junges Problem ist, dass man während eines Durchganges keinen neuen starten kann. Beziehungsweise man kann einen neuen starten, aber wird anscheinenden in einem neuen Thread durchgeführt. Deswegen wollte ich den Thread zuvor beenden. Dies ginge anscheinend mit interrupt(), aber das Problem blieb bestehen.
Wahrscheinlich liegt das Problem an der Herangehensweise, denn ich rufe eine andere Methode aus einer anderen Klasse auf und dort passiert alles. Oder einfacher ausgedrückt, wenn ich den Game-Loop in der Klasse 'CollisionProcess' hätte, würde interrupt() funktionieren. Aber ich habe dennoch eine Lösung gefunden.
Ich überprüfe, ob der Durchgang noch läuft und zwar nicht anhand des Threads, sondern anhand der Bedingung für die While-Schleife. Diese Bedingung wird ja schließlich nach einem Durchgang auf 'false' gesetzt, damit die Durchgänge pausieren.
Eine neue Runde lässt sich nun erst nach einem Durchgang beginnen. Somit ist dieses Problem vom Tisch.

5.6 Bitte einer nach dem anderen

Zu Beginn meines Projekt war ich fast am Verzweifeln, da meine Swing-Komponenten nicht angezeigt wurden. Ich hatte extra sauber gearbeitet und alle Komponenten einem Container von dem JFrame hinzugefügt und dann ein unergründlicher Fehler.
Mittlerweile weiß ich, dass Swing nicht mit mehreren Zugriffen auf ihre Komponenten klar kommt. Deswegen müssen Zugriffe in eine Warteschlange eingereiht werden.
Dies wird mit 'EventQueue.invokeLater(() -> { *Swing-Komponenten* });'

6 Schlussworte

Dieses Projekt war sehr aufschlussreich und damit auch erfolgreich. Hinzu kommt, dass man das End-Produkt durch aus begutachten kann. Es gibt dort noch viele Redundanzen oder ähnelt teilweise einem Spaghetti-Salat und dennoch mache ich Fortschritte.
Vor allem bin ich von den Früchten meines privaten Engagement begeistert. In den Sommerferien habe ich angefangen so wirklich privat irgendetwas zu programmieren. Und dies hat mir bei diesem Projekt sehr geholfen. Ich war begeistert wie viel leichter es mir fällt.
Zuvor hatte ich einen FlappyBird-Klon mit einem Tutorial geschrieben. Dabei habe ich beispielsweise die Wichtigkeit von Schnittstellen gesehen und konnte diese gleich hier in diesem Projekt anwenden. Auch im Game-Loop und der generellen Struktur wurde mir durch diese Erfahrung sehr geholfen.
Aber ich konnte in diesem Projekt nicht nur Erfahrung einsetzen, sondern auch gewinnen. Ich habe viele verschiedene Dinge wie beispielsweise das Arbeiten mit Threads oder das Erstellen von Menüs gelernt.

Es wurden mir generell die Konzepte der OOP nähergebracht. Naja bis auf das Vererben. Dieses Konzept konnte ich nicht gebrauchen, aber bei Spielen oder anderen Projekten ist es sinnvoll. Dabei kann man Objekte und dessen Hierarchie besser ordnen. Auch in diesem Projekt hätte es zum Einsatz kommen können, aber ich denke dies wäre eher unnötig gewesen.

Insgesamt ein sehr erfolgreiches Projekt und ein Projekt, welches viel vollendeter ist als all meine bisherigen Projekte. Darauf bin ich aufjdenfall stolz.

6.1 Verbesserung

Ich habe schon eindeutig 'besser' programmiert als bei meinen bisherigen Projekten, aber dennoch sollte das Konzept noch besser durchdacht sein, damit ich beispielsweise nicht Swing mit AWT mische.
Auch meinen Spaghetti-Code möchte ich in Zukunft noch mehr zurückfahren.
Außerdem möchte ich nächstes Mal mehr mit dem Schlüsselwort 'final' arbeiten. Wie ich in Erfahrung bringen konnte, kann der Einsatz dieses Schlüsselwortes die Lesbarkeit steigern. Denn der Leser weiß sofort, dass es sich hierbei um eine unveränderliche Konstante handelt.
Zudem fände ich es schön mal andere Methoden zum Zeichnen zu nehmen, als immer nur Swing oder AWT(Canvas).

7 Quellen

In diesem Projekt habe ich so einiges durch andere Projekte, welche ich gemacht habe gelernt/übernommen.
Zum Beispiel die Struktur mit dem Interface.
Trotzdem kam ich nicht dabei drum herum im Internet nach Lösungen und Hilfe zu suchen... ich mein das ist ein wichtiger Bestandteil des Informatikerdarseins.

https://www.youtube.com/watch?v=0ctk4BxwKH0
Hier habe ich die Formel zur Berechnung des Stoßes her. Dies ist der unelastische Stoß (für die 'echte' Welt).

https://www.youtube.com/watch?v=d470OpVDzRs
Hier habe ich mich allgemein über Stöße informiert.

https://www.youtube.com/watch?v=HEfHFsfGXjs
Hier habe ich die Idee von dem Projekt her.

http://openbook.rheinwerk-verlag.de/javainsel9/javainsel_19_026.htm
Von diesem Auszug eines Buches weiß ich, dass Swing nicht mit mehreren Zugriffen klar kommt und diese Zugriffe eingereiht werden müssen.

https://www.java-tutorial.org/jslider.html
Von diesem Artikel weiß ich wie man JSlider erstellt.

https://java-tutorial.org/jpopupmenu.html
Auf dieser Website konnte ich lernen JMenus zu erstellen.

https://www.javatpoint.com/java-jmenuitem-and-jmenu
Hier konnte ich ebenfalls lernen wie man JMenus programmiert.

https://www.dpunkt.de/java/Programmieren_mit_Java/Multithreading/3.html
http://wwwbroy.in.tum.de/lehre/vorlesungen/info3/doc/threads.pdf
Durch diese PDF-Datei und den Artikel im Internet, habe ich gelernt Threads zu benutzen.

https://www.genmymodel.com/
Die verwendete Entwicklungsumgebung für das Sequenzdiagramm.

http://dia-installer.de/index.html.de
Hier kann man die Entwicklungsumgebung für das Erstellen von Klassendiagrammen downloaden.