Schriftliche Dokumentation zum Softwareentwurf des Projekts „Kugel-Lineal“

Datum: 20.10.15

Version 2.0

Inhalt

1 Ziel des Projektes

Ziel des Projektes ist es, ein Programm zu schreiben, um das Schießen einer Papierkugel mit einem Lineal zu simulieren. Das Programm soll an Schulen zum Einsatz kommen, um die Verschmutzung in den Klassenzimmern zu reduzieren und Aggressionen bei Schülern abzubauen.

Genauere Anforderungen können dem Pflichtenheft entnommen werden.

2 Grundlegende Komponenten

Eine Software nicht in Komponenten zu unterteilen kann zu verschlechterter Wartbarkeit, Übersichtlichkeit sowie Wiederverwendbarkeit führen.

Deshalb orientiert sich unsere Grundarchitektur an dem MVC-Prinzip, d. h.:

1. Es gibt eine Komponente für alles, was dem Benutzer angezeigt wird (View)

2. Es gibt eine Komponente für die Logik des Programmes (Controller)

3. Es gibt eine Komponente zum Halten, Speichern und Laden der Daten (Model)

In unserem Fall unterteilt sich der Controller nochmals in 2 Subkomponenten, die **Engine** (Seite **3**) und das **Spielfeld** (Seite **5**) bzw. die **Spielfeldelemente** (Seite **6**)**.**

Die **Engine** kümmert sich um grundlegende Aufgaben wie z. B. **Eventmanagement** oder **Prozessmanagement.** Die **Spielfeldelemente** sind die logische Repräsentation der grafischen Objekte wie z. B. der **Kugel** oder des **Katapults**.

3 Engine

Die Engine besteht aus sieben Hauptkomponenten: **Graphics**, **Physics**, **Processmanager**, **Eventmanager**, **Controller**, **Timer** und die **Engine** selbst.

Diese Komponenten erben von der Klasse **EngineElement**. Diese Klasse enthält als Attribute eine Prozess-ID und eine Instanz von sich selbst, auf welche über **getInstance** zugegriffen wird. Ausserdem sind Methoden zum Initialisieren und Beenden der Komponenten vorhanden, welche im Konstruktor **EngineElement** als **Eventcallbacks** registriert werden. Die **run-Funktion** wird als **Process** registriert, dessen **ID** abgespeichert wird.

Alle Komponenten werden als Referenz in **Engine** abgespeichert. Über Getter-Methoden kann auf diese zugegriffen werden.

3.1 Graphics

Graphics enthält eine Liste aller **Graphicsobjects**. Diese können über **addGraphicsobject** oder **removeGraphicsobject** hinzugefügt oder entfernt werden. Mit **drawScene** werden die **Graphicsobjects** über deren **render-Funktion** gezeichnet, während mit **clear** der Bildschirm geleert wird. Die **Graphicsobejcts** enthalten eine zu zeichnende **Texture** und einen **Counter**, wie auch eine Methode **runAnimation**, welche zum Darstellen einer Animation erforderlich sind.

3.2 Physics

Physics enthält eine Liste aller **Physicsobjects**. Über die Methode **simulateNext** wird die Physik-Szenerie aktualisiert. Mit **simulateNextNSteps** wird dieser Schritt N-Mal wiederholt und abgespeichert, um so eine Vorrausberechnung zu erhalten. Dabei wird auf Kollisionen mit der Methode **checkforCollisions** geprüft. Über **addPhysicsobject** und **removePhysicsobject** können **Physicsobjects** hinzugefügt oder entfernt werden. Die Attribute **hullbox**, **static** und **weight** spezifizieren die Eigenschaften eines **Physicsobjects**.

3.3 Processmanager

Der **Processmanager** enthält eine Liste mit allen Prozessen. Über **removeProcess** und **addProcess** können Prozesse hinzugefügt oder entfernt werden. **RunProcesses** verarbeitet alle **Processe**. Solche Prozesse enthalten selbst Referenzen auf Objekte von Klassen, die das Interface **ProcessOwner** implementieren und somit über eine aufzurufende **Callback-Funktion** verfügen.

3.4 Eventmanager

Der **Eventmanager** enthält eine Liste aller **Events** und einen **Vektor** aller **Eventlistener**. Über **registerListener** und **unregisterListener** können **Eventlistener** hinzugefügt oder entfernt werden. Diese werden im **Eventlistenervector** abgespeichert. Eine Klasse die das **Interface Eventlistener** implementiert enthält eine Callback Funktion, welche beim Triggern des **Events** mit einer bestimmten **ID** aufgerufen wird. Dies passiert in der Methode **processEvents**, in der die **Eventlist** abgearbeitet wird.

3.5 Controller

Der **Controller** enthält einen **Vektor** in dem alle **Controls** gespeichert sind, sowie eine Referenz auf die zurzeit aktiven Controls. Über **setControls** und **addControls** können Controls hinzugefügt und aktiviert werden. Solche **Controls** enthalten einen **Vektor**, der **(Maus)Tasten** auf **Events** mappt, was durch die Methode **setControl** definiert wird.

3.6 Timer

Der **Timer** gibt die **Frequenz** des Programms vor, welche über **setFrequence** und **getFrequence** gesetzt und abgerufen werden kann. Die Methode **waitTimer** wartet die restliche Zeit bis zum nächsten **Tick**.

3.7 Vorgänge innerhalb der Engine

Beim erstmaligen Aufrufen von **getInstance** aller **EngineElements** durch **getInstance** im Konstruktor der **Engine**, werden die **run-Prozesse** erstellt und die Funktionen **startup** und **shutdown** als Eventcallbacks registriert.

Anschliessend wird das **StartupEvent** dem **Eventmanager** gesendet, wodurch sich alle **EngineElements** in ihren **startup-Funktionen** initialisieren. Womöglich muss dieser Vorgang etwas unsauber gelöst werden, da der **EventManager** noch nicht initialisiert ist.

Nun ist die **Engine** gestartet und es beginnt eine Schleife, die erst beim Senden **Shutdown-Events** wieder verlassen wird. Innerhalb dieser Schleife wird zuerst die Funktion **ProcessManager::runProcesses** ausgeführt, in der die Prozesse einzeln verarbeitet werden.

Die **Tickrate** der Prozesse wird mit der bereits gewarteten Zeit verglichen und die **Callback-Funktion** des **Processowners** entsprechend aufgerufen.

EventManager::processEvents ist einer dieser Prozesse. Alle eingereihten **Events** werden abgearbeitet und die entsprechend registrierten **Listener** mit ihren **Callback-Funktionen** aufgerufen.

Innerhalb dieser Funktionen wird die Spiellogik ausgeführt, die Grafik gezeichnet und auf Eingaben überprüft. Hier können wiederrum Prozesse erstellt oder gelöscht werden. Ebenfalls können hier Events erstellt werden. Die Kommunikation mit dem **EventManager** und dem **ProcessManager** erfolgt über die **Getter-Funktionen** der **Engine**.

Wir das **Shutdown-Event** gesendet, so werden die **shutdown-Funktionen** aufgerufen und alle **EngineElements** beenden sich selbst.

Am Schluss werden ihre **Destruktoren** aufgerufen.

4 Menüs

4.1 Erstellung der Menüoberfläche:

Es können eine ganze Reihe an Packages importiert werden, die alle Standartmäßig in Java 8 vorhanden sind. Für die Erstellung des Menüs werden folgende benötigt:

1. javafx.application.*
2. javafx.stage.*
3. javafx.scene.*
4. javafx.scene.layout.*
5. javafx.scene.control.*

4.2 Main

Die Klasse Main erbt von der Klasse „Application“ in dieser sind einige Methoden bereits implementiert. Als erstes wird die Methode **launch(args)** aus „Application“ aufgerufen, diese bildet die Grundlage für jedes JavaFX Programm. Die Methode **void start( )** wird aufgerufen und alle in dieser anzuzeigenden Buttons und Szenen (unterschiedliche Menüfester, z.B. Hauptmenü, Settings…) werden darin implementiert.

Das Interface **EventHandler** gibt die Methode **handler( )** vor, mit der alle Buttons mit Funktionen versehen werden.

4.3 Menünavigation

Startpunkt ist das Hauptmenü. Her kann über Buttons eine der vier Oberflächen der zweiten Reihe aufgerufen werden oder die Anwendung beendet werden.

Zu beachten ist, dass man vom Hauptmenü über das Einstellungsmenü nicht in das Pausenmenü kommt und andersherum.

5 Spielfeld

5.1 Playingfield

Die Klasse **Playingfield** fungiert als eine Art Container, in dem alle in einem Level vorkommenden Objekte gesammelt werden. Außerdem kann über das **Playingfield** auch auf die Objekte zugegriffen werden. Die **ArrayList obstacles** enthält alle Hindernisse, die auf dem **Playingfield** existieren. Da alle Hindernisse von der Klasse **Entity** erben, können sowohl runde, als auch eckige Hindernisse in der Liste gespeichert werden. Außerdem kann über das **Playingfield** noch auf folgende Elemente zugegriffen werden:

1. Die Kugel (**ball**)
2. Das Ziel (**target**)
3. Der Balken, der die Schusskraft repräsentiert (**bar**)
4. Die Punkteanzeige (**score**)
5. Den Windmesser (**wind)**
6. Das Katapult (**catapult**)
7. Den Luftwiderstand (**airDensity**)

5.2 PowerBar

Die Klasse **PowerBar** repräsentiert die Schusskraftanzeige am Rande des Bildschirms. Über die Methode **launch( )** wird die Anzeige gestoppt und die aktuelle Kraft wird zurückgegeben. Die anderen Methoden dienem dem Zugriff auf die beiden Attribute **power** (Kraft mit der auf das Lineal gedrückt wird) und **speed** (Geschwindigkeit, mit der die Anzeige fluktuiert).

5.3 ScoreBoard

Die Klasse **ScoreBoard** repräsentiert die Punkteanzeige über der Schusskraftanzeige am Bildschirmrand. Über die Methoden **add( )** sowie **reduce( )** kann der Punktestand (**score**) bearbeitet werden, ohne direkt auf ihn zugreifen zu müssen.

5.4 WindMeter

Die Klasse **WindMeter** repräsentiert den Windmesser in der rechten oberen Bildschirmecke. Über den Windmesser kann auf die Windrichtung (**direction**) sowie die Windstärke (**strength**) zugegriffen werden.

6 Spielfeldelemente

6.1 Vector

Die Klasse **Vector** dient dazu, im Zweidimensionalen möglichst einfach arbeiten zu können. Die Klasse **Vector** enthält zwei Attribute **x** und **y**. Durch diese Werte lassen sich sowohl Punkte, als auch Richtungen definieren. Durch die Methoden **addVector** sowie **multiplyVector** wird das Rechnen mit Vektoren vereinfacht.

6.2 Entity

Die Klasse **Entity** enthält grundlegende Elemente, die auch alle anderen Spielfeldelemente benötigen. In der **ArrayListe positions** wird gespeichert, wo das Element ist. Das Attribut **dampening** gibt an, wie hoch der Dämpfungsfaktor ist, falls es zu Kontakt mit einem anderen Element kommt.

6.3 Circle

Die Klasse **Circle** entspricht einem kreisförmigen Hindernis. Zusätzlich zu den in **Entity** bereits definierten Attributen gibt es hier noch das Attribut **radius**, welches den Radius des Kreises festlegt. Über die Methode **getBorders( )** werden näherungsweise alle Punkte zurückgeliefert, die auf dem Kreis liegen.

6.4 Ball

Die Klasse **Ball** entspricht der vom Katapult gefeuerten Kugel. Zusätzlich zu den in **Circle** definierten Attributen gibt es hier noch die Richtung der Kugel (**direction**), die Geschwindigkeit der Kugel (**speed**) und das Gewicht der Kugel (**weight**). Über die Methoden **speedUp( )**, sowie **speedDown( )**, kann die Geschwindigkeit der Kugel beeinflusst werden, ohne direkt auf den Wert zugreifen zu müssen.

6.5 TargetCircle

Die Klasse **TargetCircle** repräsentiert ein kreisförmiges Ziel. Sie enthält die gleichen Attribute wie die Klasse **Circle**.

6.6 Rectangle

Die Klasse **Rectangle** entspricht einem rechteckigen Hindernis. Über die Methode **getBorders( )** werden näherungsweise alle Punkte zurückgeliefert, die auf dem Hindernis liegen.

6.7 TargetRectangle

Die Klasse **TargetRectangle** repräsentiert ein rechteckiges Ziel. Sie enthält die gleichen Attribute wie die Klasse **Rectangle**.

6.8 Catapult

Das Attribut **rubber** ist ein Dreieck, **contactPoint** ist der Punkt, an dem sich Lineal und Dreieck berühren (der Drehpunkt).

- **fire( )** animiert das Lineal zu einer Drehbewegung
- **move*( )** ändert die Position des Lineals nach links oder rechts

7 Berechnung der Flugbahn

Die Berechnung der Flugbahn erfolgt in der Klasse **DragTrajectory**.

Die Methode **calculatePositions**() berechnet die einzelnen Positionen der Flugbahn, die dann in **mPositions** gespeichert werden.

Die Berechnung der Flugbahn mit Luftwiderstand beachtet folgende Faktoren:

- Fläche des Durchschnitts der Kugel
- Masse der Kugel
- Luftdichte (fest)
- Luftwiderstandskonstante ist abhängig von Form des Körpers

Die Berechnung der Flugbahn erfolgt schrittweise, da so am besten auf Kollision mit Objekten reagiert werden kann.