Dokumentation für die Billard-Simulation

Von

Vincenzo Di Renzo

Kurs: TIF19A

Vorlesung: Python

Inhalt

[1 Einleitung 1](#_Toc52289252)

[1.1 Ursprüngliche Projektbeschreibung 1](#_Toc52289253)

[1.2 Status Abgabe 1](#_Toc52289254)

[1.3 Installation 2](#_Toc52289255)

[1.4 Bemerkung 2](#_Toc52289256)

[2 Layout 3](#_Toc52289257)

[3 Kollisionen 4](#_Toc52289258)

[3.1 Kugel zu Kugel 4](#_Toc52289259)

[3.2 Kugel zu Bande 5](#_Toc52289260)

[3.3 Kugel zu Loch 5](#_Toc52289261)

[4 Module und Klassen 6](#_Toc52289262)

[4.1 Externe Abhängigkeiten 6](#_Toc52289263)

[4.2 Modul scenebase.py 6](#_Toc52289264)

[4.3 Modul main.py 6](#_Toc52289265)

[4.4 Modul titlescene.py 6](#_Toc52289266)

[4.5 Modul gamescene.py 7](#_Toc52289267)

[4.6 Modul elements.py 9](#_Toc52289268)

[4.6.1 Klasse “Ball” 9](#_Toc52289269)

[4.6.2 Klasse “Hole” 9](#_Toc52289270)

[4.6.3 Klasse “Line” 9](#_Toc52289271)

[4.6.4 Klasse “Queue” 9](#_Toc52289272)

[4.6.5 Klasse “Power” 9](#_Toc52289273)

[4.7 Modul tools.py 10](#_Toc52289274)

[5 Fazit 11](#_Toc52289275)

# Einleitung

## Ursprüngliche Projektbeschreibung

Billard-Simulation

Das Projekt ist ein Billardspiel, das allerdings nicht alle Funktionen eines Spiels mit sich

bringt. Interessant finde ich die (augenscheinliche) korrekte Physik der Stöße mit

Rollwiderstand.

In-Scope:

* GUI mit Spielfläche und Kugeln
* Steuerung per Tastatur
* Möglichkeit zu zielen und weiße Kugel in Bewegung zu versetzen
* In 1. Näherung korrekte Kollisionen von Kugeln untereinander
* In 1. Näherung korrekte Kollisionen von Kugeln zur Bande
* In 1. Näherung korrekte Kollisionen von Kugeln mit Löchern
* Kugeln besitzen Rollwiderstand
* Gleiche Spieleigenschaften bei verschiedener Hardwareleistung (verschiedenen

Clients)

Out-of-Scope:

* Verschiedene Gamescenes (Startmenu, Untermenu, Optionsmenu, Spielmenu)
* Korrekte und vollständige Implementierung von Spielregeln
* Verschiedene Spielmodi
* Mehrspieler 1 gegen 1, lokal, abwechselnd
* „Roll“ Animation der Kugeln

Ich finde es schwer einzuschätzen, wie groß der Aufwand für die „In-Scope“ Punkte sind und

würde ggf. Punkte aus dem „Out-of-Scope“ ebenfalls umsetzten.

## Status Abgabe

Letztendlich wurden folgende Aspekte umgesetzt bzw. nicht umgesetzt:

Umgesetzt:

* Startmenu (TitleScene)
* Spielmenu (GameScene)
* GUI: Billardtisch fester Größe (1680 x 880 Pixel)
* Korrekte Proportionen
  + Tisch zu Kugeln zu Löchern
* Zielen mit einem „Pfeil“ um 360°
* Einstellbare Schussstärke
* Kollisionen mit
  + horizontalen u. vertikalen Spielfeld-Banden
  + diagonalen Banden in den Ecklöchern
  + Bällen untereinander
  + korrekte Berechnung der resultierenden Winkel und Geschwindigkeiten angenommen 100% elastische Stöße
* Kugeln besitzen Rollwiderstand
* Gleiche Spieleigenschaften bei verschiedener Hardwareleistung

Nicht umgesetzt:

* Spielregeln
* Kugel-Spin
* Spielmusik
* Animationen (z.B. Kugelrollen)
* Kollisionen mit einzelnen Punkten (zu wenig Zeit gehabt)

## Installation

* Externe Abhängigkeiten installieren: „pip install -r requirements.txt“
  + (ggf. eine virtuale Environment z.B. mit “python -m venv venv” benutzen)
* Dann die main.py starten, über „python main.py“.

## Bemerkung

Diese Dokumentation könnte um einiges umfangreicher sein, allerdings finde ich, dass das den Rahmen sprengen würde. Ich habe versucht etwas Theorie zu den Kollisionen reinzubringen, und versuche zu jedem Modul und jeder Klasse den Zweck in kurzer Form zu übermitteln.

# Layout

Einfachheitshalber wurde das Programmfenster auf eine fixe Anzahl Pixel festgesetzt. Die Platzierung und Skalierung der einzelnen Elemente, wie Banden, Löcher und Kugeln und die daraus berechneten Kanten zur Kollisionserkennung wurden zu aufwendig, um die Größe dynamisch zu halten.

Wichtige Punkte wurden mit Google SketchUp ermittelt. (siehe Abbildung 2.1)

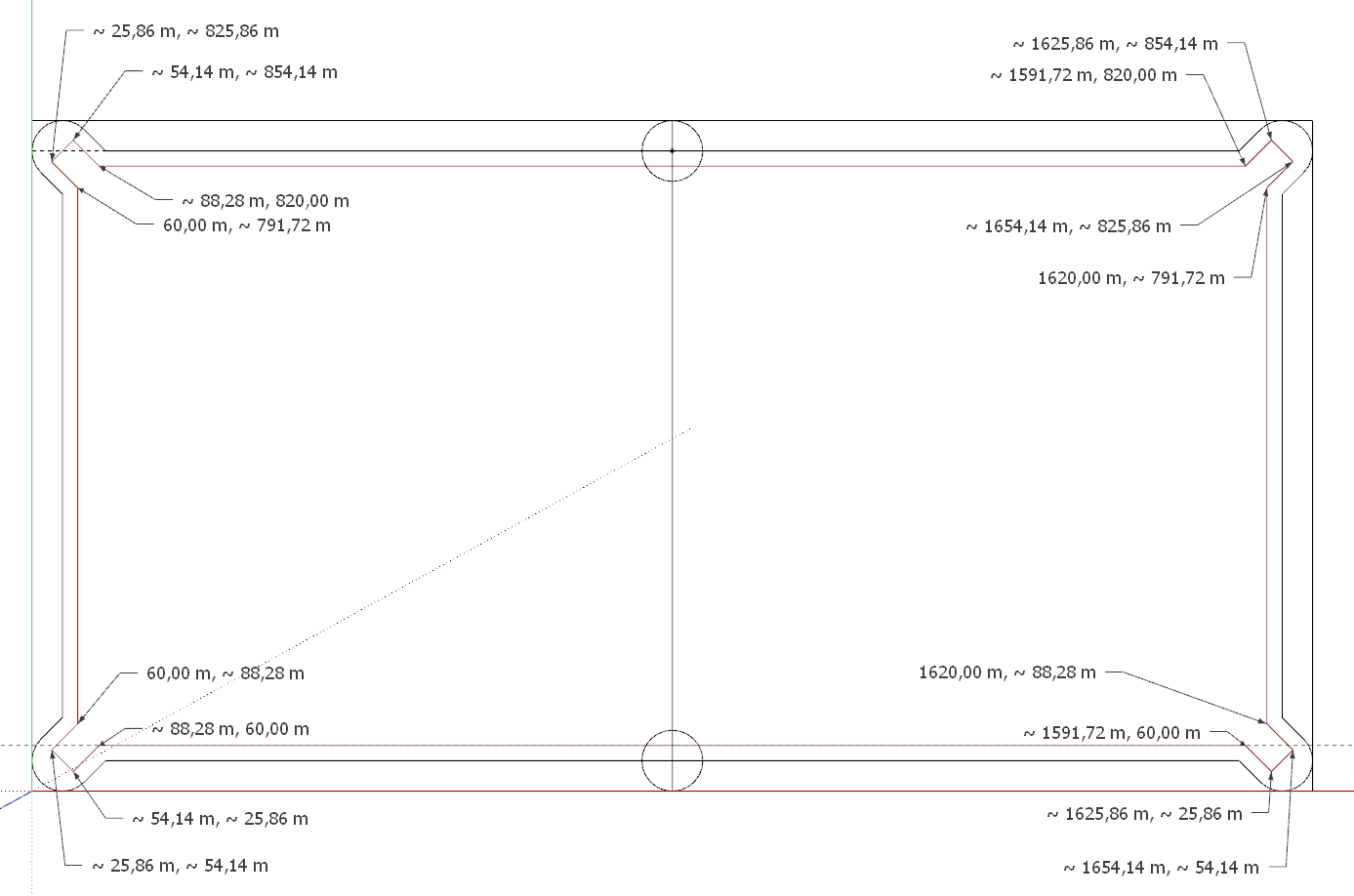


Abbildung . Das Tisch-Layout

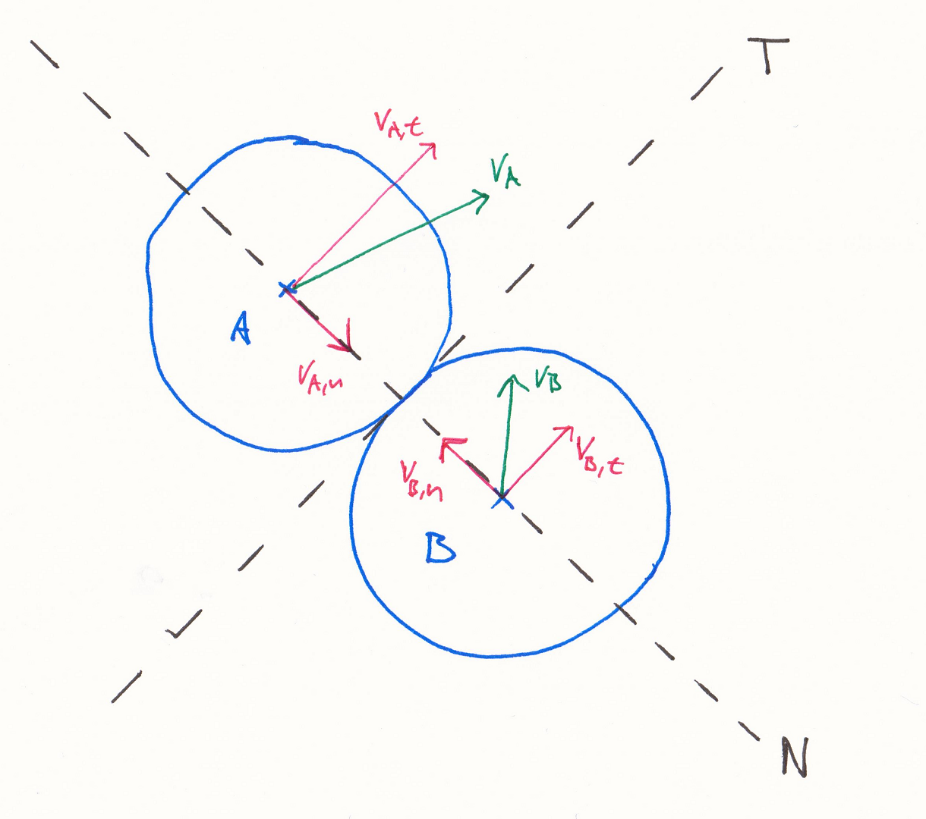
# Kollisionen

Alle Kollisionen sind zu 100% elastisch. D.h. eine Verringerung der Bewegungsenergie ist nur über zurückgelegten Weg möglich, da es eine Rollreibung gibt.

## Kugel zu Kugel

Im Folgenden wird vorgestellt, wie Kollisionen zwischen Kugeln berechnet werden. Kollisionen finden immer zwischen zwei Kugeln statt. Das Ergebnis einer Kollisionsberechnung zwischen zwei Kugeln ist, dass jeder Kugel ein neuer Geschwindigkeitsvektor zugeordnet wird. Um den Eindruck gleichzeitiger Kollisionen zu erwecken, werden alle Kugeln (d.h. alle Paarkombinationen) auf Kollision geprüft und berechnet, wodurch ggf. neue Geschwindigkeitsvektoren gesetzt werden, die im Laufe weiterer Frames die Kugel bewegen. Bei „gleichzeitigen“ Kollisionen, werden alle in einem Frame berechnet.

Die Kollision findet in 2D statt. Eine Bewegung in Z-Ebene ist nicht möglich. Die Berechnung ist allgemein gehalten. D.h. Winkel und Bewegungszustand zum Zeitpunkt des Kollidierens können beliebig sein.



Durch das Superpositionsprinzip lässt sich jeder schiefe elastischer Stoß auf einen zentralen elastischen Stoß reduzieren. Die Geschwindigkeiten der Kugeln (grünen Pfeile), lassen sich in tangentiale und normale Komponenten aufteilen (roten Pfeile). Die tangentialen Komponenten der beiden Kugeln verlaufen parallel zueinander und wechselwirken daher nicht miteinander. Der gesamte Stoß lässt sich auf die Wechselwirkung der normalen Komponenten zurückführen, welche einen zentralen elastischen Stoß gleicher Massen darstellen. Bei solch einem Stoß werden lediglich die Impulse, hier also Geschwindigkeiten, getauscht. vA,n wird zu vB,n und umgekehrt.

Die Herausforderung liegt also darin, die Anfangsgeschwindigkeit in die richtigen Komponenten aufzuteilen und den normalen Anteil zu vertauschen. Dies geschieht im Modul „elements.py“, Klasse „Ball“, Methode „ball\_to\_ball\_collision“ mit Hilfe von numpy und linearen Gleichungssystemen.

Eine Kollision wird erkannt, wenn der Abstand der Mittelpunkte zweier Kugeln den Durchmesser unterschreiten. Dann werden die Kugeln zuerst auseinandergezogen, bis sie sich nicht mehr überschneiden und anschließend wird die Kollision berechnet.

## Kugel zu Bande

Eine Kugel darf sich mit ihrem Mittelpunkt innerhalb der roten Linie bewegen, ansonsten wird eine Kollision mit der Bande detektiert (siehe Abbildung 2.1).

Bei einer Kollision mit der oberen bzw. unteren Bande wird die vertikale Komponente des Geschwindigkeitsvektors gespiegelt.

Bei einer Kollision mit der linken bzw. rechten Bande wird die horizontale Komponente des Geschwindigkeitsvektors gespiegelt.

Bei einer Kollision mit schrägen Linien eines Ecklochs, wird die Kollisionberechnung vom „Loch“-Objekt übernommen.

Kollisionen an einzelnen Punkten mit unvorhergesehenem Ausgang finden nicht statt.

## Kugel zu Loch

Ecklöcher besitzen innere Kanten im 45° Winkel, um das Einlochen interessanter zu gestalten. So kann eine Kugel auch mal wieder „herausspringen“.

Die Kollision an einer schrägen Linie gestaltet sich wie folgt:  
Jede Linie besitzt eine dazugehörige Funktionsgleichung.  
Zuerst wird geprüft, ob sich eine Kugel mit ihrem Mittelpunkt in der Nähe der Linie befindet. D.h. ist die X-Koordinate im Bereich start.x < X < end.x und die Y-Koordinate im Bereich start.y < Y < end.y.  
Dann kann abgefragt werden, ob sich die Kugel oberhalb oder unterhalb der Linie befindet, indem man die Bedingungen y > f(x) bzw. y < f(x) prüft. Welche der Bedingung zu einer Kollision führt hängt letztendlich von der Plazierung der Linie ab.

# Module und Klassen

## Externe Abhängigkeiten

Externe eingesetzte Packages sind:

* Numpy
* Pygame

Pygame wird zur Darstellung der GUI und zur Detektion von Tastatureingaben eingesetzt. Die GUI ist eine Zusammensetzung aus einfarbigen Textelementen, Kreisen, Rechtecken und Polygonen ohne Texturen. Lediglich die Kugeln besitzen eine Textur.

Numpy wird zur Vektorrechnung benötigt.

## Modul scenebase.py

Enthält die Klasse SceneBase, die ein Grundgerüst für einen Scene darstellt, um verschiedene „Bildschirme“ anzuzeigen.

Jede Scene muss die drei Methoden „ProcessInput()“, „Update()“ und „Render()“ enthalten. In jedem Frame wird jede dieser Methoden einmal aufgerufen. Sie managen die Tastatureingaben, die Spielberechnungen und zeichnen den graphischen Output. Darüber hinaus gibt es die Methode „SwitchToScene()“ um die Scene zu wechseln und „Terminate()“ um die Scene und das Programm zu beenden.

## Modul main.py

Die main.py dient als Einstiegspunkt des Programms. Die Start-Scene wird festgelegt und die unendliche Schleife, in der die drei Haupt-Methoden einer Scene (ProcessInput, Update, Render) gerufen werden, wird gestartet. Darüber hinaus, horcht die main.py auf Eingaben, die das Programm beenden würden.

In jedem Frame wird die Scene zur active\_scene.next gewechselt, die aber in der Regel auf sich selbst zeigt. Erst wenn active\_scene.next auf eine andere Scene gesetzt wird (über die Methode SwitchToScene) findet ein Scene-Wechsel statt.

## Modul titlescene.py

Enthält die Klasse TitleScene, die das Hauptmenu wie in Abbildung 4.1. unten darstellt. Es werden lediglich pygame-Textobjekte gerendert und auf die Eingabe der Enter, Backspace oder Escape Taste gehorcht.

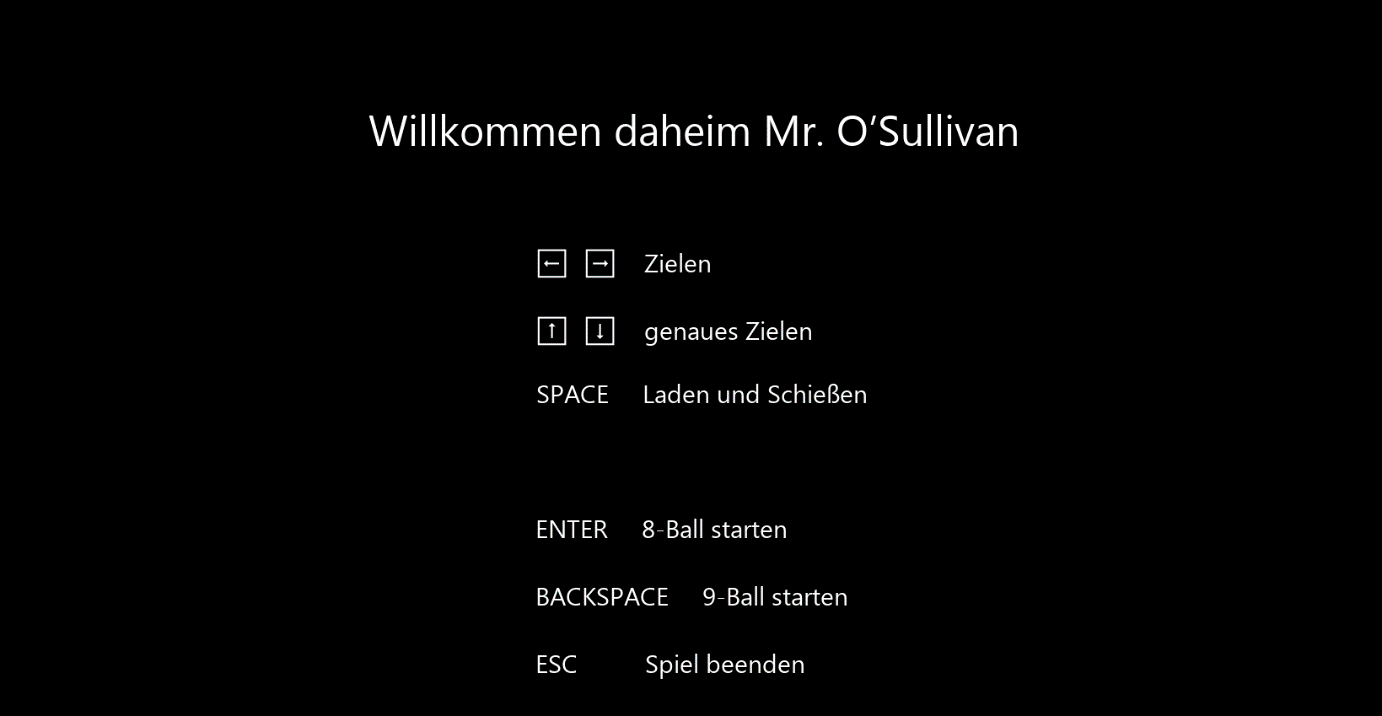


Abbildung . Das Hauptmenu (TitleScene)

## Modul gamescene.py

Enthält die Klasse GameScene, die den Spielmodus wie in Abbildung 4.2 darstellt.  
Die erforderlichen Bausteine für die GameScene sind

* Die Löcher
* Die Kugeln
* Ein Queue (der Zielpfeil)
* Eine Poweranzeige
* Der Tisch

Der Tisch ist rein optisch und wird über das Hilfsmodul tools.py gezeichnet.

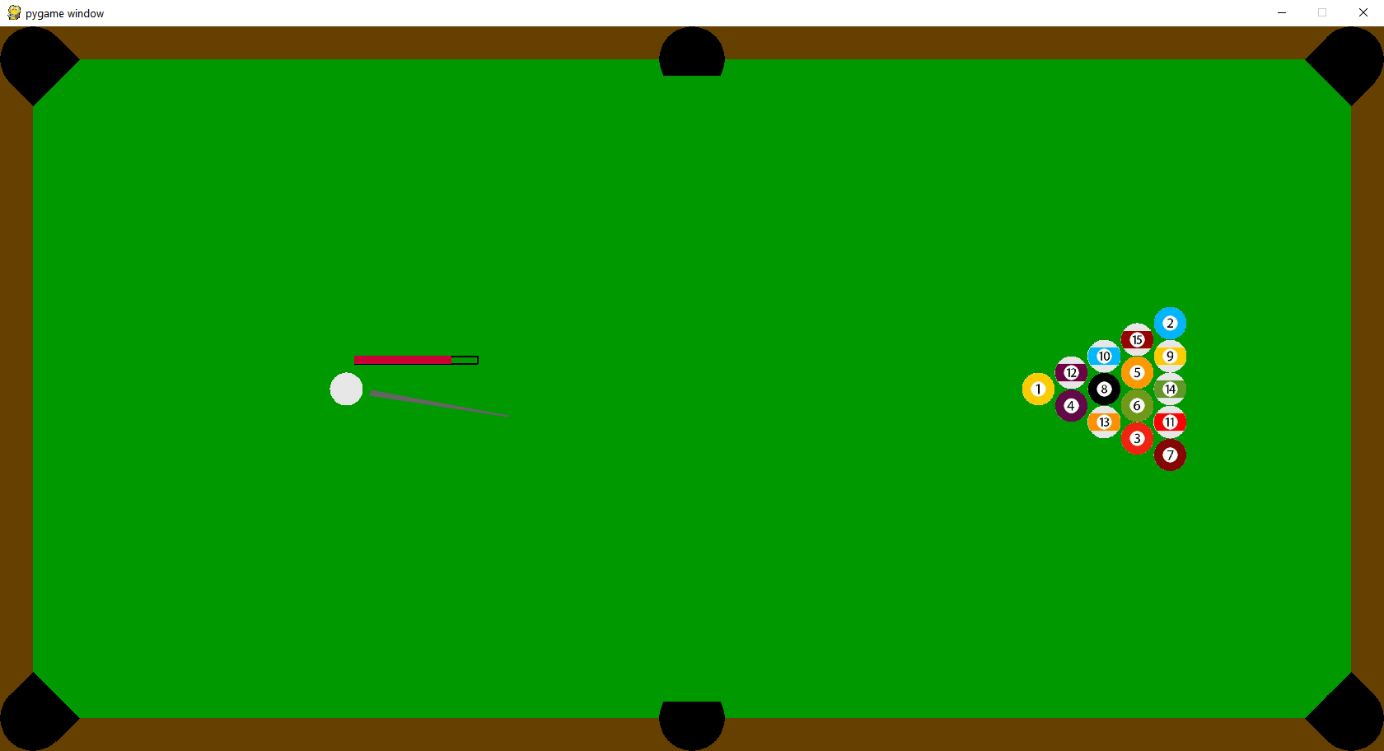


Abbildung . Das Spielmenu (GameScene)

Alle anderen Bausteine sind in Klassen umgesetzt, die sich in der elemets.py befinden. Jeder Baustein bringt die für sich wichtigen Methoden mit, um korrekt zu funktionieren. Dazu mehr im Kapitel elements.py

Die main.py ruft in jedem Frame die drei Methoden ProcessInput(), Update() und Render() des GameScene-Objekts auf, die den Spielfluss steuern.

In ProcessInput()

* werden die Tastatureingaben abgefangen, wie in Abbildung 4.1 beschrieben
* beim Druck der Leertaste wird die Powerleiste geladen
* beim Loslassen der Leertaste wird die gesammelte Power abgefragt und die weiße Kugel bewegt

In Update()

* Werden die Kugeln aktualisiert, d.h.
  + Falls die jeweilige Kugel nicht eingelocht wurde
    - Sie bewegen sich in Abhängigkeit von der Zeit seit dem letzten Frame
    - werden Kollisionen mit anderen Kugeln geprüft
    - werden Kollisionen mit Banden geprüft
    - werden Kollisionen mit Löchern geprüft
  + Falls die weiße Kugel gelocht wurde
    - wird ein neuer freier Platz gesucht

In Render()

* Wird der Tisch gezeichnet (mit Hilfe von tools.Tools.draw\_table)
* Falls die weiße Kugel still steht wird der Queue gezeichnet
* Die Powerbar wird gezeichnet

## Modul elements.py

### Klasse “Ball”

Die Klasse einer Billardkugel.  
Enthält unter anderem das Attribut der Geschwindigkeit, ob gelocht, ob in Bewegung und eine absolute Position.

In der Update-Methode wird die Kugel solange neu positioniert, bis die Geschwindigkeit 0 ist. Die Geschwindigkeit verringert sich mit jedem Frame.

Die Methoden „ball\_to\_ball\_collision“ und „ball\_to\_border\_collision“ prüfen auf Kollision und verarbeiten diese. Sie werden von der GameScene in jedem Frame gerufen.

### Klasse “Hole”

Die Klasse eines Lochs.  
Enthält unter anderem das Attribut des Typs. Abhängig vom Typ (rechts oben, links unten, …) wird beim Rendern ein anderes Polygon gewählt und die Linien zur Kollisionsberechnung werden anders gewählt.  
self.upper\_line und self.lower\_line sind Banden der Ecklöcher.  
self.goal\_line beschreibt eine Linie (mit Anfang und Ende, keine Gerade), die beim Überschreiten ein Einlochen signalisiert.

Die Methode collision() prüft, ob eine Kugel gelocht wurde. Aufgrund der unterschiedlichen Natur von Ecklöchern und Mittellöchern muss hier eine unterscheidung stattfinden.

### Klasse “Line”

Eine Hilfsklasse, die für Ecklöcher verwendet wird.

Die \_\_init\_\_() nimmt zwei Punkte als Input und berechnet daraus die Funktionsgleichung.   
Die zwei Methoden is\_above\_line() und is\_below\_line() prüfen, ob sich ein Punkt darüber bzw. darunter befinden. Anmerkung: die y-Achse ist invertiert, Vergleichsoperatoren sind dadurch auch invertiert.

### Klasse “Queue”

Die Klasse Queue stellt den Zielpfeil dar. Er lässt sich langsam und schnell rotieren und besitzt eine Richtung die abgefragt werden kann. Der Zielpfeil befindet sich immer an der weißen Kugel.

Die Rotation bei Tastendruck wurde händisch umgesetzt. D.h. der Zielpfeil wird zuerst zum Ursprung verschoben (Translation), dann rotiert, und dann wieder an die weiße Kugel gesetzt. Es gibt Rotationsmatrizen für die schnelle und langsame rotation.

### Klasse “Power”

Eine Powerbar. Wird nur gerendert, falls „active“. Ist nur active, falls geladen.

Solange die Leertaste gedrückt wird, akkumuliert die Powerbar „power“ (ein Zahlenwert). Zahlenwert kann abgefragt werden, wird dabei auf 0 gesetzt.

## Modul tools.py

Ein Hilfsmodul, das Hilfsarbeiten übernimmt.

Die Methode draw\_table(), zeichnet den Tisch ohne Bälle und Löcher. Der Tisch ist kein eigenständiges Objekt und enthält keine Funktionalität.

Die Methode create\_balls() erzeugt eine Ballformation (8-Ball oder 9-Ball), mischt die Positionen, erzeugt die Kugelobjekte (mit Zuordnung der Textur) und returned eine Liste dieser Kugeln.

Die Methode create\_holes() erzeugt die Löcher und returned eine Liste von Löchern.

Die Methode jederMitJedem()returned eine Liste von Kugelpaaren, die benutzt wird um Kollisionen zwischen den Kugeln zu berechnen. Diese Kombinationen werden nur einmal vor Spielbeginn erzeugt. Doppelte Paarungen werden ausgelassen. (1te Kugel mit den restlichen 14, 2te mit den restlichen 13, usw…)

Die Methode find\_free\_pos() findet eine neue Position für die weiße Kugel, falls sie gelocht werden sollte. Ausgehend von einem Startpunkt werden mit Hilfe einer Archimedische Spirale, die Positionen um den Startpunkt herum auf genügend Platz geprüft.  
In „if \_\_name\_\_ == "\_\_main\_\_":“ lässt sich die Spirale begutachten.

Die Methode arrow() wurde bei der Entwicklung genutzt, um einen kleinen roten Pfeil an einer bestimmten Position zu zeichnen. Wird für die normale Ausführung nicht gebraucht.

Eine Klasse „Color“, die Farben enthält. Allerdings konnte sie in elements.py aufgrund zirkulär Abhängigkeiten nicht genutzt werden.

# Fazit

Was kann man besser machen:

* Mehr Kommentare im Code
* Es werden fast keine Fehler abgefangen
* Es gibt keine Tests, keine Qualitätssicherung
* Es gibt keine Möglichkeiten aus dem Spielmenu ins Hauptmenu zu kommen, da dies eine zirkuläre Abhängigkeit erzeugt. Lösung: Ein zusätzliches Modul anlegen um die zirkuläre Abhängigkeit zu lösen.
* Es gibt kein Spielende =)
* Viel zu schwer zu spielen. Lieber auf korrekte Proportionen verzichten und das Spiel leichter gestalten. Evtl. ein besseres Zielsystem einsetzen

Alles in allem, wurden die Spezifikationen der ursprünglichen Projektbeschreibung alle erfüllt und darüber hinaus auch eine Möglichkeit eingebaut verschiedene „Szenen“ darzustellen. Das Spiel, das kein Spiel ist und deshalb Simulation betitelt wurde, macht keinen Spaß, da es sinnlos ist. Aber die Kollisionen, die im Endeffekt viel leichter auszurechnen waren als gedacht, sind toll anzusehen, finde ich (vielleicht auch nur weil ich‘s selbst geschrieben hab :D)