

# Zagraj w hokeja z ładunkami elektrycznymi

## Dokumentacja

Kamil Sikora, Maciej Ładoś, Michał Bar

April 2020

## Spis treści

<b>1</b>	<b>Opis projektu</b>	<b>3</b>
<b>2</b>	<b>Model fizyczny</b>	<b>3</b>
<b>3</b>	<b>Model symulacyjny</b>	<b>4</b>
<b>4</b>	<b>Implementacja</b>	<b>5</b>
4.1	Struktura kodu . . . . .	5
4.2	Wybrane stałe . . . . .	5
4.3	Wybrane modele i ich metody . . . . .	6
4.4	Kontrollery . . . . .	9
<b>5</b>	<b>Literatura</b>	<b>9</b>

## 1 Opis projektu

Projekt polega na symulacji gry w hokeja z dodatnio naładowanym krążkiem poruszającym jedynie poprzez wpływ innych ładunków elektrycznych. Gracz sam decyduje o rozmieszczeniu ładunków. Celem gry jest umieszczenie krążka w bramce. Gra kończy się po strzeleniu gola lub dotknięciu przez krążek któregoś z ładunków bądź przeszkód.

Projekt zdecydowaliśmy się wykonać w JavaScript

## 2 Model fizyczny

Fizyka w grze jest oparta na prawie Coulomba, którego treść brzmi: Siła wzajemnego oddziaływania dwóch naładowanych cząstek jest wprost proporcjonalna do iloczynu wartości tych ładunków i odwrotnie proporcjonalna do kwadratu odległości między nimi.

$$F = k \frac{q_1 \cdot q_2}{r^2}$$

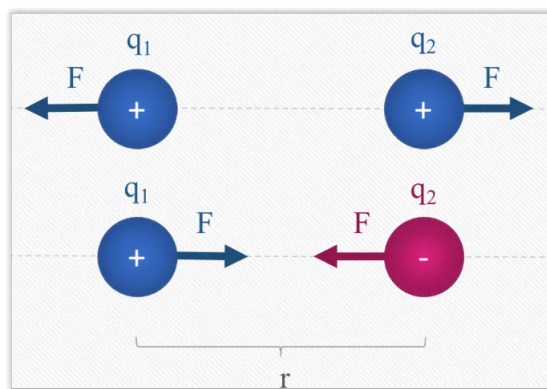
$F$  - siła elektrostatyczna

$q_1, q_2$  - ładunki elektryczne

$r$  - odległość

$k$  - stała elektrostatyczna w przybliżeniu równa  $9 \cdot 10^9 \frac{N \cdot m^2}{C^2}$

Oba ładunki mają również masę, lecz w świecie cząstek, atomów i cząsteczek chemicznych, możemy całkowicie zaniedbać ich wzajemne oddziaływania grawitacyjne.



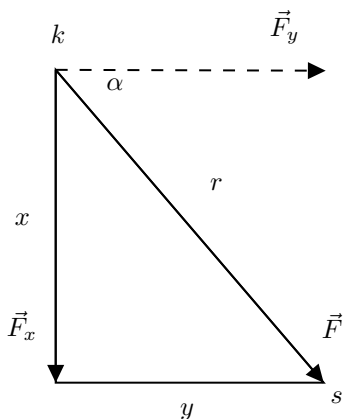
Rysunek 1: Prawo Coulomba- oddziaływanie ładunków

### 3 Model symulacyjny

Nasza symulacja operuje w płaszczyźnie dwuwymiarowej. Każdy obiekt będzie posiadał dwie współrzędne -  $x$  i  $y$ . Odległość między ciałami może być zatem wyznaczona jako odległość punktów na płaszczyźnie.

$$r = \sqrt{(x_b - x_a)^2 + (y_b - y_a)^2}$$

W celu uproszczenia obliczeń siłę  $\vec{F}$  oddziałującą na ładunki rozkładamy na składowe  $\vec{F}_x$  oraz  $\vec{F}_y$ . Zakładamy, że środkiem naszego układu współrzędnych będzie krążek – pozwoli nam to na skorzystanie z funkcji trygonometrycznych do wyliczenia tych składowych.



Korzystając z funkcji trygonometrycznych otrzymamy:

$$\sin \alpha = \frac{\vec{F}_x}{r} = \frac{x}{r} \quad \cos \alpha = \frac{\vec{F}_y}{r} = \frac{y}{r}$$

Przyjmujemy, że wartości ładunków są takie same, więc pomijamy je w obliczeniach. Wyprowadzamy wzory na składowe:

$$\vec{F}_x = \frac{\sin \alpha}{r^2} \quad \vec{F}_y = \frac{\cos \alpha}{r^2}$$

Korzystając z II zasady dynamiki Newtona, wyprowadzamy wzór na przyspieszenie ładunku:

$$F = m \cdot a \Rightarrow a = \frac{F}{m}$$

Ostatecznie wzory:

$$a_x = k \cdot \frac{\sin \alpha}{m \cdot r^2} \quad a_y = k \cdot \frac{\cos \alpha}{m \cdot r^2}$$

## 4 Implementacja

### 4.1 Struktura kodu

1. const - folder zawiera pliki przechowujące wszelkie stałe wykorzystywane w programie
2. assets - folder z plikami graficznymi oraz plikiem css
3. controllers - są tu zawarte klasy obsługujące interakcje klienta z programem oraz akcje wykonywane przez program
4. events - zawiera klasę która zapewnia obsługę zdarzeń w grze
5. extensions - folder z rozszerzeniami wykorzystywanymi w programie
6. models - zawarte są tu wszelkie klasy tworzące obiekty w grze oraz podfolder physics, który przechowuje obiekty fizyczne oraz model fizyczny

### 4.2 Wybrane stałe

1. COULOMB\_FORCE\_FACTOR - odpowiada stałej  $k$  z obliczeń z sekcji 'Model Symulacyjny'
2. CHARGE\_MIN\_DISTANCE - minimalna odległość na jakiej utrzymywane są ładunki, aby nie dopuścić do współistnienia krążka i ładunku w tym samym miejscu i czasie zgodnie z zasadą Pauliego. W związku z tym zakładamy w naszym modelu dystans między ładunkami na minimum 25 jednostek odległości
3. PUCK\_VELOCITY\_DIVIDER - dzielnik prędkości aby wizualizacja symulacji była lepiej dostrzegalna

### 4.3 Wybrane modele i ich metody

1. *BoundingBox* - główna klasa zapewniająca ruch oraz kolizje obiektów na planszy. Jej pola to  $x, y$  składowe położenia oraz szerokość *width* i wysokość *height* obiektu.

Zawiera trzy metody do obsługi kolizji: *touches()*, *contains()*, *intersects()*. *move()* przypisuje nowe położenie obiektu na podstawie przekazanych parametrów dotyczących przemieszczenia.

*distance()* oblicza dystans między obiektami w grze.

2. *GameObject* - rozszerza *BoundingBox*, bazowa abstrakcyjna klasa dla wielu modeli w grze. Deklaruje metody *update()*, *render()*.
3. *Group* - klasa implementująca działania na tablicach obiektów tworzonych jako grupy w grze
4. *HockeyGoal* - klasa implementująca rysowanie bramki na podstawie tworzonych *BoundingBox*
5. *Obstacle* - klasa dzięki której tworzy i renderuje przeszkody w grze
6. *Trace* - klasa wykorzystywana przez klasę *Puck* do rysowania trasy krążka
7. *ElectricCharge* - bazowa klasa dla klas *NegativeCharge*, *PositiveCharge* tworząca odpowiedni ładunek na podstawie przekazanego typu *type*. Typ jest zawarty domyślnie w konstruktorach klas *NegativeCharge*, *PositiveCharge*.
8. *Puck* - klasa dziedzicząca po *ElectricCharge* tworząca krążek jako ładunek ujemny lub dodatni.  
Krążek posiada własną masę *mass* oraz promień *radius*.  
*acceleration*, *velocity* to odpowiednio przyspieszenie i prędkość krążka potrzebne do wyznaczania jego ruchu.  
*trace* gdzie przypisany jest obiekt klasy *Trace* oraz *traceIsActive* dotyczą możliwości rysowania przebytej przez krążek trasy.  
*update()*, *render()* zapewniają odpowiednio ruch oraz rysowanie krążka.

9. VectorField - klasa tworząca obiekty składające się na pole wektorowe. Posiada ona ustalony zbiór punktów, i dla każdego z nich oblicza siłę która działałaby na krążek w danym miejscu. Następnie dla koordynatów tworzony jest obiekt *FieldVector*, który rysowany jest na bazie przekazanej mu siły. Klasa pola wektorowego oblicza również medianę wszystkich sił, by w prosty sposób można było na jej podstawie odpowiednio przeskalować i pokolorować wektory, w celu lepszej wizualizacji pola.

```
20  ..render(ctx){
21  ....const angle = Math.atan2(this.fy, this.fx);
22  ....const dx = Math.cos(angle) * this.length;
23  ....const dy = Math.sin(angle) * this.length;
24  ....let oldColor = ctx.fillStyle;
25
26  ....ctx.fillStyle = this.color;
27  ....ctx.beginPath();
28  ....ctx.arrow(this.x - dx, this.y - dy, this.x + dx, this.y + dy, [
29  .....0,
30  .....1,
31  .....-10,
32  .....1,
33  .....-10,
34  .....5,
35  ....]);
36  ....ctx.fill();
37  ....ctx.fillStyle = oldColor;
38  ..}
```

Rysunek 2: Funkcja rysująca wektor, kształt strzałki wyznaczany jest na bazie jej bazowej długości i kąta wektora siły. Implementacja wyznaczania siły używa metody calculate opisanej poniżej. Podmieniana jest jedynie pozycja krążka by zasymulować oddziaływanie w punkcie.

10. *CoulombForce* - klasa implementująca obliczenia modelu fizycznego. Oblicza przyspieszenie krążka na podstawie siły między dwoma przekazanymi ładunkami. Jej obiekt jest tworzony podczas dodawania ładunku w *GameController* i dodawany do *forces* w klasie *Game*

```
calculate() {
    this.displacement = {
        x: this.charge2.x - this.charge1.x,
        y: this.charge2.y - this.charge1.y,
    };

    this.r = Math.max(this.charge1.distance(this.charge2), CHARGE_MIN_DISTANCE);
    this.rCube = Math.pow(this.r, 3);

    this.x = this.calculateComposite(this.displacement.x);
    this.y = this.calculateComposite(this.displacement.y);
}
```

Rysunek 3: Funkcja *calculate* obliczająca przyspieszenie krążka na podstawie siły między ładunkami

*calculate* - główna funkcja obliczająca przyspieszenie wypadkowe krążka, wykorzystywana w *updateForces()* w *Game*

*displacement* przechowuje przemieszczenie między krążkiem a ładunkiem dla danych składowych

*r* odległość między krążkiem a ładunkiem

*rCube* odległość podniesiona do sześcianu, upraszcza obliczenia w *calculateComposite*

*x, y* składowe wyznaczonej siły

*calculateComposite()* funkcja oblicza przyspieszenie dla podanego przemieszczenia składowego.

Odpowiednik w modelu fizycznym

$$a_x = k \cdot \frac{\sin \alpha}{m \cdot r^2} \quad a_y = k \cdot \frac{\cos \alpha}{m \cdot r^2}$$

Odpowiednikiem sinusa lub cosinusa jest  $\frac{\text{displacement}}{r}$ . Jako że w mianowniku występuje również  $r^2$ , wyrażenie  $\frac{\text{displacement}}{r \cdot r^2}$  zostało zastąpione przez  $\frac{\text{displacement}}{r^3}$

Czynniki *charge.getSign()* w liczniku odpowiadają za wyznaczenie odpowiedniego kierunku przyspieszenia



## 4.4 Kontrollery

1. Controller - bazowa klasa dla pozostałych kontrolerów zawierająca pole *eventBus* przechowujące obiekt klasy *Eventbus* do obsługi zdarzeń w grze.
2. InputController - klasa obsługująca zdarzenia wymuszone przez klienta przez funkcjonalności takie jak: przycisk, checkbox, pole typu 'input' i kliknięcia myszą. Dla każdego typu interakcji użytkownika z programem jest zaimplementowana jedna z funkcji *registerButtonListeners()*, *registerCheckboxListeners()*, *registerRadioListeners()*, *registerInputListeners()*, *registerMouseListener()* wysyłająca odpowiedni komunikat dzięki *eventBus.emit()*.
3. GameController - klasa obsługująca wszelkie zdarzenia w grze.  
*game* pole przechowujące aktualną sesję gry.  
*registerListeners()* główna funkcja implementująca reakcję na konkretne zdarzenie wysłane przez grę lub użytkownika poprzez *InputController*. Przez *eventBus.on()* sprawdza czy aktywne jest dane zdarzenie  
*clear()* wyczyszczenie planszy i stanu gry  
*onDifficultyChange()* - funkcja reagująca na zmianę poziomu trudności  
*placeCharge()* - funkcja obsługująca umiejscowienie ładunków na planszy  
*showGoalMessageAnimation()*, *showFailureMessageAnimation()* - funkcje wyświetlające animacje tekstu w zależności od zdarzenia

## 5 Literatura

1. Zbigniew Kąkol, Kamil Kutorasiński. Prawo Coulomba.
2. Zbigniew Kąkol. Fizyka. Kraków 2019
3. David Halliday, Robert Resnick, Jearl Walker. Podstawy fizyki. Tom 3