Zagraj w hokeja z ładunkami elektrycznymi

Dokumentacja

Kamil Sikora, Maciej Ładoś, Michał Bar

April 2020

Spis treści

1 Opis projektu

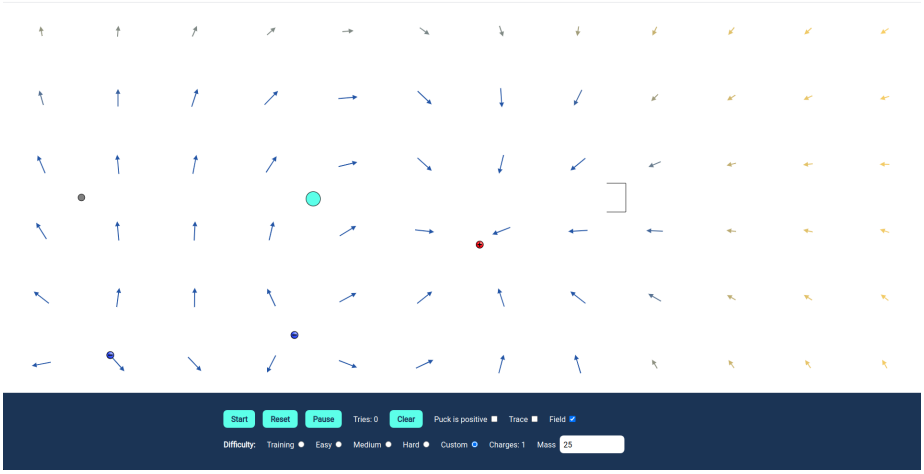
Projekt polega na symulacji gry w hokeja z dodatnio naładowanym krążkiem poruszanym jedynie poprzez wpływ innych ładunków elektrycznych. Gracz sam decyduje o rozmieszczeniu ładunków. Celem gry jest umieszczenie krążka w bramce. Gra kończy się po strzeleniu gola lub dotknięciu przez krążek któregoś z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

2 Możliwości symulatora

Aplikacja pozwala na śledzenie dokładnego zachowania ładunku w polu elektrycznym, jednocześnie udostępniając ciekawą rozgrywkę zachęcającą gracza do dokładniejszego prześledzenia interakcji wzajemnie oddziałujących między sobą ładunków. Lewy przycisk myszki dodaje ładunek ujemny we wskazane miejsce na planszy, a prawy dodatni.

Aby ułatwić wizualizację całego zachowania, możemy włączyć pole wektorowe które pozwoli nam zaobserwować siły działające na krążek w danym punkcie. Gracz może również po zaznaczeniu opcji "Traceżysować ślad poruszającego się krążka, a także z użyciem opcji "Puck is positiveźmienić jego domyślny ładunek z dodatniego na ujemny.

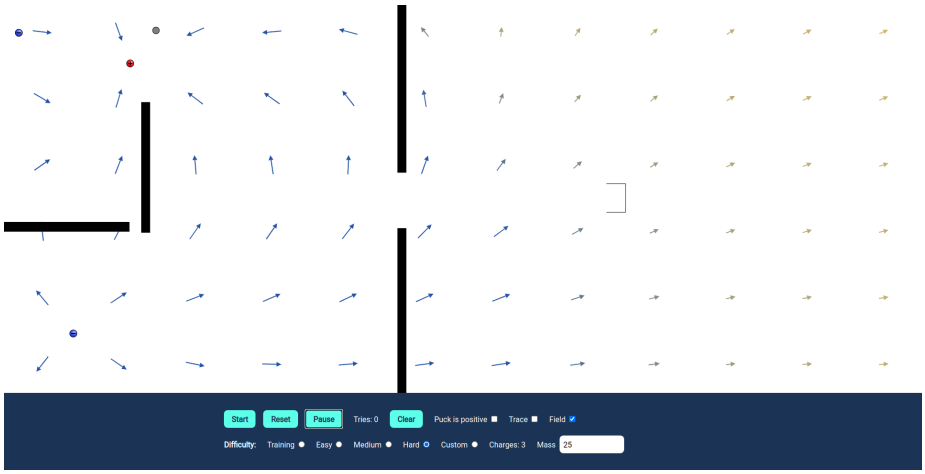
Możemy również wyczyścić planszę, zatrzymać i wznowić grę, przywrócić krążek na pozycję startową, czy też zmienić masę krążka wpływając co zmienia siłę oddziaływania na niego przez ładunki.



Rysunek 1: Pole wektorowe, izotop

Dla urozmaicenia i utrudnienia zabawy, użytkownikowi udostępnione zostały poziomy różnicujące poziom rozgrywki. Przeszkody w nich zawarte blokują

przebieg krążka, i w razie ich dotknięcia kończy się rozgrywka. Poziom Ćustom"dodaje jeszcze specjalny obiekt - izotop, który emituje ładunki w losowej chwili i o zmiennym kierunku zmieniając tym samym rozkład sił na planszy i tym samym jeszcze bardziej wzobgacając zabawę.



Rysunek 2: Poziom z przeszkodami

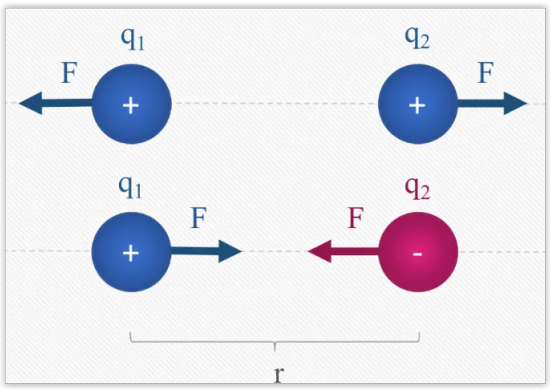
3 Model fizyczny

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.

$$F = k\frac{q1 \cdot q2}{r^2}$$

F - siła elektrostatyczna
$q1, q2$ - 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.



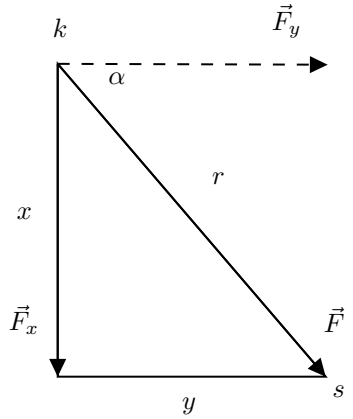
Rysunek 3: Prawo Coulomba- oddziaływanie ładunków

4 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:

$$\sin\alpha = \frac{\vec{F_x}}{r} = \frac{x}{r} \qquad \cos\alpha = \frac{\vec{F_y}}{r} = \frac{y}{r}$$

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:

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

5 Opis gry

Celem gry jest umieszczenie krążka w bramce za pomocą wstawiania odpowiednich ładunków i w ten sposób generowania pola elektrycznego. Za pomocą prawego przycisku myszy wstawiamy ładunki dodatnie, a za pomocą lewego ujemne. Znak krążka który domyślnie jest ujemny możemy zmienić zaznaczając/odznaczając pole 'Puck is positive'. Dostępne są cztery poziomy trudności do wyboru: training, easy, medium, hard. Różnią się one ilością i rozmieszczeniem przeszkód. W grze dostępne mamy przyciski do zarządzania akutalnym stanem planszy. 'Start' i 'Stop' służą rozpoczynania i zatrzymywania symulacji, 'Reset' ustawia pozycję krążka na domyślną, 'Clear' usuwa ładunki i ustawia pozycję krążka na domyślną. Licznik 'Tries' to ilość naszych błędnych prób, 'Charges' to ilość ładunków na planszy. Istnieje możliwość narysowania trasy przebytej przez krążek za pomocą pola 'Trace' oraz sił pola wektorowego za pomocą pola 'Field'.

6 Implementacja

6.1 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

6.2 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

6.3 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 obektów tworzonych jako grupy w grze

4. HockeyGoal - klasa implementująca rysowanie bramki na podstawie tworzonego *BoundingBoxa*

5. Obstacle - klasa dzięki która 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.

```
  render(ctx) {
    const angle = Math.atan2(this.fy, this.fx);
    const dx = Math.cos(angle) * this.length;
    const dy = Math.sin(angle) * this.length;
    let oldColor = ctx.fillStyle;

    ctx.fillStyle = this.color;
    ctx.beginPath();
    ctx.arrow(this.x - dx, this.y - dy, this.x + dx, this.y + dy, [
      0,
      1,
      -10,
      1,
      -10,
      5,
    ]);
    ctx.fill();
    ctx.fillStyle = oldColor;
  }
```

Rysunek 4: 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,  y: 3);

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

Rysunek 5: 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 wyznaczanej 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 $\frac{displacement}{r}$. Jako że w mianowniku występuje również r^2, wyrażenie $\frac{displacement}{r \cdot r^2}$ zostało zastąpione przez $\frac{displacement}{rCube}$

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

6.4 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 kliknęcia myszą. Dla każdego typu interakcji użytkownika z programem jest zaimplementowana jedna z funkcji *registerButtonListeners(), registerCheckboxListeners(), registerRadioListeners(), registerInputListeners(), registerMouseListeners()* wysyłająca odpowiedni komunikat dzięki *eventBus.emit()*.

3. GameControler - 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

7 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