# Cel Projektu

## Główny cel

Głównym celem projektu jest stworzenie systemu informatycznego symulującego zachowanie różnych obiektów pływających o napędzie żaglowym. Użytkownik systemu będzie mógł sterować parametrami statku(ustawienie żagli, ustawienie steru itd.), natomiast system będzie w czasie rzeczywistym symulował zachowanie statku i prezentował je użytkownikowi na ekranie. Symulacja będzie realizowana w przestrzeni dwuwymiarowej. System, poprzez prostą mechanikę symulacji mam wyjaśnić użytkownikowi np. jak to się dzieje ,że statek jest w stanie ‘płynąć pod wiatr’, albo jak ustawić żagle żeby dokonać zamierzony zwrot. Celem projektu nie jest stworzenie dokładnego symulatora z zaawansowaną mechaniką płynów, lecz mocno uproszczonego, lecz wciąż prezentującego esencję działania żaglowców.

## Cele dodatkowe

* Rozbudowa systemy w kierunku gry multiplayer.

## Wytrenowanie sieci neuronowej w celu sterowania okrętem ([Reinforcement machine learning](https://en.wikipedia.org/wiki/Reinforcement_learning)).

## sidebar\_position: 2

# Opis Wycinka Rzeczywistości

## Żaglowiec

[Żaglowiec](https://pl.wikipedia.org/wiki/%C5%BBaglowiec) (statek żaglowy) – statek wodny o napędzie żaglowym. Jednostka pływająca, której jedynym lub podstawowym czynnikiem napędowym jest jeden lub więcej żagli. Na danej jednostce przygotowanej do żeglugi zbiór wszystkich możliwych do zastosowania na niej rodzajów żagli stanowi jej aktualne ożaglowanie, przy czym nie wszystkie rodzaje tych żagli muszą być użyte jednocześnie.

pełnorejowiec Royal Clipper

Głównymi elementami umożliwiającymi poruszanie się statku są [żagiel](#żagiel)(oddziaływający z powietrzem) i miecz(oddziaływający z wodą). Zasada działania żagla i miecza jest niemal identyczna, a w mojej symulacji będzie dokładnie taka sama.

Przyjrzyjmy się zatem bliżej jednemu z nich np [żaglowi](#żagiel).

## Żagiel

[Żagiel](https://pl.wikipedia.org/wiki/%C5%BBagiel) – rodzaj pędnika wiatrowego stosowanego do napędzania żaglowców, jachtów, bojerów, żaglowozów itd. Jest to odpowiednio ukształtowany płat tkaniny (ew. innego tworzywa) rozpięty na omasztowaniu jednostek żaglowych, stawiany fałami i kierowany szotami lub brasami (najczęściej przy użyciu lin pomocniczych). Zadaniem żagla jest wytworzenie [siły aerodynamicznej](#siła-aerodynamiczna) napędzającej jednostkę. Zespół żagli tworzy ożaglowanie.

Przyjrzyjmy się temu co najważniejsze w żaglu, czyli sile aerodynamicznej ## Siła Aerodynamiczna

[Siła aerodynamiczna](https://pl.wikipedia.org/wiki/Si%C5%82a_aerodynamiczna) – siła wywierana na ciało przez powietrze lub inny gaz, w którym ciało jest zanurzone, będąca wynikiem ruchu ciała względem gazu. Siła wynika z dwóch przyczyn: + [siły parcia](https://pl.wikipedia.org/wiki/Parcie_hydrostatyczne) wywołanej ciśnieniem wywieranym na powierzchnię ciała, + [siły lepkości](https://pl.wikipedia.org/wiki/Lepko%C5%9B%C4%87), wynikającej z tarcia wewnętrznego w płynie w pobliżu poruszającego się ciała.

Siła parcia działa miejscowo prostopadle do powierzchni ciała, siła lepkości działa miejscowo równolegle (ścinająco) do powierzchni. Całkowita siła aerodynamiczna działająca na ciało jest wektorową sumą tych dwóch sił scałkowaną po całej powierzchni ciała.

marian + + +

Wyznaczanie dokładnych ciśnień i prędkości poruszającego się gazu to skomplikowany proces. Dlatego dla wyznaczenia siły aerodynamicznej działającej na żagiel posłużę się pewnymi uproszczeniami.

## Siła aerodynamiczna w żeglarstwie

Dążąc do określenia siły działającej na żagiel rozpatruje się następujące czynniki: + **Prędkość wiatru pozornego** - czyli prędkość wiatru względem żagla. Wartość siły aerodynamicznej jest wprost proporcjonalna do kwadratu prędkości wiatru pozornego. Dwukrotny wzrost prędkości wiatru oznacza wytworzenie czterokrotnie większej siły aerodynamicznej. + **Powierzchnia ożaglowania** - siła aerodynamiczna jest wprost proporcjonalna do powierzchni ożaglowania. + **Własności aerodynamiczne ożaglowania** - siła aerodynamiczna wytworzona na żaglach w znaczny sposób zależy od typu ożaglowania oraz cech konstrukcyjnych żagli takich jak: wybrzuszenie, smukłość, czy nawet rodzaj materiału. + **Kąt natarcia** – kąt pomiędzy kierunkiem przepływu wiatru a cięciwą żagla. Dla kursów ostrych, do prawidłowej pracy żagla, kąt natarcia powinien mieć od 10° do 20°. Dla kursów pełnych kąt natarcia jest bliski kątowi prostemu.

*Schematycznie przedstawiona siła aerodynamiczna Fae powstająca na żaglu jachtu oraz jej składowe: siła ciągu Fc i siła przechylająca Fp*

Wartość siły aerodynamicznej zależy od prędkości ruchu ciała, w przybliżeniu jest proporcjonalna do kwadratu prędkości. Zależy od powierzchni ciała, jej wielkości, kształtu, ustawienia względem kierunku ruchu w gazie, dla danego kształtu i jego ustawienia w przybliżeniu jest proporcjonalna do powierzchni ciała. Jest proporcjonalna do gęstości gazu, która zależy od rodzaju gazu, jego temperatury i ciśnienia. Dla powietrza stosuje się przybliżony wzór:

Fae = q · S · C = 0,0626 · V^2 · S · C

gdzie: + **q** - ciśnienie dynamiczne, + **V** - prędkość ciała względem gazu, + **S** - powierzchnia ciała, + **C** - współczynnik siły aerodynamicznej. — sidebar\_position: 3 —

# Przegląd Podobnych Systemów Informatycznych

## SailWay

[sailWay](https://sailaway.world/) - Realistyczny symulator żeglowania 3D. Na platformie [Steam](https://store.steampowered.com/app/552920/Sailaway__The_Sailing_Simulator/) jest oceniony na 7/10.

## Sea of Thieves

[Sea of Thieves](https://www.seaofthieves.com/pl) - Gra akcji z otwartym światem, w której gracze wcielają się w piratów. Gra jest dostępna na platformie [Steam](https://store.steampowered.com/app/1172620/Sea_of_Thieves/). Jest oceniona na 9/10. Posiada zręcznościowy model żeglowania, który nie jest realistyczny (można pływać w dowolnym kierunku).

## Porównanie z tym projektem

Ten projekt będzie realizowany w przestrzeni dwuwymiarowej, inne aplikacje/gry realizują to zagadnienie w 3D. Ten projekt ma zachować balans pomiędzy realnym odwzorowaniem mechaniki żeglowania, a grywalnością. Chcę zachować wartość edukacyjną, ale nie chcę, żeby system był zbyt skomplikowana.

# Przegląd Literatury

## Literatura

### [1]

[The Physics of Sailing](https://www.amazon.com/Physics-Sailing-Explained-Stephen-Davis/dp/1472905560)

Książka opisuje podstawy fizyki żeglowania. Jest to dość obszerny temat, więc książka nie jest w stanie opisać wszystkiego. Opisuje podstawowe zjawiska, które występują podczas żeglowania.

### [2]

[Grokking Simplicity: Taming complex software with functional thinking](https://pl.annas-archive.org/md5/b2814f0513ba83a14ac91b64eef909a2)

Książka opisuje programowanie funkcyjne. Jest to dość obszerny temat, więc książka nie jest w stanie opisać wszystkiego. Opisuje podstawowe zjawiska, które występują podczas programowania funkcyjnego. Książka podchodzi do tematu w sposób praktyczny i demonstruje przykłady w języku JavaScript.

### [3]

[Essential Mathematics for Games and Interactive Applications](https://pl.annas-archive.org/md5/9e397499a8cf3fd610b2cbfdfee24f37)

Książka opisuje podstawowe zagadnienia potrzebne do tworzenia interaktywnych aplikacji 3D. Najważniejsze z nich to: - Algebra liniowa w praktycznym zastosowaniu - renderowanie grafiki 3D

# Narzędzia, Języki Programowania, Biblioteki, Frameworki

## Języki Programowania

Język Programowania definiuje kształt pliku. Jeżeli zawartość piliku jest zrozumiała przez interpreter/kompilator jakiegoś języka to ta zawartość jest napisana w tym języku.

### TypeScript

[TypeScript](https://www.typescriptlang.org/) jest to główny język tego projektu. Wybrałem właśnie ten język, ponieważ umożliwia on kompilację do JavaScript. Systemy napisane w JS posiadają najlepszą przenośność pomiędzy urządzeniami, wystarczy, że maszyna posiada przeglądarkę internetową. Kolejną zaleta TS jest obchodzenie się z funkcjami. W TS function jest first-class citizen,to znaczy, że można ją przypisywać do zmiennych/stałych przesyłać do funkcji i zwracać z funkcji tak samo wygodnie, jak dane/wartości np. string, number, object. Kolejną wygodną rzeczą jest [closure](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Closures), która umożliwia definiowanie funkcji w dowolnym miejscu z możliwością odwołania się do wszystkich elementów języka w przestrzeni nazw, w której zastała zdefiniowana. Wadą Systemów napisanych w JS jest np. mniejsza wydajność w porównaniu, np. do C++.

### React

[React](https://pl.react.dev/) Nie korzystałem zbyt wiele z React w tym projekcie. Umożliwia on w wygodny sposób definiowanie wyglądu strony za pomocą drzewa komponentów React (coś jak klasy, albo funkcje), zamiast drzewa statycznych elementów html. Dzięki temu wygląd końcowy poszczególnych komponentów jest dynamiczny i zależy od różnych argumentów: np. globalny stan aplikacji, argument przesłany przez rodzica, albo odpowiedz uzyskana z zewnętrznego serwera. Pliki w języku react występuje w dwóch odmianach: .jsx i .tsx. Pierwszy jest na bazie JS, drugi na bazie TS. W tym projekcie używam tylko .tsx.

### Jest

Język testów interpretowany przez Jest jest na bazie TS/JS znajduje się w plikach z rozszerzeniem test.ts

## Frameworki

Framework Definiuje kształt katalogu. Jeżeli zawartość katalogu jest zrozumiała dla interpretera danego frameworka to ten katalog jest skonstruowany w tym frameworku. Framework jest jak język programowania, ale na poziomie katalogu.

### Next.js

[Next.js](https://nextjs.org/) Framework do tworzenia aplikacji webowych w React. Umożliwia on tworzenie aplikacji w sposób podobny do tworzenia aplikacji w React, ale dodaje kilka udogodnień, np. automatyczne ładowanie komponentów, które są potrzebne w danym momencie, a nie wszystkich na raz. Umożliwia on również łatwe tworzenie stron statycznych, które są ładowane szybciej niż strony dynamiczne. W tym projekcie używam Next.js do tworzenia stron statycznych. Zajmuje się też kompilacją TS do JS, oraz optymalizacją kodu JS.

### Jest

[Jest](https://jestjs.io/)

## Biblioteki

Biblioteka to użyteczny kod, który służy do określonego celu zapisany w jakimś języku programowania.

### ThreeJS

[ThreeJs](https://threejs.org/) biblioteka napisana w js/ts, stanowiąca ‘wraper’ dla [web GL](https://pl.wikipedia.org/wiki/WebGL) ,umożliwia w wygodny sposób tworzenie i manipulację elementami 3D na elemencie Canva z HTML5. Ten projekt co prawda przedstawia dwuwymiarowy model świata, ale zdecydowałem się na taką bibliotekę zamiast np [pixiJS](https://pixijs.com/) ponieważ TJs jest bardziej popularny, posiada dużą bazę przykładów, i wbudowaną bibliotekę umożliwiającą dokonywanie operacji na wektorach.

### Material UI

[Material UI](https://material-ui.com/) Biblioteka komponentów React, która umożliwia tworzenie stron w stylu [Material Design](https://material.io/design).

### detect-collisions

[detect-collisions](https://www.npmjs.com/package/detect-collisions) To biblioteka napisana w języku TS umożliwiająca detekcję kolizji pomiędzy obiektami. Zaimplementowałem system kolizji z wykorzystaniem tej biblioteki niestety nie jest dostatecznie wydajny, więc kod który wywołuję funkcję uruchamiającą systemem kolizji jest //zakomentowany .

## Narzędzia

Narzędzia służą do generowania i przetwarzania zawartości plików jak i całych katalogów.

### VS Code

[VS Code](https://code.visualstudio.com/) Najważniejsze narzędzie które służy do edycji zawartości plików w formie tekstowej. Ponadto posiada wbudowane, lub możliwe do dodania, interpretery różnych języków programowania, które umożliwiają wydajną prace programisty. Dzięki temu na bieżąco wiadomo gdzie występuje błąd językowy, albo do jakiej nazwy możemy się odwołać w bieżącej przestrzeni nazw. Ponadto umożliwia szybkie poruszanie się w kodzie pomiędzy odwołaniami (np z wywołania funkcji do definicji), albo dodaje możliwość refaktoryzacji kodu (np. zdefiniowanie metody, której jeszcze niema, ale już napisaliśmy do niej odwołanie).

### NPM

[npm](https://www.npmjs.com/) to połączenie: 1. [menadżera pakietów](https://en.wikipedia.org/wiki/Package_manager), który działa na maszynie dewelopera 2. serwer repozytoriów przechowujący pakiety do pobrania przez menadżera pakietów.

System informatyczny można przedstawić jako graf zależnych od siebie elementów (pakietów). Do poprawnego działania system potrzebuje dostępu do wszystkich elementów. Menadżer pakietów dba o to aby system posiadał wszystkie potrzebne elementy(pakiety). Potrzebne pakiety programista definiuje w pliku package.json . Po wywołaniu npm install npm doda do system (naszego katalogu) żądane pakiety i pakiety od których te pakiety zależą itd… . Dzięki temu np. nie powielają się pakiety, które już zostały pobrane.

### GIT

[git](https://git-scm.com/) to zdecentralizowany system kontroli wersji, który pierwotnie powstał na potrzeby tworzenia i rozwoju systemu operacyjnego Linux. Jest to obecnie najbardziej popularny SKW. Pomimo że ten program jest pisany przez jedną osobę i tak jest bardzo użyteczny, ponieważ umożliwia zapisywanie stanu systemu w formie commits i w razie potrzeby wczytanie checkout poprzedniego działającego stanu. commits można grupować w branches a po osiągnięciu pożądanego rezultatu branches mogą być połączone ze sobą merge. W połączeniu z zewnętrznym serwerem współpracującym z git, mogę dokonać kopi zapasowej. Jako zewnętrzny serwer git korzystam z [GitHub](https://github.com/) . Serwis GitHub umożliwia upublicznienie swojego repozytorium ,a także ułatwia współprace wielu osób nad jednym projektem informatycznym w ramach [Integration Manager Workflow](https://git-scm.com/about/distributed) GitHub oprócz serwera repozytorium git posiada serwer stron statycznych i udostępnia maszynę wirtualna w ramach [Github Actions](https://github.com/features/actions) Dzięki czemu możliwe jest zautomatyzowanie z repozytorium generowania i deponowania gotowej strony/aplikacji na ich serwerze.

GH repozyorium --> GH Actions --> aplikacja dostępna w internecie

### GitHub Copilot / Codex

[Codex](https://openai.com/blog/openai-codex) To [model językowy](https://en.wikipedia.org/wiki/Language_model) zoptymalizowany pod kontem języków programowania. Korzysta z niego rozszerzenie do VS Code o nazwie [GitHub CoPilot](https://github.com/features/copilot) . GHCP Generuje nieustannie sugestie na podstawie wcześniejszego kodu w pliku, a także na podstawie komentarzy, akceptuje się za pomocą tab. Jest bardzo użyteczny w sytuacji gdy korzystamy z nieznanej, ale popularnej biblioteki programistycznej (np. testy w Jest), lub chcemy napisać ciało jakiejś znanej funkcji np bubbleSort(arr), ale i bardziej złożonych, albo ostylować jakiś komponent, a nie znamy css.

#### Przykłady przydatnych propozycji:

* Wyznaczenie wektora prostopadłego do danego.

const botomEdge = this.positions[1].value.clone().sub(this.positions[2].value);  
const ortoganalToBotomEdge = new Vector2(-botomEdge.y, botomEdge.x); // GPT proposition

* funkcja która ustawia obrót obiekt składający się z trzech punktów. Wszystko posiada specyficzną strukturę danych a mimo to model wygenerował poprawny kod za jednym zamachem.

//GPT proposition  
setRotation(rotation: number) {  
const positionRotation = this.getPositionRotation();  
const rotationDifference = rotation - positionRotation.rotation;  
const rotationMatrix = new Vector2(Math.cos(rotationDifference), Math.sin(rotationDifference));  
this.positions[0].value.sub(positionRotation.position.value);  
this.positions[1].value.sub(positionRotation.position.value);  
this.positions[2].value.sub(positionRotation.position.value);   
  
this.positions[0].value = new Vector2(this.positions[0].value.x \* rotationMatrix.x - this.positions[0].value.y \* rotationMatrix.y, this.positions[0].value.x \* rotationMatrix.y + this.positions[0].value.y \* rotationMatrix.x);  
this.positions[1].value = new Vector2(this.positions[1].value.x \* rotationMatrix.x - this.positions[1].value.y \* rotationMatrix.y, this.positions[1].value.x \* rotationMatrix.y + this.positions[1].value.y \* rotationMatrix.x);  
this.positions[2].value = new Vector2(this.positions[2].value.x \* rotationMatrix.x - this.positions[2].value.y \* rotationMatrix.y, this.positions[2].value.x \* rotationMatrix.y + this.positions[2].value.y \* rotationMatrix.x);   
  
this.positions[0].value.add(positionRotation.position.value);  
this.positions[1].value.add(positionRotation.position.value);  
this.positions[2].value.add(positionRotation.position.value);  
}

# Metodyki tworzenia oprogramowania

W procesie powstawania oprogramowania wykorzystuje dwie metodyki: 1. [eXtreme Programming, XP](https://en.wikipedia.org/wiki/Extreme_programming) 2. [CI/CD](https://en.wikipedia.org/wiki/CI/CD)

## eXtreme Programing

Ta metodyka zakłada powstawanie systemu w sposób iteracyjny, organiczny. Cel końcowy projektu nie jest bardzo ściśle określony, ewoluuje w trakcie tworzenia, jest elastyczny.

### Najważniejsze cechy

#### Iteratywność

[Program](https://pl.wikipedia.org/wiki/Oprogramowanie) tworzy się w iteracjach (krótkie, przyrostowe kroki programistyczne) – i co ważniejsze – planuje tylko następną iterację. Efektem każdej iteracji (kilka tygodni) powinna być wersja programu spełniającą założenia dla danej iteracji. Następnie planuje się co zrobić dalej.

Odpowiada to zasadzie [Open Source](https://pl.wikipedia.org/wiki/Otwarte_oprogramowanie): „release early, release often” (wczesne i częste wydania).

#### Nie projektować z góry

Nie można z góry przewidzieć, jaka architektura będzie najlepsza dla danego problemu. Dlatego należy ją tworzyć w miarę rozszerzania programu.

#### Testy jednostkowe

[Testy jednostkowe](https://pl.wikipedia.org/wiki/Test_jednostkowy) pisze się zanim w ogóle zacznie się pisać kod – najlepiej na początku iteracji. Potem pisze się kod, który potrafi je wszystkie przejść. Takie testy dają zapewnienie (o ile testy są dobrze napisane), że to, co ważne, zostanie zaprojektowane, na to zaś, co nie jest ważne, programiści nie będą tracić czasu.

#### Ciągłe modyfikacje architektury

Architektura nie jest czymś, czego nie wolno ruszać. Jeśli [modyfikacja architektury](https://pl.wikipedia.org/wiki/Refactoring) ułatwi przejście danej iteracji i nie zepsuje wyników testów uzyskanych na poprzednich, należy ją wykonać. Pod tę zasadę podlega także usuwanie wszystkich znanych błędów przed rozszerzeniem funkcjonalności.

### Realizacja w praktyce

W iteracyjnym procesie powstawania systemu informatycznego ważną rolę pełni system kontroli wersji. W tym projekcie korzystam z [GIT](https://git-scm.com/). Każda główna iteracja oznacza stworzenie nowej gałęzi(branch), w której nazwie znajduje się numer aktualnej iteracji (major version). Główna iteracja składa się z iteracji podrzędnych (minor version), które w git występują jako commit w ich nazwie występuje odpowiedni numer iteracji podrzędnej. W momencie uznania, że aktualna wersja (major version) spełnia wymagania ustalone na początku iteracji dokonuje operacji scalania merge aktualnej gałęzi z gałęzią główną master-branch. W tym momencie tworzona jest kolejna gałąź z nazwą odpowiadającą kolejnej iteracji i cykl się powtarza.

Pseudo kod powyższego opisu:

let majorVersion =0;  
let minorVersion =0;  
while(true)  
{   
 git.newBrach(name: majorVersion) //tworzę nową gałąź  
 wyznaczNowyCel();  
 while(!czyCelZostałOsiągniety())  
 {  
 realizacjaCelu();  
 git.commit(name: minorVersion);  
 minorVersion++;  
 }  
 git.merge() // scalanie z master  
 minorVersion=0;  
 majorVersion++;  
}

Poniższy screen obrazuje powstałą w ten sposób strukturę commits i branches. Niebieska linia przedstawia gałąź master, kolorami przedstawione są wszystkie gałęzie kolejnych iteracji ## CI / CD

Jest to maksymalne zautomatyzowanie i przyspieszenie procesu przetworzenia kodu źródłowego do gotowego systemu informatycznego

### Realizacja

Wykorzystuję do tego celu dwa narzędzia: 1. framework [NextJS](https://vercel.com/solutions/nextjs?utm_source=google&utm_medium=cpc&utm_campaign=17166484775&utm_campaign_id=17166484775&utm_term=next%20js&utm_content=134252114817_626380274875&gad=1&gclid=Cj0KCQjwmtGjBhDhARIsAEqfDEct38dlVJCSHtujquAFxwOD5VZQzhVdn2ePglTMclQZeyF_oUbuzEUaAhsDEALw_wcB) 2. platformę [GitHub](https://github.com/)

#### NextJs

Umożliwia kompilacje kodu do postaci, która może być udostępniana przez serwer przeglądarkom internetowym. Projekt jest pisany w [TypeScript](https://www.typescriptlang.org/), ponadto wykorzystuje komponenty [React](https://pl.react.dev/) Ten cały kod musi zostać skompilowany do JavaScript/Html.

#### GitHub

Github jest platformą, która umożliwia: 1. hostowanie repozytorium Git <- to jest najważniejsze zadanie GitHub’a 2. hostowanie statycznych stron internetowych <- dokładnie tego potrzebuje w tym projekcie 3. dokonanie pewnych operacji w ramach [GitHub Actions](https://github.com/features/actions) . To znaczy kompilację kodu źródłowego do postaci wymaganej przez serwer i przeniesienie tego (deploy) na serwer statycznych stron GitHub. Akcją jest wyzwalana przez jakiś triger, w tym przypadku jest to modyfikacja gałęzi master, to znaczy każdy push do tej gałęzi aktywuje akcję. Zadania w ramach akcji są zdefiniowane w repozytorium w katalogu .github/workflos w pliku z rozszerzenie .yml. Tutaj korzystam z gotowego pliku przygotowanego dla tego frameworka. #### podsumowanie Dzięki temu każde wypchnięcie gałęzi master na maszynie developera po krótkiej chwili skutkuję powstaniem nowej wersji aplikacji na serwerze użytkowym.

# Symulator czasu rzeczywistego

symulator jest właściwie [automatem skończonym](https://pl.wikipedia.org/wiki/Automat_sko%C5%84czony) , a dokładniej [automatem Moor’a](https://pl.wikipedia.org/wiki/Automat_Moore%E2%80%99a)

## Definicja Automatu Moor’a

Automat Moore’a jest to rodzaj [deterministycznego automatu skończonego](https://pl.wikipedia.org/wiki/Deterministyczny_automat_sko%C5%84czony), reprezentowany przez uporządkowaną szóstkę:

⟨ Z , Q , Y , Φ , Ψ , q 0 ⟩

[Moore machine-diagram.svg](https://pl.wikipedia.org/wiki/Plik:Moore_machine-diagram.svg)

gdzie:

* Z = { z 1 , z 2 , … , z n } – zbiór sygnałów wejściowych,
* Q = { q 1 , q 2 , … , q n } – zbiór stanów wewnętrznych,
* Y = { y 1 , y 2 , … , y n } – zbiór sygnałów wyjściowych,
* Φ – funkcja przejść, q(t+1) = Φ(q(t), z(t)),
* Ψ – funkcja wyjść, y ( t ) = Ψ(q(t)) zależy tylko od stanu w którym znajduje się automat,
* q0 – stan początkowy, należy do zbioru Q.

Bardziej czytelnie można by przedstawić to w kodzie jako:

interface Stan {...} //zbiór stanów wewnętrznych  
interface Wejscie {...} //zbiór sygnałów wejściowych  
interface Wyjscie {...} //zbiór sygnałów wyjściowych  
  
const automatMoora = (  
aktualnyStan: Stan,  
funkcjaPrzejscia: (aktualnyStan: Stan, sygnałWejsciowy: Wejscie) => Stan,  
funkcjaWyjscia: (aktualnyStan: Stan) => Wyjscie,  
getWejscie: () => Wejscie,  
setWyjscie: (wyjscie: Wyjscie) => void,  
)=>{  
//uzyskuje wejscie z zewnątrz np. stan myszy  
const sygnalWejsciowy = getWejscie();   
const nowyStan = funkcjaPrzejscia(aktualnyStan, sygnalWejsciowy);  
const wyjscie = funkcjaWyjscia(nowyStan);  
setWyjscie(wyjscie); //np. render  
autmatMoora(nowyStan, funkcjaPrzejscia, funkcjaWyjscia, getWejscie, setWyjscie);  
}

## Implementacja w tym symulatorze

### WorldElement

symulator jest zbudowany z obiektów które implementują interfejs WorldElement

export interface WorldElement {  
update(): void;  
destroy(): void;  
}

metoda update jest wywoływana w każdej iteracji (funkcji przejścia). Definiuje w jaki sposób aktualizować dany element świata. Np. dla klasy FrictionInteraction:

update(): void {  
// sila tarcia zalezy od predkosci wzgledem obiektów i jest stała  
let velocityDeferace = this.dynamicElement2.velocity.clone().sub(this.dynamicElement1.velocity);  
let force: Vector2 = velocityDeferace.clone().normalize().multiplyScalar(this.frictionRate);  
let negativeForce: Vector2 = force.clone().multiplyScalar(-1);  
this.dynamicElement1.force.add(force);  
this.dynamicElement2.force.add(negativeForce);  
}

Dla każdej klasy implementującej WorlElement istnieje ‘kontener’ tych obiektów, który jest globalnym obiektem klasy WorldElements. ustaliłem że nazwa kontenera będzie jak nazwa klasy z literką ‘s’ na końcu np. dla DynamicElement kontener to DynamicElements. Każda klasa z int. WorldElement w konstruktorze dodaje się do swojego kontenera:

export class FrictionInteraction implements WorldElement {  
 dynamicElement1: DynamicElement;  
 dynamicElement2: DynamicElement;  
 frictionRate: number;  
  
 constructor(dynamicElement1: DynamicElement, dynamicElement2: DynamicElement, frictionRate: number) {  
 this.dynamicElement1 = dynamicElement1;  
 this.dynamicElement2 = dynamicElement2;  
 this.frictionRate = frictionRate;  
  
 //tutaj dodaje sie do globalnego kontenera  
 frictionInteractions.addElement(this);   
 }  
 ...  
}

Dzieki temu że kontener jest globalny mogę w wygodny sposób tworzyć nowe obiekty, które są zagnieżdżone w innych i nie potrzebuje przekazywać przez nie wszystkie referencji do kontenera. W innych językach programowania taki kontener mógłby być atrybutem statycznym, ale w TS niema takiej możliwości.

Tak wygląda klasa WorldElements:

export class WorldElements {  
 protected elements: WorldElement[] = [];  
 update() {  
 this.elements.forEach((element) => {  
 element.update();  
 })  
 }  
  
 removeElement(element: WorldElement) {  
 this.elements = this.elements.filter((e) => e != element);  
 }  
  
 addElement(element: WorldElement) {  
 this.elements.push(element);  
 }  
  
 clear() {  
 this.elements = [];  
 }  
}

Wywołanie metody update na obiekcie WorldElements powoduje wywołanie metody update na każdym obiekcie w kontenerze.

### Funkcja przejścia

Końcowa funkcja przejścia (transitionFunction) wywołuje metodę update na każdym kontenerze:

private transitionFunction() {  
 const realWorldDt = 10;  
 const dt = realWorldDt \* timeSpeed.value;  
 let SimulationMaximumDT = springInteractions.getSimulationMaximumDT();  
 SimulationMaximumDT = 0.3;  
 const iterations = Math.floor(dt / SimulationMaximumDT);  
  
 for (let i = 0; i < iterations; i++) {  
 userInteractors.update();  
 // this.collisionSystemDuration += mesureTime(() => collisionSystem.update(), 1);  
 // this.dynamicCollidingPolygonsDuration += mesureTime(() => dynamicCollidingPolygons.update(), 1)  
 // this.dynamicCollidingTrianglesDuration += mesureTime(() => dynamicCollindingTriangles.update(), 1);  
 // wraper mesureTime służy do pomiaru wydajności  
 this.springInteractionsDuration += mesureTime(() => springInteractions.update(), 1);  
 this.frictionInteractionsDuration += mesureTime(() => frictionInteractions.update(), 1);  
 this.dynamicElementsDuration += mesureTime(() => dynamicElements.update(SimulationMaximumDT), 1);  
 this.fluidInteractorsDuration += mesureTime(() => fluidInteractors.update(), 1);  
 this.trianglesDuration += mesureTime(() => triangles.update(), 1);  
 pointers.update();  
 }  
 }

ta funkcja jest przesłana do interwału aby była wywołana co 10[ms] :

this.intervals.push(setInterval(() => this.transitionFunction(), 10));

### Stan

Stan początkowy automatu/symulatora jest reprezentowany przez klasę World:

export class World {  
 constructor() {  
 pointer.pointer = new Pointer();  
 const windDynamicElement = new DynamicElement(new Position(), 9999999999);  
 windDynamicElement.velocity = wind.velocity;  
 const clouds = new ViewTexture(new PositionRotation(windDynamicElement.position), 'clouds.png', { height: 1000000, width: 1000000 }, 100, { x: 500, y: 500 });  
  
 const viewOcean = new ViewTexture(new PositionRotation(), 'water.jpg', { height: 1000000, width: 1000000 }, -10, { x: 500, y: 500 });  
   
 const ship = new Ship2();  
  
 const DynameicElementOcean = new DynamicElement(new Position(), 9999999999);  
 const friction = new FrictionInteraction(ship.hull.dynamicCollidingPolygon.centerDynamicElement, DynameicElementOcean, 0.01)  
 ...  
 }  
}

kolejne stany są przechowywane w globalnych kontenerach (obiektach klasy WorldElements)

# Elementy świata

## świat

Elementy świata posiadają referencje do innych elementów świata. Tworzą w ten sposób strukturę danych (świat). Tak wygląda diagram klas Elementów świata:

Diagram Klas nie dostarcza wystarczającej wiedzy, aby zrozumieć działanie symulatora. Należało by spojrzeć również na diagram obiektów, chociażby w stanie początkowym symulatora:

daigram obiektów

### Wybrane Elementy świata

#### Position

export class Position {  
 value: Vector2 = new Vector2(0, 0);  
 constructor(value: Vector2 = new Vector2(0, 0)) {  
 this.value = value;  
 }  
}

Pozycja jest tylko daną, nie posiada metod.

#### ViewPoint

export class ViewPoint implements View {  
 readonly position: Position;  
 readonly circle: THREE.Mesh;  
  
 constructor(position: Position) {  
 this.position = position;  
  
 const geometry = new THREE.CircleGeometry(5, 32);  
 const material = new THREE.MeshBasicMaterial({ color: 0xffff00 });  
 this.circle = new THREE.Mesh(geometry, material);  
 views.addView(this)  
 }  
  
  
 get3DObject(): THREE.Object3D<THREE.Event> {  
 return this.circle;  
 }  
 update(): void {  
 this.circle.position.set(this.position.value.x, this.position.value.y, 0);  
 }  
 destroy(): void {  
 views.removeView(this);  
 }  
}

Ten element przechowuje informacje potrzebne do renderowania obiektu na ekranie. update ‘tłumaczy’ z pozycję elementu zapisanego w formie wykorzystywanej prze inne elementy tego systemu do zrozumiałej przez Three.js.

#### DynamicElement

export class DynamicElement {  
 force = new Vector2(0, 0);  
 mass = 1;  
 acceleration = new Vector2(0, 0);  
 velocity = new Vector2(0, 0);  
 position: Position  
  
 viewPoint: ViewPoint // na potrzeby testów  
  
 constructor(position: Position, mass: number = 1) {  
 this.mass = mass;  
 this.position = position;  
  
 this.viewPoint = new ViewPoint(position);  
  
 dynamicElements.addElement(this);  
 }  
  
 update(dt: number) {  
 this.acceleration = this.force.clone().divideScalar(this.mass);  
 this.velocity.add(this.acceleration.clone().multiplyScalar(dt));  
 this.position.value.add(this.velocity.clone().multiplyScalar(dt));  
  
 this.force = new Vector2(0, 0);  
 }  
  
 remove() {  
 dynamicElements.removeElement(this);  
 }  
  
 getMomentum() {  
 return this.velocity.clone().multiplyScalar(this.mass);  
 }  
}

Jest to jeden z ważniejszych elementów świata. Przechowuje on informacje o prędkości, przyspieszeniu, masie, pędzie i sile, która działa na element. metoda update dokonuje [integracji numerycznej](https://en.wikipedia.org/wiki/Numerical_integration#Reasons_for_numerical_integration) [równań ruchu](https://en.wikipedia.org/wiki/Equations_of_motion) W ten sposób oblicza nowe wartości swoich atrybutów. Ten Obiekt nie implementuje WorldElements ponieważ korzysta ze zmodyfikowanej metody update która przyjmuje argument dt (delta time) czyli zmiana jaka będzie użyta w integracji. Należy uważać na prawidłową wartość dt. Zbyt mała może spowodować problemy z wydajnością, a zbyt duża może spowodować destabilizacje modelu dynamicznego tzn. model przestanie zachowywać stałe ruchu takie jak: zachowanie pędu, czy energii w skutek czego model się ‘rozpadnie’.

##### Stabilność modelu dynamiki

Model jest stabilny, jeżeli zachowuje pęd

Pęd jest stały, jeżeli dt jest dostatecznie małe. Istnieje takie dtMax poniżej, którego system jest stabilny. Jak podaje Wikipedia dtMax jest związane z największą częstością drgania któregokolwiek molekuła systemu. [Energy Drift](https://en.wikipedia.org/wiki/Energy_drift)

dtMax = 2^(1/2)/omega

![{t<{}0.225p}](data:image/svg+xml; charset=utf-8; profile="https://www.mediawiki.org/wiki/Specs/SVG/1.0.0";base64,)

{t<{}0.225p}

Omega jest stała, jeżeli oddziaływania się nie zmieniają, to znaczy nie powstają i nie znikają nowe obiekty klasy Interaction. Omega jest równa

omega = (k/m)^(1/2)

gdzie: + k - współczynnik sprężystości + m - masa

Dla cząstek, które podlegają wielu oddziaływaniom, zsumuję współczynnik sprężystości, tak jak by sprężyny były połączone równolegle.

Teraz znajduję największą omegę i na jej podstawie wyznaczam dtMax poniżej której system jest stabilny. Ciekawe 🤔, że to działa:

test('momentum conservation for for wsp = 2^(1/2)', () => {  
 // molecular model is stable (conservation of momentum) if dt< wsp /omegaMax  
 // where omegaMax is the highest oscilation frequency of the molecul in the system   
 // according to Wikipedia wsp should be 2^(1/2)  
 dynamicElement1.velocity = new Vector2(10, 0);  
 dynamicElement2.mass = 10000000;  
   
 for (let i = 1; i < 1000; i++) {  
 interaction.springRate = Math.random() \* 1000;  
 dynamicElement1.mass = Math.random() \* 1000;  
 dynamicElement2.mass = Math.random() \* 1000;  
 dynamicElement1.velocity = new Vector2(10, Math.random() \* 1000);  
 let maximumDt = calculatemaximumDt(interaction.springRate, dynamicElement1.mass, dynamicElement2.mass);  
 maximumDt \*= 1;  
 let momentum0 = dynamicElement1.getMomentum().add(dynamicElement2.getMomentum());  
 for (let i = 0; i < 10000; i++) {  
 interaction.update();  
 dynamicElementUpdater.update(maximumDt);  
 }  
 let momentum1 = dynamicElement1.getMomentum().add(dynamicElement2.getMomentum());  
 expect(momentum0.distanceTo(momentum1) <= 0.01 \* momentum0.length()).toBeTruthy();  
 }  
 });

Jeżeli maximumDt pomnożę, chociaż przez 1.1 to system przestaje być stabilny. 👏

#### Połączenie DynamicElement+ Position+ ViewPoint

Jeżeli połączę te trzy elementy otrzymam obiekt który może przesówać się (być przesówany?) po ekranie.

...  
const position = new Position(new Vector2(0, 0));  
const dynamicElement = new DynamicElement(position);  
const viewPoint = new ViewPoint(position);  
dynamicElement.velocity = new Vector2(1, 0);  
...

# Podsumowanie

## Porównanie pierwotnych celów z osiągniętymi rezultatami

### Podstawowy cel

Podstawowym celem projektu było stworzenie aplikacji, która będzie symulować żeglowanie. Ta aplikacja miała po przez interakcję z użytkownikiem wyjaśnić dlaczego żaglowiec potrafi Np. : - płynąć pod wiatr - pływać z prędkością większą niż prędkość wiatru

W tym symulatorku powyższe zjawiska zachodzą, i wynikają z zaimplementowanego modelu fizyki. Użytkownik może obserwować jak zmieniają się parametry żaglowca (np. prędkość), w zależności od ustawień żagli i steru.

**Mogę uznać że podstawowy cel został osiągnięty 👌.**

### Dodatkowe cele

Dodatkowymi celami było: - rozbudowa symulatora do gry multiplayer - stworzenie modelu z wykorzystaniem uczenia maszynowego, który będzie sterował żaglowcem.

## **Dodatkowe cele nie zostały zrealizowane ❌.**

## sidebar\_position: 2

# Dalszy rozwój

## Optymalizacja i naprawa wyłączonych mechanik

System musi zostać zoptymalizowany ponieważ w obecnej postaci nie jest w stanie działać płynnie na słabszych komputerach. Ponadto wiele zaimplementowanych mechanik (np. cały system kolizji!) musiały zostać całkowici //zakomentowane, ponieważ były zbyt wolne.

## Rozbudowa

Symulator powinien zastać rozbudowany o podstawowe elementy: + Interaktywne tworzenie statków, + Losowo generowane mapy,

## Realizacja celów dodatkowych

W dalszej kolejności można by zrealizować niezrealizowane cele dodatkowe.