# 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

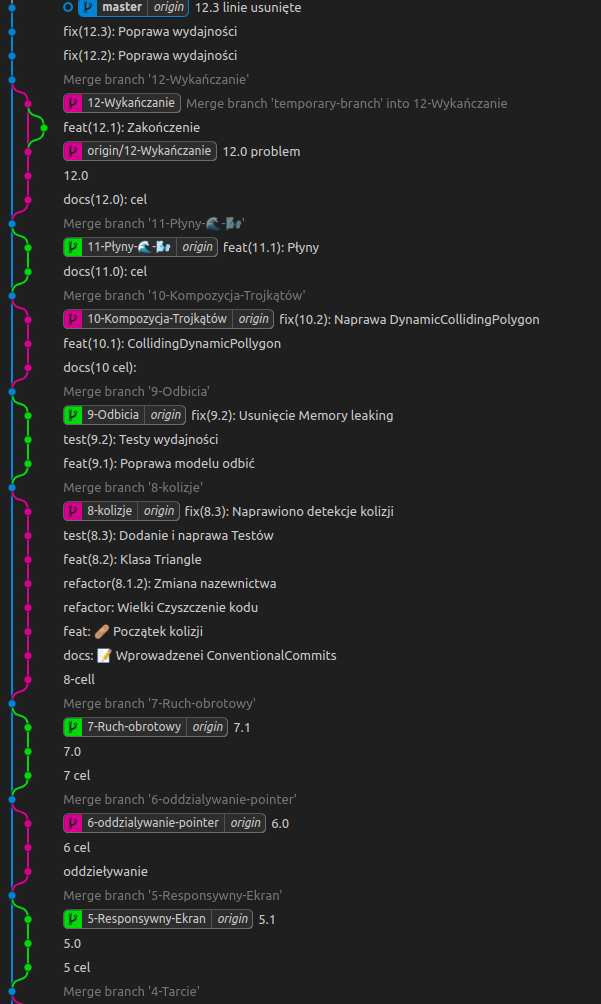
Architektura nie jest czymś, czego nie wolno ruszać. Jeśli [modyfikacja architektury](https://pl.wikipedia.org/wiki/Refactoring) ułatwi przejście danej iteracji i nie zepsuje wyników testów uzyskanych na poprzednich, należy ją wykonać. Pod tę zasadę podlega także usuwanie wszystkich znanych błędów przed 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](data:text/html; charset=utf-8;base64,)](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

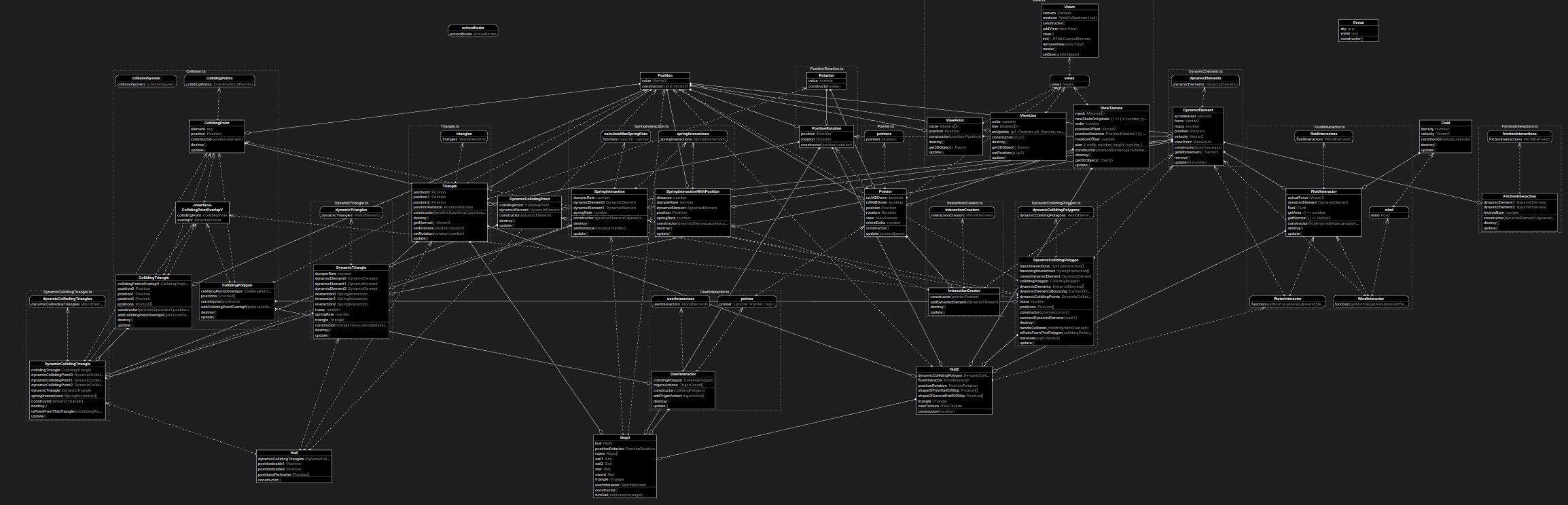
Stan początkowy automatu/symulatora jest reprezentowany przez klasę World:

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: 

Ogólnym założeniem jest stworzenie symulacji w której model fizyczny jest zbudowany z molekułów, które są związana oddziaływaniami ([Soft-body dynamics](https://en.wikipedia.org/wiki/Soft-body_dynamics)). Dzięki temu elementy statku będą mogły bardziej realistycznie się zachowywać np.żagiel wyginać przy silnym wietrze .

## 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 przesuwać się (być przesuwany?) 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);  
...

gif

Obiekt position jest jak komunikator który umożliwia współprace pomiędzy dynamicElement(nadpisuje position), a viewPoint(wyświetla na podstawie position).

## SpringInteraction

obiekty klasy SpringInteraction odpowiadają za powstawanie sił pomiędzy elementami dynamicznymi. Ten element świata implementuje model sprężyny i tłumika([Mass-spring-damper model](https://en.wikipedia.org/wiki/Mass-spring-damper_model)). ### Sprężyna Sprężyna jest modelowana zgodnie z [prawem Hooke’a](https://en.wikipedia.org/wiki/Hooke%27s_law). Siła sprężystości jest proporcjonalna do odległości od punktu równowagi.

### Tłumik

Bez tłumienia obiekty drgały by bez przerwy co nie odzwierciedlało by dobrze rzeczywistość. Tłumik generuje [siłę tarcia](https://en.wikipedia.org/wiki/Friction) stała co do długości. Siła działa tylko wzdłuż tłumika i jest odwrotnie skierowana jak ruch tłumika.

### Implementacja:

export class SpringInteraction implements WorldElement {  
 readonly dynamicElement0: DynamicElement;  
 readonly dynamicElement1: DynamicElement;  
 private distance: number;  
 springRate: number;  
 readonly dumperRate: number;  
  
 // readonly viewLine: ViewLine;  
  
 constructor(dynamicElement0: DynamicElement, dynamicElement1: DynamicElement, springRate?: number, dumperRate?: number, distance?: number) {  
 this.dynamicElement0 = dynamicElement0;  
 this.dynamicElement1 = dynamicElement1;  
 this.springRate = springRate ? springRate : calculateMaxSpringRate(Math.min(dynamicElement0.mass, dynamicElement0.mass), 1);  
 this.dumperRate = dumperRate != undefined ? dumperRate : 0.1;  
 this.distance = distance != undefined ? distance : dynamicElement0.position.value.distanceTo(dynamicElement1.position.value);  
 springInteractions.addElement(this);  
 }  
  
 update(): void {  
 const pointsShift = this.dynamicElement1.position.value.clone().sub(this.dynamicElement0.position.value);  
 const pointDirection = pointsShift.clone().normalize();  
  
 const springForceOn1 = calculateSpringForceOn1(pointsShift, pointDirection, this.springRate, this.distance);  
 // according to third law of Newton  
 const springForceOn2 = springForceOn1.clone().multiplyScalar(-1);  
  
 const velocityShift = this.dynamicElement1.velocity.clone().sub(this.dynamicElement0.velocity);  
 const dumperForceOn1 = calculateDumperForceOn1(velocityShift, this.dumperRate, pointDirection);  
 const dumperForceOn2 = dumperForceOn1.clone().multiplyScalar(-1);  
  
 this.dynamicElement0.force.add(springForceOn1);  
 this.dynamicElement1.force.add(springForceOn2);  
 this.dynamicElement0.force.add(dumperForceOn1);  
 this.dynamicElement1.force.add(dumperForceOn2);  
 }  
  
 ...  
}  
  
function calculateSpringForceOn1(pointsShift: Vector2, pointDirection: Vector2, springRate: number, distance: number) {  
 // acording to third law of Newton and spring force  
 // F1 = -F2  
 // F1 = -k \* (x1 - x2)   
 const springNeutral = pointDirection.clone().multiplyScalar(distance);  
 const springShift = pointsShift.clone().sub(springNeutral);  
 const forceOn1 = springShift.clone().multiplyScalar(springRate);  
 return forceOn1;  
}  
function calculateDumperForceOn1(velocityShift: Vector2, dumperRate: number, pointDirection: Vector2) {  
 const velocityShiftPointDirection = pointDirection.clone().multiplyScalar(velocityShift.dot(pointDirection));  
 const dumperForceOn1 = velocityShiftPointDirection.clone().multiplyScalar(dumperRate);  
 return dumperForceOn1;  
}

## Dodanie Elementu springInteraction do świata

Dodanie do świata elementu SpringInteraction Sprawia że elementy DynamicElement otrzymują wartość siły która na nie działą i z tego mogą wyznaczyć przyspieszenie.

const position1 = new Position(new Vector2(0, 0));  
const dynamicElement1 = new DynamicElement(position1);  
const viewPoint1 = new ViewPoint(position);  
dynamicElement1.velocity = new Vector2(1, 0);  
  
const position2 = new Position(new Vector2(100, 0));  
const dynamicElement2 = new DynamicElement(position2);  
const viewPoint2 = new ViewPoint(position2);  
  
const springInteraction = new SpringInteraction(dynamicElement1, dynamicElement2);

gif

Taka konstrukcja sprawi że obiekty będą poruszać się ruchem harmonicznym tłumionym wzdłuż osi x.

Właśnie te oddziaływania definiują jakie może być maksymalne dt aby model się nie rozpadł o czym już [wspomniałem](#stabilność-modelu-dynamiki).

## PositionRotation

Obiekt, który służy do przechowywania informacji o położeniu i rotacji obiektu(bryły) Skąd się bierze rotacja? o tym za chwilę

export class PositionRotation {  
 readonly position: Position = new Position();  
 rotation: Rotation = new Rotation();  
  
 constructor(position?: Position, rotation?: Rotation) {  
 this.position = position || new Position();  
 this.rotation = rotation || new Rotation();  
 }  
};  
  
export class Rotation {  
 value: number = 0;  
  
 constructor(value?: number) {  
 this.value = value || 0;  
 }  
}

## ViewTexture

Element świata podobny do ViewPoint ale zamiast kółka służy do wyświetlania tekstury. Do tego celu potrzebuje informacji o rotacji obiektu.

export class ViewTexture implements View {  
 readonly mesh: THREE.Mesh;  
 readonly positionRotation: PositionRotation | (() => PositionRotation);  
 readonly order: number = 0;  
 rotationOffset: number = 0;  
 positionOffset: THREE.Vector2 = new THREE.Vector2();  
 newSkaleOnUpdate: () => { x: number; y: number; } = () => { return { x: 1, y: 1 } };  
 size: { width: number, height: number } = { width: 0, height: 0 };  
  
 constructor(positionRotation: PositionRotation | (() => PositionRotation), picturePath: string, size: { width: number, height: number }, order: number, repeat?: { x: number, y: number },) {  
 this.order = order || -1;  
 this.positionRotation = positionRotation;  
 this.size = size;  
  
 // clone texture beter (to not have a seam)  
 const texture = new THREE.TextureLoader().load(picturePath);  
  
 const setRepeat = (repeat: { x: number, y: number }, texture: THREE.Texture) => {  
 texture.wrapS = THREE.MirroredRepeatWrapping;  
 texture.wrapT = THREE.MirroredRepeatWrapping;  
 texture.repeat.set(repeat.x, repeat.y);  
 }  
 repeat && setRepeat(repeat, texture);  
  
 //create plane  
 const geometry = new THREE.PlaneGeometry(size.width, size.height);  
 const material = new THREE.MeshBasicMaterial({  
 map: texture,  
 transparent: true,  
 // depthWrite: false,  
 side: THREE.DoubleSide  
 });  
  
 this.mesh = new THREE.Mesh(geometry, material);  
  
 //register view for rendering  
 views.addView(this)  
  
 }  
  
 update() {  
  
 const scale = this.newSkaleOnUpdate();  
 this.mesh.scale.set(scale.x, scale.y, 1);  
  
 const halfHeight = this.size.height / 4;  
 const xOffset = -(halfHeight \* scale.y - halfHeight);  
  
  
 const vec = this.positionOffset.clone();  
 vec.add(new THREE.Vector2(xOffset, 0));  
 const positionRotation = this.positionRotation instanceof Function ? this.positionRotation() : this.positionRotation;  
 vec.rotateAround(new THREE.Vector2(0, 0), positionRotation.rotation.value);  
 const position = positionRotation.position.value.clone().add(vec);  
 this.mesh.position.set(position.x, position.y, this.order);  
 this.mesh.rotation.z = positionRotation.rotation.value + this.rotationOffset;  
 }  
 ...  
}

Do konstruktora mogę przesłać referencję na obiekt PositionRotation, albo funkcję która to zwróci jest to wygodne bo daję swobodę co do sposobu wyliczenia rotacji bryły.

Metoda onScaleUpdate jest wykorzystywana np. do imitowania rozciągania żagli w zależności od prędkości wiatru.

## Triangel

Jest to element świata, który służy do wyliczania rotacji i pozycji bryły stworzonej z trzech DynamicElenent połączonych ze sobą za pomocą SpringInteraction. Chociaż sam model fizyki nie posiada przestrzeni rotacji (nie ma takich elementów jak moment bezwładności, czy moment siły), to rotacja jest symulowana. Jest wynikiem oddziaływań powstałych przez SpringInteraction. Rotacja jest wyliczana na podstawie zmiany położenia trzech obiektów DynamicElement względem siebie.

export class Triangle implements WorldElement {  
 readonly position0: Position;  
 readonly position1: Position;  
 readonly position2: Position;  
 readonly positionRotation: PositionRotation;  
  
 constructor(position0: Position, position1: Position, position2: Position, positionRotation: PositionRotation) {  
 this.position0 = position0;  
 this.position1 = position1;  
 this.position2 = position2;  
 this.positionRotation = positionRotation;  
 triangles.addElement(this)  
 }  
 update(): void {  
 const positionRotation = this.getPositionRotation();  
 this.positionRotation.position.value = positionRotation.position.value;  
 this.positionRotation.rotation = positionRotation.rotation;  
 }  
 private getPositionRotation(): PositionRotation {  
 const botomEdge = this.position0.value.clone().sub(this.position1.value);  
 const ortoganalToBotomEdge = new Vector2(-botomEdge.y, botomEdge.x); // rotate 90 degrees, GPT proposition  
 const rotationOfTriangle = Math.atan2(ortoganalToBotomEdge.y, ortoganalToBotomEdge.x);  
  
 const centerOfTriangle = this.position0.value.clone().add(this.position1.value).add(this.position2.value).divideScalar(3); // GPT proposition  
  
 const positionRotation = new PositionRotation(new Position(centerOfTriangle), new Rotation(rotationOfTriangle));  
 return positionRotation;  
 }  
 ...  
}

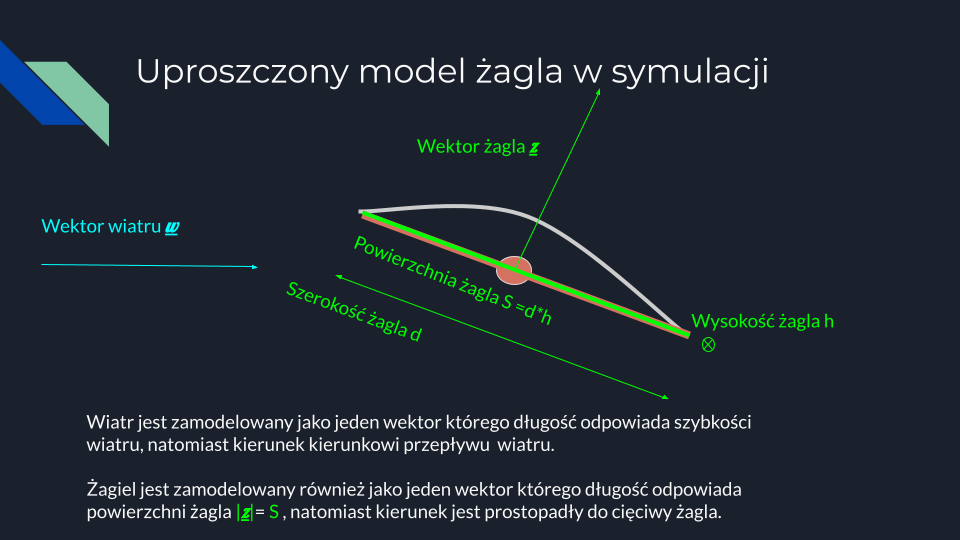
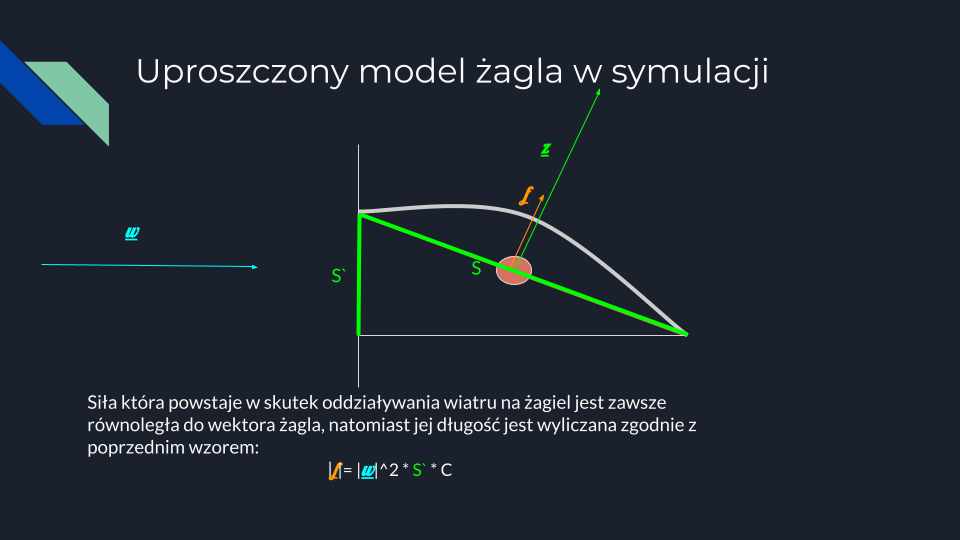
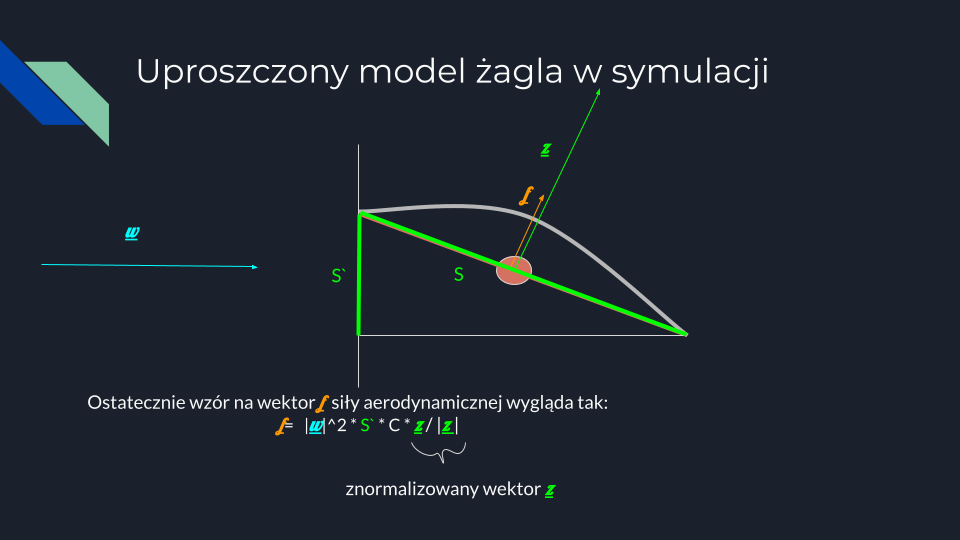
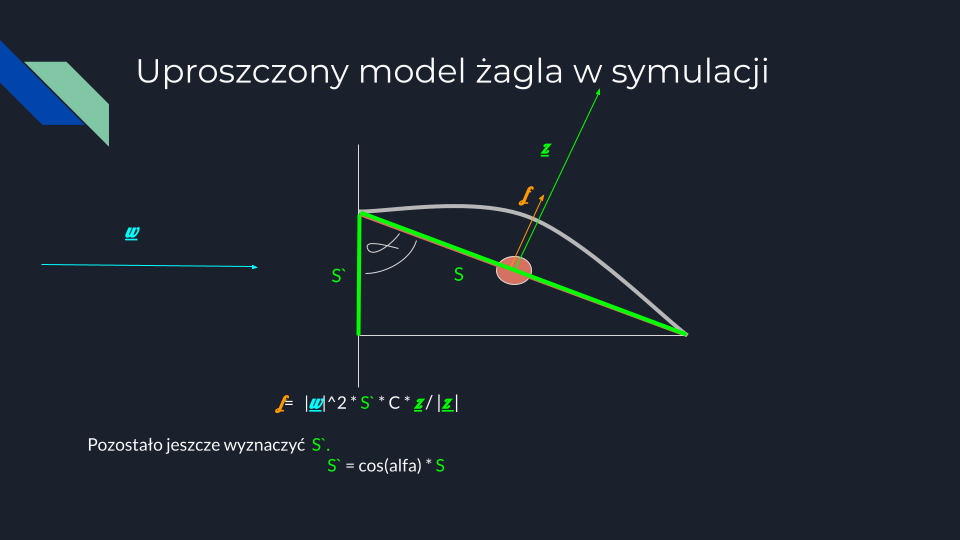
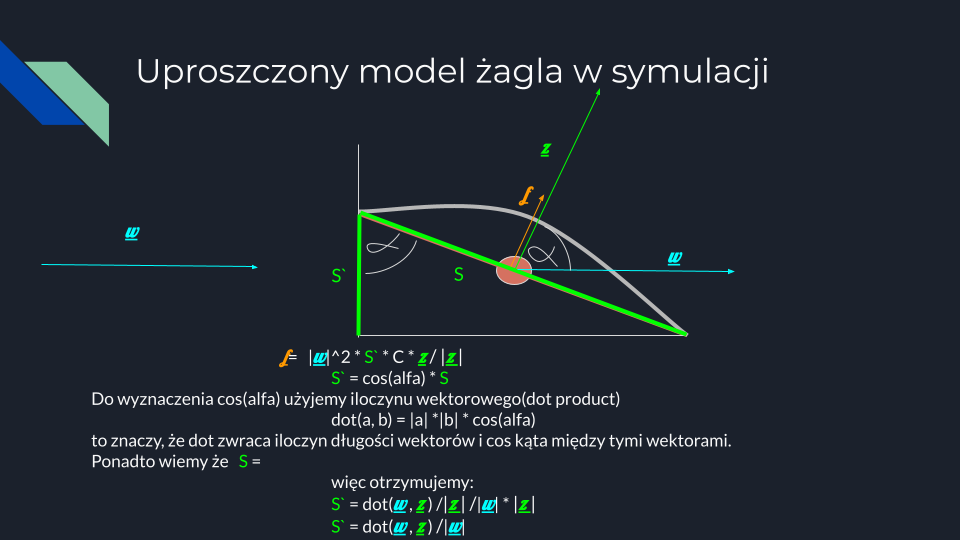
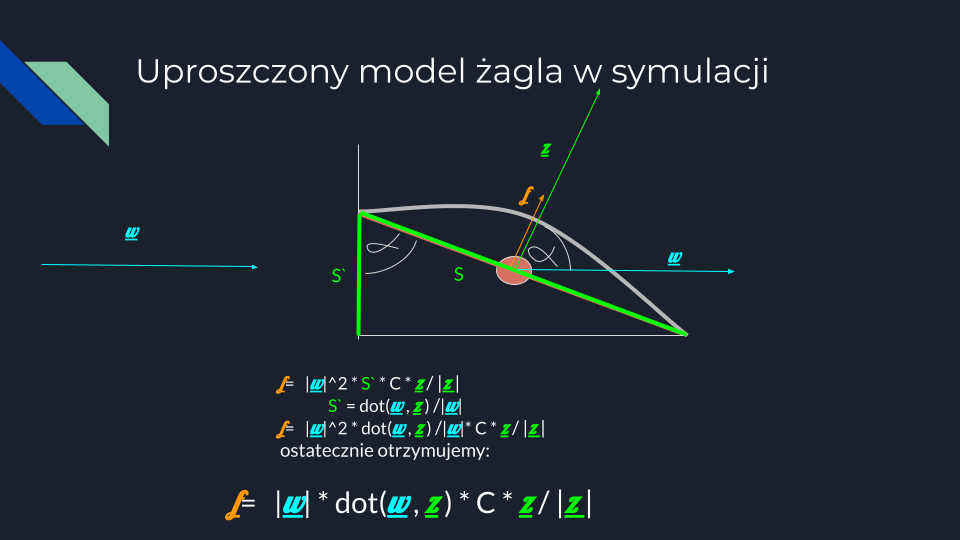
W ten sposób uzyskuję rotacje potrzebną do wyświetlenia tekstury. ### Przykład

const position0 = new Position(new Vector2(0, 0));  
const position1 = new Position(new Vector2(0, 100));  
const position2 = new Position(new Vector2(100, 0));  
  
const dynamicElement0 = new DynamicElement(position0);  
const dynamicElement1 = new DynamicElement(position1);  
const dynamicElement2 = new DynamicElement(position2);  
  
const springInteraction0 = new SpringInteraction(dynamicElement0, dynamicElement1);  
const springInteraction1 = new SpringInteraction(dynamicElement1, dynamicElement2);  
const springInteraction2 = new SpringInteraction(dynamicElement2, dynamicElement0);  
  
const positionRotation = new PositionRotation();  
  
const triangle = new Triangle(position0, position1, position2, positionRotation);  
  
const texture = new ViewTexture(positionRotation, "assets/ship.png", { width: 100, height: 100 }, 0, { x: 1, y: 1 });  
  
//dodanie prędkości  
dynamicElement0.velocity.value = new Vector2(100, 0);

Taka konfiguracja spowoduje wyświetlenia na ekranie poruszającego się w prawo i obracającego się zgodnie z ruchem wskazówek zegara tekstury “assets/ship.png”.

## FluidInteractor

Element świata, który symuluję mechanikę płynów. Jest to element, który sprawia że żagle i ster generują siłę napędową.

Poniższe grafiki prezentują koncepcje wyliczania siły napędowej w symulacji:      

Implementacja:

export class FluidInteractor implements WorldElement {  
 getNormal: () => Vector2  
 dynamicElement: DynamicElement;  
 fluid: Fluid;  
 getArea: () => number;  
 actualForce: Vector2 = new Vector2();  
  
 constructor(fluid: Fluid, normalGetter: () => Vector2, areaGetter: () => number, dynamicElement: DynamicElement) {  
  
 this.fluid = fluid;  
 this.getNormal = normalGetter;  
 this.getArea = areaGetter;  
 this.dynamicElement = dynamicElement;  
 fluidInteractors.addElement(this);  
 }  
  
 update(): void {  
 const maxFluidForce = 10000;  
 const velocity = this.fluid.velocity.clone().sub(this.dynamicElement.velocity);  
  
 const dotNormalVelocity = velocity.dot(this.getNormal());  
 const dotNormalVelocitySquared = dotNormalVelocity \* dotNormalVelocity \* (dotNormalVelocity > 0 ? 1 : -1);  
 const forceLength = dotNormalVelocitySquared \* this.fluid.density \* this.getArea();  
 const safeFluidForceLength = Math.min(Math.abs(forceLength), maxFluidForce) \* (forceLength > 0 ? 1 : -1);  
  
 const fluidForce: Vector2 = this.getNormal().clone().multiplyScalar(safeFluidForceLength);  
  
 this.actualForce.set(fluidForce.x, fluidForce.y);  
 this.dynamicElement.force.add(fluidForce);  
 }  
 destroy(): void {  
 throw new Error("Method not implemented.");  
 }  
  
}

W systemie Istnieją dwa rodzaje płynów: Wind i Water.

export class Fluid implements WorldElement {  
 density: number;  
 velocity: Vector2;  
  
 constructor(density: number, velocity: Vector2) {  
 this.density = density;  
 this.velocity = velocity;  
 }  
  
 update(): void { // na razie nie mam potrzeby aktualizowania płynów, np kierunku wiatru  
 throw new Error("Method not implemented.");  
 }  
 ...  
}  
 export const wind = new Fluid(1, new Vector2(0, 1));  
  
 const water = new Fluid(1000, new Vector2(0, 0));

Woda jest 100 razy gęstsza od powietrza, dlatego też siła napędowa generowana przez wiatr jest 100 razy mniejsza niż siła napędowa generowana przez wodę przy takich samych pozostałych parametrach.

## Ship

Z powyższych elementów bazowych mam dużą swobodę w konstruowaniu różnych obiektów pływających. Oto implementacja statku występującego w aktualnej wersji symulacji:

export class Ship2 {  
 positionRotation = new PositionRotation();  
 triangle: Triangle;  
 hull = new Hull2();  
  
 sail1: Sail  
 sail2: Sail  
  
 sword: Ster;  
  
 ster: Ster;  
  
 ropes: Rope[] = []; // liny służą do obracania żagli  
  
 userInteractor: UserInteractor;  
  
 constructor() {  
 const centerY = this.hull.dynamicCollidingPolygon.centerDynamicElement.position.value.y;  
  
  
 this.sail1 = new Sail(new Vector2(200, centerY));  
 this.sail2 = new Sail(new Vector2(500, centerY));  
  
 this.hull.dynamicCollidingPolygon.connectDynamicElement(this.sail1.mast);  
 this.hull.dynamicCollidingPolygon.connectDynamicElement(this.sail2.mast);  
  
 const ropeLeftSail1 = new SpringInteraction(this.sail1.yardLeft, this.sail2.mast, 0.1, 0.1);  
 const ropeRightSail1 = new SpringInteraction(this.sail1.yardRight, this.sail2.mast, 0.1, 0.1);  
 const ropeLeftSail2 = new SpringInteraction(this.sail2.yardLeft, this.sail1.mast, 0.1, 0.1);  
 const ropeRightSail2 = new SpringInteraction(this.sail2.yardRight, this.sail1.mast, 0.1, 0.1);  
  
 this.ropes.push({ side: 'left', sail: 'front', interaction: ropeLeftSail1 });  
 this.ropes.push({ side: 'right', sail: 'front', interaction: ropeRightSail1 });  
 this.ropes.push({ side: 'left', sail: 'back', interaction: ropeLeftSail2 });  
 this.ropes.push({ side: 'right', sail: 'back', interaction: ropeRightSail2 });  
  
 this.triangle = new Triangle(this.sail1.mast.position, this.sail2.mast.position, this.hull.shapeOfFirstHalfOfShip[5], this.positionRotation);  
  
  
 this.ster = new Ster(new Position(new Vector2(60, centerY)), this.triangle);  
 this.hull.dynamicCollidingPolygon.connectDynamicElement(this.ster.dynamicElement);  
 this.ster.rotationOfSter.value = Math.PI / 8;  
  
 this.sword = new Ster(new Position(new Vector2(350, centerY)), this.triangle);  
 this.hull.dynamicCollidingPolygon.connectDynamicElement(this.sword.dynamicElement);  
  
 this.userInteractor = new UserInteractor(this.hull.dynamicCollidingPolygon.collidingPolygon);  
 ...  
 }  
 ...  
}

Najważniejsze Elementy Statku to: + kadłub - Hull2-> Łączy ze dobą wszystkie elementy statku + żagle - Sail -> produkują się napęd, którą przekazuja do kadłuba po prze SpringInteraction. Połączone są z kadłubem w trzech punktach: mast, yardLeft, yardRight, liny (SpirngInteraction) zmieniają swoją długość i w ten sposób obracają żagle. + sword - Ster -> obiekt który jest ‘przypięty’ do kadłuba na środku i oddziałuje z wodą, nie może się obracać. + ster - Ster -> obiekt który jest ‘przypięty’ do kadłuba z tyłu i może się obracać.

### Kadłub

Łączy ze sobą wszystkie elementy statku

export class Hull2 {  
  
 dynamicCollidingPolygon: DynamicCollidingPolygon;  
 shapeOfFirstHalfOfShip: Position[];  
 shapeOfSecondHalfOfShip: Position[];  
 viewTexture: ViewTexture;  
 triangle: Triangle;  
 positionRotation = new PositionRotation();  
 fluidInteractor: FluidInteractor;  
 constructor(location?: Vector2) {  
 this.shapeOfFirstHalfOfShip = [  
 new Position(new Vector2(17, 176)),  
 new Position(new Vector2(208, 205)),  
 new Position(new Vector2(362, 218)),  
 new Position(new Vector2(539, 210)),  
 new Position(new Vector2(606, 195)),  
 new Position(new Vector2(650, 167)),  
 new Position(new Vector2(672, 127)),  
 ];  
  
 this.shapeOfSecondHalfOfShip = this.shapeOfFirstHalfOfShip.map((position) => {  
 return new Position(new Vector2(position.value.x, -position.value.y + 220));  
 });  
  
 const reverserSecondhalf = this.shapeOfSecondHalfOfShip.slice().reverse();  
 const shapeOfShip = this.shapeOfFirstHalfOfShip.concat(reverserSecondhalf);  
  
 this.dynamicCollidingPolygon = new DynamicCollidingPolygon(shapeOfShip, 1000);  
  
 this.triangle = new Triangle(this.shapeOfSecondHalfOfShip[0], this.shapeOfFirstHalfOfShip[0], this.dynamicCollidingPolygon.centerDynamicElement.position, this.positionRotation);  
  
 this.viewTexture = new ViewTexture(this.positionRotation, 'kadlub.png', { width: 680, height: 220 }, 1);  
 this.viewTexture.positionOffset = new Vector2(190, 0);  
  
 this.fluidInteractor = WaterInteractor(  
 () => {  
 const normal = this.triangle.getNormal();  
 return normal;  
 },  
 () => 0.00001,  
 this.dynamicCollidingPolygon.centerDynamicElement  
 )  
 }  
}

DynamicCollidingPolygon to klasa która generuje bryłę z podanych punktów, po przez związanie ich za pomocą SpringInteraction. Ponadto obiekt ten potrafi kolidować z innymi, niestety funkcjonalność ta została wyłączona ponieważ nie jest zbyt wydajna. Znajduje się tutaj już obiekt definiujący wygląd - zwykła przyklejona tekstura. Model kadłuba realizuje [soft body dynammics](https://en.wikipedia.org/wiki/Soft-body_dynamics), czyli jest elastyczny. Jadnak z braku czasu nie zaimplementowałem takiego wyglądu.

### Żagiel

class Sail {  
 yardView: ViewTexture;  
 sailView: ViewTexture;  
 positionRotation: PositionRotation;  
 triangle: Triangle;  
  
 mast: DynamicElement;  
 yardLeft: DynamicElement;  
 yardRight: DynamicElement;  
 aditionalDynamicElement: DynamicElement;  
  
 interactions: SpringInteraction[] = [];  
  
 windInteractor: FluidInteractor  
 // dynamicTriangle: DynamicTriangle;  
 sailArea = 1;  
 unfurling = 1;  
  
 constructor(position: Vector2) {  
 const width = 400;  
 const height = 50;  
 this.mast = new DynamicElement(new Position(position));  
 this.yardLeft = new DynamicElement(new Position(new Vector2(0, width / 2).add(position)));  
 this.yardRight = new DynamicElement(new Position(new Vector2(0, -width / 2).add(position)));  
  
 this.aditionalDynamicElement = new DynamicElement(new Position(new Vector2(-height, 0).add(position)));  
  
 this.interactions.push(new SpringInteraction(this.mast, this.yardLeft, 0.1, 0.1));  
 this.interactions.push(new SpringInteraction(this.mast, this.yardRight, 0.1, 0.1));  
 this.interactions.push(new SpringInteraction(this.yardLeft, this.aditionalDynamicElement, 0.1, 0.1));  
 this.interactions.push(new SpringInteraction(this.yardRight, this.aditionalDynamicElement, 0.1, 0.1));  
 this.interactions.push(new SpringInteraction(this.mast, this.aditionalDynamicElement, 0.1, 0.1));  
  
 const getNormal = () => {  
 const vectorYard = this.yardLeft.position.value.clone().sub(this.yardRight.position.value);  
 const yardPerpendicular = new Vector2(-vectorYard.y, vectorYard.x);  
 yardPerpendicular.normalize();  
 return yardPerpendicular;  
 }  
 this.windInteractor = WindInteractor(() => getNormal(), () => this.sailArea \* this.unfurling, this.mast);  
  
  
 this.positionRotation = new PositionRotation();  
 this.triangle = new Triangle(this.yardLeft.position, this.yardRight.position, this.aditionalDynamicElement.position, this.positionRotation);  
  
 this.yardView = new ViewTexture(this.positionRotation, 'yard.png', { height, width }, 1);  
 this.yardView.rotationOffset = Math.PI / 2;  
 this.yardView.positionOffset = new Vector2(-height / 2, 0);  
  
  
 this.sailView = new ViewTexture(this.positionRotation, 'plutno.png', { height, width }, 3);  
 this.sailView.rotationOffset = Math.PI / 2;  
 this.sailView.positionOffset = new Vector2(-height / 2, 0);  
  
 const windForce = this.windInteractor.actualForce  
  
 this.sailView.newSkaleOnUpdate = () => {  
 return { x: 1, y: Math.cbrt((-windForce.clone().dot(getNormal()) \* 50)) }  
 };  
 }  
 changeSailArea(changeUnfurling: number) {  
 this.unfurling += changeUnfurling;  
 this.unfurling = Math.min(Math.max(this.unfurling, 0), 1);  
  
 this.sailView.newSkaleOnUpdate = () => {  
 return { x: this.unfurling, y: Math.cbrt((-this.windInteractor.actualForce.clone().dot(this.triangle.getNormal()) \* 50)) }  
 };  
 }  
}

Sam żagiel jest bryłą zbudowaną z czterech DynamicElement i SpringInteraction pomiędzy nimi. Do masztu jest doczepiony FluidInteractor. Tekstura żagla jest skalowana w zależności od siły która na niego działa. Żagiel może być zwijany i rozwijany.

### Ster

class Ster {  
 dynamicElement: DynamicElement;  
 fluidInteractor: FluidInteractor;  
 triangleOfShip: Triangle;  
 rotationOfSter: Rotation = new Rotation();  
 area = 0.01;  
 view: ViewTexture;  
 constructor(position: Position, triangleOfShip: Triangle) {  
 this.triangleOfShip = triangleOfShip;  
 const positionRotation = triangleOfShip.positionRotation;  
 this.dynamicElement = new DynamicElement(position);  
  
 const getNormal = () => {  
 const shipNormal = this.triangleOfShip.getNormal();  
 const rotatedNormal = shipNormal.clone().rotateAround(new Vector2(), this.rotationOfSter.value);  
 return rotatedNormal;  
 }  
  
 this.fluidInteractor = WaterInteractor(() => getNormal(), () => this.area, this.dynamicElement);  
  
 const getPositionRotation = () => {  
 const rotation = new Rotation(this.triangleOfShip.positionRotation.rotation.value);  
 rotation.value += this.rotationOfSter.value + Math.PI / 2;  
 const position = this.dynamicElement.position;  
 return new PositionRotation(position, rotation);  
 }  
  
 this.view = new ViewTexture(getPositionRotation, 'ster.png', { width: 50, height: 10 }, 1);  
 }  
}

Ster Jest prostszy od żagla. Nie jest bryłą, składa sią z tylko jednego DynamicElement. Do niego jest doczepiony FluidInteractor. Ster może być obracany. Rotacja steru w przestrzeni świata jest wyliczana na podstawie rotacji statku i rotacji steru względem statku.

## Collisions

System Posiada zaimplementowany model kolizji, jednakże nie spełnia wymagań wydajności, więc nie będą go omawiał.