LABO 5: GIT INTRO & SHOOTER

GIT DEEL 1

Volg de tutorial op https://github.com/devinehowest/git-intro .
We volgen de tutorial tot aan het deel “Samenwerken met git” wat we volgend labo bekijken.

OEFENING: SHOOT ‘M UP

Analyseren project

Doel van deze oefening is een eenvoudige shooter game te bouwen. We zullen onze code opdelen in aparte **modules** en mbv webpack en babel een cross-browser bundle maken.

Het spel wordt gespeeld door 1 **speler** die vanuit zijn vliegtuig **kogels** kan afschieten om de aankomende **vijanden** uit te schakelen. Via de linker en rechter **pijltjestoetsen** kan de speler zijn vliegtuig besturen. De vijanden komen in een random tijdsinterval tevoorschijn en bewegen naar de speler toe.
Echter wanneer een vijand er in slaagt om de speler te raken, is het spel ten einde.
Er vindt een **explosie** plaats wanneer:

- een kogel een vijand raakt. (**botsing**)
- een vijand de speler raakt. (**botsing**)

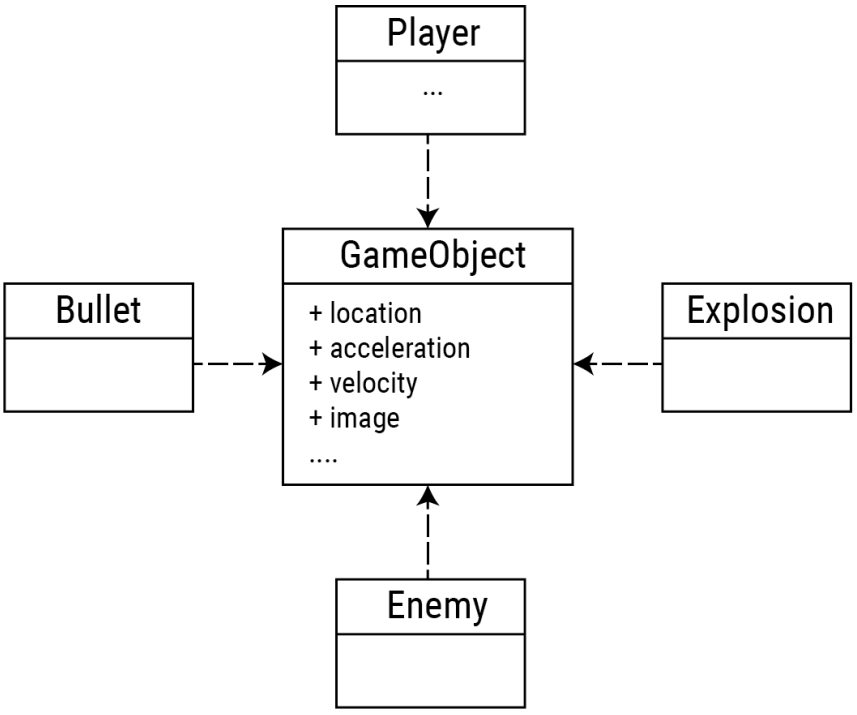
Als we de opgave analyseren kunnen we concluderen dat we volgende verschillende klasses zouden kunnen gebruiken met elk hun eigen verantwoordelijkheid:

- **Keyboard** class
    - Deze klasse moet instaan voor het opvangen van de keyboard toetsen.
- **CollisionDetector** class

- Deze klasse moet instaan voor het detecteren van een collision tussen 2 elementen. Wanneer er een collision is gebeurd dan moet deze klasse dat 'laten weten'.

- **Player** class
- **Enemy** class
- **Bullet** class
- **Explosion** class

Als we eventjes verder denken dan zien we dat zowel Player, Enemy, Bullet en Explosion eigenlijk voor een groot stuk dezelfde verantwoordelijkheden en logica hebben. Daarom is het een goed idee om die verantwoordelijkheden te bundelen in een meer generieke klasse (**GameObject**) waarvan we dan zullen overerven.



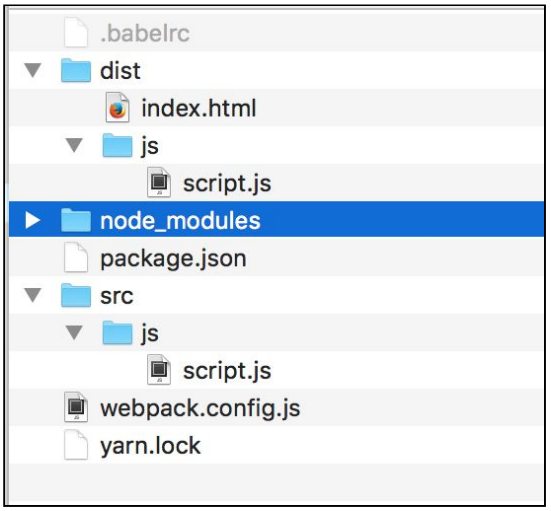
Los van deze (nieuwe) bovenstaande klassen die we zullen moeten programmeren kunnen we voor dit game ook gebruik maken van onze **Vector** klasse en van onze **lib.js** library uit vorige labo's.

Nu we al een goed beeld hebben van de structuur van de game kunnen we starten met de setup en initialisatie van het project.

Initialiseren project

Maak een index.html bestand aan en voeg een canvas (600x600px) tag toe. Vervolgens spreek je de canvas aan in een script.js file en je initialiseert de context.
Doe de setup verder zoals we vorig labo hebben gedaan gebruikmakend van Webpack en Babel.

Uiteindelijk moet je volgende structuur hebben:



Vervolgens voeg je ook nog onze **lib.js** en **Vector.js** uit vorig labo toe. We zullen deze code hergebruiken en onze library functions uitbreiden.

- lib.js voeg je toe in **src > js > functions**.
- Vector.js voeg je toe in **src > js > classes**.

Er zijn ook enkele images meegeleverd. Plaats deze in de folder **dist > img**

GameObject

We zullen eerst deze base class schrijven waarvan zowel Player, Enemy, Bullet en Explosion van overerven.
Als we een nieuw GameObject instantiëren zullen we telkens een **x en y positie** meegeven als ook een afbeelding. Hiervan vertrekkende kunnen we alvast volgende properties instellen:

- **this.location** : een Vector geïnitialiseerd met x en y
- **this.image** : de afbeelding die moet gebruikt worden.

Spritesheets

De images die zijn bijgeleverd zijn allemaal spritesheets. Niet elke image heeft evenveel frames en sommige spritesheets moeten minder snel afgespeeld worden dan andere.
Daarom zullen we in onze GameObject klasse een **frameRate** property instellen. Met een property **numFrames** zullen we het aantal frames van de spritesheet instellen.
(idem zoals we in het labo van week 2)

In de GameObject klasse voorzien we ook een **update()** en **draw()** methode.

- in de update() methode berekenen we welke frame van de spritesheet moet getoond worden
- in de draw() methode gebruiken we translate() om onze frame te verplaatsen en drawImage() om het juiste frame uit de spritesheet te 'knippen'.

We krijgen volgende GameObject klasse:

```
import Vector from './Vector.js';

export default class GameObject{
  constructor(x,y,image){
       this.location  = new Vector(x,y);
       this.image = image;
       this.size = this.image.height;
       this.frameRate = 60;
       this.frameNr = 0;
       this.localFrameNr = 0;
       this.numFrames = 1;
  }
  update(){
       this.frameNr ++;
       this.localFrameNr = Math.floor(this.frameNr/(60 / this.frameRate));
       this.localFrameNr = this.localFrameNr%this.numFrames;
  }
  draw(ctx){
       ctx.save();
       ctx.translate(this.location.x, this.location.y);
       ctx.drawImage(this.image, this.localFrameNr * this.size,0, this.size,
this.size, -this.size/2,-this.size/2,this.size,this.size);
       ctx.restore();
  }
}
```

Image Catalog

Je zal meerdere images moeten inladen in deze applicatie. Om te vermijden dat we een variabele aanmaken per image, zullen we alle images opslaan in 1 globaal object.
Elke image geven we een id, dat we gebruiken als property in dit globale object.

In onze script.js maken we een nieuwe globale variabele aan om de images op basis van een id in op te slaan.

```
let catalog = {};
```

Uiteindelijk willen we op een dynamische manier extra properties gaan toekennen in dit catalog object. Bijvoorbeeld willen we in catalog.playerImage de image instantie met de player.png opslaan.
Wanneer je de naam voor een in-te-stellen property in een variabele bewaart, kun je niet gebruik maken van de standaard dot notatie:

```
let obj = {};
let key = 'propnaam';
let value = 'de waarde voor de property';
```

```
obj.key = value; //zal de value opslaan in een .key, en niet in
.propnaam
```

Je moet gebruik maken van square brackets – net zoals bij array addressering – om de properties in te stellen en uit te lezen:

```
obj[key] = value; //nu zit de waarde in obj.propnaam
```

Op deze manier kun je zelfs properties gaan definiëren met dots, dashes of andere karakters in de property name…

Met deze kennis in ons achterhoofd zullen we een nieuwe functie toevoegen aan onze **lib.js** die images kan inladen in een catalog object. Deze functie is identiek aan onze loadImage() functie maar nu krijgen we naast een URL ook een id (die we als key zullen gebruiken) en een catalog object (het object waarin we de resultaten zullen opslaan) binnen.

```
export const loadImageInCatalog = (url,id,catalog) => {
  return new Promise((response,reject) => {
      //image load
      catalog[id] = new Image();
      catalog[id].addEventListener(`load`,event =>
response(catalog[id]));
      catalog[id].addEventListener(`error`,event => reject(event));
      catalog[id].setAttribute('src',url);
      if(catalog[id].complete){
      response(catalog[id]);
      };
  });
}
```

In de init functie van onze script.js laden we die de 4 images in mbv promise chaining.

```
Promise.all([
```

```
      loadImageInCatalog(`img/bullet.png`,`bullet`,catalog),
      loadImageInCatalog(`img/player.png`,`player`,catalog),
      loadImageInCatalog(`img/enemy.png`,`enemy`,catalog),
      loadImageInCatalog(`img/explosion.png`,`explosion`,catalog)
]).then(loaded);
```

Een GameObject instantiëren

Tijd nu om onze GameObject klasse te testen.
Start met in je script.js je GameObject klasse te importeren:

```
import GameObject from './classes/GameObject.js';
```

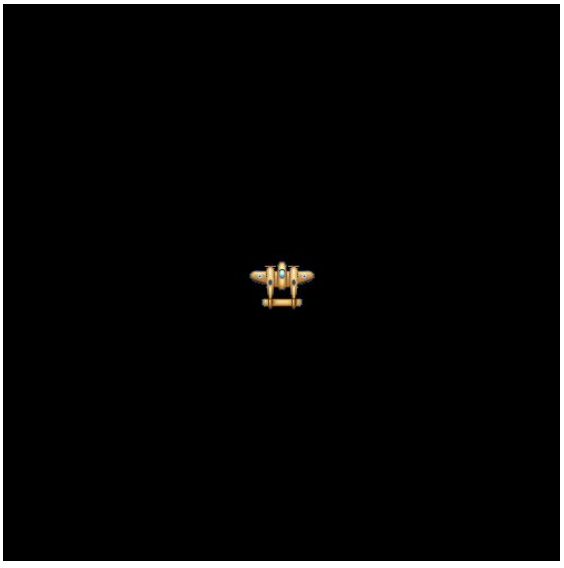
Wanneer alle images zijn ingeladen maken we een instantie aan van onze GameObject klasse.

```
myGameObject = new GameObject(canvas.width/2,canvas.height/2,
catalog.player);
myGameObject.numFrames = 3;
```

In de draw functie roep je de update en draw functie van de GameObject klasse op:

```
myGameObject.update();
myGameObject.draw(ctx);
```

Als alles goed is verlopen krijg je volgend resultaat in de browser te zien:



Nu we weten dat onze GameObject klasse werkt, kunnen we deze gebruiken als basis voor onze Player, Enemy, Bullet en Explosion.

Player

Player klasse

Zoals uit onze analyse is gebleken moet de Player klasse overerven van onze GameObject klasse.

1. Maak een **Player.js** bestand in src>js>classes
2. Importeer de GameObject klasse en erft ervan over.
3. Roep in de constructor ook de super constructor aan.
4. Stel de **frameRate** property in op 20 (zodat de motoren niet te snel draaien)
5. De player spritesheet bestaat uit 3 frames, dus stel je de **numFrames** property in op 3.
6. Voorzie een extra property boolean **killed** en stel deze standaard in op false.

Een Player instantiëren

Analoog aan wat we getest hebben met de GameObject klasse maak je nu een instantie aan van de Player klasse. (je mag de GameObject instantie verwijderen, dit was enkel om te testen).

Plaats een Player onderaan gecentreerd op de canvas. De player mag enkel getoond worden wanneer killed niet true is.

Keyboard

Keyboard klasse

De keyboard besturing gaan we afhandelen in een aparte klasse **Keyboard**. Deze klasse bevat 2 eventlisteners en bijhorende handler methods.

We voorzien ook een **isDown(keyCode)** method die we kunnen oproepen wanneer we willen weten of een bepaalde toets is ingedrukt of niet. Daarnaast staan er 3 constanten gedefinieerd voor de LEFT, RIGHT en SPACE toets.

```
export default class Keyboard{
  constructor(){
       this.keys = {};
       window.addEventListener(`keydown`, event => this.handleKeyDown(event));
       window.addEventListener(`keyup`, event => this.handleKeyUp(event));
       Keyboard.LEFT = 37;
       Keyboard.RIGHT = 39;
       Keyboard.SPACE = 32;
  }
  handleKeyDown(event){
       this.keys[event.keyCode] = true;
  }
  handleKeyUp(event){
       delete this.keys[event.keyCode];
  }
  isDown(keyCode){
       return this.keys[keyCode];
  }
}
```

Een Keyboard instantiëren

Instantieer een Keyboard object in je script.js wanneer de images zijn ingeladen. (vergeet ook niet de Keyboard klasse te importeren).

In de draw functie controleer je telkens welke toets is ingedrukt. (LEFT, RIGHT of SPACE).
De toetsen mogen niet meer werken wanneer de player killed boolean true is.
Vervolgens zullen we nu kijken hoe we de locatie kunnen aanpassen van de Player instantie.

Player forces

De player zijn locatie moet worden aangepast als we op het linkse of rechtse pijltje drukken.
We zouden rechtstreeks de **player.location.x** waarde kunnen aanpassen maar dan verspringt de player onmiddellijk naar de nieuwe positie.

Het zou beter zijn als dit met een vloeiende animatie gebeuren.

Hiervoor moeten we onze GameObject klasse een beetje uitbreiden:

1. Maak 2 nieuwe properties aan van het vector type: **velocity** en **acceleration**.
2. In de update() method tellen we de acceleration op bij de velocity en passen we de velocity toe op de location:

```
this.velocity.add(this.acceleration);
this.location.add(this.velocity);
```

3. In de **globale draw()** functie zouden we nu de acceleratie kunnen instellen wanneer er op de pijltjes wordt gedrukt.

```
player.acceleration.add(new Vector(-1, 0)); // linkse pijltje
player.acceleration.add(new Vector(1, 0)); // rechtse pijltje
```

4. Als je dit test dan zal de player alsmaar versnellen tot ver buiten het scherm. Dus we moeten een **vertraging** inbouwen. Dit kun je doen door de velocity elk frame een beetje te verkleinen.

```
this.velocity.mult(0.9);
```

5. Doe ook nog een reset van de acceleration vector, zodat deze niet blijft runnen.

```
this.acceleration.mult(0);
```

6. Test de applicatie. De player beweegt nu met een vloeiende beweging naar links en naar rechts.

In plaats van de acceleration property rechtstreeks aan te passen buiten het GameObject, is het beter om dit via een functie te doen. Zo blijven de interne properties van het GameObject beter afgeschermd van de buitenwereld.

Definieer een **applyForce** method in de GameObject klasse waarin je een vector binnenkrijgt en toepast op de acceleration.

```
applyForce(force){
   this.acceleration.add(force);
}
```

Maak nu gebruik van de applyForce functie in de draw() functie van script.js in plaats van rechtstreeks te werken op de acceleration property van de Player.

We krijgen nu volgende code voor onze LINKS/RECHTS besturing:

```
if(!player.killed){
  if(keyboard.isDown(Keyboard.LEFT)){
      player.applyForce(new Vector(-0.5,0));
  }
  if(keyboard.isDown(Keyboard.RIGHT)){
      player.applyForce(new Vector(0.5,0));
  }
  if(keyboard.isDown(Keyboard.SPACE)){
    console.log(`shoot`);
  }
}
```

Bullet

Shooting Bullets

Maak idem aan de Player class een Bullet class die overerft van de GameObject class. Hierin toon je de bullet.png.
Definieer een globale array **playerBullets** waarin je deze bullets zal bewaren.

Bij het indrukken van de spatiebalk maak je een nieuwe bullet aan op de player location.
Push de bullet in de array playerBullets en zorg dat de update en draw functie van alle bullets opgeroepen wordt.
Voor je de update uitvoert, stel je de velocity in op een vaste, negatieve waarde (BVB -5).
Vanaf nu schiet de player bullets af naar de bovenkant van het scherm.

Cleanup Bullets

Bij het afvuren van bullets maken we constant nieuwe instanties aan en voegen we deze toe aan de globale array. Ook al is een bullet niet meer zichtbaar (y coordinaat kleiner dan 0), toch bestaat deze nog en wordt deze 60 keer per seconde ge-update.
Na een tijdje zal onze game trager beginnen werken, omdat we nog met een heleboel onzichtbare bullets zitten).
In de draw functie zal je op het einde de bullets **filteren**. Wanneer ze niet meer zichtbaar zijn, verwijder je ze uit de playerBullets array. Je maakt opnieuw gebruik van de **Array filter functie**. Meer info : https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_Objects/Array/filter

```
playerBullets = playerBullets.filter(bullet => bullet.location.y >0);
```

Enemy

Enemy klasse

Een volgende stap is om enemies toe te voegen die in de richting van de player bewegen.
Maak een nieuwe klasse aan en noem deze Enemy.

Identiek aan de Player klasse stel je een frameRate, numFrames en killed property in.

Enemies instantiëren

Maak een globale array enemies aan. voeg in de draw functie op willekeurige momenten een enemy toe. Dit kun je doen door een check te doen op de terugkeerwaarde van Math.random():

```
if(Math.random() < 0.05){
  let enemy = new Enemy(canvas.width * Math.random(),0,catalog.enemy);
  enemies.push(enemy);
}
```

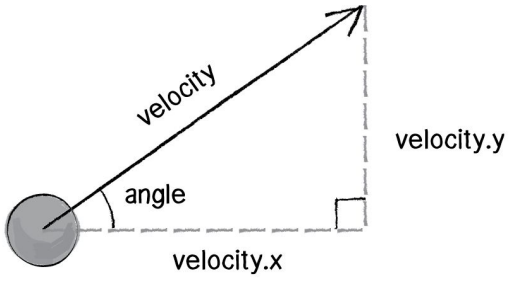
Voer de draw functie van alle enemies uit, zodat deze op het scherm verschijnen.

Enemy Movement

De enemies moeten bewegen in de richting van de player.
Bereken de richtingsvector tussen de enemy en de player. Gebruik een deel van de genormaliseerde vector en pas deze via de applyForce functie van de enemy toe, zodat deze beweegt in de richting van de player.

Test de applicatie. De enemies vliegen richting de player, alleen klopt de hoek van de graphic niet echt en vliegen ze zijwaarts.

We zullen de **graphic** moeten **roteren**, zodat deze in de richting van de player kijkt. Hiervoor gebruiken we opnieuw driehoeks-meetkunde. We zullen de waarden van onze richtingsvector gebruiken om de richting van de vector te berekenen.



tangent(angle) = velocity.y/velocity.x

De tangens van de hoek is gelijk aan velocity.y gedeeld door velocity.x.
De hoek zelf kun je dan bepalen door de boogtangens van deze deling te nemen.

if $tangent(angle) = velocity_y / velocity_x$

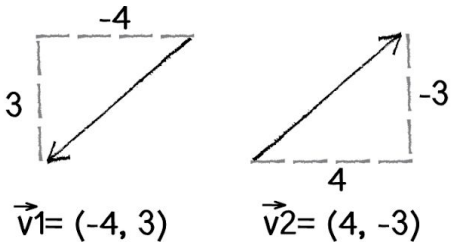
then $angle = arctangent(velocity_y / velocity_x)$

We zullen de enemy draw functie overschrijven en de hoek toepassen.

1. Kopieer de draw functie van de GameObject class in de Enemy class.
2. Bereken de tangens van de velocity, en pas dit toe op de graphic.
3. We voegen nog een extra rotatie toe (-Math.PI / 2) omdat de graphic reeds gedraaid staat:

```
ctx.translate(this.location.x, this.location.y);
let angle = Math.atan(velocity.y / velocity.x) - Math.PI / 2;
ctx.rotate(angle);
```

We zitten nog met een klein probleem: **twee tegengestelde vectoren zullen dezelfde hoek returnen** bij het berekenen van een boogtangens:

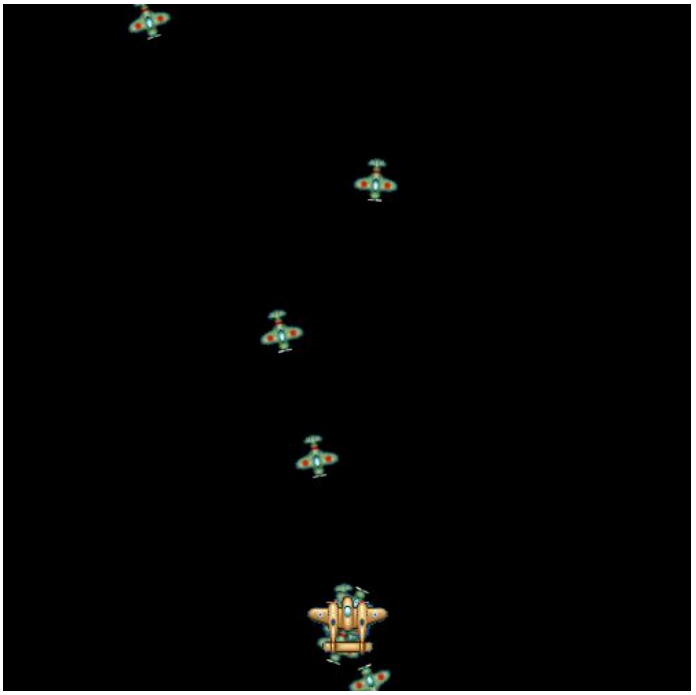


V1 ⇒ angle = atan(-4/3) = atan(-1.25) = -0.9272952 radians = -53 degrees
V2 ⇒ angle = atan(4/-3) = atan(-1.25) = -0.9272952 radians = -53 degrees

Gelukkig is er een ingebouwde functie, die de boogtangens berekent, rekening houdend met de richting van de vec: **Math.atan2**. Let op: je geeft er eerst de y-waarde mee, en als tweede argument de x-waarde:

```
ctx.translate(this.location.x, this.location.y);
let angle = Math.atan2(this.velocity.y,this.velocity.x) - Math.PI/2;
ctx.rotate(angle);
```

Test de applicatie. De enemies draaien zich nu in de richting van de player.



Cleanup

Zorg als laatste stap dat de enemies worden verwijderd wanneer ze buiten beeld vliegen.
Dit kan je doen zoals we met de bullets hebben gedaan. (array filter functie).

CollisionDetector

CollisionDetector klasse

We zullen collisions detecteren tussen de playerBullets en de enemies. Deze collisions zullen we in een aparte class gaan managen, zodat onze code modulair is en elke klasse een beperkte set van verantwoordelijkheden heeft.

Omdat deze klasse moet kunnen laten weten wanneer een collision heeft plaatsgevonden, zal deze klasse events moeten kunnen uitsturen. Hiervoor gaan we gebruik maken van een third party klasse die dit voor ons mogelijk maakt.

Download daarom eventemitter 2 van op https://github.com/asyncly/EventEmitter2 (lib > eventemitter2.js).
Plaats dit bestand in een **src>js>vendors** map.

Alvorens we starten met de implementatie van de CollisionDetector klasse gaan we een toevoeging doen aan onze **GameObject** klasse.
We zullen een methode voorzien om een botsing tussen 2 GameObjecten te detecteren.
Wanneer twee objecten met elkaar botsen is de afstand tussen beide zeer klein.
Dit resulteert in volgende code:

```
calculateDistance(otherElement){
      let distance = Vector.sub(otherElement.location ,
this.location).mag();
      distance = distance - otherElement.size/2 - this.size/2;
      return distance;
  }
  collidesWith(otherElement){
      return (this.calculateDistance(otherElement)<=0);
  }
```

Definieer nu een **CollisionDetector** klasse die overerft van EventEmitter2:

```
import EventEmitter2 from '../vendors/eventemitter2.js';

export default class CollisionDetector extends EventEmitter2{
  constructor(){
      super({});
  }
}
```

Definieer een functie detectCollisions in de CollisionDetector die 2 arrays binnenkrijgt: een array met elementen waarvan je collisions wil detecteren met de elementen in de tweede array:

```
detectCollisions(elements1, elements2) {

}
```

In deze functie overloop je alle elementen in de elements1 array. Voor elk element uit deze eerste array, overloop je alle elementen in de tweede array (elements2). Je moet botsingen detecteren tussen het element in de eerste array en het element in de tweede array. Dit kan je doen door de **collidesWith** methode op te roepen die we daarnet hebben toegevoegd aan de GameObject klasse.

Bij het detecteren van een botsing, stuur je een ‘collision’ event uit, en geef je deze twee elementen mee als extra parameters:

```
this.emit('collision', element1, element2);
```

Maak in de globale init functie nu een globale instantie aan van deze CollisionDetector, waarmee we botsingen tussen de bullets en de enemies willen detecteren. Koppel een functie aan het collision event:

```
playerBulletEnemyCollisionDetector = new CollisionDetector();
playerBulletEnemyCollisionDetector.on('collision',
playerBulletEnemyCollision);
```

Deze functie zal 2 parameters binnenkrijgen: de betreffende bullet en de enemy:

```
const playerBulletEnemyCollision = (playerBullet, enemy) => {

};
```

Plaats voorlopig een log statement in deze functie, zodat je kan checken of deze opgeroepen wordt.

In je globale draw functie moet je tenslotte nog de collisionDetection triggeren. Je geeft de array met player bullets en de array met enemies mee als parameters:

```
playerBulletEnemyCollisionDetector.detectCollisions(playerBullets,
enemies);
```

Test de code. Je zou logs moeten zien verschijnen bij een botsing tussen een bullet en een enemy.

In de playerBulletEnemyCollision functie zullen we in plaats van een log de playerBullet en de enemy verwijderen. Dit kun je doen door gebruik te maken van de filter methode op een array. Zo kan je ervoor zorgen dat de bullet en enemy instantie waartussen de botsing gebeurd is verwijderd worden uit respectievelijk de playerBullets en enemies array.

Explosion

We willen een explosie animatie tonen op de plaats waar we een enemy neerschieten. Maak een nieuwe Explosion class aan die werkt met de explosion.png image.
Deze animatie mag maar 1 keer afspelen. We zullen hiervoor de update functie uitbreiden:

```
update() {
  super.update();
  this.localFrameNr = Math.floor(this.frameNr / (60 / this.frameRate));
  if(this.localFrameNr >= this.numFrames) {
      this.killed = true;
```

```
    }
}
```

Maak een globale array aan waarin je de explosies zal bijhouden. In de collision detection event handler maak je een explosion aan op de locatie van de enemy die neergeschoten werd. Geef deze explosion nog een fractie van de enemy velocity, zodat deze nog een beetje zal bewegen.

In de globale draw functie roep je tenslotte de update en draw functies op van deze explosions, zodat we ze op het scherm zien verschijnen. Na de update call op de explosion, gooi je de explosions die "killed" zijn weg:

```
explosions = explosions.filter(explosion => !explosion.killed);
```

Player Killed

We moeten nog collisions detecteren tussen een enemy en de player. Maak een 2de instantie aan van onze collisiondetector class en koppel opnieuw een event handler aan het collision event.
In de draw functie moeten we nu de detectCollisions functie oproepen, met als parameters 2 arrays. We hebben echter maar 1 player; wrap deze dus in een array wanneer je deze doorgeeft aan de collision detection functie:

```
playerEnemyCollisionDetector.detectCollisions([player], enemies);
```

Bij een collision tussen de player en de enemy toon je een explosion op de plaats van de player.

De game is gedaan: zorg ervoor dat we niet meer kunnen schieten, en dat de enemies vanaf nu rechtdoor vliegen (niet langer naar de positie van de player).

Extra functionaliteiten

- Laat de enemies ook bullets afvuren richting speler & detecteer botsingen
- Voorzie een systeem van levens: een player kan 3x geraakt worden alvorens hij sterft.
- Zorg dat de game moeilijker wordt naarmate de tijd vordert: meer enemies die sneller bewegen.