WebGL/canvas/PIXI.js

WebGL er en JavScript API til at rendere interaktive 2D og 3D grafik i en kompatibel web browser, uden brug af plug-ins. WebGL er fuldt integreret med andre Webstandarder. WebGL gør brug af **GPU** accelereret data, physics og image afvikling, som en del af en websides canvas.

https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API

Hvad er GPU:
En grafikprocessor enhed, grafisk bearbejdningsenhed eller GPU er et computer komponent, der er specifikt designet til at vise tredimensionel grafik på en computerskærm med høj billedhastighed. En GPU er som regel på et separat Grafikkort, men den kan også være integreret på ens Motherboard eller CPU.

Der findes mange biblioteker og frameworks, der arbejder med WebGL og canvas. Vi vil fokusere på PIXI.js:

Hvad er PIXI:
Pixi er et renderings bibliotek, som kan udvikle: "Rich, Interative graphic experiences" og spil og som er nemt tilgængelig og nemt at bygge i med.

https://www.pixijs.com/
https://www.pixijs.com/faq

Andre Super Soniske Seje biblioteker/Frameworks:
https://gist.github.com/dmnsgn/76878ba6903cf15789b712464875cfdc

For at arbejde med PIXI.js kræves en web browser … og vores sædvanlige Miljø/boilerplate, hvor vi indtil nu har arbejdet med HTML5, CSS/SCASS og JavaScript.
Som udgangspunkt er PIXI.js installeret i vores boilerplate, så vi kan bare kaste os ud i det. VI ser lige på hvad canvas elementet er for noget dingel dangel.

Hvad er Canvas:
Canvas er interaktivt tegnebræt, hvor du kan tegne grafik med JavaScript. Canvas er et HTML5 element <canvas id="canvas"></canvas>.

Se mere her:
https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API

Vi skal lave et lille spil, eller indledningerne til et spil. Vi vil lave den motor, som efterfølgende kan bruges til at lave flere levels. Vi laver et "Ninja" spil, hvor det går ud på at smadre så mange fjender, som muligt. fjenderne komme ind fra siderne og det er så brugerens opgave at eliminere så mange som muligt. Spillet vil arbejde med animationer og lyd og masser af interaktion.

Vi arbejder med samme model, som vi har gjort indtil videre, så vi har vores sædvanlige opsætning. Du kan omdøbe Initalize.js til Game.js og kalde den via App.js og importer PIXI:

```
import * as PIXI from 'pixi.js';
```

I Pixi.js kan vi oprette animationer, grafik, tekst og lyd, som bliver implementeret via PIXI´s version af canvas elementet. I dette canvas arbejder vi. Vi kommer til at arbejde med mange områder af pixi, efterhånden som vores spil bliver udviklet.

Der er en del opsætning til opretning af vores canvas, så lad os lave det i en selvstændig klasse. Vi laver vores spil, så det kan indgå i et webview i en app. Så vi udvikler til tablets og mobil.

Opret en klasse "Stage.js".

Importer PIXI:

```
import * as PIXI from 'pixi.js';
```

Importer Stage klassen I Game klassen med det samme. Så nu bliver Game klassen kaldt fra App.js og vi har importeret Stage klassen i Game klassen.

Check om alt kører som det skal. Husk at køre en: npm run webpack

Vi starter med at oprette nogle variabler i Stage klassens konstruktør, som vi vil bruge senere til at styre skaleringen af spillet, så de kan tilpasses alle devices:

```
//Stage.js

import * as PIXI from 'pixi.js';
import '../css/style.scss';

export default class Stage {

  constructor() {

    this.targetWidth = 1700;
    this.targetHeight = 768;
    this.targetCenter = 1024; // centers interactive area for objects (not backgroun
d)

   this.appWidth = window.innerWidth;
    this.appHeight = window.innerHeight;

    this.scaleFactor = this.appWidth / this.targetWidth;

  }

}
```

Vi skal nu oprette vores canvas igennem PIXI.Application Dette gør vi også i Stage Klassen:

```
//Stage.js

this.app = new PIXI.Application({

        autoResize: true,
        resolution: devicePixelRatio,
        backgroundColor: 0xcccccc,
        width: this.appWidth,
        height:this.appHeight,
        antialias: true,
        resolution: window.devicePixelRatio || 1
    });

    document.body.appendChild(this.app.view);

I nogle tilfælde, uvist af hvad, kan det være nødvendigt at tilføje noget css, for at få browseren til at skalerer din canvas korrekt.

 this.app.view.style.width = this.appWidth + "px";
    this.app.view.style.height = this.appHeight + "px";
```

Så er vores canvas klar og skalerbar.

I PIXI kan man arbejde med containere. Vi skal oprette to af disse. En til baggrunden og en til alle vores objekter. De bliver skalerbare, så de tilpasser sig vores canvas og dermed alle devices (næsten)

```
//Stage.js

this.bg = new PIXI.Container();
        this.bg.x = this.appWidth / 2;
        this.bg.y = this.appHeight / 2;
        this.bg.pivot.x = this.targetWidth * 0.5;
        this.bg.pivot.y = this.targetHeight * 0.5;
        this.bg.interactive = true;
        this.app.stage.addChild(this.bg);
        //this.bg.interactiveChildren = false;
        this.bg.getBounds();
        this.bg.scale.x = this.bg.scale.y = this.scaleFactor;
        this.bg.scale.y = this.bg.scale.x = this.appHeight / this.targetHeight;


        this.scene = new PIXI.Container();
        this.scene.x = this.appWidth / 2
        this.scene.y = 0;
        this.scene.pivot.x = this.targetCenter * 0.5;
```

```
        //this.actors.pivot.y = this.targetCenter * 0.5;
        this.app.stage.addChild(this.scene);
        this.scene.getBounds();
        //this.actors.scale.x = this.actors.scale.y = this.scaleFactor;
        //this.actors.scale.y = this.actors.scale.x = this.scaleFactor;
        this.scene.scale.x = this.scene.scale.y = this.scaleFactor;
        this.scene.scale.y = this.scene.scale.x = this.appHeight / this.targetHeight;

        this.app.renderer.resize(this.appWidth, this.appHeight);
```

Til sidst render vi alt det vi har lavet indtil videre.

Vi vil gerne have mulighed for at tilgå en del af de ”settings” variabler, som vi har brugt i Stage klassen, så det sidste vi gør i Stage klassen er at lave en get funktion, som returnere nogle settings, som vi kan tilgås fra Game klassen.

Så i Stage klassen opretter du en get funktion:

```
//Stage.js

get stageInfo() {

    let si = {

        appWidth: this.appWidth,
        appHeight: this.appHeight,
        targetHeight: this.targetHeight,
        targetWidth: this.targetWidth,
        scaleFactor: this.scaleFactor,
        app: this.app

    }

    return si;

}
```

Så er din Stage klasse færdig og du kan nu bruge den i f.eks Game klassen til oprette grafik, som kan kan indlejres i de to containere og som vil blive skaleret efter device størrelse. Jeg har lavet en scss fil, der som det eneste sætter margin på body til 0 Q

Din Stage klasse skulle gerne se sådan her ud nu:

```
//Stage.js
```

```
import * as PIXI from 'pixi.js';
import '../css/style.scss';
export default class Stage {




constructor() {

  this.targetWidth = 1700;
  this.targetHeight = 768;
  this.targetCenter = 1024; // centers interactive area for objects (not background)

  this.appWidth = window.innerWidth;
  this.appHeight = window.innerHeight;


    this.scaleFactor = this.appWidth / this.targetWidth;

    this.app = new PIXI.Application({
        autoResize: true,
        resolution: devicePixelRatio,
        backgroundColor: 0xcccccc,
        width: this.appWidth,
        height:this.appHeight,
        antialias: true,
        resolution: window.devicePixelRatio || 1
    });

    document.body.appendChild(this.app.view);

    this.bg = new PIXI.Container();
        this.bg.x = this.appWidth / 2;
        this.bg.y = this.appHeight / 2;
        this.bg.pivot.x = this.targetWidth * 0.5;
        this.bg.pivot.y = this.targetHeight * 0.5;
        this.bg.interactive = true;
        this.app.stage.addChild(this.bg);
        //this.bg.interactiveChildren = false;
        this.bg.getBounds();
        this.bg.scale.x = this.bg.scale.y = this.scaleFactor;
        this.bg.scale.y = this.bg.scale.x = this.appHeight / this.targetHeight;


        this.scene = new PIXI.Container();
        this.scene.x = this.appWidth / 2
        this.scene.y = 0;
```

```
        this.scene.pivot.x = this.targetCenter * 0.5;
        //this.actors.pivot.y = this.targetCenter * 0.5;
        this.app.stage.addChild(this.scene);
        this.scene.getBounds();
        //this.actors.scale.x = this.actors.scale.y = this.scaleFactor;
        //this.actors.scale.y = this.actors.scale.x = this.scaleFactor;
        this.scene.scale.x = this.scene.scale.y = this.scaleFactor;
        this.scene.scale.y = this.scene.scale.x = this.appHeight / this.targetHeight;

        this.app.renderer.resize(this.appWidth, this.appHeight);

  }

  get stageInfo() {

    let si = {

        appWidth: this.appWidth,
        appHeight: this.appHeight,
        targetHeight: this.targetHeight,
        targetWidth: this.targetWidth,
        scaleFactor: this.scaleFactor,
        app: this.app

    }

    return si;
  }
}
```

Vi skal nu oprette en Game.js fil og i den opretter vi en Game klasse. Vi importer vores Stage klasse. Derefter initialiser vi vores Stage klasse i Game konstruktøren. Vi arbejder fremadrettet i konstruktøren:

```
//Game.js

import Stage from './Stage'

export default class Game{

constructor(){

    this.myStage = new Stage();

        }}
```

Det næste vi skal er at hente vores to containere, scene og bg og gøre dem klar til brug:

```
//Stage.js

export default class Game{

constructor(){

    this.myStage = new Stage();
    this.scene = this.myStage.scene;
    this.scene.sortableChildren = true; // Så kan vi bruge z-index på children
    this.background = this.myStage.bg;
    this.si = this.myStage.stageInfo;

        }
}
```

VI bruger sortableChildren på en parent for at fortælle at vi vil have mulighed for at bruge z-index i vores container.

Den sidste klasse variable, som vi henter ind: this.si = this.myStage.stageInfo er at vi kan tilgå vores get funktion med vores objekt egenskaber, som vi oprettede i Stage klassen.

I Pixi kan man oprette en loader, PIXI.Loader, der sørger for at data, man vil bruge, er hentet/loaded. I den forbindelse vil vi oprette et array, der henviser til vores grafik og som vi kan kalde i vores loader. Så opretter vi vores Loader og bruger add metoden af Loader objeket til at hente de data vi skal bruge. Du kan oprette lige så mange add metoder som du har brug for. Når alle data er klar kaldes load metoden og så kan vi begynde at oprette vores grafiske brugerflade:

```
//Game.js

import {gsap} from 'gsap';
import Stage from './Stage'
import Enemy from './Enemy'
import * as PIXI from 'pixi.js';
import {Spine} from 'pixi-spine';
import {howl, howler} from 'howler';

export default class Game{

constructor(){
```

```
    this.myStage = new Stage();
    this.scene = this.myStage.scene;
    this.scene.sortableChildren = true;
    this.background = this.myStage.bg;
    this.si = this.myStage.stageInfo;

    let assets = [
        '../assets/spritesheet/ninjarack.json',
        './assets/images/background.jpg',
        '../assets/images/ninja-jump.png',
        '../assets/images/play.png'
      ];

      const loader = PIXI.Loader.shared
       .add(assets)

      .add('alienspine', '../assets/spritesheet/alien-spine/alienboss.json')

      .load((loader,res) => {
          console.log('ready for game');
        })
  }

}
```

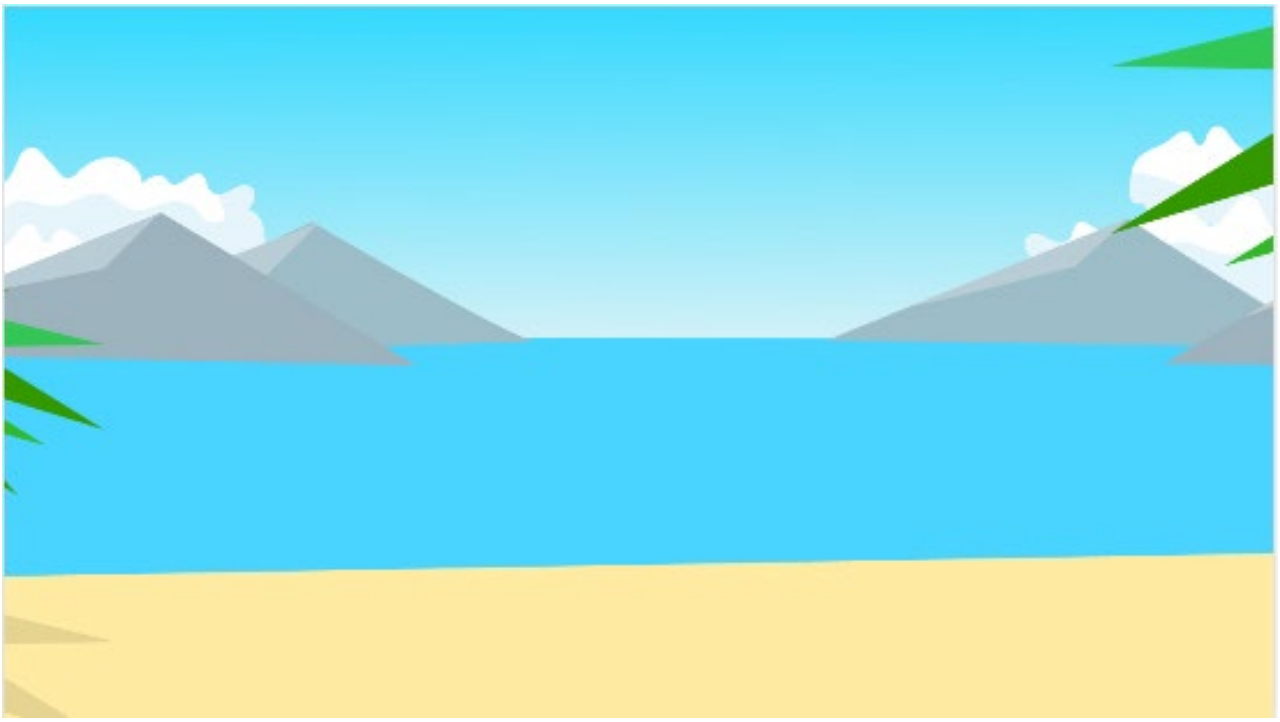
Vi kan starte med at smide et baggrundsbillede ind i vores load funktion. Det vil vi placere i vores scene container. I Pixi henter vi en texture for derefter at oprette en Sprite, som vi kan tilføje til en container.

Læs mere om sprites og textures i Pixi:

https://pixijs.download/dev/docs/PIXI.Sprite.html

```
//Game.js

let bgTexture = PIXI.Texture.from('./assets/images/background.jpg');
        let _bg = new PIXI.Sprite(bgTexture);
       this.background.addChild(_bg);
```

Så skulle du gerne have følgende:



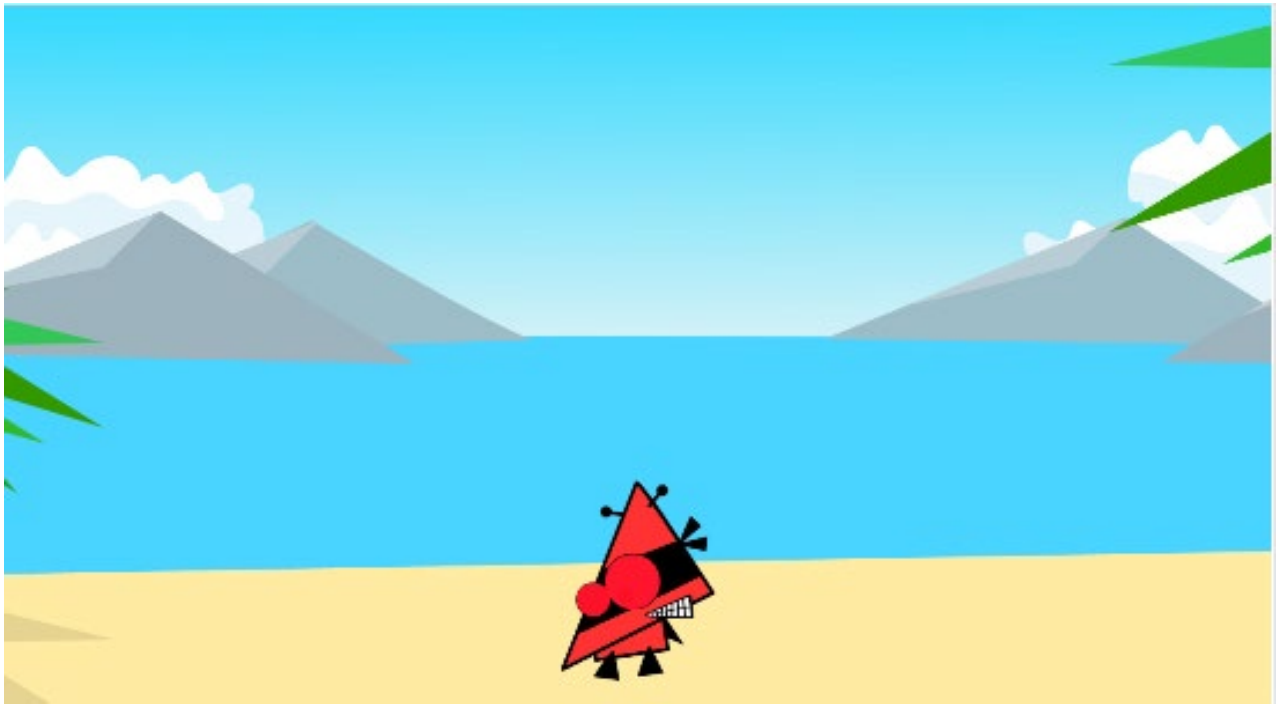
Vi skal nu have vores Ninja på scenen. Nija har en idle animation og animationen er udformet, som en Spritesheet. Et Spritesheet er en række billeder der er samlet til et stykke grafik og har en json struktur til at placere og animerer de enkelte sprites, der indgår i et spritesheet. Ninja´s Spritesheet er lavet i TexturePacker, hvor der er importeret en række billeder, som giver en animation, som derefter er eksporteret med en json fil. Selve Ninja Animationen er laver i Adobe Animate.

Som du kan se i vores assets Array henviser vi til en json fil ”ninjarack.json”, den har informationer og alle de enkelte billeder der til sammen giver en animation, desuden har den også en reference til det samlede spritesheet billede.

Først vil vi lave en reference til den ressource, altså vores spritesheet, som vi vil bruge. Derefter indsætter vi den animerede sprite med PIXI.AnimatedSprite(). Når vi bruger AnimatedSprite refererer vi til strukturen i det spritesheet vil vi l bruge. Som udgangspunkt er alle henvisninger til de enkelte grafikstykker, som udgør animationen navngivet med et unikt navn efterfulgt af en underscore og et fortløbende nummer:

"alien_0001.png", "alien_0002.png" …osv. Så kan AnimateSprite få fat i de data den skal bruge for at lave animationen. Derefter er det bare at placere Ninja, hvor vi vil have den på scenen. VI implementerer Ninja i vores Scene container. Det gør vi fremadrettet med alt vores grafik i dette projekt. Når den er placeret kan vi med play() metoden af AnimateSprite starte vores animation.

```
//Game.js

let sheet = PIXI.Loader.shared.resources['../assets/spritesheet/ninjarack.json'].spritesheet;

      this.ninja = new PIXI.AnimatedSprite(sheet.animations['alien']);
      this.ninja.anchor.set(0.5);
      this.ninja.x = 512;
      this.ninja.y = 768 - 150;

      this.ninja.interactive = true;
      this.ninja.buttonMode = true;
      this.ninja.zIndex = 2;
      this.ninja.animationSpeed = 0.5;

      this.ninja.play();
      this.scene.addChild(this.ninja);
```



Det næste vi vil er at, når vi trykker hvor som helst på skærmen hopper vores ninja derhen. Til det vil vi bruge GSAP, og vi vil lægge lyd på med Howler. Vi vil også skifte Ninjas texture ud når den angriber og skifter texture tilbage, når den er færdig med at angribe. Vi vil bruge en eventListener ”on” til det. Når

objekter skal interageres med i Pixi, skal de sættes til at være interaktive. I dette tilfælde er det vores Pixi.Stage vi vil gøre aktivt:

```
//Game.js
this.si.app.stage.interactive = true;

  this.si.app.stage.on('pointerdown', (event) => {

        this.ninja.stop(); // stop ninja animation
        this.ninja.texture = PIXI.Texture.from('../assets/images/ninja-
jump.png'); //skift texture til jump;


       })
```

Så skal vi, når der trykkes, hvor som helst på skærmen få vores ninja til at hoppe derhen og når den animation er færdig hoppe tilbage igen og skifter texture tilbage til idle animationen. Det bruger vi GSAP til. Så den skal importeres i Game klassen. Desuden skal vi finde musens/fingerens position når vi trykker på skærmen, som vi derefter så kan bruge i vores GSAP animation:

```
//Game.js

this.si.app.stage.on('pointerdown', (event) => {

        this.ninja.stop(); // stop ninja animation
        this.ninja.texture = PIXI.Texture.from('../assets/images/ninja-
jump.png'); //skift texture til jump;

        let newPosition = event.data.getLocalPosition(this.background);

        gsap.to(this.ninja, {
            duration: 0.2,
            x: newPosition.x-300,
            y:newPosition.y,

            ease: "Circ.easeOut",

            onComplete: () => {

              gsap.to(this.ninja, {
                duration: 0.2,

                x: 500,
                y:768-150,
```

```
                ease: "Circ.easeOut",
              });

              this.ninja.play();
            }, //END: onComplete
          });
    })
```

Som du kan se skal vi også sikre os at, uanset hvilken retning ninja springer, skal han vender den rigtige vej, når han angriber. Denne skal også placeres i eventListneren:

```
//Game.js
let mX = event.data.global.x;

   mX > this.si.appWidth/2 ? this.ninja.scale.x = -1 : this.ninja.scale.x = 1;
```

Vi bruger en event.data.global for at få positionen af vores museposition og hvis den er større end appWidth/2, så skal Ninja scale.x være lig -1 hvilket vil få den til at roterer horisontalt og ellers skal scale.x være lig 1

Lyd i PIXI

Vi skal også have noget lyd på og til det vil vi bruge Howler. Vi vil i første omgang bare have en lyd på, når han angriber. Lad os starte med at importer Howler ind i vores Game klasse og videre kalde Howl i eventHandleren og sætte den til at afspille lyden når vi trykker på skærmen. Den skal placeres i din ninja eventhandler.

```
//Game.js

this.hitSound = new Howl({
            src:['./assets/sound/effekt_swish.mp3'],
            volume:0.5
            })

            this.hitSound.play();
```

Vi smider mere lyd på senere. Det næste der skal ske, er at vi gerne vil have en start ikon, der, når man klikker på det, starter spillet. Så vi skal have importeret et stykke grafik, som skal placeres midt på skærmen, som skal hentes ind i Pixi.Load sammen med det andet grafik, der allerede er hentet ind. I kan oprette den i jeres konstruktør, men ikke i eventHandleren.

```
//game.js

let playTexture = PIXI.Texture.from("./assets/images/play.png");

        let play = new PIXI.Sprite(playTexture);
        play.anchor.set(0.5);
        play.x=512;
        play.y=250;
        play.interactive = true;
        play.buttonMode = true;
        this.scene.addChild(play);
```

Så skulle det hele gerne se sådan her ud:

Som du kan se kan du stadigvæk hopper rundt med din Ninja, selvom spillet ikke er sat i gang endnu. Så vi flytter vores aktivering af vores stage ind i en eventHandler, så den først bliver aktiv, når man har trykket på start. Så lad os oprette en eventHandler på vores startknap og sætte vores interaktivitet til at være true på vores stage og så slette den vi oprettede tidligere. Så skulle Ninja ikke hoppe før du har trykket på start:

```
//Game.js

play.on('pointerdown', (event) => {

        event.stopPropagation();

        this.si.app.stage.interactive = true;


        })
```

.. og når man har trykket på skiltet animerer vi skiltet ud af skærmen. Det bruger vi GSAP til. Vi vil også have en lyd når skiltet animeres ud. Det bruger vi Howler til. Vi sætter en timer på lyden, da der er sat en easing på animationen, der gør at den lige skal i gang 😊 før den forsvinder ud af skærmen. Desuden starter vi noget ambient sound.

```
//Game.js

play.on('pointerdown', (event) => {

        event.stopPropagation();
        this.si.app.stage.interactive = true;

        gsap.to(event.currentTarget, {
            duration: 0.5,
            delay:0.2,
              y: play.y - 350,
              ease: "Elastic.easeInOut",
            });

            let soundSwirp = new Howl({
                src:['./assets/sound/effekt_swish.mp3'],
                volume: 0.2
                })

                let timerid = setTimeout(() => {
                  soundSwirp.play();
                }, 500);

                let sound = new Howl({
                    src:['./assets/sound/musicloop.mp3'],
                    autoplay: true,
                    loop: true,
```

```
                volume: 0.5
                })
            })
```

A hero needs an Enemy … or the other way around

Som tidligere skrevet skal der komme ”fjender” ind fra siden. Vi laver en Enemy klasse, som opretter et antal fjender, som kommer random ind fra enten højre eller venstre side. Vores fjende vil blive oprette, som en Spine animation, som er en meget komprimeret spritesheet animation, som bliver animeret i Spine:

http://esotericsoftware.com/

Start med at lave en Enemy.js fil og Opret klassen Enemy og importer GSAP. desuden skal du hente, via npm, set-random-interval og importer den i Enemy klassen og brug en parameter items i konstruktøren.

```
//Enemy.js

import {gsap} from 'gsap';
import setRandomInterval from 'set-random-interval';

export default class Enemy{

constructor(items){

    console.log(items);

     }

}
```

Lad os test den i Game.js. Vi vil starte den op i play eventListeneren, som vi oprettede til vores start knap i Game klassen. Start med at importer Enemy klassen ind i Game klassen. Vi skal have to parametre med over til Enemy klassen fra Game klassen, nemlig vores reference til til vores enemy spritesheet og vores container, hvor hvor vores fjender skal oprettet i.Men først skal vi have en klasse variabel, til vores enemy, da vi skal tilgå den senere. Så i toppen af konstruktøren i Game klassen sætter vi en klasse variabel

```
//Enemy.js
this.enemy
```

Og så i play eventlisteren:

```
 //Enemy.js

this.enemy = new Enemy({
    name:res.alienspine,
```

```
    addTo:this.scene

});
```

Installer spine og PIXI og importer disse.

```
import * as PIXI from 'pixi.js';
import {Spine} from 'pixi-spine';
```

Så lige så snart vi starter spillet er vi i gang med at hente fjender, så dem skal vi have gang i. I vores Enemy klasse skal vi have oprettet vores fjende med en Spine animation, som kommer gående ind enter fra højre eller venstre, så vi skal i Enemy klassen have oprettet en del variabler.

```
//Enemy.js

import {gsap} from 'gsap';

import setRandomInterval from 'set-random-interval';

export default class Enemy{

constructor(items){

    this.resname = items.name;
    this.container = items.addTo;

    this.startFrom = 0;
    this.endAt = 0;
    this.front = 0;
    this.enemyArray= [];
    this.enemyDuration = [20,21,22,23,24,25,26,27,28,29,30,40,50];

    this.from = ["left", "right"];

    this.counter=0;

     }

}
```

Så skal vi til at oprette vores Enemy, som vi opretter i en ny Pixi container og som bliver oprettet, så der altid dukker nye Enemis op i et interval. Så det første vi gør er at oprette en setInterval, men vi gør brug af

den pakke, som vi installerede og allerede har importeret tidligere: set-random-interval. Så lad os oprette den i konstruktøren i Enemy klasse.

```
//Enemy.js

const interval = setRandomInterval(() =>{
      console.log("im the enemy");
   },1000,5000)
```

Vi kan teste om vores interval kører, som den skal ved at smide en console.log. Så nu bliver vores interval "skudt" af mellem en værdi fra 1000 til 5000 millisekunder.

Det næste vi gør er at op en variabel, som henter random fra vores this.from array, da vi vil bruge den string vi får tilbage til at bestemme fra hvilken side vores Enemy skal komme ind fra.

```
 //Enemy.js

const interval = setRandomInterval(() =>{

        var getFrom = this.from[Math.floor(Math.random() * this.from.length)];
        this.counter++;

      if(getFrom == "left") {

        this.startFrom= -400;
        this.endAt = 1700;
        this.front = 1 ; // Hvis man bruger en scale.x egenskab på congtaineren o
g sætter den til -1 vil den flippe horizintalt


      }else{

        this.startFrom= 1700;
        this.endAt = -400;
        this.front = 1; ;// Hvis man bruger en scale.x egenskab på congtaineren o
g sætter den til -1 vil den flippe horizintalt
      }

      console.log(getFrom, this.startFrom, this.endAt);
```

Prøv at consol.log getFrom, this.StartFrom, this.endAt. Så kan du se hvad der sker. Det er disse data vi vil bruge til at animere vores fjendes indtog på scenen.

Nu har vi de data vi skal bruge til at vores Enemy kan bygges. Som sagt bruger vi en spine animation, som er pakket ind i en Pixi container. Det betyder at vi kan animere containeren med GSAP og spine animationen vil indeholde de animationer, som skal bruges til at gå, angribe og dø.

Så først opretter vi vores Enemy container i vores interval.

```
//Enemy.js
this.enemyContainer = new PIXI.Container();

      this.enemyContainer.x=this.startFrom;
      this.enemyContainer.data=this.enemyDuration[Math.floor(Math.random() * this.e
nemyDuration.length)];// Denne bruger vi til GSAP
//Enemy.js

this.enemyContainer.alive = true; // Denne kan sættes til false hvis denne specifik
ke enemy er død og sikre at vi ikke interagere med den når den er død.
       this.enemyContainer.id=this.counter;
      this.enemyContainer.y=768-50; // placering på y aksen
      this.enemyContainer.scale.x =  this.front; // hvilken vej skal fjenden vende
      this.enemyContainer.zIndex = 1;
      this.container.addChild( this.enemyContainer );

      this.enemyArray.push(this.enemyContainer); // alle enemies bliver lagt I et a
rray, så vi kan styre de enkelte enemies.
```

Så skal vi have oprettet vores spine i vores enemyContainer. Som det første skal du have importeret vores Pixi-spine pakke i Vi smider også en hitarea på vores enemy, så vi har bedre kontrol over det område, vi senere skal bruge til at detecte noget intersection mellem vores Ninja og enemy. Til det bruger vi Pixi.Graphics og opretter et rectangle, som vi placere centreret i vores enemyContainer.

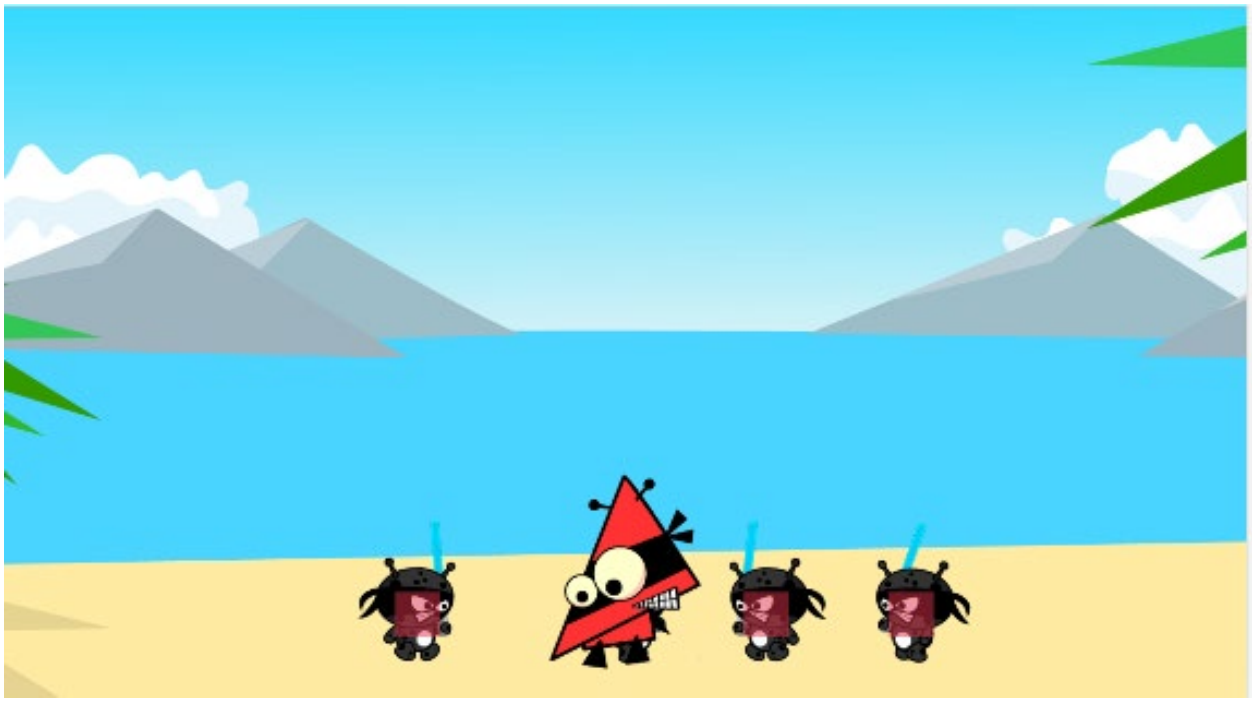
```
 //Enemy.js

       const alienEnemy = new Spine(this.resname.spineData);
      alienEnemy.x=0;
      alienEnemy.y=0;
      alienEnemy.state.setAnimation(0, 'walk', true);
      this.enemyContainer.addChild(alienEnemy);

      const hitarea = new PIXI.Graphics();
      hitarea.beginFill(0xDE3249);
      hitarea.drawRect(-25, -75, 50, 50);
```

```
      hitarea.alpha=0.5;
      hitarea.endFill();
      this.enemyContainer.addChild(hitarea);
```

Når vi afvikler dette, sker der ikke noget, umiddelbart. Så lad os få animeret vores container med GSAP, så vores enemies begynder at enter scenen.

```
//Enemy.js

gsap.to(this.enemyContainer, {
        duration:this.enemyContainer.data,
         x: this.endAt,
         ease: "Power0.easeNone",
         onComplete: () => {

          this.container.removeChild(this.enemyArray[0]);
          this.enemyArray.shift();

         }

        });
```

Som du kan se bruger vi nogle af vores settings variabler til at animere vores enemy. Desuden sikrer vi os også, at når vores spine container er færdig med animationen smider vi referencen til vores container fra arrayet ud og tømmer arrayet for den aktuelle spine container. Så skulle alt se sådan her ud og du skulle gerne have nogle enemies der kommer ind på scenen.



Så er vi næsten færdige med vores Enemy klasse. Vi skal dog lige oprette en enkelt get funktion, som returner det array, som vi har fyldt op med enemies, så vi kan tilgå disse fra Game klassen.

```
 //Enemy.js

get enemies(){

        return this.enemyArray;

        }
```

Intersection with the enemies

Når vores Ninja hopper og rammer enemies skal de selvfølgelig bare dø og til det vil vi i Game klassen oprette en Pixi.ticker. En ticker i Pixi, som jo er canvas og egenligt bare en en requestAnimationFrame, som kan afvikle et loop, som andre objekter kan lytte på. I dette tilfælde vil vi lytte på, hvornår Ninja kollidere eller har en intersection med en enemy. Til det vil vi oprette en ny klasse og skrive en meget ”simpel” HItTest klasse der lytter på om to objekter rammer hinandens x og y koordinater. Så opret en ny klasse der hedder HitTest.js

```
//Enemy.js
```

```
export default class Hittest {

    constructor() {

    }

    checkme(a,b){

        var ab = a.getBounds();
        var bb = b.getBounds();
        return ab.x + ab.width > bb.x && ab.x < bb.x + bb.width && ab.y + ab.height >
 bb.y && ab.y < bb.y + bb.height;

    }

};
```

Når vi bruger pixi.getBounds() kan vi finde vores objekters bredde og højde, som vi så returnere og sammenligner vores to objekter på x og x og på bredde og højde. Denne klasse kan vi så bruge i Game klassen til at detecte intersection mellem to objekter. Vi importer hitTest klassen ind i Game klasse og initialisere den med det samme i konstruktøren og så bruger vi den i Pixi.Ticker, som så skal holde øje med hvornår to objekter kolliderer.

```
//Game.js

import Hittest from "./Hittest";
```

Initialiser den i konstruktøren

```
//Game.js

this.ht = new Hittest();
```

Og nu skal vi så have den sat op i Pixi.Ticker i konstruktøren. Placer den nederst i din Game klasse konstruktør

https://pixijs.download/dev/docs/PIXI.Ticker.html

```
//Game.js

let ticker = PIXI.Ticker.shared;

      ticker.add((delta) => {

        console.log('ticker');
```

```
})
```

Så kan vi begynde at detecte med vores hitTest klasse i vores ticker, som er sat op til at holde øje med vores stage.

Det første vi skal sikre os er at vores enemy er klar og laver vi et loop (forEachs), som vi bruge til at sørge for at alle vores enemy bliver tjekket om de bliver kollideret med. Vi henter faktisk vores get funktion fra Enemy klassen, som jo returnere et array. Når det er på plads bruger vi vores hitTest klasse til at detecte og når de kollidere skifter vores enemy animation ud med en ”die” animation.

```
//Game.js

let ticker = PIXI.Ticker.shared;

    ticker.add((delta) => {

      if(this.enemy != undefined){

        this.enemy.enemies.forEach(_enemy => {

//Game.js

if (this.ht.checkme(this.ninja, _enemy.getChildAt(1))) {

            const currentEnenmySpriteSheet = _enemy.getChildAt(0); //henter objekt fra
            currentEnenmySpriteSheet.state.setAnimation(0, 'die', true);//henter ny animation
             }// END if

          })//END forEach
        } // END if


      }) // END ticker
```

Vi testet om det hele funker.

Vi får fat i den aktuelle enemy ved at bruge getChildAt på vores enemy container og så kan vi sætte en animation på den, da det er vores Spine animation vi får fat i. Det andet objekt der er i vores enemy container er jo et graphic objekt, som vi bruger til at hitTeste med.

Det næste vi skal er at vi vil bruge GSAP til at få vores enemy til at springe op og falde ned, for derefter at forlade skærmen, så den får en mere ”naturlig død”

Hvis vi bruger vores ”normale” GSAP metode, kunne det sådan her ud

```
//Game.js
gsap.to(_enemy, {
                  duration: 0.7,
                  y:200,
                  ease: "Circ.easeOut",
                  onComplete: () => {

                    gsap.to(_enemy, {
                      duration: 0.5,
                     y:1200,
                     ease: "Circ.easeIn",
                     onComplete: () => {
                       this.scene.removeChild(_enemy);
                     }

                    })
                  }
                });
```

Det ser noget rodet ud. GSAP har en timeline animation, hvor vi kan få det sat mere overskueligt op.

https://greensock.com/docs/v3/GSAP/Timeline

```
//Game.js


let enemyDieTimeline = gsap.timeline({
                onComplete: () =>{
                  this.scene.removeChild(_enemy);

                }
              });
              enemyDieTimeline.to(_enemy, {y: 300, duration: .7, ease: "Circ.easeOut"});
              enemyDieTimeline.to(_enemy, {y: 1200, duration: .5, ease: "Circ.easeIn"});
```

Meget nemmere at overskue. Så når timeline animationen er færdig fjerner vi vores enemy med Pixi´s removeChild() metode.

Som i kan se når i tester rammer vores enemies ind i mellem Ninja og får den til at hoppe lidt igen før den dør. Det er vi ikke interesseret i. Vi gav vores enemy en egenskab, da vi oprettede den.

```
//Game.js

this.enemyContainer.alive = true;
```

Den vil vi nu gøre brug af for at sikre os at når en enemy dør, så dør den. Vi starter med at sætte vores alive egenskab til false, når de kolliderer og sammen med den detect vi laver en condition der siger at alive skal være true for at en kollision kan forekomme.

```
//Game.js

let ticker = PIXI.Ticker.shared;

      ticker.add((delta) => {

        if(this.enemy != undefined){

          this.enemy.enemies.forEach(_enemy => {



if (this.ht.checkme(this.ninja, _enemy.getChildAt(1) ) && _enemy.alive == true) {

                const currentEnenmySpriteSheet = _enemy.getChildAt(0);
                currentEnenmySpriteSheet.state.setAnimation(0, 'die', true);


               let enemyDieTimeline = gsap.timeline({
                onComplete: () =>{
                  this.scene.removeChild(_enemy);

                }
              });
              enemyDieTimeline.to(_enemy, {y: 300, duration: .7, ease: "Circ.easeOut"});
              enemyDieTimeline.to(_enemy, {y: 1200, duration: .5, ease: "Circ.easeIn"});


              _enemy.alive = false;

            }// END if
```

```
        })//END forEach
      } // END if


    }) // END ticker
```

Vi vil også gerne have lyd på når Ninja rammer en enemy. Der gør vi også brug af vores enemy egenskab alive, da lyden kun skal afvikles, hvis enemy egenskaben alive er lig true.

```
//Game.js
let ticker = PIXI.Ticker.shared;

      ticker.add((delta) => {

        if(this.enemy != undefined){

          this.enemy.enemies.forEach(_enemy => {


            if (this.ht.checkme(this.ninja, _enemy.getChildAt(1) ) && _enemy.alive == true) {

                const currentEnenmySpriteSheet = _enemy.getChildAt(0);
                currentEnenmySpriteSheet.state.setAnimation(0, 'die', true);

          if(_enemy.alive){
                  this.hitSound = new Howl({
                    src:['./assets/sound/effekt_hit.mp3'],
                    volume: 0.2
                    })

                    this.hitSound.play();

                }

               let enemyDieTimeline = gsap.timeline({
                onComplete: () =>{
                  this.scene.removeChild(_enemy);

                }

              });
              enemyDieTimeline.to(_enemy, {y: 300, duration: .7, ease: "Circ.easeOut"});
              enemyDieTimeline.to(_enemy, {y: 1200, duration: .5, ease: "Circ.easeIn"});

              _enemy.alive = false;
```

```
          }// END if

        })//END forEach
      } // END if
    }) // END ticker
```

Så kunne vi også smide en lyd på når Ninja hopper. Den smider vi bare på vores stage eventListeneren

```
//Game.js

this.ia = new Howl({
              src:['./assets/sound/ia1.mp3'],
              volume: 0.1
              })

              this.ia.play();
```

Og for sjov skyld kunne vi da også lave lidt random lyd. Opret et array i starten af din konstruktør

```
this.SoundArray = ["ia1", "ia2"];
```

Og ændre så i din Howl

```
 //Game.js

let getFromSoundArray = this.SoundArray[Math.floor(Math.random() * this.SoundArray.length)];

            this.ia = new Howl({
              src:['./assets/sound/' + getFromSoundArray + '.mp3'],
              volume: 0.1
              })

              this.ia.play();
```

Hit that hero

Der skal selvfølgelig også være mulighed for ar vores enemies kan ramme vores Ninja. Vi skal have lavet et hitarea, som vores enemies kan ramme. Vi laver det på den måde, at hvis en enemy kommer inden for et

område begynder den at angribe og rammer vores Ninja. Så vi starter med at oprette et stks. graphic bagved vores Ninja. Det gør vi Game klassen lige inde vores oprettelse af vores ninja Sprite.

```
        this.hitareaNinja = new PIXI.Graphics();
        this.hitareaNinja.beginFill(0xDE3249);
        this.hitareaNinja.drawRect(500-150, 550, 300, 200);//x, y, w, h
        this.hitareaNinja.alpha=0.5;
        this.hitareaNinja.endFill();
        this.scene.addChild(this.hitareaNinja);
```

Så skal vi havde lavet endnu en kollisionstest med vores HitTest klasse, så når en enemy rammer vores Ninja skal der ske noget. Den opretter vi også i vores Pixi.Ticker, inden for vores forEachs løkke og efter vores første hitTest. Vi skal også sætte en egenskab mere i vores enemy klasse. Desuden skal vi have lagt en ny egenskab på vores spine container i Enemy klassen.

```
if (this.ht.checkme(this.hitareaNinja, _enemy.getChildAt(1)) && _enemy.attack == true) {

              console.log("ninja hit");

            }
```

Og så sætte den nye egenskab på vores spritesheet i vores enemy klasse.

```
this.enemyContainer.attack = true;
```

Lad os teste om vores hitTest gør det den skal.

Vi kan lige så godt sætte den nye egenskab attack til false i begge hitTest, da vi jo ikke skal teste om den rammer hvis den allerede er død.

Når vores enemy kollidere med Ninja vil vi gerne have at vores enemy hopper lidt op og begynder at svinge sit sværd. Så først skal vi ændre vores animation fra ”walk” til ”attack” og så bruge vi en GSAP med en timeout, så vi har lidt styr over, hvornår vores Ninja reagere på at blive ramt. Så i vores hitTest statement. Starter vi med at finde vores enemys attakc animation

```
const currentEnenmySpriteSheetAttack = _enemy.getChildAt(0);
       currentEnenmySpriteSheetAttack.state.setAnimation(0, 'attack', true);
```

Derefter opretter vi vores timer og sætter nyt grafik ind, så vi virkelig kan se at han bliver ramt og bruger GSAP til at få vores ninja til at hoppe lidt.

```
 let timeToNinjaIsHurt =  setTimeout(() => {

    this.ninja.stop();
    this.ninja.texture = PIXI.Texture.from('../assets/images/ninja-hurt.png');

    gsap.to(this.ninja, {
    duration: 0.7,
    y:550,
    ease: "Circ.easeOut",
      onComplete: () => {
           this.ninja.play();

  gsap.to(this.ninja, {
      duration: 0.4,
      y:768-150
      })
}
})

},300)

_enemy.alive = false;
_enemy.attack = false;
```

Vi vil også gerne have voree enemies til at hoppe lidt, så uden for timeren kan vi lave lidt GSAp magic på vores enemies og når en enemy er førsig med at angribe, sætter vi animationen tilbage til ”walk”

```
 gsap.to(_enemy, {
            duration: 0.7,
            y:550,
            ease: "Circ.easeOut",
            onComplete: () => {

             gsap.to(_enemy, {
              duration: 0.5,
              y:768-50,
              ease: "Circ.easeOut"

             })

             currentEnenmySpriteSheetAttack.state.setAnimation(0, 'walk', true);

              }

              }) // END gsap
```

Til sidst kunne vi jo smide noget mere lyd på, når vores enemy hopper og vores Ninja bliver ramt. Men det må i selv klare.