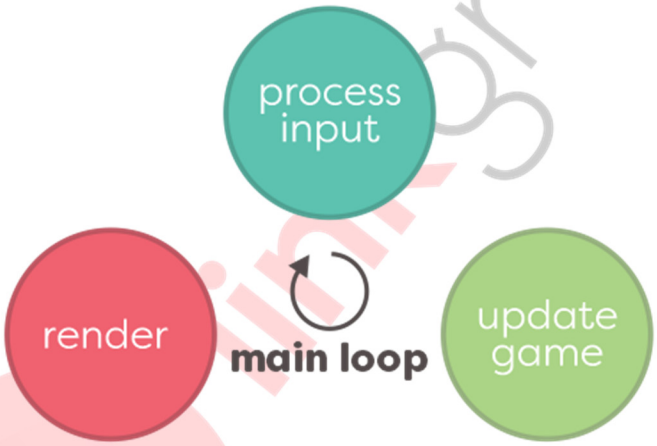
# Glavna petlja i pozadina igre

U procesu razvoja *Jump & Run* igre, u prethodnoj lekciji obavili smo prvo crtanje unutar canvas elementa. Reč je o gradijentu koji predstavlja najdalji sloj pozadine naše igre. Naš sledeći korak jeste realizacija ostalih slojeva pozadine. Svi oni će biti animirani, odnosno kretaće se zdesna nalevo kako bi simulirali kretanje glavnog junaka. Stoga će prvi korak u ovoj lekciji biti kreiranje glavne petlje, koja će ciklično da obavlja ažuriranje elemenata naše igre. Takav posao za nas nije potpuna nepoznanica. Ipak, sada će pojmu glavne petlje biti posvećena znatno veća pažnja.

## Šta je glavna petlja?

Glavna petlja animacije (engl. *main loop*) jeste logika koja konstantno, tokom izvršavanja igre, ponavlja operacije ažuriranja i crtanja, pri čemu uzima u obzir korisničku interakciju i na taj način predstavlja svojevrstan motor igre (slika 10.1).




*Slika 10.1. Glavna petlja*

Sa slike 10.1. možete videti osnovne korake koji se obavljaju tokom izvršavanje neke igre. Oni se ponavljaju sve dok je igra aktivna. Ovo praktično znači da je svaka igra na svom osnovnom nivou sastavljena iz tri operacije koje se *beskonačno* ponavljaju, a to su:

- obrada korisničke interakcije,
- ažuriranje igre,
- crtanje grafike.

Tokom izvršavanja bilo koje igre, tri navedene operacije ponove se hiljadama puta, sve dok je igra aktivna. Svako ponavljanje prikazanih operacija proizvodi jedan **kadar** (engl. *frame*), koji se crta na displeju uređaja. Izgled susednih kadrova uglavnom se veoma malo razlikuje, te se njihovim brzim smenjivanjem postiže efekat animacije. Logika koja opisane operacije konstantno ponavlja naziva se glavna petlja.

U dosadašnjem toku ovoga kursa mogli ste da vidite da je glavnu petlju moguće kreirati korišćenjem nekoliko tajming funkcija koje web pregledači nama izlažu na korišćenje. Ipak, u prethodnim lekcijama prikazani su osnovni pristupi za obavljanje takvog posla. Stoga ćemo se mi u nastavku posvetiti kreiranju nešto naprednije petlje animacije.

## requestAnimationFrame()

Iako je glavnu petlju animacije moguće postići korišćenjem bilo koje tajming funkcije, najčešća je praksa da se za obavljanje takvog posla koristi metoda `requestAnimationFrame()`. Tako nešto nije slučajno s obzirom na to da se korišćenjem ove metode tajming izvršavanja funkcije prepušta web pregledaču, koji nastoji da logiku prosleđenu ovoj funkciji izvršava onoliko puta u sekundi koliko je displej korisničkog uređaja sposoban da prikaže.

U prethodnim lekcijama je ova metoda za kreiranje glavne petlje iskorišćena na sledeći način:

```
function main() {
    window.requestAnimationFrame(main);

    //clear, update and draw
}

main();
```

Upravo kreirana funkcija `main()` jeste glavna petlja animacije. Ona trenutno poseduje samo jednu naredbu, kojom se obavlja pozivanje `requestAnimationFrame()` metode. Njoj se prosleđuje referenca na `main()` funkciju, čime se ona prosleđuje na ponovno izvršavanje neposredno pre narednog osvežavanja korisničkog okruženja. Umesto ostatka logike glavne petlje, unutar tela `main()` funkcije je postavljen komentar, koji ćemo u nastavku da zamenimo logikom za brisanje grafike iz `canvas` elementa, njeno ažuriranje i ponovno crtanje.

Poziv `main()` funkcije koji je naveden na kraju prikazanog primera pokreće glavnu petlju.

**requestAnimationFrame() je najbolje pozvati što pre unutar glavne petlje**

Ovako dolazimo i do prvog unapređenja u odnosu na do sada viđene glavne petlje. Poziv metode `requestAnimationFrame()` je naveden na samom početku `main()` funkcije. Naime, dobra praksa je da se web pregledaču što pre stavi do znanja da je i pre narednog osvežavanja potrebno da izvrši logiku naše glavne petlje. Zbog toga se poziv metode `requestAnimationFrame()` nalazi na samom vrhu funkcije `main()`.

**Brisanje nacrtane grafike**

Prvi korak u konstrukciji narednog kadra jeste brisanje kompletne grafike koja je nacrtana u prethodnom ciklusu. U prethodnim lekcijama mi smo to uglavnom obavljali pozivanjem `clearRect()` metode. Sada tako nešto možemo da obavimo crtanjem najdaljeg sloja pozadine naše igre – gradijenta koji predstavlja noćno nebo. Stoga ćemo logiku iz prethodne lekcije da upakujemo unutar jedne zasebne funkcije i da poziv takve funkcije postavimo unutar naše glavne petlje:

```
window.onload = init;
let ctx;
function init() {
    let canvas = document.getElementById("my-canvas");
    if (canvas.getContext) {
        ctx = canvas.getContext('2d');
        main();
    } else {
        alert("Canvas is not supported.");
    }
}
function main() {
    window.requestAnimationFrame(main);
    clearCanvas();
    //update and draw
}
function clearCanvas() {
    let linearGradient = ctx.createLinearGradient(ctx.canvas.width / 2, 0,
ctx.canvas.width / 2, ctx.canvas.height);
    linearGradient.addColorStop(0, '#0D0E20');
    linearGradient.addColorStop(0.5, '#26303E');
    linearGradient.addColorStop(1, '#445664');
    ctx.fillStyle = linearGradient;
    ctx.fillRect(0, 0, ctx.canvas.width, ctx.canvas.height);
}
```

Ovo je sada kompletan kod koji se nalazi unutar `game.js` fajla. Logika za crtanje pravougaonika ispunjenog gradijentom je prebačena unutar zasebne funkcije (`clearCanvas`), a poziv takve funkcije je postavljen unutar `main()` funkcije. Kako bi takvoj logici na raspolaganju bila referenca na kontekst za crtanje, promenljiva `ctx` je izmeštena na globalni nivo. Glavna petlja se pokreće iz `init()` funkcije pozivanjem funkcije `main()`.

Ukoliko nakon ovih izmena HTML dokument otvorite unutar web pregledača, moći ćete da vidite da se vizuelno ništa nije promenilo. Ipak, sada se u pozadini, približno 60 puta u sekundi, unutar `canvas` elementa obavlja crtanje pravougaonika sa gradijentom. To se ne može primetiti zato što u svakom ciklusu pravougaonik izgleda identično.

## Realizacija animiranih slojeva pozadine

Nakon prvog sloja pozadine koji je u potpunosti statičan, na redu je realizacija ostalih slojeva pozadine, koji se kreću zdesna nalevo i tako simuliraju kretanje glavnog junaka. Logika po kojoj će biti realizovani svi slojevi pozadine će biti identična.

Svi elementi pozadine, osim sloja noćnog neba koji je već realizovan, konstantno će se kretati zdesna nalevo. Pri tome će se brzine njihovog kretanja razlikovati kako bi se što realnije dočarao efekat kretanja. Naime, najbrže će se kretati oni slojevi pozadine koji su najbliži korisniku (zemlja, odnosno podloga). Logično, najmanju brzinu kretanja imaće zvezde.

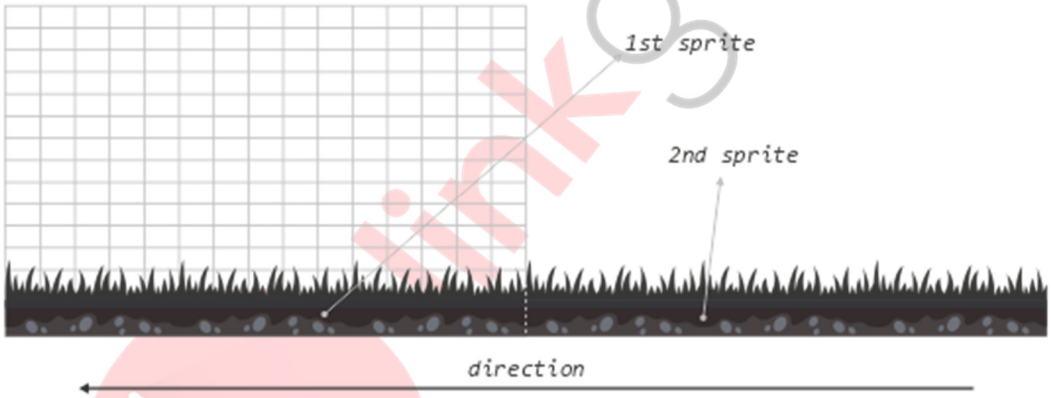
S obzirom na gotovo identične osobine svih slojeva koji će činiti pozadinu naše igre, kompletnu logiku takvih slojeva mi ćemo objediniti unutar klase `InfiniteScrollingBackground`. Reč je o klasi koja će sadržati osnovnu logiku kojom će se postići beskonačno kretanje pozadine (engl. *infinite scrolling background*).

Svaki sloj pozadine igre *Jump & Run* predstavljaće se korišćenjem spritesheeta sa dva sprajta.

**Sprite i spritesheet**

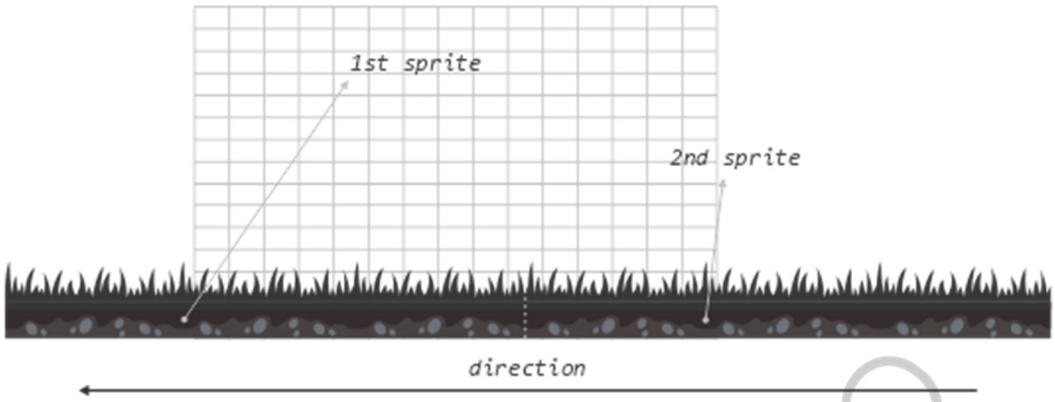
Sprajt se definiše kao dvodimenzionalna slika kojom se predstavlja jedan element igre. Grupa sprajtova se naziva spritesheet.

Kako biste bolje razumeli logiku koja će biti upotrebljena prilikom realizacije slojeva pozadine koji se beskonačno pomeraju zdesna nalevo, pogledajte sliku 10.2.



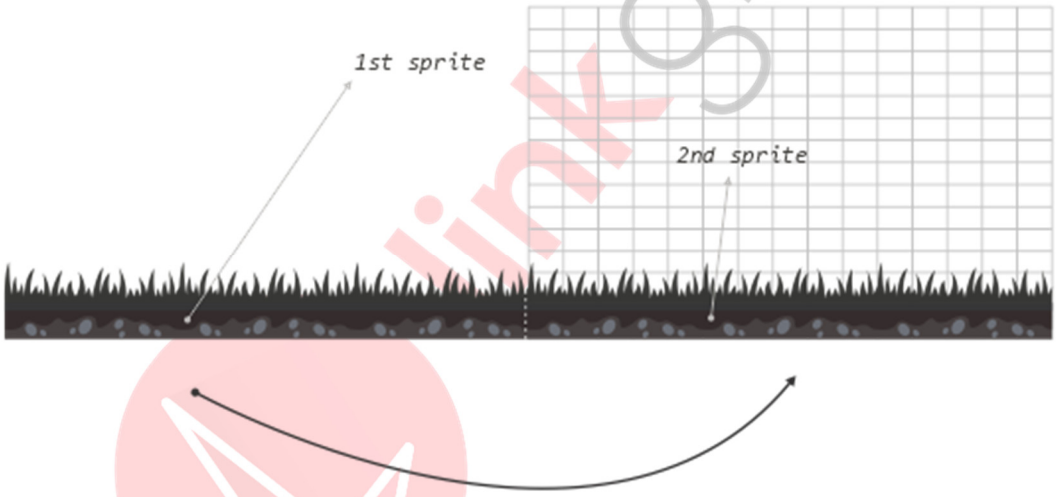

*Slika 10.2. Svaki sloj pozadine realizuje se korišćenjem spritesheeta sa dva sprajta*

Sa slike 10.2. možete videti dva sprajta koja se crtaju prilikom izvršavanja *Jump & Run* igre. Početak drugog sprajta postavljen je na kraj prvog, pri čemu je na slici granica između takvih sprajtova označena jednom uspravnom isprekidanom linijom. Mreža koju možete da vidite ilustruje granice `canvas` elementa. Strelica na dnu slike dočarava kretanje kompletne pozadine zdesna nalevo. Pogledajmo sada šta se dešava prilikom takvog kretanja (slika 10.3).



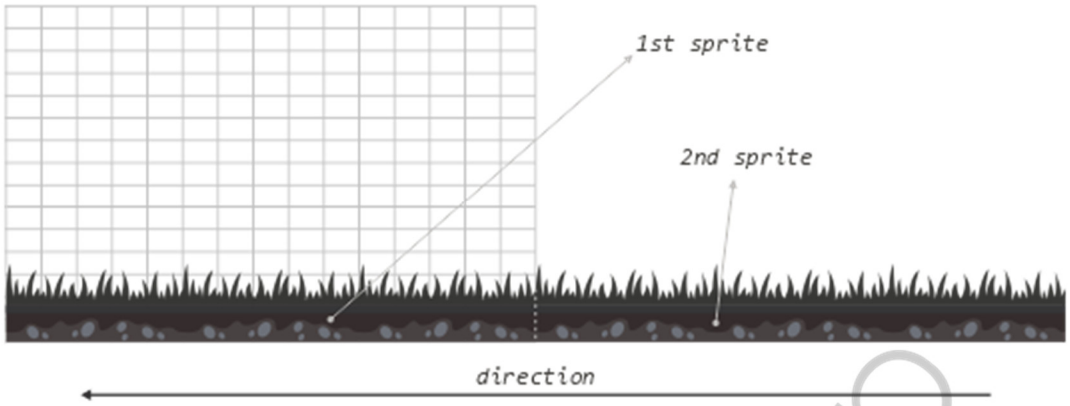

*Slika 10.3. Kretanje sprajtova koji predstavljaju pozadinu*

Slika 10.3. ilustruje kretanje pozadine igre *Jump & Run* zdesna nalevo. Možete videti da su u trenutku prikazanom slikom vidljiva oba sprajta od kojih je sačinjena podloga igre. Ipak, jasno je da će konstantnim kretanjem pozadine zdesna nalevo doći trenutak kada će prvi sprajt u potpunosti izaći iz okvira canvas elementa (slika 10.4).




*Slika 10.4. Osnovna logika promene lokacije sprajtova*

Slika 10.4. ilustruje situaciju u kojoj je prvi sprajt pozadine izašao iz okvira canvas elementa. U takvom trenutku u potpunosti je vidljiv samo drugi sprajt. Kako bi se postigao efekat beskonačnog pomeranja pozadine korišćenjem samo dva sprajta, u ovom trenutku se vrši vraćanje sprajtova na njihovu originalnu poziciju (slika 10.5).




*Slika 10.5. Sprajtovi se vraćaju na originalnu poziciju, a s obzirom na to da su identični, korisnik ne vidi nikakvu promenu*

Prikazanom logikom postiže se efekat pozadine koja se beskonačno pomera zdesna nalevo, i to korišćenjem samo dva sprajta za svaki od slojeva pozadine. Ključna osobina takvih sprajtova je da su u potpunosti identični. Stoga, kada se drugi sprajt zameni prvim, korisnik ne može da primeti nikakvu razliku.

Na upravo prikazanim slikama dočaran je samo jedan element pozadine – podloga po kojoj glavni lik trči. Svi ostali slojevi pozadine realizuju se na identičan način.

Slojeve pozadine, ali i sve ostale elemente od kojih treba da bude sačinjena naša igra, mi ćemo realizovati korišćenjem klasa, što je funkcionalnost koja je u JavaScript uvršćena sa pojavom ES5 verzije.

**JavaScript klase**

Unutar klasa enkapsuliraju se svojstva i ponašanja koja opisuju jedan tip objekata. Objekti se kreiraju na osnovu klasa, a na osnovu jedne klase se može kreirati proizvoljan broj objekata. Svi objekti kreirani korišćenjem iste klase poseduju identičan skup svojstava i metoda.

U JavaScriptu klase su specijalna vrsta funkcija. Ipak, umesto ključne reči `function`, za njihovo kreiranje koristi se ključna reč `class`:

```
class MyClass {
  ...
}
```

Nakon ključne reči `class`, navodi se naziv klase. Između vitičastih zagrada definiše se telo klase. Unutar tela klase mogu se naći klasni članovi – konstruktor i metode.

Konstruktor je specijalna metoda unutar klase, koja se poziva prilikom kreiranja objekata, a najčešće se koristi da postavi vrednosti svojstava objekta koji se kreira.

Jedna klasa može da nasledi neku drugu klasu. Ključna reč kojom se obavlja nasleđivanje između klasa u jeziku JavaScript jeste **extends**. Ova ključna reč navodi se između naziva klase koja nasleđuje i nasleđene klase:

```
class MyClass extends SomeOtherClass {
  ...
}
```

Pozivanje konstruktora roditeljske klase iz nasleđene klase može se obaviti korišćenjem ključne reči `super`:

```
class MyClass extends SomeOtherClass {
  constructor(param){
    super(param);
  }
}
```

`super` je ključna reč koja omogućava da se obavi pozivanje konstruktora ili pristup svojstvima i metodama roditeljske klase.

Klasa sa osnovnom logikom za realizaciju funkcionalnosti slojeva pozadine zvaće se `InfiniteScrollingBackground`:

```
import { Sprite } from './sprite.js';

export class InfiniteScrollingBackground {

    constructor(ctx) {
        this.ctx = ctx;
        this.gameWidth = ctx.canvas.width;
        this.gameHeight = ctx.canvas.height;
        this.horizontalOffset = 1;
        this.spritesheet;
        this.sprites = [new Sprite(), new Sprite()];
        this.spritesNo = 2;
    }

    load(loadingFinished) {

        this.spritesheet = new Image();

        this.spritesheet.addEventListener('load', () => {

            this.spriteHeight = this.spritesheet.height;
            this.spriteWidth = this.spritesheet.width / this.spritesNo;

            this.sprites[0].Source.x = 0;
            this.sprites[0].Source.y = 0;
            this.sprites[0].Source.width = this.spriteWidth;
            this.sprites[0].Source.height = this.spriteHeight;
```

```
            this.sprites[1].Source.x = this.spriteWidth;
            this.sprites[1].Source.y = 0;
            this.sprites[1].Source.width = this.spriteWidth;
            this.sprites[1].Source.height = this.spriteHeight;

            loadingFinished();

        });

        this.spritesheet.src = this.spritesheetUrl;

    }

    update() {

        this.horizontalOffset += this.velocity;

        if (this.horizontalOffset > this.gameWidth) {
            this.horizontalOffset = this.horizontalOffset -
this.gameWidth;
        }

        this.sprites[0].Destination.x = 0 - this.horizontalOffset;
        this.sprites[0].Destination.y = 0;
        this.sprites[0].Destination.width = this.gameWidth;
        this.sprites[0].Destination.height = this.gameHeight;

        this.sprites[1].Destination.x = this.gameWidth -
this.horizontalOffset;
        this.sprites[1].Destination.y = 0;
        this.sprites[1].Destination.width = this.gameWidth;
        this.sprites[1].Destination.height = this.gameHeight;

    }

    draw() {

        for (let i = 0; i < this.sprites.length; i++) {

            this.ctx.drawImage(this.spritesheet,
                this.sprites[i].Source.x,
                this.sprites[i].Source.y,
                this.sprites[i].Source.width,
                this.sprites[i].Source.height,
                this.sprites[i].Destination.x,
                this.sprites[i].Destination.y,
                this.sprites[i].Destination.width,
                this.sprites[i].Destination.height);

        }

    }
}
```

Unutar prikazane klase inkorporirano je sve ono o čemu je bilo reči u prethodnim redovima. Najznačajniji elementi ove klase i njihova namena su razjašnjeni u nastavku:

- `constructor()` – metoda unutar koje se obavlja inicijalizacija svih promenljivih koje će biti korišćene tokom funkcionisanja objekata ove klase,
- `load()` – metoda kojom se inicira učitavanje resursa; u našem primeru je to slika, odnosno spritesheet kojim će konkretan sloj pozadine da bude predstavljen unutar igre; bitno je primetiti da ova metoda prihvata i jedan parametar koji se odnosi na callback funkciju, koja će biti pozvana kada se učitavanje slike završi; na ovaj način se omogućava da glavna logika unutar `game.js` fajla dobije dojavu o završetku učitavanja slike,
- `update()` – metoda unutar koje se u svakoj iteraciji glavne petlje vrši ažuriranje osobina grafike; preciznije, vrši se ažuriranje pozicije na kojoj se crtaju sprajtovi koji predstavljaju sloj pozadine; ovu metodu ćemo nešto kasnije da pozivamo direktno iz glavne petlje animacije,
- `draw()` – još jedna metoda koja će se pozivati u svakoj iteraciji glavne petlje; unutar nje se nalazi logika za crtanje sprajtova unutar `canvas` elementa.

**Metoda drawImage()**

Osnova upravo prikazane logike jeste metoda `drawImage()` koja omogućava crtanje jednog isečka slike:

```
drawImage(image, sx, sy, sWidth, sHeight, dx, dy, dWidth, dHeight)
```

Uloga prikazanih parametara je sledeća:

- `image` – slika čiji će isečak biti nacrtan,
- `sx` – x-koordinata referentne tačke isečka slike unutar izvorne slike,
- `sy` – y-koordinata referentne tačke isečka slike unutar izvorne slike,
- `sWidth` – širina isečka,
- `sHeight` – visina isečka,
- `dx` – x-koordinata referentne tačke isečka unutar `canvas` elementa,
- `dy` – y-koordinata referentne tačke isečka unutar `canvas` elementa,
- `dWidth` – širina koju će isečak zauzimati unutar `canvas` elementa,
- `dHeight` – visina koju će isečak zauzimati unutar `canvas` elementa.

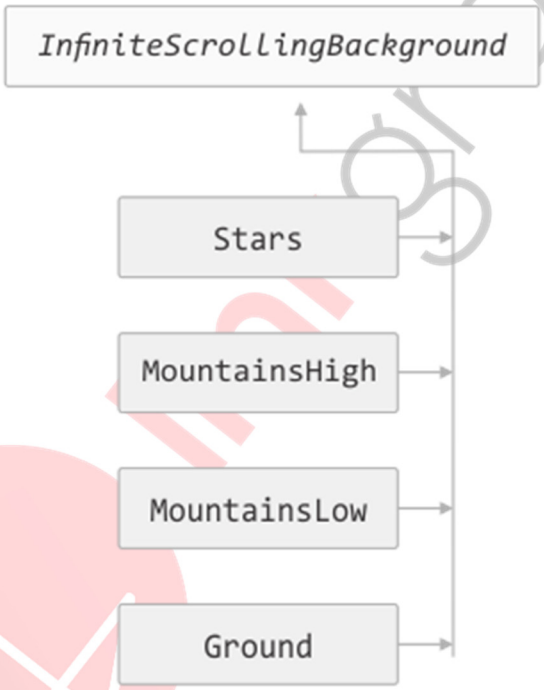
Klasa `InfiniteScrollingBackground` koristi i objekte još jedne klase. Reč je o klasi za enkapsulaciju osobina jednog sprajta:

```
export class Sprite {

    constructor() {

        this.Source = {
            x: 0,
            y: 0,
            width: 0,
            height: 0
```

```
            }

            this.Destination = {
                x: 0,
                y: 0,
                width: 0,
                height: 0
            }
        }
    }
```

Unutar klase `Sprite` nalaze se svojstva koja predstavljaju dva objekta sa po četiri parametra. Ovo je samo olakšica za predstavljanje parametara koji se prosleđuju metodi `drawImage()`.

Klasa `InfiniteScrollingBackground` nije zamišljena za direktno instanciranje, već nju treba da naslede konkretne klase koje će predstavljati slojeve pozadine (slika 10.6).




*Slika 10.6. Klase koje se koriste za realizaciju slojeva pozadine*

U nastavku će biti prikazano kako će izgledati upravo navedene klase koje će predstavljati pojedinačne slojeve pozadine.

**Klasa Stars:**

```
import { InfiniteScrollingBackground } from
'./infiniteScrollingBackground.js';

export class Stars extends InfiniteScrollingBackground {
```

```
        constructor(ctx) {

            super(ctx);

            this.velocity = 1;
            this.spritesheetUrl = 'spritesheets/stars.png';
        }

    }
```

**Klasa MountainsHigh:**

```
    import { InfiniteScrollingBackground } from
    './infiniteScrollingBackground.js';

    export class MountainsHigh extends InfiniteScrollingBackground {


        constructor(ctx) {

            super(ctx);

            this.velocity = 1.05;
            this.spritesheetUrl = 'spritesheets/mountains_high.png';

        }

    }
```

**Klasa MountainsLow:**

```
    import { InfiniteScrollingBackground } from
    './infiniteScrollingBackground.js';

    export class MountainsLow extends InfiniteScrollingBackground {


        constructor(ctx) {

            super(ctx);

            this.velocity = 1.2;
            this.spritesheetUrl = 'spritesheets/mountains_low.png';
        }

    }
```

**Klasa Ground:**

```
import { InfiniteScrollingBackground } from
'./infiniteScrollingBackground.js';

export class Ground extends InfiniteScrollingBackground {


    constructor(ctx) {

        super(ctx);

        this.velocity = 2;
        this.spritesheetUrl = 'spritesheets/ground.png';
    }

}
```

Sada možete da vidite kolika ušteda je napravljena enkapsulacijom funkcionalnosti pokretnih pozadina unutar jedne roditeljske klase. Unutar konkretnih klasa nalazi se samo konstruktor unutar koga se definišu dva podatka: brzina sloja pozadine i lokacija na kojoj se nalazi spritesheet. Sve ostalo se nalazi unutar klase `InfiniteScrollingBackground`.

**JavaScript moduli**

Do sada ste mogli da vidite da kod koji smo pisali koristi funkcionalnosti ES6 modula. Svaka klasa se piše u zasebnom fajlu, a zatim se kao modul uvozi unutar drugih fajlova u kojima se javi potreba za njihovim korišćenjem.

ES6 moduli funkcionišu po vrlo jednostavnom principu:

- svaki JavaScript fajl je zaseban modul,
- dve osnovne operacije prilikom korišćenja modula su uvoz (engl. *import*) i izvoz (engl. *export*),
- jedan modul može da omogući izvoz proizvoljnog broja entiteta,
- jedan modul može da uveze proizvoljan broj entiteta iz drugih modula.

**Eksportovanje elemenata iz jednog modula**

Sve promenljive i funkcije definisane unutar jednog modula privatne su i vidljive samo unutar takvog modula. Kako bismo ih učinili vidljivim i izvan modula, neophodno je da ih izvezemo. To se postiže korišćenjem JavaScript naredbe **export**:

```
export var variable1 = 1234;
var variable2 = 56789;
export function sayHello(name) {
  //...
}
```

Prikazani kod ilustruje sadržaj jednog `.js` fajla. Samim tim, reč je o jednom JavaScript modulu. Unutar takvog modula nalaze se dve promenljive i jedna funkcija. Promenljiva `variable1` i funkcija `sayHello()` obeležene su naredbom `export`. Na taj način su one izvezene iz modula, odnosno omogućen je njihov uvoz u neke druge module. Ipak, promenljivu `variable2` neće biti moguće uvesti u neki drugi modul zato što prethodno nije izvezena iz modula u kojem je definisana.

**Objedinjeno eksportovanje većeg broja elemenata**

Izvoz je moguće obaviti i objedinjeno, na kraju jednog dokumenta:

```
export { variable1, sayHello };
```

**Importovanje elemenata iz drugih modula**

Kako bi se ovakvi entiteti uvezli unutar nekog drugog modula (fajla), koristi se naredba **import**, na sledeći način:

```
import { variable1, sayHello } from 'app.js';
```

Ovakva `import` naredba funkcionisaće ukoliko je prethodno prikazani kod smešten unutar fajla sa nazivom `app.js`.

Entitete je moguće importovati i pojedinačno:

```
import variable1 from 'app.js';
```

**Imenovanje elemenata prilikom importovanja**

Prilikom importovanja, elementima koji se uvoze moguće je dati i neko posebno ime, koje će se koristiti u tekućem modulu:

```
import variable1 as Variable from 'app.js';
```

Na ovaj način je uvezen element sa nazivom `variable1`, kojim će se u tekućem dokumentu rukovati korišćenjem naziva `Variable`.

**Importovanje svih elemenata odjednom**

Moguće je obaviti uvoz svih entiteta koji su izvezeni u nekom modulu, korišćenjem sledećeg pristupa:

```
import * as App from 'app.js';
```

Karakter zvezdica se odnosi na sve eksportovane elemente jednog modula. Tako je prikazanom naredbom rečeno da će iz fajla `app.js` biti uvezeni svi eksportovani elementi i da će oni biti dostupni korišćenjem naziva `App`. Stoga će biti moguće napisati nešto ovako:

```
App.sayHello('Ben');
```

**Podrazumevani izvoz i uvoz**

Na kraju, postoji mogućnost da se unutar jednog modula jedan element obeleži kao podrazumevani element za izvoz:

```
function sayHello(name) {
  //...
}
export default sayHello;
```

Na ovaj način otvara se mogućnost uvoza bez definisanja naziva elementa koji želimo da uvezemo:

```
import Hello from 'app.js';
```

`Hello` se odnosi na naziv koji je podrazumevano izvezenom elementu dodeljen unutar modula u koji se uvozi.

**Uključivanje skripte koja predstavlja modul u HTML dokument**

Na kraju, ukoliko je neophodno da neku skriptu koja predstavlja modul uključite unutar HTML dokumenta, potrebno je iskoristiti atribut **type="module"**. U našem projektu je to potrebno obaviti unutar `index.html` fajla, prilikom uključivanja `game.js` fajla:

```
<script type="module" src="js/game.js"></script>
```

Kako bi se iskoristila funkcionalnost modula, fajlove je neophodno hostovati na nekom HTTP serveru.

Upravo kreirane klase biće iskorišćene unutar `game.js` fajla na sledeći način:

```
import { Ground } from './ground.js';
import { Stars } from './stars.js';
import { MountainsLow } from './mountainsLow.js';
import { MountainsHigh } from './mountainsHigh.js';

window.onload = init;

let ctx;


let stars;
let mountainsHigh;
let mountainsLow;
let ground;

function init() {
    let canvas = document.getElementById("my-canvas");
```

```
    if (canvas.getContext) {
        ctx = canvas.getContext('2d');

        stars = new Stars(ctx);
        stars.load(loaded);

        mountainsHigh = new MountainsHigh(ctx);
        mountainsHigh.load(loaded);

        mountainsLow = new MountainsLow(ctx);
        mountainsLow.load(loaded);

        ground = new Ground(ctx);
        ground.load(loaded);

    } else {
        alert("Canvas is not supported.");
    }
}

let loaderCounter = 0;

function loaded() {

    loaderCounter++;

    if (loaderCounter < 4)
        return;

    main();
}


function main() {
    window.requestAnimationFrame(main);

    clearCanvas();
    update();
    draw();

}

function clearCanvas() {
    let linearGradient = ctx.createLinearGradient(ctx.canvas.width / 2,
0, ctx.canvas.width / 2, ctx.canvas.height);
    linearGradient.addColorStop(0, '#0D0E20');
    linearGradient.addColorStop(0.5, '#26303E');
    linearGradient.addColorStop(1, '#445664');
    ctx.fillStyle = linearGradient;
    ctx.fillRect(0, 0, ctx.canvas.width, ctx.canvas.height);
}
```

```
function update() {
    stars.update();
    mountainsHigh.update();
    mountainsLow.update();
    ground.update();
}


function draw() {
    stars.draw();
    mountainsHigh.draw();
    mountainsLow.draw();
    ground.draw();
}
```

Na fajlu `game.js` učinjene su sledeće izmene:

- objekti koji predstavljaju slojeve pozadine kreirani su unutar `init()` funkcije i tom prilikom je obavljen poziv `load()` metode svakog kreiranog objekta,
- prilikom svakog poziva metode `load()`, njoj je prosleđena referenca na istu metodu – `loaded()`; to znači da će se metoda `loaded()` pozivati svaki put kada se pojedinačni spritesheet učita,
- logika za pokretanje petlje, koja je do sada bila unutar metode `init()`, sada je prebačena unutar metode `loaded()`,
- metoda `loaded()` poseduje logiku kojom se proverava da li je obavljeno učitavanje svih resursa; to se postiže proverom broja pozivanja metode `loaded()`,
- unutar metode glavne petlje, odnosno unutar metode `main()`, sada se pozivaju dve dodatne metode – `update()` i `draw()`,
- unutar metode `update()` pozivaju se `update()` metode svih objekata koji predstavljaju slojeve pozadine,
- unutar metode `draw()` pozivaju se metode `draw()` svih objekata koji predstavljaju slojeve pozadine.

Sve ovo će rezultovati prikazom kao na animaciji 10.1.


*Animacija 10.1. Animacija pozadine Jump & Run igre*

## Brzina smenjivanja kadrova (Frame rate)

Već više puta do sada je spomenut pojam brzine smenjivanja kadrova (engl. *frames per second*). Rečeno je da se u idealnim uslovim funkcija koja predstavlja parametar metode `requestAnimationFrame()` poziva približno 60 puta u sekundi, što stvara frekvenciju osvežavanja od 60 FPS. U narednim redovima ćemo po prvi put moći da vidimo da li je to zaista tako i kolika je zapravo brzina petlje koju smo kreirali u ovoj lekciji.

Kako bi se izračunala frekvencija osvežavanja, neophodno je znati koliko vremena protekne između dva susedna poziva funkcije koja predstavlja glavnu petlju. U obavljanju takvog posla pomoći će nam sama `requestAnimationFrame()` metoda.

Naime, callback funkcija koja se prosleđuje `requestAnimationFrame()` metodi automatski dobija jedan parametar, koji predstavlja vreme proteklo od početka epohe. Takav parametar se može koristiti kako bi se izračunalo vreme koje je proteklo između dva sukcesivna izvršavanja glavne petlje:

```
let delta;
let previousTime;
function loaded() {
    ...
    previousTime = performance.now();
    window.requestAnimationFrame(main);
}
function main(currentTime) {
    window.requestAnimationFrame(main);
    delta = currentTime - previousTime;
    clearCanvas();
    update();
    draw();

    previousTime = currentTime;
}
```

Pored definisanja parametra `currentTime` koji sada prihvata metoda `main()`, na kodu je napravljeno još nekoliko izmena:

- definisane su dve globalne promenljive: `delta` – vreme proteklo između dve iteracije, i `previousTime` – vreme prethodne iteracije,
- inicijalna vrednost promenljive `previousTime` se dobija pozivom metode `performance.now()`,
- kod kojim započinje izvršavanje glavne petlje je izmenjen, pa se tako umesto direktnog pozivanja `main()` metode ona prosleđuje `requestAnimationFrame()` metodi; time se osigurava da će i prva iteracija da dobije vrednost ulaznog parametra `currentTime`,
- unutar `main()` metode obavlja se oduzimanje prethodnog vremena od trenutnog i time se računa period između dve iteracije,
- na kraju `main()` metode vrednost promenljive `currentTime` dodeljuje se promenljivoj `previousTime` i na taj način je sve spremno za sledeću iteraciju.

S obzirom na to da smo u prethodnim redovima napisali logiku za računanje perioda između susednih iteracija glavne petlje (`delta`), frekvenciju osvežavanja je moguće izračunati vrlo lako:

```
let fps = parseInt(1000 / delta);
```

Kako bismo lako mogli da pratimo frekvenciju osvežavanja animacije, unutar HTML dokumenta će biti postavljen još jedan element, unutar koga će se ispisivati podaci o brzini animacije:

```
<div id="game-stats">FPS: <span id="fps"></span></div>
```

Ovakav element dodajte na kraj `body`a, pre `script` elementa. On će biti stilizovan na sledeći način:

```
#game-stats {
    color: white;
}
```

Na kraju, evo i funkcije u kojoj će se računati i prikazivati FPS:

```
function showFps() {
    let fps;
    let fpsSpan = document.getElementById("fps");

    if (delta != 0) {
        fps = parseInt(1000 / delta);
    }
    fpsSpan.innerHTML = fps;
}
```

Funkciju je potrebno pozvati unutar glavne petlje i to će za efekat imati ispisivanje frekvencije osvežavanja u gornjem, levom uglu HTML dokumenta.

## Vremenski bazirana animacija

Računanje frekvencije osvežavanja animacije u prethodnim redovima nismo implementirali samo kako bismo takav podatak mogli da prikažemo na vrhu HTML dokumenta. Frekvenciju osvežavanja ćemo u nastavku koristiti za obavljanje jednog mnogo značajnijeg posla – za postizanje vremenski bazirane animacije.

Vremenski bazirana animacija jeste ona animacija čiji progres zavisi od vremena, a ne od brzine smenjivanja kadrova u jednoj sekundi. Naime, u prethodnim redovima mogli ste da vidite da je naša animacija osvežavana približno 60 puta u jednoj sekundi. Ipak, tako nešto kod vas ne mora biti slučaj.

Naime, ukoliko bismo prikazani kod pokrenuli na nekom uređaju sa displejem koji je sposoban da prikaže 120 kadrova po sekundi, naša animacija bi imala dva puta veću brzinu osvežavanja. Takođe, u nekim situacijama uređaj na kome se animacija izvršava možda neće biti sposoban da generiše 60 kadrova animacije u sekundi, pa će frekvencija osvežavanja biti 40, 30 ili manje kadrova po sekundi. Sve ovo praktično znači da frekvencija osvežavanja animacije zavisi od osobina hardverskih komponenata uređaja na kome se izvršava.

Loša stvar je ta što smo mi u dosadašnjem toku ove lekcije brzinu kretanja slojeva pozadine koncipirali na osnovu brzine izvršavanja glavne petlje, odnosno na osnovu broja kadrova koji se generišu po sekundi. To na kraju znači da brzina kretanja elemenata u našoj igri neće biti identična na svim uređajima i u svim situacijama, već da će zavisiti od broja kadrova koji je hardver uređaja u stanju da generiše.

Prilikom kreiranja animacije unutar igara, potrebno je težiti postizanju vremenski bazirane animacije. To je animacija kod koje brzina kretanja elemenata igre zavisi od vremena, a ne od frekvencije osvežavanja, odnosno broja kadrova koji se generišu u jednoj sekundi.

Prvi korak u realizaciji opisanog ponašanja biće kreiranje promenljive unutar koje će da bude smeštena brzina kojom je potrebno sprovoditi našu animaciju. Vrednost takve promenljive će da bude računata na sledeći način:

```
let speed;
let speedFactor = 1;
let BASE_SPEED = 16;

function main(currentTime) {
    window.requestAnimationFrame(main);

    delta = parseInt(currentTime - previousTime);
    speed = speedFactor * delta / BASE_SPEED;

    clearCanvas();
    update();
    draw();
    showFps();

    previousTime = currentTime;
}
```

Vrednost promenljive `speed` računa se unutar metode koja predstavlja glavnu petlju. Za računanje brzine animacije koja je zavisna od proteklog vremena koristi se već viđena promenljiva `delta` (vreme proteklo između dve iteracije petlje), ali i po jedna promenljiva i konstanta:

- `speedFactor` – promenljiva čija je vrednost 1, a koju ćemo nešto kasnije da koristimo za globalno ubrzavanje ili usporavanje kompletne igre,
- `BASE_SPEED` – bazna brzina, odnosno referentna vrednost razmaka između dve iteracije; za vrednost je postavljeno 16; takva vrednost nije odabrana slučajno; naime, vrednost od 16 ms jeste vreme koje protekne između dva kadra kada je frekvencija osvežavanja 60 FPS; to je frekvencija osvežavanja koju smo mi uzeli kao baznu.

Ukoliko je frekvencija osvežavanja animacije 60 FPS, vrednost promenljive `speed` će biti 1:

```
speed = (1*16)/16 = 1
```

Kada hardver klijentskog uređaja nije sposoban da generiše 60 kadrova u jednoj sekundi, vrednost promenljive `speed` će biti veća od 1. Na primer:

```
speed = (1*32)/16 = 2
```

Ovo je izraz koji podrazumeva frekvenciju osvežavanja od 30 FPS. Kao što vidite, tada će promenljiva `speed` da ima vrednost 2. To na kraju znači da će biti potrebno da naše elemente igre u svakom kadru pomeramo za duplo veću distancu kako bismo kompenzovali duplo manji broj kadrova po sekundi.

Vrednost promenljive `speed` je potrebno prosleđivati `update()` metodi svakog objekta koji predstavlja sloj pozadine:

```
function update() {
    stars.update(speed);
    mountainsHigh.update(speed);
    mountainsLow.update(speed);
    ground.update(speed);
}
```

Zatim će vrednost brzine ovi objekti da koriste na sledeći način:

```
this.horizontalOffset += this.velocity * speed;
```

Prilikom računanja distance za koju je potrebno pomeriti sloj pozadine, sada se u obzir uzima i prosleđena vrednost promenljive `speed`. Tako će na kraju pozadina biti pomerana za veći broj piksela što je frekvencija osvežavanja manja, čime se osigurava identična brzina animacije u svim situacijama.

Brzinu animacije ćemo dodati i unutar `div` elementa sa globalnim podacima o izvršavanju naše igre.

*index.html:*

```
<div id="game-stats">FPS: <span id="fps"></span> Speed: <span
id="speed"></span></div>
```

*game.js:*

```
function showStats() {
    let fpsSpan = document.getElementById("fps");
    let speedSpan = document.getElementById("speed");
    let fps;
    if (delta != 0) {
        fps = parseInt(1000 / delta);
    }
    fpsSpan.innerHTML = fps;
    speedSpan.innerHTML = speed;
}
```

## Realizacija postepenog ubrzavanja igre

U okviru uvodnog izlaganja o osobinama *Jump & Run* igre rečeno je da će se brzina igre linearno povećavati kako bi, kako vreme protiče, postajala sve komplikovanija za igrača. Takva osobina će biti realizovana korišćenjem upravo prikazane promenljive `speedFactor`. Naime, postepenim uvećavanjem vrednosti ove promenljive povećavaće se i vrednost promenljive `speed`. Mogli ste da vidite da vrednost promenljive `speed` dobijaju svi elementi igre, pa je stoga jasno da će se intervencijom nad `speedFactor` promenljivom ubrzavati svi elementi igre.

Sve što je potrebno uraditi kako bi se realizovalo postepeno ubrzavanje igre jeste dodati sledeću logiku na kraj glavne petlje:

```
function main(currentTime) {
    window.requestAnimationFrame(main);

    delta = parseInt(currentTime - previousTime);
    speed = speedFactor * delta / BASE_SPEED;

    clearCanvas();
    update();
    draw();
    showStats();

    previousTime = currentTime;

    if (speed < 8) {
        speedFactor += 0.001;
    }

}
```

Na kraj glavne petlje postavljena je naredba kojom se u svakom ciklusu vrednost promenljive `speedFactor` uvećava za `0.001`. Takođe, kako se igra ne bi ubrzavala do beskonačnosti, te kako u nekom trenutku ne bi postala neupotrebljiva, dodat je i uslovni blok, kojim se proverava da je vrednost promenljive `speed` manja od 8. To praktično znači da će maksimalna brzina igre biti 8. Naravno, vi taj podatak možete menjati po sopstvenom nahođenju.

**Pitanje**

Brzina kretanja grafičkih elemenata igre treba da zavisi od:

a) frekvencije osvežavanja,
**b) proteklog vremena,**
c) jačine hardvera korisničkog uređaja,
d) sposobnosti displeja.

***Objašnjenje:***

*Prilikom kreiranja animacije unutar igara, potrebno je težiti postizanju vremenski bazirane animacije. To je animacija kod koje brzina kretanja elemenata igre zavisi od vremena, a ne od frekvencije osvežavanja, odnosno broja kadrova koji se generišu u jednoj sekundi.*

## Rezime

- Glavna petlja jeste logika koja konstantno, tokom izvršavanja igre, ponavlja operacije ažuriranja i crtanja, te na taj način predstavlja svojevrstan motor igre.
- Jedno izvršavanje petlje igre proizvodi jedan kadar (*frame*).
- Kreiranje glavne petlje podrazumeva upotrebu `requestAnimationFrame()` metode.
- `requestAnimationFrame()` poziva prosleđenu funkciju neposredno pre narednog osvežavanja korisničkog okruženja, odnosno po jednom za svaki ciklus osvežavanja.
- `requestAnimationFrame()` je najbolje pozvati što pre unutar glavne petlje.
- Prvi korak u konstrukciji narednog kadra jeste brisanje kompletne grafike koja je nacrtana u prethodnom ciklusu.
- Svi animirani slojevi pozadine *Jump & Run* igre beskonačno će se kretati zdesna nalevo i tako simulirati trčanje glavnog karaktera.
- Svi slojevi pozadine *Jump & Run* igre realizovani su korišćenjem sprajtova.
- Svaki sloj pozadine predstavlja se korišćenjem dva identična sprajta, koja su objedinjena unutar jednog spritesheeta.
- Osnovna metoda za rad sa sprajtovima jeste metoda `drawImage()` sa 9 parametara, koja omogućava crtanje jednog isečka slike.
- Frekvencija osvežavanja animacije izražava se brojem proizvedenih kadrova u sekundi (*frames per second, FPS)* .
- Callback funkcija koja se prosleđuje `requestAnimationFrame()` metodi automatski dobija jedan parametar, koji predstavlja vreme proteklo od početka epohe; takav parametar se može koristiti kako bi se izračunalo vreme koje je proteklo između dva sukcesivna izvršavanja glavne petlje, pa samim tim i za računanje frekvencije osvežavanja.
- Vremenski bazirana animacija jeste ona animacija čiji progres zavisi od vremena, a ne od brzine smenjivanja kadrova u jednoj sekundi.