Glavni junak i protivnici, obrada korisničke interakcije

U prethodnim lekcijama ovoga modula u potpunosti je realizovan jedan deo *Jump & Run* igre. Reč je o pozadini koja je u konstantnom kretanju sa desna na levo. U ovoj lekciji realizovaćemo preostale elemente, pa ćemo tako na kraju lekcije dobiti potpuno funkcionalnu igru. Stoga će u ovoj lekciji prvo biti prikazana realizacija glavnog karaktera igre. Nakon realizacije glavnog karaktera biće prikazano kako obraditi korisničku interakciju, odnosno kako učiniti da pritiskom tastera space, glavni lik igre učini skok. Nakon realizacije glavnoj junaka, biće kreirani i protivnici, a posle toga i sistem koji će omogućiti detekciju kolizije između glavnoj junaka i protivnika. Razvoj *Jump &* Run igre zaokružićemo dodavanjem zvučnih efekata, propratnih poruka i sistema za prikaz ostvarenog rezultata (*score, engl.*).

Kreiranje glavnog junaka

Logika za kreiranja glavnog junaka *Jump & Run* igre, sasvim razumljivo, jeste drugačija od one koja je prikazana u prethodnom poglavlju. Glavni junak naše igre se ne pomera po horizontalnoj osi, već se njegovo kretanje simulira pomeranjem pozadine. Sa druge strane, glavni karakter se može pomerati po vertikalnoj osi i to tokom skoka.

Glavni junak *Jump & Run* igre, može se naći u nekoliko različitih stanja, koja će diktirati i različitu grafiku kojom će on biti predstavljen. Takva različita stanja su sledeća:

- mirovanje (STANDING)
- trčanje (RUNNING)
- skakanje (JUMPING)

Pre nego što korisnik pokrene igru, naš glavni junak treba da bude u stanju mirovanja (STANDING). Takođe, kada dođe do kontakta sa nekim od neprijatelja, glavnog junaka je potrebno zaustaviti, odnosno prebaciti u stanje STANDING. Pored stanja mirovanja, glavni junak može biti u stanju trčanja i skakanja. Sva tri navedena stanja predstavljaju se korišćenjem nekoliko sprajtova (slika 11.1).

Slika 11.1 – Grupa sprajtova za predstavljanje glavnog junaka Jump & Run igre

Slika 11.1 predstavlja jedan spritesheet, sastavljen od ukupno 9 sprajtova (slika 11.2).

Slika 11.2– Grupa sprajtova podeljena na pojedinačne sprajtove

Na slici 11.2 prikazane su granice pojedinačnih sprajtova unutar spritesheet-a. Prva dva sprajta koriste se za dočaravanje stanja mirovanja. Narednih 5 sprajtova koriste se za stanje trčanja, dok se poslednja dva sprajta upotrebljavaju prilikom skoka.

Glavni junak igre, biće modelovan korišćenjem klase `Runner`:

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

export class Runner {

    constructor(ctx) {
        this.ctx = ctx;
        this.spritesheetUrl = 'spritesheets/runner.png';
        this.gameWidth = ctx.canvas.width;
        this.gameHeight = ctx.canvas.height;

        this.velocity = 0.1;

        this.NO_SPRITES = 9;
        this.currentSpriteIndex = 0;

        this.RUNNER_STATE = {
            STANDING: 'standing',
            RUNNING: 'running',
            JUMPING: 'jumping'
        }

        this.Sprite = new Sprite();

        this.runnerState = this.RUNNER_STATE.RUNNING;

        this.JUMP_DURATION = 80;
        this.jumpState = 0;
    }


    load(loadingFinished) {

        this.spritesheet = new Image();

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

            this.spriteHeight = this.spritesheet.height;
            this.spriteWidth = this.spritesheet.width / this.NO_SPRITES;
```

```
                this.initialY = this.gameHeight - this.spriteHeight -
parseInt(this.gameHeight * 0.12);
                this.jumpY = this.initialY;
                this.jumpLimit = this.spriteHeight * 1.2;

                loadingFinished();

            });

            this.spritesheet.src = this.spritesheetUrl;

        }


        update(speed) {

            speed = 3 + speed - 3 * Math.sqrt(speed);

            switch (this.runnerState) {

                case this.RUNNER_STATE.STANDING:

                    //cycle trough 2 sprites
                    this.currentSpriteIndex = this.currentSpriteIndex +
(this.velocity * speed);

                    if (this.currentSpriteIndex >= 2) {
                        this.currentSpriteIndex = 0;
                    }

                    //set running sprite
                    this.Sprite.Source.x =
Math.floor(this.currentSpriteIndex) * this.spriteWidth;
                    this.Sprite.Source.y = 0;
                    this.Sprite.Source.width = this.spriteWidth;
                    this.Sprite.Source.height = this.spriteHeight;

                    //set sprite position on screen
                    this.Sprite.Destination.x = 25;
                    this.Sprite.Destination.y = this.initialY;
                    this.Sprite.Destination.width = this.spriteWidth;
                    this.Sprite.Destination.height = this.spriteHeight;

                    break;
                case this.RUNNER_STATE.RUNNING:

                    //cycle trough 4 sprites
                    this.currentSpriteIndex = this.currentSpriteIndex +
(this.velocity * speed);

                    if (this.currentSpriteIndex > 5) {
                        this.currentSpriteIndex = 2;
                    }

                    //set running sprite
```

```
                    this.Sprite.Source.x =
Math.floor(this.currentSpriteIndex) * this.spriteWidth;
                    this.Sprite.Source.y = 0;
                    this.Sprite.Source.width = this.spriteWidth;
                    this.Sprite.Source.height = this.spriteHeight;

                    //set sprite position on screen
                    this.Sprite.Destination.x = 25;
                    this.Sprite.Destination.y = this.initialY;
                    this.Sprite.Destination.width = this.spriteWidth;
                    this.Sprite.Destination.height = this.spriteHeight;

                    break;
                case this.RUNNER_STATE.JUMPING:

                    //raise The Runner
                    if (this.jumpState <= this.JUMP_DURATION / 3) {

                        this.currentSpriteIndex = 7;

                        this.Sprite.Source.x =
Math.floor(this.currentSpriteIndex) * this.spriteWidth;
                        this.Sprite.Source.y = 0;
                        this.Sprite.Source.width = this.spriteWidth;
                        this.Sprite.Source.height = this.spriteHeight;

                        this.jumpY = this.jumpY - Math.abs(this.jumpLimit *
(1 - (this.JUMP_DURATION / 3 - this.jumpState) / (this.JUMP_DURATION / 3)));

                        if (this.jumpY < this.initialY - this.jumpLimit) {
                            this.jumpY = this.initialY - this.jumpLimit;
                        }


                        //raise up the Runner
                        this.Sprite.Destination.x = 25;
                        this.Sprite.Destination.y = this.jumpY;
                        this.Sprite.Destination.width = this.spriteWidth;
                        this.Sprite.Destination.height = this.spriteHeight;


                        //hold
                    } else if (this.jumpState > this.JUMP_DURATION / 3 &&
this.jumpState < this.JUMP_DURATION / 3 * 2) {


                        this.jumpY = this.initialY - this.jumpLimit;
                        this.Sprite.Destination.x = 25;
                        this.Sprite.Destination.y = this.jumpY;
                        this.Sprite.Destination.width = this.spriteWidth;
                        this.Sprite.Destination.height = this.spriteHeight;

                        //low down The Runner
                    } else if (this.jumpState >= this.JUMP_DURATION / 3 * 2
&& this.jumpState < this.JUMP_DURATION) {
                        //set second jump frame
```

```
                        this.currentSpriteIndex = 8;

                        this.Sprite.Source.x =
Math.floor(this.currentSpriteIndex) * this.spriteWidth;
                        this.Sprite.Source.y = 0;
                        this.Sprite.Source.width = this.spriteWidth;
                        this.Sprite.Source.height = this.spriteHeight;


                        this.jumpY = this.jumpY + Math.abs(this.jumpLimit *
((this.jumpState - this.JUMP_DURATION / 3 * 2) / this.JUMP_DURATION / 3));

                        if (this.jumpY > this.initialY) {
                            this.jumpY = this.initialY;
                        }

                        //raise up the Runner
                        this.Sprite.Destination.x = 25;
                        this.Sprite.Destination.y = this.jumpY;
                        this.Sprite.Destination.width = this.spriteWidth;
                        this.Sprite.Destination.height = this.spriteHeight;

                        //jump is finished, return to running
                    } else {
                        this.jumpState = 0;
                        this.jumpY = this.initialY;
                        this.runnerState = this.RUNNER_STATE.RUNNING;
                    }

                    this.jumpState += speed;
                    break;

            }

        }

        draw() {
            this.ctx.drawImage(this.spritesheet,
                this.Sprite.Source.x,
                this.Sprite.Source.y,
                this.Sprite.Source.width,
                this.Sprite.Source.height,
                this.Sprite.Destination.x,
                this.Sprite.Destination.y,
                this.Sprite.Destination.width,
                this.Sprite.Destination.height);
        }

        jump() {
            if (this.runnerState == this.RUNNER_STATE.RUNNING) {
                this.runnerState = this.RUNNER_STATE.JUMPING;
            }
        }

        startRunning() {
            this.currentSpriteIndex = 2;
            this.runnerState = this.RUNNER_STATE.RUNNING;
```

```
        }
        stopRunning() {
            this.currentSpriteIndex = 0;
            this.runnerState = this.RUNNER_STATE.STANDING;
        }
    }
```

Klasa `Runner` poseduje sledeće najznačajnije elemente:

- `constructor()` - konstruktorska funkcija unutar koje se obavlja deklarisanje i inicijalizovanje različitih svojstava koje će imati objekat koji predstavlja glavnog junaka igre
- `load()` - metoda za učitavanje spritesheet-a; učitavanje funkcioniše kao i kod objekata za predstavljanje pozadine
- `update()` - metoda za ažuriranje grafike koja predstavlja glavnog junaka; metoda je korišćenjem jednog `switch` bloka podeljena na tri segmenta koji odgovaraju stanjima u kojima se glavni junak može naći (mirovanje, trčanje, skok)
- `draw()` - metoda za crtanje sprajta koji predstavlja glavnoj junaka igre
- `jump()` - metoda koja će inicirati stanje skoka
- `startRunning()` - metoda koja će inicirati stanje tračanja
- `stopRunning()` - metoda koja će inicirati stanje mirovanja

Napomena

O samoj unutrašnjoj logici klase `Runner`, možete više da saznate u video lekciji.

Obrada korisničke interakcije

Nakon realizacije glavnog junaka igre, biće prikazana logika za obradu korisničke interakcije. Korisnička interakcija će se odnositi na mogućnost korisnika da pokrene igru i da izvrši skok. Obe akcije obavljaće se pritiskom na taster space ili klikom bilo gde unutar površine našeg `canvas` elementa.

Za početak, potrebno je da učinimo da igra ne započinje automatski, prilikom učitavanja HTML dokumenta. Tako nešto ćemo obaviti korišćenjem sledeće logike unutar glavne petlje:

```
let running = false;
function main(currentTime) {
    if (running) {
        window.requestAnimationFrame(main);
    }
    delta = parseInt(currentTime - previousTime);
    speed = Math.abs(speedFactor * delta / BASE_SPEED);
    clearCanvas();
    update();
    draw();
    showStats();
    previousTime = currentTime;
    if (speed < 8) {
        speedFactor += 0.001;
    }

}
```

Kao što možete da vidite, uvedena je jedna nova promenljiva - `running`. Reč je o promenljivoj pomoću koje se kontroliše izvršavanje glavne petlje. Početna vrednost ove promenljive je `false`, što znači da će se nakon učitavanja HTML dokumenta, glavna petlja izvršiti samo jednom. To će biti dovoljno da se svi elementi naše igre iscrtaju jednom. Ipak, kako bi se animacija igre pokrenula, neophodno je da vrednost promenljive `running` postane `true` i da se još jednom obavi poziv metode `main()`. Mi ćemo takav posao obaviti na sledeći način.

Kod koji je potrebno postaviti na kraj `init()` funkcije:

```
document.body.onkeyup = function(e) {
    if (e.keyCode == 32) {
        handleUserAction();
    }
}

canvas.onclick = handleUserAction;
```

Na ovaj način je obavljena pretplata na dva događaja - na pritisak tastera space na tastaturi i na klik na taster miša. U oba slučaja će se aktivirati identična funkcija:

```
function handleUserAction() {
    if (!running) {
        running = true;
        window.requestAnimationFrame(main);
        runner.startRunning();
    } else {
        runner.jump();
    }
}
```

Prvim klikom na miš ili pritiskom na taster space, pokreće se glavna petlja i vrednost promenljive `running` postaje `true`. Na taj način započinje animacija. Svakim narednim klikom na taster miša ili dugme space, obavlja se pozivanje `jump()` metode `Runner` objekta, čime se inicira skok.

Pitanje

Koji `keyCode` ima taster space na tastaturi?

a) 31
b) 32
c) 33
d) 133

Objašnjenje:

U upravo prikazanom primeru obrade korisničke interakcije, možete da vidite da taster space ima kod 32. Takav kod se koristi kako bi se detektovalo da je korisnik kliknuo baš na space, a ne na neki drugi karakter.

Realizacija protivnika

Protivnici su poslednji element *Jump & Run* igre, o kome još nije bilo reči. Unutar *Jump & Run* igre, postoji ukupno 5 različitih protivnika, koji se u proizvoljnim trenucima pojavljuju duž staze kojom naš glavni junak trči. Tajming kreiranja neprijatelja i rastojanje između njih će biti proizvoljni.

Protivnici unutar *Jump & Run* igre biće realizovani korišćenjem dve klase:

- `Enemies` - klasa unutar koje će se objediniti funkcionalnosti za prikaz neprijatelja; reč je o klasi koja će biti zadužena da na početku igre kreira objekte svih neprijatelja i da zatim brine o njihovom nasumičnom prikazivanju
- `Enemy` - klasa unutar koje će biti enkapsulirane funkcionalnosti koje se tiču jednog neprijatelja

Osnovna klasa za rukovanje neprijateljima jeste klasa `Enemies`. Klasa za reprezentovanje pojedinačnih neprijatelja jeste `Enemy`. Bitno je reći da će svi neprijatelji biti kreirani prilikom instanciranja `Enemies` klase i da će tom prilikom biti smešteni unutar jednog niza. Zatim će se iz takvog niza, u određenim trenucima, prikazivati proizvoljni neprijatelji. Kada neprijatelj izađe iz okvira `canvas` elementa, njegov objekat se neće uništiti, već samo vratiti na početnu poziciju, neposredno izvan desne ivice `canvas` elementa, kako bi bio u stanju pripravnosti za neko naredno prikazivanje. Na taj način će biti kreiran takozvani *pool* objekata tipa `Enemy`, čime će se izbeći uništavanje i ponovno kreiranje objekata tokom izvršavanja igre. Upravo opisana logika klase `Enemies`, izgledaće ovako:

```
import { Enemy } from './enemy.js';

export class Enemies {

    constructor(ctx) {
        this.ctx = ctx;
        this.gameWidth = ctx.canvas.width;
        this.gameHeight = ctx.canvas.height;
        this.images = ['spritesheets/obstacle1.png',
            'spritesheets/obstacle2.png',
            'spritesheets/obstacle3.png',
            'spritesheets/obstacle4.png',
            'spritesheets/obstacle5.png'
        ];

        this.list = [];
        this.loaderCounter = 0;
        this.notifyLoaded;

        this.accumulator = 0;
        this.obstacleTrigger;
    }

    load(loaded) {

        this.notifyLoaded = loaded;

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

```
                let enemy = new Enemy(this.ctx, this.images[i]);
                this.list.push(enemy);

                enemy.load(loaded);

            }

        }


        loaded() {
            this.loaderCounter++;

            if (this.loaderCounter < 5)
                return;

            this.notifyLoaded();
        }



        update(speed) {

            this.accumulator += speed;
            //obstacleTrigger determines when new obstacle is going to be
created
            if (!this.obstacleTrigger) {
                this.obstacleTrigger = this.randomNumber(200, 600);
            }

            //when is time to show new obstacle
            if (this.accumulator > this.obstacleTrigger) {

                //create new obstacle and add it to obstacle list
                let enemy = this.list[this.randomNumber(0, 4)];
                enemy.isActive = true;

                //reset control variable and frame counter for next obstacle
creation
                this.obstacleTrigger = undefined;
                this.accumulator = 0;

            }

            //update every obstacle on screen
            //loop trough obstacle list and update
            for (let i = 0; i < this.list.length; i++) {
                //if obstacle is visible call update method
                if (this.list[i].isActive) {
                    this.list[i].update(speed);
                }
            }
        }

        draw() {
            //update every obstacle on screen
            //loop trough obstacle list and update
```

```
            for (let i = 0; i < this.list.length; i++) {
                //if obstacle is visible call update method
                if (this.list[i].isActive) {
                    this.list[i].draw();
                }
            }
        }
        reset() {
            for (let i = 0; i < this.list.length; i++) {
                this.list[i].reset();
            }
            this.accumulator = 0;
            this.obstacleTrigger;
        }

        randomNumber(min, max) {
            return Math.floor(Math.random() * (max - min + 1)) + min;
        }

    }
```

Kao što je rečeno, klasa `Enemies`, koristi i klasu `Enemy` za modelovanje pojedinačnih neprijatelja:

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

    export class Enemy {

        constructor(ctx, imageUrl) {
            this.ctx = ctx;
            this.gameWidth = ctx.canvas.width;
            this.gameHeight = ctx.canvas.height;
            this.horizontalOffset = 1;
            this.imageUrl = imageUrl;
            this.x;
            this.y;

            this.velocity = 2;

            this.isActive = false;
        }

        load(loadingFinished) {

            this.image = new Image();

            this.image.addEventListener('load', () => {
                this.height = this.image.height;
                this.width = this.image.width;
                this.x = this.gameWidth;
                this.y = this.gameHeight - this.height -
parseInt(this.gameHeight * 0.12);

                loadingFinished();
            });
```

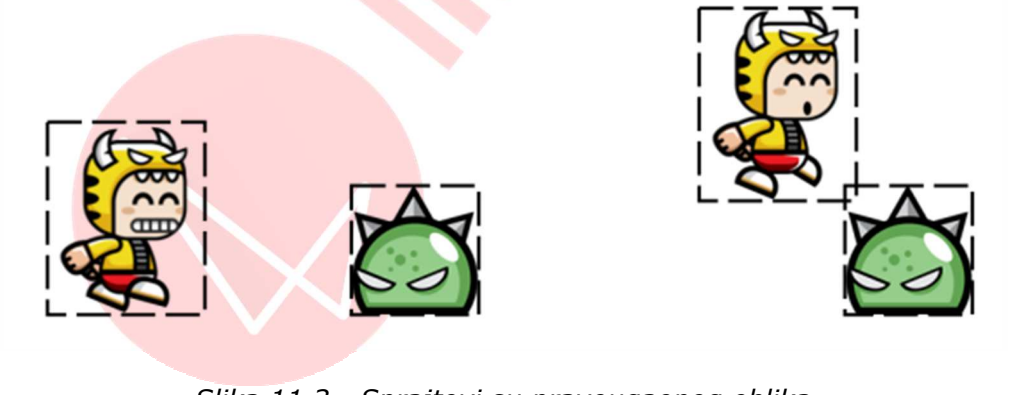
```
            this.image.src = this.imageUrl;
        }
        update(speed) {
            this.x -= this.velocity * speed;
            if (this.x < 0 - this.width) {
                this.reset();
            }
        }
        reset() {
            this.isActive = false;
            this.x = this.gameWidth;
        }

        draw() {
            this.ctx.drawImage(this.image, this.x, this.y);
        }
    }
```

Detekcija kolizije

Nakon kreiranja glavnog junaka i neprijatelja, kao logičan se nameće korak detekcije kolizije između takve dve vrste elemenata naše igre. Naime, prilikom kontakta našeg glavnog junaka i nekog od neprijatelja, potrebno je doći do negativnog okončanja igre. Stoga je potrebno kreirati određeni sistem za detekciju dodira glavnog junaka i neprijatelja.

Kako biste razumeli na koji način će logika za detekciju koliziju da funkcioniše, potrebno je da razumete način na koji se sprajtovi crtaju unutar canvas elementa. Naime, svaki sprajt jeste pravougaonik unutar koga se nalazi grafika sprajta. Ukoliko bi se prikazali okviri takvih imaginarnih pravougaonika, dobili bismo prikaz kao na slici 11.3.



Slika 11.3 - Sprajtovi su pravougaonog oblika

Detekcija kolizije se može realizovati logikom koja bi utvrđivala trenutak u kome dođe do kontakta između pravougaonika koji predstavlja granice sprajta glavnog junaka i pravougaonika koji predstavlja granice sprajta nekog od protivnika. Ipak, sa slike 11.3 možete da vidite da takav sistem ne bi u potpunosti bio merodavan. Naime na desnoj polovni slike 11.3 možete videti da su se pravougaonici dva sprajta dodirnuli iako se vizuelno, kontakt između glavnoj junaka i neprijatelja još uvek nije dogodio. Zbog ovakvih situacija, mi detekciju kolizije nećemo utvrđivati korišćenjem pravougaonih okvira sprajtova. Pravougaone okvire zamenićemo krugovima (slika 11.4).

Slika 11.4 - Imaginarni krugovi u kojima se nalaze sprajtovi

Sa slike 11.4 možete videti da su kružni okviri sprajtova znatno prikladniji za detekciju kolizije u našoj situaciji. Kako bi se stvorili ovakvi imaginarni krugovi i na osnovu njih obavila detekcije kolizije, biće korišćena sledeća funkcija:

```
      function detectCollision() {

          for (let i = 0; i < enemies.list.length; i++) {
              if (enemies.list[i].isActive) {

                  let circle1 = {
                      radius: runner.Sprite.Destination.width * 0.4,
                      x: runner.Sprite.Destination.x +
runner.Sprite.Destination.width / 2,
                      y: runner.Sprite.Destination.y +
runner.Sprite.Destination.height / 2
                  };
                  let circle2 = {
                      radius: enemies.list[i].width * 0.4,
                      x: enemies.list[i].x + enemies.list[i].width / 2,
                      y: enemies.list[i].y + enemies.list[i].height / 2
                  };

                  var dx = circle1.x - circle2.x;
                  var dy = circle1.y - circle2.y;
                  var distance = Math.sqrt(dx * dx + dy * dy);

                  if (distance < circle1.radius + circle2.radius) {
                      isGameOver = true;
                  }

              }
          }
      }
```

Funkcija `detectCollision()` pozivaće se u svakoj iteraciji petlje. Unutar nje se utvrđuje kolizija između imaginarnog kruga koji predstavlja glavnoj junaka i imaginarnih krugova kojima se predstavlja svaki od aktivnih neprijatelja.

Imaginarni krugovi se kreiraju na osnovu pozicije i veličine sprajtova. Takvi krugovi se unutar prikazane funkcije predstavljaju promenljivim `circle1` i `circle2`. Za poluprečnik krugova se uzima 40% širine sprajta, dok se za centar krugova postavlja centar sprajta. Centar sprajta se dobija tako što se na `x` i `y` koordinate, dodaje polovina širine i visine, respektivno.

Poluprečnici su postavljeni na 40% širine sprajta, a ne na 50%, kako bi se omogućio nešto dublji kontakt između glavnoj junaka i neprijatelja i kako se ne bi događale situacije da se kolizija detektuje i kada nje vizuelno nema, zbog neregularnog oblika grafike koja predstavlja sprajtove.

Nakon kreiranja imaginarnih krugova, njihova kolizija se detektuje praćenjem rastojanja između njihovih centara. Ukoliko je rastojanje između dva kruga manje od zbira njihovih poluprečnika, to je signal da je došlo do kolizije. U tom slučaju se vrednost promenljive `isGameOver` postavlja na `true`.

Konkretno rastojanje između centara krugova, dobija se Pitagorinom teoremom.

Funkciju `detectCollision()` je potrebno pozivati na kraju `update()` metode.

Kraj igre

U prethodnim redovima ste mogli da vidite da se prilikom detekcije kolizije, vrednost promenljive `isGameOver` postavlja na `true`. Vrednost ove promenljive se proverava unutar `draw()` funkcije:

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

    if (isGameOver) {
        gameOver();
    }
}
```

Kada se utvrdi da je `isGameOver` `true`, poziva se metoda `gameOver()`:

```
function gameOver() {

    running = false;

    ctx.font = 'bold 80px Luckiest Guy';
    ctx.fillStyle = "white";

    let text = ctx.measureText('Game Over');

    ctx.fillText('Game Over', ctx.canvas.width / 2 - text.width / 2,
ctx.canvas.height / 2);

}
```

Metoda `gameOver()` obavlja dva zadatka: zaustavlja glavnu petlju i ispisuje poruku *Game Over*. Glavna petlja se zaustavlja postavljanjem promenljive `running` na `false`. Crtanje *Game Over* teksta obavlja se korišćenjem `fillText()` metode i nekoliko svojstava za stilizovanje.

Takođe, kako bi tekst bio centriran unutar canvas-a, koristi se i metoda measureText() kojom se utvrđuje koliki prostor će tekst zauzima unutar canvas elementa.

Svojstva i metode za rad sa tekstom

Metoda za crtanje teksta unutar canvas elementa je fillText():

```
fillText(text, x, y)
```

Metoda za crtanje teksta, prihvata sledeće parametre:

- text - tekst koji je potrebno nacrtati
- x - x koordinata početne tačke za ispisivanje teksta (gde će tekst unutar canvas-a biti nacrtan)
- y - y koordinata početne tačke za ispisivanje teksta

Veličina, familija i stil teksta, definišu se svojstvom font:

```
ctx.font = 'bold 80px Luckiest Guy';
```

Boja teksta, definiše se svojstvom fillStyle:

```
ctx.fillStyle = "white";
```

Prostor koji će po širini tekst da zauzme unutar canvas elementa, dobija se metodom measureText():

```
let text = ctx.measureText('Game Over');
```

Reč je o metodi koja omogućava da se tekst izmeri pre nego što bude nacrtan i da se na taj način stekne uvid u prostor koji će biti potreban za njegov prikaz.

Bitno je da primetite da se prilikom crtanja teksta *Game Over* koristi jedan web font koji je na sledeći način uključen u naš projekat:

```
<link
href="https://fonts.googleapis.com/css2?family=Luckiest+Guy&display=swap
" rel="stylesheet">
```

Reč je o jednom od Google Web fontova, a prikazani link element je potrebno da se nađe unutar head odeljka našeg HTML dokumenta. Ipak, bitno je znati da ukoliko se ovakav web font ne koristi nigde unutar HTML dokumenta, već samo za crtanje u canvas elementu, web pregledač će zahtev za njegovim učitavanjem poslati tek kada naredba za crtanje teksta dođe na izvršavanje. Stoga se može dogoditi da inicijalno crtanje teksta bude obavljeno upotrebom nekog drugog, podrazumevanog fonta, dok web pregledač u potpunosti ne obavi učitavanje web fonta. Najlakši način za prevazilaženje ovoga problema, jeste da se konkretan web font upotrebi i na nekom HTML elementu dokumenta. Na taj način će ga web pregledač učitati tokom parsiranja dokumenta. Mi u našem primeru, možemo na primer, da logo koji je do sada bio predstavljen korišćenjem vektora, pretvorimo u tekst koji će za vrednost font-family CSS svojstva da ima Luckiest Guy:

```
<h1 style="font-family: Luckiest Guy">Jump & Run</h1>
```

Naravno, potrebno je da adekvatno izmenite i stilizaciju, tako da ovakav tekst izgleda kao i vektor koji je do sada korišćen.

Dodavanje zvuka

Unutar igre *Jump & Run* određeni događaji propraćeni su reprodukcijom odgovarajućeg zvuka. Reč je o veoma kratkim zvucima koji se reprodukuju kada:

- započne igra
- glavni junak napravi skok
- glavni junak ostvari kontakt sa neprijateljom

Zvukovi koje je potrebno reprodukovati nalaze se unutar foldera `sounds`:

- `click.wav` - kada igre započne
- `fail.wav` - kada dođe do kontakta između glavnoj junaka i nekog od protivnika
- `jump.wav` - kada glavni junak napravi skok

Reprodukcija zvukova podrazumevaće programabilno kreiranja `audio` elemenata za svaki od ova tri zvuka. Tako kreirani `audio` elementi postojaće samo unutar interne memorije web pregledača, odnosno mi njih nećemo prikazivati na stranici.

Kompletna logika za učitavanje zvučnih efekata, biće enksapulirana unutar jedne konstruktorske funkcije:

```
function Sound(src) {
    this.sound = document.createElement("audio");
    this.sound.src = src;
    this.sound.setAttribute("preload", "auto");
    this.sound.setAttribute("controls", "none");

    this.play = function() {
        this.sound.play();
    }
    this.stop = function() {
        this.sound.pause();
    }
}
```

Ovo je konstruktorska funkcija za kreiranje objekata koji će predstavljati zvučne efekte. Funkciji `Sound()` se prosleđuje putanja do fajla koji predstavlja zvučni efekat, a ona obavlja kreiranje i konfigurisanje `audio` elementa.

Kreiranje svih zvučnih efekata se može obaviti na sledeći način:

```
let startSound;
let jumpSound;
let endSound;

function loadAudio() {
    startSound = new Sound("sounds/click.wav");
    jumpSound = new Sound("sounds/jump.wav");
    endSound = new Sound("sounds/fail.wav");
}
```

Na ovaj način se kreiraju tri objekta, koji predstavljaju zvučne efekte naše igre. Funkciju `loadAudio()` je potrebno pozvati unutar `init()` metode.

Na kraju, zvučne efekte je potrebno reprodukovati u odgovarajućim trenucima. Prvo, unutar funkcije za obradu korisničke interakcije:

```
function handleUserAction() {

    if (!running) {
        resetGame();
        window.requestAnimationFrame(main);
        runner.startRunning();
        startSound.play();
    } else {
        if (runner.jump()) {
            jumpSound.play();
        }
    }
}
```

Kada igra započne (if uslovni blok), reprodukuje se `startSound`, pozivanjem metode `play()`. Kada korisnik napravi skok, reprodukuje se `jumpSound()`. Ipak, bitno je da primetite da je reprodukcija zvuka koji prati skok, upakovana unutar jednog uslovnog bloka. Naime, skok je radnja koja traje neko vreme, a korisnika ništa ne sprečava da tokom trajanja skoka pokušava da ponovo inicira skok klikom na dugme space ili taster miša. U takvim situacijama, bez navedene provere, svakim novih klikom bio bi reprodukovan zvuk koji predstavlja skok. Stoga je logika metoda `jump()`, klase `Runner` proširena, tako da vraća jednu `boolean` vrednost:

```
    jump() {
        if (this.runnerState == this.RUNNER_STATE.RUNNING) {
            this.runnerState = this.RUNNER_STATE.JUMPING;
            return true;
        }

        return false;
    }
```

Sve dok je skok u toku, metoda `jump()` će emitovati `false` povratnu vrednost. Takva povratna vrednost se kao što ste mogli da vidite, koristi kao kontrolna vrednost, u odnosu na koju se vrši reprodukovanje zvuka koji predstavlja skok.

Na kraju, zvučni efekat je potrebno reprodukovati i prilikom kontakta glavnog junaka i nekog od neprijatelja:

```
function gameOver() {
    endSound.play();
    running = false;
    ctx.font = 'bold 80px Luckiest Guy';
    ctx.fillStyle = "white";
    let text = ctx.measureText('Game Over');
    ctx.fillText('Game Over', ctx.canvas.width / 2 - text.width / 2,
ctx.canvas.height / 2);

}
```

Na početak metode `gameOver()` je postavljena naredba za reprodukciju zvuka koji predstavlja neuspešan završetak igre.

Prikaz Scora-a

Kako bi korisnik bio u mogućnosti da prati progres i izmeri uspešnost, unutar Jump & Run igre obavićemo i prikaz scora-a:

```
let score = 0;

function drawScore() {

    score += speed;

    let scoreText = "Score: " + parseInt(score);

    ctx.font = 'bold 36px Luckiest Guy';
    ctx.fillStyle = "white";

    let text = ctx.measureText(scoreText);

    ctx.fillText(scoreText, 50, 50);

}
```

Score će se računati unutar promenljive `score`, tako što će u svakoj iteraciji petlje, brzina igre da bude dodavana vrednosti `score` promenljive. Score će se prikazivati u gornjem, levom uglu igre, a za crtanje teksta koji predstavlja `score` iskorišćeni su već viđeni pristupi za crtanje teksta.

Metodu `drawScore()` je potrebno pozivati unutar `draw()` metode:

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

    if (running) {
        drawScore();
    }

    if (isGameOver) {
        gameOver();
    }
}
```

Unutar metode `draw()` je definisano da će se `score` računati i prikazivati, samo kada je igra aktivna, proverom vrednosti promenljive `running`.

Napomena

Kompletan proces kreiranja *Jump & Run* igre ilustrovan je u pratećim video lekcijama. Takođe, kompletan projekat sa završenom *Jump & Run* igrom, možete da preuzmete sa sledećeg linka:

`jump_and_run_FINAL.rar`

Rezime

- glavni junak igre se ne pomera po horizontalnoj osi, dok se po vertikalnoj osi može pomeriti samo tokom skoka
- glavni junak može se naći u tri različita stanja: mirovanje, trčanje, skok
- grafika glavnog junaka definisana je korišćenjem jednog skupa sprajtova
- sprajt se definiše kao dvodimenzionalna slika kojom se predstavlja jedan element igre
- grupa sprajtova se drugačije naziva spritesheet
- obrada korisničke interakcije podrazumeva pretplatu na događaje klika na taster miša i pritiska dugmeta space na tastaturi
- u igri *Jump & Run* postoji ukupno 5 protivnika koji se u proizvoljnim trenucima pojavljuju duž staze kojom naš glavni junak trči
- kada neprijatelj izađe iz okvira `canvas` elementa, njegov objekat se ne uništava, već samo vraća na početnu poziciju, neposredno izvan desne ivice `canvas` elementa
- detekcija kolizije se obavlja tako što se utvrđuje kontakt između imaginarnih krugova kojima se interno predstavljaju sprajtovi
- u igri Jump & Run koriste se tri zvučna efekta, koji se reprodukuju kada igra započne, kada korisnik napravi skok i kada dođe do kontakta između glavnog junaka i neprijatelja